

JAVASCRIPT

1. EXPRESSION & STATEMENT

Expression adalah suatu nilai / menghasilkan nilai.

```
// Expression
5;
3+2;
```

Statement adalah suatu aksi atau kondisi. Salah satunya deklarasi variable dan sebagainya.

```
// Statement
var name;
let age;
```

2. COMMENTS

Memberikan komentar pada code dapat menggunakan “//” atau “/* */”

```
// one row
/* first row
second row */
```

3. VARIABLE

Variable biasanya digunakan untuk menyimpan suatu nilai. Terdapat tiga jenis variable pada javascript yaitu :

- a. **Var** : Sangat tidak disarankan karena dapat menimbulkan bug
- b. **Let** : Dapat menyimpan dan mengubah nilai secara langsung. Deklarasi Variable biasanya **let statement = expression;**

```
let myName;
console.log(myName); // output : undefined
// need to assign value
myName = "John";
console.log(myName); // output : John

let myAge = 20; // declare and assign value
```

Note : let expression1 = (statement2 = expression); // outpunya error

Kecuali : let statement1 = (expression1 = expression2);

c. **Const** : Dapat menyimpan tapi sulit diubah secara langsung

```
const _construct = "Harus ada Value";  
const myId = "19990101"; // constant variable  
const kd_fjpp7 = 1070108; // huruf, angka, underscore, $  
const $dolar = 1000;
```

Lalu, tidak dapat diubah seperti ini, karena sifat constant

```
const z = 100;  
console.log(z);  
z = 200;  
console.log(z); // error
```

Note*

- Harus dimulai dengan huruf atau underscore (_).
- Dapat terdiri dari huruf, angka, dan underscore (_) dengan berbagai kombinasi.
- Tidak boleh mengandung spasi (whitespace). Jika penamaan variabel lebih dari dua kata, tuliskan secara camelCase. Contoh firstName, lastName, catName, dll.
- Tidak boleh mengandung karakter spesial (! , / \ + * = dll.)

4. TIPE DATA

Tipe data merupakan jenis/klasifikasi data berdasarkan kesamaan, misalnya tanggal, angka, huruf dsb.

a. **Undefined** : Ketika kita mendeklarasikan suatu variabel tapi **tidak menyimpan nilai**

```
let x;  
console.log(typeof (x)); // output : undefined
```

b. **Numbers** : Nilai yang disimpan **berupa angka**. Baik bilangan decimal atau bilangan bulat

```
let xy = 10;  
console.log(typeof (xy)) /* output: number */  
  
let yz = 17.25;  
console.log(typeof (yz)) /* output: number */
```

Operator :

Operator	Fungsi	Contoh
+	Penjumlahan	10 + 10 = 20

-	Pengurangan	$10 - 5 = 5$
/	Pembagian	$10 / 5 = 2$
*	Perkalian	$10 * 5 = 50$
%	Sisa Hasil Bagi	$10 \% 3 = 1$
**	Perpangkatan	$10 ** 2 = 100$
++	Ditambah 1	10++ atau ++10 Ditambah setelah output atau ditambah duluan sebelum output
--	Dikurangi 1	10-- atau --10

- c. **BigInt** : Ketika nilai **number** tidak cukup, maka kita membutuhkan tipe **BigInt**, perbedaannya terletak pada **karakter n diakhir Angka**. Selain itu, dia juga dapat menyimpan angka kecil juga lohyyy.

```
const bigNumber = 1234567890123456789012345678901234567890n;
const myInt = 1234567890123456789012345678901234567890;

console.log(bigNumber);
console.log(myInt);

/* output
1234567890123456789012345678901234567890n
1.2345678901234568e+39
*/
```

Operator :

```
console.log(5n + 2n); // output : 7n
console.log(5n - 2n); // output : 3n
console.log(5n * 2n); // output : 10n
console.log(5n / 2n); // output : 2n, bukan 2.5n
console.log(5n % 2n); // output : 1n
```

- d. **Strings** : Nilai berupa teks yang di tambahkan tanda khusus yaitu “ teks “ / ‘ teks ’ / ` teks ` . Solusinya, gunakan **backslash(\)** untuk mengurangi ambiguitas dalam **tanda petik**.

```
let greet = "Hello";
let moreGreet = 'Hallo';
let greetAgain = `Hai`;
console.log(typeof (greet)) /* output: string */
console.log(typeof (moreGreet)) /* output: string */
console.log(typeof (greetAgain)) /* output: string */
```

String concatenation : Menggabungkan string dengan menggunakan tanda tambah (+)

```
let otherGreet = greet + " " + greetAgain;
console.log(otherGreet) /* output: Hello Hai */
```

String Interpolation : Memasukkan nilai dari **variable** ke dalam teks string dengan menggunakan **backticks (`)** dan **\$(variable)** didalamnya

```
const namaLengkap = "Dimas Rendy Sugara";
console.log(`Hello, my name is ${namaLengkap}.`);
/* output: Hello, my name is Dimas Rendy Sugara. */
```

- e. **Boolean** : Hanya memiliki nilai **True** atau **False**. Sering digunakan dalam pengkodisian

```
let myBool = true;
console.log(typeof (myBool))
/* output: boolean */
```

Contoh Pengkodisian :

```
const a = 10;
const b = 12;

let isGreater = a > b;
let isLess = a < b;

console.log(isGreater); // false
console.log(isLess); // true
```

- f. **Null** : Serupa dengan undefined, namun null **perlu diinisialisasikan pada variabel**. Null biasa digunakan sebagai **nilai sementara pada variabel**, tapi sebenarnya nilai tersebut “tidak ada”.

```
let someLaterData = null;
console.log(someLaterData); // output : null
```

- g. **Symbol** :]digunakan untuk menunjukkan **identifier yang unik**.

```
const id1 = Symbol("id");
const id2 = Symbol("id");
console.log(id1); // output : Symbol(id)
console.log(id2); // output : Symbol(id)
console.log(id1 == id2); // output : false, karena setiap symbol adalah unik
```

Note : Symbol ini umumnya digunakan sebagai **nama property** dari **Object**. Object sendiri merupakan tipe data kompleks untuk menyimpan berbagai struktur data. Kita akan segera bertemu dan mempelajari tentang object pada modul Data Structure.

5. OPERATOR

Operator dalam bahasa pemrograman sendiri adalah simbol yang memberi tahu interpreter untuk **melakukan operasi seperti matematika, relasional, atau logika** untuk memberikan hasil tertentu.

- a. **Assignment Operator (=)** : Operator ini digunakan untuk **memberikan nilai pada variabel**.

```
let ab = 10;
let bx = ab;
console.log(bx); // output : 10
```

Jika digunakan pada operator matematika

```
let ab = 10;
let bx = ab;
console.log(bx); // output : 10

ab += bx; // artinya -> ab = ab + bx; output : 20
ab -= bx; // artinya -> ab = ab - bx; output : 0
ab *= bx; // artinya -> ab = ab * bx; output : 100
ab /= bx; // artinya -> ab = ab / bx; output : 1
ab %= bx; // artinya -> ab = ab % bx; output : 0
```

- b. **Comparison Operator** : Operator **pembandingan** sebagai **logika dasar** dalam membandingkan nilai pada JavaScript.

Operator	Fungsi	Contoh
==	Membandingkan kedua nilai apakah sama (tidak identik).	<pre>let xyz = 10; let abc = 15; console.log(xyz == abc); // output : false</pre>

!=	Membandingkan kedua nilai apakah tidak sama (tidak identik).	<pre>console.log(xyz != abc); // output : true</pre>
===	Membandingkan kedua nilai apakah identic (termasuk tipe data).	<pre>console.log(xyz === abc); // output : false</pre>
!==	Membandingkan kedua nilai apakah tidak identik (termasuk tipe data).	<pre>console.log(xyz !== abc); // output : true</pre>
>	Membandingkan dua nilai apakah nilai pertama lebih dari nilai kedua.	<pre>console.log(xyz > abc); // output : false</pre>
>=	Membandingkan dua nilai apakah nilai pertama lebih atau sama dengan nilai kedua.	<pre>console.log(xyz >= abc); // output : false</pre>
<	Membandingkan dua nilai apakah nilai pertama kurang dari nilai kedua.	<pre>console.log(xyz < abc); // output : true</pre>
<=	Membandingkan dua nilai apakah nilai pertama kurang atau sama dengan nilai kedua.	<pre>console.log(xyz <= abc); // output : true</pre>

c. **Logical Operator** : Dengan logical operator, kita dapat **menggunakan kombinasi dari dua nilai boolean atau bahkan lebih dalam menetapkan logika.**

a. **&&** : Operator dan (and). Logika akan **menghasilkan nilai true apabila semua kondisi terpenuhi** (bernilai true).

```
let first = 10;
let second = 12;

console.log(first < 15 && second > 10); // (true && true) -> true
console.log(first > 15 && second > 10); // (false && true) -> false
```

b. **||** : Operator atau (or). Logika akan **menghasilkan nilai true apabila ada salah satu kondisi terpenuhi** (bernilai true).

```
console.log(first < 15 || second > 10); // (true || true) -> true
console.log(first > 15 || second > 10); // (false || true) -> true
```

- c. **!** : Operator tidak (**not**). Digunakan untuk **membalikkan suatu kondisi**.

```
console.log(!(first < 15)); // !(true) -> false
console.log(!(first < 15 && second > 10)); // !(true && true) -> !(true) -> false
```

6. IF/ELSE IF/ ELSE

Statement if akan menguji suatu kondisi. Jika kondisi **bernilai true**, maka blok kode di dalamnya akan **dijalankan**. Sebaliknya, jika **bernilai false**, maka proses yang ditentukan akan **dilewatkan**.

```
let jika = true;
if (jika) {
  console.log("Jika benar, maka ini akan dijalankan");
} else {
  console.log("Nilai Salah");
}
// output : Jika benar, maka ini akan dijalankan
```

Selain itu, kita juga dapat melakukan pengkondisian dalam perbandingan operator

```
let language = "French";
let greeting = "Selamat Pagi"

if (language === "English") {
  greeting = "Good Morning!";
} else if (language === "French") {
  greeting = "Bonjour!";
} else if (language === "Japanese") {
  greeting = "Ohayou Gozaimasu!";
}

console.log(greeting); // output : Bonjour!
```

7. TERNARY OPERATOR / CONDITIONAL EXPRESSION

Penulisan if – else statement dalam satu baris saja, Penulisannya sebagai berikut

condition ? true expression : false expression

Contoh :

```
const isMember = false;
const discount = isMember ? 0.1 : 0;
console.log(`Anda mendapatkan diskon sebesar ${discount * 100}%`);
// output : Anda mendapatkan diskon sebesar 0%
```

8. SWITCH CASE

Pengecekan berdasarkan kasus-kasus yang telah ditentukan, dengan format seperti berikut:

```
switch (expression / variable) {  
  case value1:  
    // do something;  
    break;  
  case value2:  
    // do something;  
    break;  
  default:  
    // do something;  
}
```

Contoh :

```
let country = "French";  
let time = null;  
  
switch (country) {  
  case "English":  
    time = "Good Morning!";  
    break;  
  case "French":  
    time = "Bonjour!";  
    break;  
  case "Japanese":  
    time = "Ohayou Gozaimasu!";  
    break;  
  default:  
    time = "Selamat Pagi!";  
}  
  
console.log(greeting);  
/* output Bonjour! */
```

9. LOOPING

Looping digunakan untuk **melakukan hal yang sama berulang-ulang**.

- a. **For loop** : Digunakan untuk melakukan pengulangan hingga mendapatkan pengkondisian bernilai true atau kondisi sudah memenuhi (bernilai false)

```
for (inisialisasi variabel; test kondisi; perubahan nilai variabel) {  
  // do something  
}
```

Contoh :

```
for (let i = 0; i < 5; i++) {  
  console.log(i);  
}  
  
/* output  
0  
1  
2  
3  
4 // berhenti ketika i = 5,  
   dan tidak menampilkan nilai lagi  
*/
```


- b. **For of loop** : Perulangan yang dilakukan dengan **Jumlah proses looping-nya pun akan menyesuaikan dengan ukuran dari array**. Array tersebut dimasukkan kedalam sebuah variable satu per satu.

```
let myArray = ["Luke", "Han", "Chewbacca", "Leia"];
/* for (const variableBaru of Array sebelumnya (myArray)){
  // do something
} */
for (const arrayItem of myArray) {
  console.log(arrayItem)
}

/* output
Luke
Han
Chewbacca
Leia
*/
```

- c. **While and do-while** : Sama seperti for, instruksi while **mengevaluasi ekspresi boolean dan menjalankan kode di dalam blok while ketika bernilai true**.

```
let i = 1;

while (i <= 100) {
  console.log(i);
  i++;
}

// output : 1 sampai 100
```

Bentuk lain dari do-while :

```
let loop = 1;

do {
  console.log(loop);
  loop++;
} while (loop <= 100);

// output : 1 sampai 100
```

- d. **Infinite looping** : Diakibatkan karena tidak adanya penambahan variable pembatas atau biasanya (i++);

```
let i = 1;

while (i <= 5) {
  console.log(i);
} infinite loops dengan output 1 sebanyak-banyaknya
```

10. OBJECT

Object mampu **menyimpan nilai dari beragam tipe data dan membentuk data yang lebih kompleks**. Untuk **menetapkan objek** pada sebuah variabel kita **gunakan tanda kurung kurawal {}**.

```
const object = {};
```

Object berisi pasangan **key dan value** yang juga dikenal dengan property. **Key** berperan mirip seperti nama variabel yang menyimpan sebuah nilai. Sementara, **value** berisi nilai dengan tipe data apa pun termasuk objek lain.

```
let object = { key1: "value1", key2: "value2", key3: "value3" }
let contohObject = {
  key1: "value1",
  $key2: "value2",
  _key3: "value3",
  "key keempat": "value4",
}
```

Note : Meskipun key merupakan string, kita tidak perlu menuliskan tanda petik kecuali ada karakter khusus seperti spasi.

Kemudian untuk **mengakses nilai dari properti object**, kita dapat memanggil nama object lalu tanda titik dan diikuti nama propertinya. Contoh:

```
`${namaObject.keyValue}` -> dipisahkan oleh titik
namaObject["keyValue"]; -> menggunakan kurung siku
```

```
const user = {
  firstName: "Luke",
  lastName: "Skywalker",
  age: 19,
  isJedi: true,
  "home world": "Tattooine"
};

console.log(`Halo, nama saya ${user.firstName} ${user.lastName}`);
console.log(`Umur saya ${user.age} tahun`);
console.log(`Saya berasal dari ${user["home world"]}`);

/* output
Halo, nama saya Luke Skywalker
Umur saya 19 tahun
Saya berasal dari Tattooine
*/
```

Contoh :

Untuk **mengubah nilai properti** di dalam object kita gunakan **assignment operator (=)**.

```
object.key1 = "value1"; // bentuk $, hanya untuk penggunaan string
user.age = 21;
user["home world"] = "Chefchaouen";
```

Kita juga dapat **menghapus property** pada **object** menggunakan keyword **“delete”** seperti berikut:

```
delete object.key1; // menghapus key isJedi
delete user.isJedi;
```

11. ARRAY

Array adalah **Sebuah Variable** yang menampung banyak nilai/ value yang dipisahkan oleh koma yang berada di dalam kurung siku [].

Perbedaan array dengan object adalah **data pada array** disusun secara **berurutan** dan diakses menggunakan **index**. Untuk **mengakses nilai di dalam array**, kita gunakan **tanda kurung siku []** yang di dalamnya berisi angka yang merupakan **posisi nilai yang ingin diakses**.

```
let varArray = ["Cokelat", 42.5, 22, true, "Programming"];
console.log(varArray);
console.log(varArray[1]);
console.log(typeof varArray);
console.log(Array.isArray(varArray));
/* output:
[ 'Cokelat', 42.5, 22, true, 'Programming' ]
42.5
object
true */
```

Note : Jika index diluar array maka nilai defaultnya adalah **undefined**. Dan Jika ingin mengubah sesuatu pada array, kita dapat melakukan berbagai aksi yaitu :

Nama Function	Fungsi	Contoh
contohArray.length	Menghitung jumlah value pada array	<pre>console.log("jumlah data saat ini: ", varArray.length);</pre>
contohArray.push("newValue");	Menambahkan value pada akhir array	<pre>const month = ['January', 'March', 'April', 'May']; month.push('June'); // June pada index terakhir console.log(month); // output: ['January', 'March', 'April', 'May', 'June']</pre>

<code>contohArray.pop("newValue")</code>	Menghapus value index terakhir	<pre>month.pop(); console.log(month); // output: ['January', 'March', 'April', 'May']</pre>
<code>contohArray.shift();</code>	Menghapus value index pertama	<pre>month.shift(); console.log(month); // output: ['March', 'April', 'May']</pre>
<code>contohArray.unshift("newValue")</code>	Menambahkan value index pertama	<pre>month.unshift('February'); console.log(month); // output: ['February', 'March', 'April', 'May']</pre>
<code>delete contohArray[index];</code>	Mengeluarkan value pada index yg dipilih, tapi indexnya tetap ada	<pre>delete month[1]; console.log(month); // output: ['February', <1 empty item>, 'April', 'May']</pre>
<code>contohArray.splice(index, jumlah value);</code>	Menghapus banyak value dari suatu index	<pre>month.splice(0, 2); console.log(month); // output: ['April', 'May']</pre>
<code>contohArray.splice(index, jumlah value dihapus (0 tidak ada, dan 1 seterusnya jumlah value yg dihapus), "newValue", "newValue", "dan seterusnya")</code>	Menambah banyak value pada index yang diinginkan	<pre>month.splice(1, 0, 'June', 'July', 'August'); console.log(month); // output: ['April', 'June', 'July', 'August', 'May']</pre>
<code>contohArray.filter((perItem, indexItem) => { //do something });</code>	Menentukan apakah currentValue(item) ada didalam index kalau true masuk ke array kalo false diabaikan	<pre>const uniqueValue = contohArray.filter((item, index) => args.indexOf(item) === index); atau biar lebih mudah dipahami const uniqueValue = contohArray.filter((item, index) => { return args.indexOf(item) === index });</pre>
<code>contohArray.indexOf(item)</code>	Menentukan apakah item tersebut ada pada salah satu index	<pre>contohArray.indexOf('June') // 2</pre>

12. SPREAD OPERATOR

Spread operator digunakan untuk mengubah bentuk array menjadi string.

```
const favorites = ["Seafood", "Salad", "Nugget", "Soup"];
console.log(favorites);
console.log(...favorites);
/* output
[ 'Seafood', 'Salad', 'Nugget', 'Soup' ]
Seafood Salad Nugget Soup
*/
```

Note : Spread operator dapat digunakan untuk **menggabungkan dua buah array ke dalam array baru**. Selain array, hal yang sama berlaku untuk object.

```
const favorites = ["Seafood", "Salad", "Nugget", "Soup"];
const others = ["Cake", "Pie", "Donut"];

const allFavorites = [...favorites, ...others];
console.log(allFavorites);
/* output
[ 'Seafood', 'Salad', 'Nugget', 'Soup', 'Cake', 'Pie', 'Donut' ]
*/
```

13. DESTRUCTURING OBJECT

Pada **pengelolaan data dengan format JSON** (JavaScript Object Notation), **contoh format json** :

```
let data = [
  {
    userId: 1,
    nameId: "Dimas Rendy",
  },
  {
    "userId": 2,
    "nameId": "Fulan bin Fulan",
  }
]
```

- a. **Destructuring Object** : Menggunakan **object literal { }** dengan posisi **diawal**, dan **variable nya belum pernah di deklarasikan**

```
let data = [
  {
    userId: 1,
    nameId: "Dimas Rendy",
  },
  {
    "userId": 2,
    "nameId": "Fulan bin Fulan",
  }
]

let { userId, nameId } = data[0];
console.log(userId); // output : 1
console.log(nameId); // output : Dimas Rendy
```

- b. **Destructuring Assignment** : Perbedaan dengan destruct object adalah **variable sudah pernah di deklarasikan sebelumnya**. Oleh karena itu, perlu **ditambahkan kurung ()** didalamnya **expression dan statement**

```
let firstName = "Dimas";
let lastName = "Rendy";

const profile = {
  firstName: "John",
  lastName: "Doe",
};

({ firstName, lastName } = profile);

console.log(firstName);
console.log(lastName);
/* output
John
Doe
*/
```

- c. **Default Values** : Jika terdapat data yang kosong, atau variable tidak ada pada data json. Kita dapat **membuat default value** agar data tidak 'undefined' dan jika data yang dimaksud ternyata ada, **maka data json yang ditampilkan bukan default value**.

```
let firstName = "Dimas";
let lastName = "Rendy";

const profile = {
  firstName: "John",
  lastName: "Doe",
};

({ firstName, lastName, userName = "tidak ada" } = profile);

console.log(firstName);
console.log(lastName);
console.log(userName);
/* output
John
Doe
tidak ada
*/
```

- d. **Assigning to Different Local Variable Names** : Ketika **Mengubah nama variable** atau **variable tempat penyimpanan object** berbeda dengan **object**. Perbedaannya terletak pada **penggunaan titik dua** :

```
const profile = {
  firstName: "John",
  lastName: "Doe",
};

let { firstName: namaDepan, lastName: namaBelakang } = profile;

console.log(namaDepan);
console.log(namaBelakang);
/* output
John
Doe
*/
```

14. DESTRUCTURING ARRAY

Destructuring pada array menggunakan **tanda kurung siku []** dan **bekerja sesuai urutan indexnya**.

```
const buku = ["Game of Thrones", "Harry Potter", "Lord of the Rings"];
let [katalog1, katalog2, katalog3] = buku;
console.log(katalog1);
console.log(katalog2);
console.log(katalog3);
// game of thrones
// harry potter
// lord of the rings
```

Cara untuk destructuring array beberapa data saja dapat dilakukan dengan mengosongkan variable dan menyisakan koma saja.

```
const buku = ["Game of Thrones", "Harry Potter", "Lord of the Rings"];
let [, , katalog3] = buku;
console.log(katalog3);
// lord of the rings
```

- a. **Destructuring Assignment** : sama seperti object, yaitu jika sudah pernah di deklarasikan **tapi tidak perlu tanda kurung**

```
let namaBuku = "Buku Default";
const buku = ["Game of Thrones", "Harry Potter", "Lord of the Rings"];
[namaBuku] = buku;
console.log(namaBuku);
// output: Game of Thrones
```

Selain itu, pertukaran nilai pada Array dapat dilakukan dengan hanya :

```
[a, b] = [b, a];
```

- b. **Default Values** : sama seperti pada object, tinggal tambahkan **sama dengan dan nilai default**

```
const buku = ["Game of Thrones"];
let [judulBuku, pengarangBuku = "Tidak ada"] = buku;
console.log(judulBuku);
console.log(pengarangBuku);
// output: Game of Thrones
// output: Harry Potter
```


15. MAP

Map sama seperti object, namun **map dapat menuliskan key lebih bebas dari pada object** yang hanya string dan symbol saja. Berikut adalah contoh deklarasi tipe data map.

```
const myMap = new Map();
```

Atau jika ingin sekaligus dengan valuenya :

```
const myMap = new Map([
  ['1', 'a String key'],
  [1, 'a number key'],
  [true, true]
]);
```

Contoh Function yang ada pada object map.

Function	Fungsi	Contoh
myMap.size	Menghitung total data pada object map	<pre>const myMap = new Map([['1', 'a String key'], [1, 'a number key'], [true, true]]); console.log(myMap.size);</pre>
myMap.get("key")	Mencari data dengan key tertentu	<pre>console.log(myMap.get(1));</pre>
myMap.set("key", "value")	Menambahkan data pada object map	<pre>console.log(myMap.set('2', 'another string'));</pre>
myMap.has("key")	Melihat apakah key tersebut ada pada salah satu data object map	<pre>console.log(myMap.has('2'));</pre>
myMap.delete("key")	Menghapus data sesuai key yang sama pada object map	<pre>console.log(myMap.delete('2'));</pre>

16. SET

Set merupakan sebuah **tipe data yang menampung banyak nilai tanpa adanya indeks dan semua valuenya bersifat unik alias tidak ada duplikasi.**

```
const numberSet = new Set([1, 4, 6, 4, 1]);
console.log(numberSet);
// output: {1, 4, 6}
```

Function yang ada pada Set

Function	Fungsi	Contoh
mySet.add("value")	Menambahkan value pada object set	<pre>numberSet.add(5);</pre>

<code>mySet.delete("value")</code>	Menghapus value pada object set	<code>numberSet.delete(4);</code>
------------------------------------	---------------------------------	-----------------------------------

17. DECLARE FUNCTION

Function merupakan sebuah fungsi yang **sudah ada pada javascript** atau dapat dibuat dan dapat digunakan untuk berbagai macam kondisi. Selain itu, function dapat mengembalikan sebuah nilai.

Penulisan sebuah function ialah :

```
function namaFunction(addSomethingOptional) {
    // do something
}
```

Penggunaan Function:

```
namaFunction("optionalData");
```

Contoh Penggunaan **Return** :

```
function welcome(namaAnda, negaraAnda) {
    if (negaraAnda == "Indonesia") {
        return `Selamat Datang, ${namaAnda}`;
    } else {
        return `Welcome, ${namaAnda}`;
    }
}

let message = welcome("Dimas", "Indonesia");
```

NOTE * : KETIKA STATEMENT RETURN TEREKSEKUSI, MAKA FUNGSI AKAN LANGSUNG TERHENTI DAN MENGEMBALIKAN NILAI.

Bentuk format penulisan function lainnya adalah **expression function**. Kita memasukkan sebuah **function** ke dalam sebuah **variable** yang telah dideklarasikan (bisa let bisa const).

```
let salamKenal = function (myNama, myNegara) {
    if (myNegara == "Indonesia") {
        return `Selamat Datang, ${myNama}`;
    } else {
        return `Welcome, ${myNama}`;
    }
}

let myMessage = salamKenal("Dimas", "Indonesia");
console.log(myMessage);
```

18. FUNCTION PARAMETER

Pada pembuatan function, terdapat parameter. Yaitu nilai yang di input dalam tanda kurung, dan dapat bertipe apa saja termasuk object.

```
let myObject = {
  myUserId: 1,
  myUserName: "Dimas Rendy",
}

function myFunction({ myUserId, myUserName }) {
  console.log(myUserId, myUserName);
}

myFunction(myObject);
// output : 1 'Dimas Rendy'
```

- a. **Default parameters** : Sebagai antisipasi jika salah satu parameter bersifat 0 ATAU tidak diisi, maka default param dapat mengatasi hal ini dengan menambahkan sama dengan dan nilai default values.

```
function exponentsFormula(baseNumber, exponent = 2) {
  const result = baseNumber ** exponent;
  console.log(`${baseNumber}^{exponent} = ${result}`);
}

exponentsFormula(3); // output : 3^2 = 9
```

- b. **Rest parameters** : Berkebalikan dari spread operator pada array yang menjabarkan nilai array, selain menggunakan tiga titik (...) rest param mengubah banyak argument menjadi 1 array.

```
function sum(...numbers) {
  let result = 0;
  for (let number of numbers) {
    result += number;
  }
  return result;
}

console.log(sum(1, 2, 3, 4, 5));
// output : 15
```

19. ARROW FUNCTION EXPRESSION

Arrow function adalah salah satu bentuk function yang **bersifat EXPRESSION** saja dan di tandai dengan menggunakan **fat arrow** (=>) dan kita **tidak perlu menuliskan keyword “function”** setiap membuat fungsi.

Kita **tetap menuliskan parameter** di dalam tanda kurung lalu **diikuti dengan tanda panah (=>)** sebelum kurung kurawal.

```
// function expression
const sayHello = (greet) => {
  console.log(`${greet}!`)
}
```

Atau Jika **CUMAN 1 PARAMETER**

```
const sayHello = greet => {
  console.log(`${greet}!`)
}
```

Atau, jika **TIDAK ADA PARAMETER SAMA SEKALI**

```
const sayHello = () => {
  console.log(`WELCOME!`)
}
```

Atau, Jika kita ingin membuat **dalam satu baris**, kita tidak perlu membuat kurung kurawal dan **TIDAK PERLU MEMBUAT “RETURN”** khusus 1 line saja otomatis dia akan mengembalikan nilainya.

```
const sayHello = () => console.log(`WELCOME!`);
```

1 baris versi tanpa return :

```
const howMuch = (a, b) => a * b;
console.log(howMuch(2, 3));
```

20. VARIABLE SCOPE

Variable scoping adalah keadaan di mana kita membutuhkan **variabel untuk diakses di seluruh script** yang kita buat. Tetapi ada juga keadaan di mana kita ingin **variabel tersebut hanya dapat diakses pada cakupan fungsi dan fungsi turunannya saja**.

Contoh :

```
// global variable, dapat diakses pada parent() dan child()
const asd = 'a';

function parent() {
  // local variable, dapat diakses pada parent() dan child(), tetapi tidak dapat diakses di luar dari fungsi
  const b = 'b';

  function child() {
    // local variable, dapat diakses hanya pada fungsi child().
    const c = 'c';
  }
}
```

a. **Global scope** : Dapat diakses semua parent dan child walau berbeda-beda function, **const asd**

b. **Local scope** : Hanya dapat diakses pada function yang sama, **const b** dan **const c**

NOTE * : Perlu kita perhatikan, jika kita **lupa menuliskan keyword let, const, atau var** pada script ketika membuat sebuah variabel, **maka variabel tersebut akan menjadi global**. Usahakan mengurangi dampak tabrakan antar variable maupun function.

21. CLOSURE

Closure ialah sebuah kiasan untuk **FUNCTION yang dapat menggunakan nilai variable sebuah local scope** atau sebutan lainnya **lexical scope**.

JavaScript **tidak memiliki cara untuk mendeklarasikan suatu fungsi atau variabel menjadi private** seperti bahasa Java. Sehingga sebuah fungsi atau variabel bisa diakses dari mana pun. Kenapa kita membutuhkan private method? Salah satunya adalah **untuk membatasi akses ke fungsi atau variable dan mencegah bug akibat global variable scope**.

Contoh function expression 'add', terdapat variable counter yang bersifat local scope dan tidak dapat diakses diluar function 'add'.

```
const add = () => {
  let counter = 0;
  return () => {
    return ++counter;
  };
}

const addCounter = add();
```

22. OOP, OBJECT ORIENTED PROGRAMMING

OOP adalah paradigma (konsep dasar) pemrograman yang banyak diterapkan ketika membangun aplikasi.

- a. **Property** : Nilai di dalam object yang **menyimpan informasi** object tersebut. Perumpamaan dalam kehidupan sehari-hari adalah sebuah mobil memiliki property yaitu warna mobil, merek mobil, nomor plat, dan sebagainya
- b. **Method** : fungsi yang **menggambarkan aksi** yang dapat dilakukan oleh object tersebut. Perumpamaan dalam kehidupan sehari-hari adalah sebuah mobil memiliki method gas maju, mundur, belok, dan berhenti.

Contoh oop

```
const car = {  
  // properties  
  brand: 'Ford',  
  color: 'red',  
  maxSpeed: 200,  
  chassisNumber: 'f-1',  
  // methods  
  drive: () => {  
    console.log('driving');  
  },  
  reverse: () => {  
    console.log('reversing');  
  },  
  turn: () => {  
    console.log('turning');  
  }  
}  
  
console.log(car.brand); // Ford  
console.log(car.color); // red  
console.log(car.maxSpeed); // 200  
console.log(car.chassisNumber); // f-1  
car.drive(); // driving  
car.reverse(); // reversing  
car.turn(); // turning
```

- a. **Object blueprint** : Ketika memiliki banyak object, kita hanya cukup membuat objek yang sudah terdefiniskan macam-macam properti dan method-nya. Sehingga kita **cukup menggunakan cetakan tersebut untuk membuat objek yang serupa, tetapi kita bisa menentukan nilai-nilai properti yang berbeda**. Akan kita bahas di constructor function.

NOTE : JAVASCRIPT BUKANLAH CLASS-BASED LANGUAGE, MELAINKAN PROTOTYPE-BASED LANGUAGE. MENARIKNYA, KONSEP-KONSEP OOP YANG MEMANFAATKAN CLASS SEPerti

PEWARISAN DAPAT DILAKUKAN DENGAN MEMANFAATKAN PROTOTYPE. ALIH-ALIH MENGGUNAKAN CLASS, PROTOTYPE-LAH YANG DIADAPTASI JAVASCRIPT.

23. CONSTRUCTOR FUNCTION

Constructor function digunakan saat inisialisasi atau mempersiapkan data untuk objek yang dapat dipakai berulang-ulang.

```
function Car(brand, color, maxSpeed, chassisNumber) {
  this.brand = brand;
  this.color = color;
  this.maxSpeed = maxSpeed;
  this.chassisNumber = chassisNumber;
}
// method
Car.prototype.drive = function () {
  console.log(`${this.brand} ${this.color} is driving`);
}

Car.prototype.reverse = function () {
  console.log(`${this.brand} ${this.color} is reversing`);
}

Car.prototype.turn = function () {
  console.log(`${this.brand} ${this.color} is turning`);
}

// Membuat objek mobil dengan constructor function Car
const car1 = new Car('Toyota', 'Silver', 200, 'to-1');
const car2 = new Car('Honda', 'Black', 180, 'ho-1');
const car3 = new Car('Suzuki', 'Red', 220, 'su-1');

console.log(car1);
console.log(car2);
console.log(car3);

car1.drive();
car2.drive();
car3.drive();
```

Penulisan :

- Nama function diawal harus KAPITAL, contoh “function Car (brand,){ this.brand = brand, ... }”
- Memanfaatkan keyword ‘this’ yang bernilai objek (instance) dirinya sendiri. Pada function constructor **this.brand** dan digunakan sebagai value yaitu **\${this.brand}**
- Constructor function memiliki internal property bernama prototype. Properti ini digunakan untuk mendefinisikan method-method yang akan dimiliki oleh objek yang dibuat. Alasan method perlu

didefinisikan di dalam prototype agar mudah untuk diwarisi ketika melakukan pewarisan.

Contohnya method drive, **Car.prototype.drive = function () { //dosomething }**

- Anda harus memanggilnya dengan menambahkan keyword 'new'. Contohnya, **const car1 = new Car("Toyota",);**

NOTE : Perlu Anda ingat bahwa **constructor function** hanya dapat dibuat dengan **regular function**.

Anda **tidak dapat membuat constructor function dengan arrow function**. Arrow function tidak dapat dipanggil dengan keyword new.

- a. **Sintaks Class di ES6** : Menggunakan keyword "class" agar lebih mirip oop di class seperti java. Selain itu, tampilan class seperti ini lebih mudah dipahami.

```
class Car {
  constructor(brand, color, maxSpeed, chassisNumber) {
    this.brand = brand;
    this.color = color;
    this.maxSpeed = maxSpeed;
    this.chassisNumber = chassisNumber;
  }

  drive() {
    console.log(`${this.brand} ${this.color} is driving`);
  }

  reverse() {
    console.log(`${this.brand} ${this.color} is reversing`);
  }

  turn() {
    console.log(`${this.brand} ${this.color} is turning`);
  }
}

// Membuat objek mobil dengan constructor function Car
const car1 = new Car('Toyota', 'Silver', 200, 'to-1');
const car2 = new Car('Honda', 'Black', 180, 'ho-1');
const car3 = new Car('Suzuki', 'Red', 220, 'su-1');

console.log(car1);
console.log(car2);
console.log(car3);

car1.drive();
car2.drive();
car3.drive();
```


24. PROPERTY AND METHOD

Pada sebuah class terdapat 2 hal yang bisa kita definisikan, yaitu :

- a. **Property** : Berisi nilai-nilai, informasi, selain dapat di definisikan sebuah property juga dapat di input manual oleh sistem atau input default.

```
class Car {  
  constructor(brand, color, maxSpeed) {  
    this.brand = brand;  
    this.color = color;  
    this.maxSpeed = maxSpeed;  
    this.chassisNumber = `${brand}-${Math.floor(Math.random() * 1000) + 1}`;  
  }  
}
```

Tambahan : chassisNumber tetap dapat diubah misal "car1.chassisNumber = 'BMW-1'". Hal ini dikarenakan data property langsung menampung nilai.

- b. **Properti Getter dan Setter** : Pada Accessor property, nilai properti dikontrol oleh sebuah getter dan setter. Nilai yang didapatkan dari properti tersebut dikontrol oleh method `get` dan cara menetapkan nilai tersebut dikontrol oleh method `set`.

```
class User {  
  constructor(firstName, lastName) {  
    this.firstName = firstName;  
    this._lastName = lastName;  
  }  
  
  get lastName() {  
    return this._lastName;  
  }  
  
  set lastName(lastName) { // harus ada parameter  
    console.log(`Data tidak bisa diubah`);  
  }  
}  
  
const user1 = new User('John', 'Doe');  
console.log(user1.lastName); // Doe  
user1.lastName = 'Dimas'; // Data tidak bisa diubah  
console.log(user1.lastName); // Doe
```

Keterangan : Pada penamaan constructor kita menambahkan underscore untuk menandakan tipe accessor property, dan membuat getter dan setter tanpa underscore (yang akan digunakan di luar class). Walau masih bisa merubah data `_lastName`, tapi itu sangat tidak direkomendasikan.

- c. **Method** : Berisi Tindakan atau suatu aksi selain itu, method juga dapat menciptakan return nilai yang bahkan bisa dipakai oleh property class itu sendiri.

```
class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this._defaultMoney = this._generateMoney();
  }

  get defaultMoney() {
    return this._defaultMoney;
  }

  set defaultMoney(value) {
    console.log(`Sorry you can't change default money`);
  }

  _generateMoney() {
    return Math.floor(Math.random() * 1000000) + 1;
  }
}

const user1 = new User('Dimas', 'Rendy');
console.log(user1); // object User
user1.defaultMoney = 1000000; // ubah tapi tidak bisa
console.log(user1.defaultMoney); // object user.defaultMoney
```

- d. **Member Visibility** : hak akses pada sebuah properti dan method di dalam class. Secara default, seluruh properti dan method yang dibuat di dalam class bersifat public, alias dapat diakses di luar dari kode class via instance. Selain public, kita juga bisa membuat properti dan method bersifat private, terutama ketika kita ingin properti atau method tersebut hanya digunakan dalam cakupan kode di dalam class saja (penggunaan internal).

Walau sudah menambahkan ***“underscore” yang menandakan property atau method private***, tapi **sebenarnya data tetap dapat diakses secara public** (alias langsung menggunakan property underscore / via instance).

NOTE : Untuk menyelesaikan masalah ini, JavaScript versi ES2022 secara resmi mengenalkan cara dalam **menetapkan hak akses private pada properti dan method objek**, yakni dengan **menambahkan tanda # di awal penamaan properti atau method**. Kalau sebelumnya `_nameProperty`, sekarang `#nameProperty`

```

class User {
  #defaultMoney = null; // enclosing class

  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.#defaultMoney = this.#generateMoney();
  }

  get defaultMoney() {
    return this.#defaultMoney;
  }

  set defaultMoney(value) {
    console.log(`Sorry you can't change default money`);
  }

  #generateMoney() {
    return Math.floor(Math.random() * 1000000) + 1;
  }
}

const user1 = new User('John', 'Doe');
console.log(user1.#generateMoney); // ts(18013)
console.log(user1.#defaultMoney); // object user.defaultMoney

```

Property '#generateMoney' is not accessible outside class 'User' because it has a private identifier. ts(18013)

View Problem (Ctrl+K N) Quick Fix... (Ctrl+.)

Keterangan:

- Kalau property perlu menambahkan enclosing class, contoh **#defaultMoney = null;**
- Jika class private tersebut diakses, maka akan memunculkan **'syntax error'**

25. PEWARISAN / INHERITANCE

Inheritance atau pewarisan sifat adalah **salah satu bentuk penggunaan antara superclass (parent) dan subclass (child)** agar dapat **mewariskan property atau method yang sama pada subclass**. Jika kita ingin menggunakan property atau method superclass pada subclass, kita perlu menambahkan “**extends namaSuperClass**” pada saat deklarasi subclass. Contohnya :

```
// Superclass
class MailService {
  constructor(sender) {
    this.sender = sender;
  }

  sendMessage(message, receiver) {
    console.log(`${this.sender} sent ${message} to ${receiver}`);
  }
}

// Subclass
class WhatsAppService extends MailService {
  sendBroadcastMessage(message, receivers) {
    for (const receiver of receivers) {
      this.sendMessage(message, receiver);
    }
  }
}

// Subclass
class EmailService extends MailService {
  sendDelayedMessage(message, receiver, delay) {
    setTimeout(() => {
      this.sendMessage(message, receiver);
    }, delay);
  }
}
```

Keterangan :

- Pada subclass kita perlu menambahkan **extends MailService**, lalu kita bisa menggunakan method **this.sendMessage** yang didapat dari superclass.
- Pada masing-masing subclass, kita juga dapat menggunakan method dari superclass di dalam method subclass itu sendiri.
- Property **this.sender** adalah property yang bersifat superclass, maka property ini juga dapat digunakan untuk subclass, lebih mudah silahkan lihat code pemanggilan berikut.

Pemanggilan :

```
const whatsapp = new WhatsAppService('+6281234567890');
const email = new EmailService('dimas@dicoding.com');

whatsapp.sendMessage('Hello', '+6289876543210');
whatsapp.sendBroadcastMessage('Hello', ['+6289876543210', '+6282234567890']);
// whatsapp.sendDelayedMessage(); // Error karena milik EmailService

email.sendMessage('Hello', 'john@doe.com');
email.sendDelayedMessage('Hello', 'john@doe.com', 3000);
// email.sendBroadcastMessage(); // Error karena milik WhatsAppService
```

Sebelum adanya class pada ES6, bentuk inheritance digunakan tetap sama yaitu menggunakan prototype.

Contohnya pada superclass:

```
function MailService(sender) {
  this.sender = sender;
}

MailService.prototype.sendMessage = function (message, receiver) {
  console.log(`${this.sender} sent ${message} to ${receiver}`);
}
```

Tambahan **Function instanceof**: digunakan untuk melihat hasil boolean dari sebuah variable dengan bentuk instance (misal **myArray instanceof Array**, atau pada class bisa seperti **user1 instanceof User** , bahkan bisa mengecek superclassnya juga misal di **whatsapp1 instanceof MailService**.

26. OVERRIDING / MENIMPA

Overriding juga termasuk kedalam bentuk pewarisan di mana subclass dapat **mendefinisikan implementasinya sendiri pada sebuah method yang pernah di definisikan superclassnya**. Hal ini tentu sangat berguna jika mau membuat sebuah method yang lebih spesifik di subclass ataupun menambahkan property yang baru. **Overriding dapat dilakukan pada constructor dan method class**.

- a. **Constructor Overriding** : Overriding constructor dilakukan sesimpel kita **mendefinisikan kembali method constructor()** pada sebuah subclass. Tapi perlu ditambahkan **"super(propertySuperClass);"** pada constructor yang di override

```
class MailService {
  constructor(sender) {
    this.sender = sender;
  }
}

class WhatsAppService extends MailService {
  // overriding constructor
  constructor(sender, isBusiness) {
    super(sender); // super() harus dipanggil sebelum this
    this.isBusiness = isBusiness;
  }
}
```

- b. **Method Overriding** : Method overriding biasanya dilakukan **ketika kita ingin mengubah implementasi method warisan superclass**. Tidak ada syarat khusus seperti menambahkan keywords super, kita hanya perlu mendefinisikan ulang method tersebut di dalam subclass.

```
class MailService {
  constructor(sender) {
    this.sender = sender;
  }

  sendMessage(message, receiver) {
    console.log(`${this.sender} sent ${message} to ${receiver}`);
  }
}

class WhatsAppService extends MailService {
  constructor(sender, isBusiness) {
    super(sender);
    this.isBusiness = isBusiness;
  }

  // Overriding method
  sendMessage(message, receiver) {
    // memanggil method sendMessage pada superclass
    super.sendMessage(message, receiver);

    console.log('message sent via WhatsApp');
  }
}
```

27. OBJECT COMPOSITION

Object composition adalah SOLUSI untuk pewarisan method yang digunakan berulang-ulang tapi tidak semua subclass memerlukannya. Jadi misalkan dirumah ada AC, tapi tidak seluruh ruangan memerlukan AC, misalnya ruang makan dan kamar mandi tidak membutuhkan AC. Tapi ruang kerja dan kamar tidur AC. Kalau kita buat dalam bentuk superclass, tentu itu menjadikan method dapat diakses oleh semua subclass dengan inheritance, solusinya adalah dengan **Object Composition/Memecahkan object dengan menstrukturkan kode berdasarkan KEMAMPUAN YANG DILAKUKAN** kalau inheritance berdasarkan identity class.

Keterangan Code Contoh :

- Kita tidak membuat sebuah inheritance atau override method, tapi membuat object composition. Jadi kita **tidak perlu menambahkan keyword extends, atau super**
- Walau demikian, kita **tetap bisa menggunakan property constructor**, contohnya this.name.
- Perlu diingat, **developer.name** merupakan **hasil dari pemanggilan constructor new Developer**.

```
class Developer {
  constructor(name) {
    this.name = name;
  }

  commitChanges() {
    console.log(`${this.name} is committing changes...`);
  }
}

function canBuildUI(developer) {
  return {
    buildUI: () => {
      console.log(`${developer.name} is building UI...`);
    }
  }
}

function canBuildAPI(developer) {
  return {
    buildAPI: () => {
      console.log(`${developer.name} is building API...`);
    }
  }
}

function canDeployApp(developer) {
  return {
    deployApp: () => {
      console.log(`${developer.name} is deploying app...`);
    }
  }
}
```

```
function createFrontEndDeveloper(name) {
  const developer = new Developer(name);
  return Object.assign(developer, canBuildUI(developer));
}

function createBackEndDeveloper(name) {
  const developer = new Developer(name);
  return Object.assign(developer, canBuildAPI(developer));
}

function createDevOps(name) {
  const developer = new Developer(name);
  return Object.assign(developer, canDeployApp(developer));
}

function createFullStackDeveloper(name) {
  const developer = new Developer(name);
  return Object.assign(developer, canBuildUI(developer), canBuildAPI(developer), canDeployApp(developer));
}
```

```

const frontEndDeveloper = createFrontEndDeveloper('Fulan');
frontEndDeveloper.commitChanges();
frontEndDeveloper.buildUI();
console.log(`is ${frontEndDeveloper.name} developer? `, frontEndDeveloper instanceof Developer);

const backEndDeveloper = createBackEndDeveloper('Fulana');
backEndDeveloper.commitChanges();
backEndDeveloper.buildAPI();
console.log(`is ${backEndDeveloper.name} developer? `, backEndDeveloper instanceof Developer);

const devOpsDeveloper = createDevOps('Fulani');
devOpsDeveloper.commitChanges();
devOpsDeveloper.deployApp();
console.log(`is ${devOpsDeveloper.name} developer? `, devOpsDeveloper instanceof Developer);

const fullStackDeveloper = createFullStackDeveloper('Fulanah');
fullStackDeveloper.buildUI();
fullStackDeveloper.buildAPI();
fullStackDeveloper.deployApp();
console.log(`is ${fullStackDeveloper.name} developer? `, fullStackDeveloper instanceof Developer);
console.log(fullStackDeveloper);

```

Tambahan :

- **Return Object.assign(developer, CanBuildUI(developer), canBuildAPI(developer));** adalah salah satu fungsi **Object.assign()**, mengomposisikan kemampuan-kemampuan object yang dibutuhkan.
- Hasil dari Object.assign fullstackdeveloper

```

Developer {
  name: 'Fulanah',
  buildUI: [Function: buildUI],
  buildAPI: [Function: buildAPI],
  deployApp: [Function: deployApp]
}

```


28. BUILT-IN CLASS

Built-in class atau class bawaan adalah class yang sudah ada pada javascript seperti Data, Object, Array, Math, Number, dan String.

- a. **Date class** : Digunakan untuk membuat constructor yang berhubungan dengan waktu dan tanggal.

```
const date = new Date();

const timeInJakarta = date.toLocaleString('id-ID', {
  timeZone: 'Asia/Jakarta',
});

const timeInTokyo = date.toLocaleString('ja-JP', {
  timeZone: 'Asia/Tokyo',
});

const timeInMakassar = date.toLocaleString('id-ID', {
  timeZone: 'Asia/Makassar',
});

console.log(timeInJakarta); // output : 12/10/2021, 10:00:00
console.log(timeInTokyo); // output : 12/10/2021, 11:00:00
console.log(timeInMakassar); // output : 12/10/2021, 09:00:00
```

- b. **Array class** : Digunakan untuk membuat constructor array yang dapat digunakan untuk pewarisan

```
class UniqueArray extends Array {
  constructor(...args) {
    // make sure args is unique before passing it to super
    const uniqueValue = args.filter((item, index) => args.indexOf(item) === index);
    super(...uniqueValue);
  }

  push(item) {
    // make sure only unique item is added
    if (!this.includes(item)) {
      super.push(item);
    }
  }
}

const someArray = new UniqueArray('a', 'b', 'c', 'a', 'b', 'c');
console.log(someArray); // ['a', 'b', 'c']
someArray.push('d');
console.log(someArray); // ['a', 'b', 'c', 'd']
someArray.push('a');
console.log(someArray); // ['a', 'b', 'c', 'd']
```

Keterangan :

- function `.filter` digunakan untuk **melakukan perbandingan per item** di sebuah array dengan **kondisi yg disesuaikan**
- kondisi filter secara otomatis membuat sebuah perulangan untuk per item, jadi kita tidak perlu melakukan perulangan for dsb

29. PARADIGMA FUNCTIONAL PROGRAMMING / FP

Paradigma Functional Programming adalah **suatu bentuk lebih spesifik dari onject composition**. Paradigma pemrograman di mana **proses komputasi didasarkan pada fungsi matematika murni**. Functional Programming (selanjutnya akan kita singkat menjadi FP) ditulis dengan **gaya deklaratif yang berfokus pada “what to solve”** dibanding “how to solve” yang dianut oleh gaya imperatif.

- a. **Imperatif styles** : Menggunakan prinsip *how to solve* alias bagaimana dapat mengisi nilai array yang baru dengan memikirkan bagaimana cara melakukan perulangan *for*, kapan harus berhenti `array.length`, dan memasukkan nilai dengan function `push`. *Terkesan sangat memerlukan sedikit fokus*

```
// imperative programming
const names = ['Harry', 'Ron', 'Jeff', 'Thomas'];

const newNamesWithExcMark = [];

for (let i = 0; i < names.length; i++) {
  newNamesWithExcMark.push(`${names[i]}!`);
}
```

- b. **Deklaratif styles** : Menggunakan prinsip *what to solve*, Jadi intinya apapun caranya selama lebih mudah, ringkas, dan efisien untuk mencapai tujuan. Kita tidak perlu pusing untuk memikirkan cara manual untuk mencapai sebuah tujuan. Tidak ada proses looping manual; Tidak perlu tahu kapan harus berhenti dari looping; Kita cukup fokus pada TUJUAN SAJA.

```
const names = ['Harry', 'Ron', 'Jeff', 'Thomas'];

const newNamesWithExcMark = names.map((name) => `${name}!`);

console.log(newNamesWithExcMark);

/* output:
[ 'Harry!', 'Ron!', 'Jeff!', 'Thomas!' ]
*/
```

30. KONSEP-KONSEP FP

a. Pure Function

konsep dari pembuatan fungsi yang mengharuskan fungsi untuk **tidak bergantung terhadap nilai yang berada di luar fungsi atau parameternya**. Tanpa harus **mendeklarasi variable** dengan nilai yang seharusnya **tetap**.

```
// Pure Function, pi langsung 3.14
const hitungLuasLingkaran = (jariJari) => {
  return 3.14 * (jariJari * jariJari);
}
```

Sedangkan, dan lebih banyak baris.

```
// Impure Function, pi dapat berubah melalui variable PI
let PI = 3.14;

const hitungLuasLingkaran = (jariJari) => {
  return PI * (jariJari * jariJari);
}
```

Selain dilarang untuk bergantung terhadap nilai luar, pure function juga **dilarang keras untuk mengubah nilai yang berada di luar baik secara sengaja atau tidak sengaja**. Pure function tidak boleh menimbulkan efek samping (*no side effect*) ketika digunakan.

Bentuk unpure function, karena nilai person berubah.

```
const createPersonWithAge = (age, person) => {
  person.age = age;
  return person;
};

const person = {
  name: 'Bobo'
};

const newPerson = createPersonWithAge(18, person);

console.log({
  person,
  newPerson
});

/**
 * Output:
 * {
 *   person: { name: 'Bobo', age: 18 },
 *   newPerson: { name: 'Bobo', age: 18 }
 * }
 */
```

Bentuk Pure Function, nilai person tidak berubah

```
const createPersonWithAge = (age, person) => {  
  // spread operator object, ketika dibuat akan menggabungkan  
  // semua properti termasuk age yang ada di dalam object person  
  return { ...person, age };  
};  
  
const person = {  
  name: 'Bobo'  
};  
  
const newPerson = createPersonWithAge(18, person);  
  
console.log({  
  person,  
  newPerson,  
});  
  
/**  
 * Output:  
 * {  
 *   person: { name: 'Bobo' },  
 *   newPerson: { name: 'Bobo', age: 18 }  
 * }  
 */
```

3 konsep Pure Function :

- Mengembalikan nilai yang sama bila inputannya (nilai parameter) sama.
- Hanya bergantung pada argumen yang diberikan.
- Tidak menimbulkan efek samping.

b. Immutability

Immutable berarti **sebuah objek tidak boleh diubah setelah objek tersebut dibuat**. Kontras dengan mutable yang artinya objek boleh diubah setelah objek tersebut dibuat.

```
const daftarNama = ['Harry', 'Ron', 'Jeff', 'Thomas'];

const newNamaTandaSeru = daftarNama.map((name) => `${name}!`);

console.log({
  daftarNama,
  newNamaTandaSeru,
});

/**
 * {
 *   daftarNama: [ 'Harry', 'Ron', 'Jeff', 'Thomas' ],
 *   newNamaTandaSeru: [ 'Harry!', 'Ron!', 'Jeff!', 'Thomas!' ]
 * }
 */
```

Jika kita ingin mengubah nilai IMMUTABLE, maka kita **terapkan perubahan pada saat *return object baru yang dibuat***. Tapi tidak merubah nilai *usser*.

```
const usser = {
  firstname: 'Harry',
  lastName: 'Protter', // ups, typo!
}

// kita buat function untuk membuat object baru dengan nama yang benar
const createUserWithNewLastName = (newLastName, usser) => {
  // ubah property lastName menjadi newLastName
  return { ...usser, lastName: newLastName }
}

// kita memanfaatkan function dan object yang sudah ada
const newUser = createUserWithNewLastName('Potter', usser);

console.log({ usser, newUser });

/**
 * output:
 * {
 *   usser: { firstname: 'Harry', lastName: 'Protter' },
 *   newUser: { firstname: 'Harry', lastName: 'Potter' }
 * }
 */
```

c. Rekursif

Rekursif merupakan **teknik pada sebuah function yang memanggil dirinya sendiri**. Biasanya kita buat **untuk menggantikan sintaksis iterasi seperti for, foreach, while**. Kita sebenarnya bisa menggantikan operasi iterasi dengan rekursi. Namun tidak sebatas itu saja, dengan rekursi kita dapat membuat dan mengolah data structures seperti Tree dan Array.

```
const countdown = start => {
  console.log(start);
  if (start > 0) countdown(start - 1);
};

countdown(10);
// output : 10 - 0
```

Sedangkan bentuk do while :

```
const countdown = start => {
  do {
    console.log(start);
    start -= 1;
  }
  while (start > 0);
};

countdown(10);
// output : 10 - 1
```

d. High-Order Function

JavaScript memiliki kemampuan **First Class Functions**, karena itu **fungsi pada JavaScript dapat diperlakukan layaknya sebuah data**. Kita bisa **menyimpan function dalam variabel**, **memberikan function sebagai parameter pada fungsi lainnya**, hingga **mengembalikan function di dalam function**.

```
// menyimpan function dalam variabel
const hello = () => {
  console.log('Hello!')
};

// memberikan function sebagai parameter pada fungsi lainnya say(hello)
// memanggil function hello diatas
const say = (someFunction) => {
  someFunction();
}

// mengembalikan function di dalam function
const letsHello = () => {
  return () => {
    console.log('Hello!');
  }
}

hello();
say(hello);
letsHello()();

/**
  Hello!
  Hello!
  Hello!
 */
```

Sulit dipahami, yapp tapi ada beberapa hal yang menarik. Higher-Order Function menjadi bagian konsep pada paradigma FP karena Higher-Order Function **merupakan fungsi yang dapat menerima fungsi lainnya pada argumen; mengembalikan sebuah fungsi; atau bahkan keduanya.**

Teknik Higher-Order Function biasanya digunakan untuk:

- Mengabstraksi atau mengisolasi sebuah aksi, event, atau menangani alur asynchronous menggunakan callback, promise, dan lainnya.
- Membuat utilities yang dapat digunakan di berbagai tipe data.
- Membuat teknik [currying](#) atau function composition.

31. REUSABLE FUNCTION

Dengan menerapkan konsep-konsep yang ada di dalam paradigma FP, **fungsi yang Anda buat akan bersifat *reusable*.** Karena fungsi yang Anda buat merupakan pure function, ia tidak akan dipengaruhi ataupun mempengaruhi keadaan di/dari luar. Hal ini tentu membuat fungsi dapat digunakan berkali-kali tanpa khawatir mendapatkan hasil di luar ekspektasi Anda.

Berikut ini adalah beberapa reusable yang sudah ada di javascript

- a. **Array Map** : Fungsi bawaan yang dapat dipanggil dari sebuah data bertipe array dan menerima satu buah callback function. Callback function tersebut akan **dipanggil sebanyak jumlah panjang array dan akan memiliki akses pada index array sesuai dengan iterasinya.** fungsi **map** akan mengembalikan array baru. Nilai tiap item pada array yang dikembalikan, dihasilkan dari kembalian **callback function-nya**. Karena **callback function** dapat mengakses item array, biasanya developer menggunakannya untuk menghasilkan nilai baru.

```
const newArray = ['Harry', 'Ron', 'Jeff', 'Thomas'].map((name) => `${name}!`);  
// const newArray = ['value1', 'value2', 'value3', 'value4'].map((perValue) => { //do something });  
console.log(newArray);  
// output : [ 'Harry!', 'Ron!', 'Jeff!', 'Thomas!' ]
```

- b. **Array Filter** : Fungsi bawaan yang sangat berguna untuk **melakukan proses penyaringan (*filtering*)** terhadap nilai array yang ada. Bila Anda memiliki kasus di mana Anda **ingin menghilangkan beberapa item di array berdasarkan spesifikasi tertentu**, ini yang paling cocok. Cara kerja fungsi ini mirip seperti array.map(). Namun, **callback function dari fungsi ini harus mengembalikan boolean.** Di mana nilai boolean ini digunakan untuk **menentukan apakah item array lolos saring atau tidak.** Sama seperti fungsi map(), fungsi filter() juga akan mengembalikan array yang telah disaring dalam bentuk array baru.

```
const truthyArray = [1, '', 'Hallo', 0, null, 'Harry', 14].filter((item) => Boolean(item));  
// const truthyArray = [1, '', 'Hallo', 0, null, 'Harry', 14].filter((perItem) => { //filter true });  
console.log(truthyArray);  
// output : [ 1, 'Hallo', 'Harry', 14 ]
```

Atau, contoh jika key suatu object didalam sebuah array harus bernilai diatas 85

```
const students = [
  {
    name: 'Harry',
    score: 60,
  },
  {
    name: 'James',
    score: 88,
  },
  {
    name: 'Ron',
    score: 90,
  },
  {
    name: 'Bethy',
    score: 75,
  }
];

const eligibleForScholarshipStudents = students.filter((student) => student.score > 85);
// const eligibleForScholarshipStudents = arrayObject.filter((perObjectArray) => { //perObjectArray.keyObject true });
console.log(eligibleForScholarshipStudents);
// output : [ { name: 'James', score: 88 }, { name: 'Ron', score: 90 } ]
```

- c. **Array Reduce** : Fungsi bawaan dari data yang bertipe array yang digunakan untuk **mengeksekusi fungsi reducer pada setiap elemen array dan hanya mengembalikan output satu nilai saja**. Callback function dari fungsi ini dapat diolah untuk manipulasi data **currentValue** dan menyimpannya pada **accumulator**. Selain itu, fungsi reduce juga memiliki nilai awal yang dapat didefinisikan pada bagian **initialValue**. Kira kira seperti ini bentuk konsep reusable reduce :

```
arr.reduce(callback(accumulator, currentValue, [currentIndex], [array]), [initialValue])
// [...] adalah opsional parameter
```

Contoh Array Reduce untuk mencari total nilai siswa :

```
const banyakStudent = [
  {
    name: 'Harry',
    score: 60,
  },
  {
    name: 'James',
    score: 88,
  },
  {
    name: 'Ron',
    score: 90,
  },
  {
    name: 'Bethy',
    score: 75,
  }
];

const totalScore = banyakStudent.reduce((acc, student) => acc + student.score, 0);
// const totalScore = arrayObject.reduce((accumulator, perObjectArray) => { //acc nol, tapi terus ditambah score });
console.log(totalScore); // 60 + 88 + 90 + 75 = 313 atau output : 313
```


- d. **Array Some** : Fungsi bawaan dari array yang akan menghasilkan nilai Boolean yang dihasilkan didasarkan pada pernyataan apakah ada setidaknya satu dari deretan nilai dalam array tersebut lolos berdasarkan kriteria yang kita tuliskan dalam *callback function*. Konsep :

```
arr.some(callback(element, [index], [array]), [thisArg])
```

Contoh penggunaannya misalkan kita ingin mengetahui apakah dalam deretan angka terdapat angka genap.

```
const array = [1, 2, 3, 4, 5];
const even = array.some(element => element % 2 === 0);

console.log(even); // true
```

- e. **Array Find** : digunakan untuk mencari apakah di dalam deretan nilai terdapat nilai yang sesuai dengan kriteria yang kita definisikan pada *callback function*. Yang membedakan `array.find()` dengan `array.some()`, **find** akan menghasilkan satu nilai dari elemen yang pertama kali ditemukan berdasarkan kriteria tertentu dan akan menghasilkan nilai `undefined` bila tidak ada kriteria yang terpenuhi pada item array. Konsep :

```
arr.find(callback(element, [index], [array]), [thisArg]);
```

Contoh kita akan mencari siswa dengan nama 'James'.

```
const studentss = [
  {
    name: 'Harry',
    score: 60,
  },
  {
    name: 'James',
    score: 88,
  },
  {
    name: 'Ron',
    score: 90,
  },
  {
    name: 'Bethy',
    score: 75,
  }
];

const findJames = studentss.find(student => student.name === 'James');
console.log(findJames); // { name: 'James', score: 88 }
```

- f. **Array Sort** : Fungsi bawaan dari array yang berguna untuk melakukan pengurutan nilai dari sebuah deretan nilai. Secara *default*, fungsi `sort` akan mengubah semua nilai dalam deretan menjadi bentuk string dan mengurutkannya secara *ascending*. Konsep :

```
arr.sort([compareFunction])
```

Contoh sorting :

```
const months = ['March', 'Jan', 'Feb', 'Dec'];
months.sort();
console.log(months);
// sesuai abjad
// output: [ 'Dec', 'Feb', 'Jan', 'March' ]

const array1 = [1, 30, 4, 1000, 101, 121];
array1.sort();
console.log(array1);
// sesuai urutan angka
// output: [ 1, 1000, 101, 121, 30, 4 ]
```

Contoh pengurutan di atas **didasarkan pada pengurutan bentuk tipe data string**. Oleh karena itu, ketika kita ingin **mengurutkan sesuai dengan kriteria yang kita inginkan (berdasarkan tanggal ataupun berdasarkan nilai siswa)** maka kita perlu membuat *compare function* tersendiri.

```
const array1 = [1, 30, 4, 1000];
// const compareNumber = (a, b) => { //do something };
const compareNumber = (a, b) => {
  return a - b;
};
// misal, a = 1, b = 30, maka a - b = -29, maka a sebelum b
// misal, a = 30, b = 4, maka a - b = 26, maka b sebelum a
// dan a = 30, b = 100, maka a - b = -70, maka a sebelum b
const sorting = array1.sort(compareNumber);
console.log(sorting); // [ 1, 4, 30, 1000 ]
```

Pada compare function, fungsi akan membandingkan **2 nilai yang akan menghasilkan 3 result** yaitu negatif (-), 0, dan positif (+).

- Jika, **negative** maka **`a`** akan diletakkan **sebelum `b`**
- Jika, **positive** maka **`b`** akan diletakkan **sebelum `a`**
- Jika, **0** maka **tidak ada perubahan** posisi.

- g. **Array Every** : Fungsi bawaan dari array yang digunakan **untuk mengecek apakah semua nilai dari sebuah array sesuai dengan kriteria yang didefinisikan**. Kembalian dari `array.every()` adalah **nilai Boolean**. Konsep :

```
arr.every(callback(element, [index], [array]))
```

Contoh, kita akan mengecek apakah seorang siswa telah lulus semua uji materi:

```
const scores = [70, 85, 90];
const minimumScore = 65;

const examPassed = scores.every(score => score >= minimumScore);
// const examPassed = array.every((perArray) => { //every score is more than minimumScore });
console.log(examPassed); // true
```

- h. **Array forEach** : Berfungsi untuk **memanggil fungsi callback pada setiap iterasi index array**. Berbeda dari fungsi array lain yang sudah kita bahas, **fungsi ini tidak mengembalikan nilai apa pun**. Jadi fungsi ini secara harfiah hanya **berfungsi untuk memanggil fungsi callback-nya saja**, tak lebih dari itu. Melalui fungsi ini, Anda **dapat mengubah sintaks perulangan berdasarkan jumlah array secara imperatif menjadi deklaratif**. Konsep :

```
array.forEach(callback(element, [index], [array]), [thisArg])
```

Contoh Imperatif style :

```
const names = ['Harry', 'Ron', 'Jeff', 'Thomas'];

for (let i = 0; i < names.length; i++) {
  console.log(`Hello, ${names[i]}!`);
}

// Hello, Harry!
// Hello, Ron!
// Hello, Jeff!
// Hello, Thomas!
```

Contoh Deklaratif style :

```
const names = ['Harry', 'Ron', 'Jeff', 'Thomas'];
// names.forEach((name) => //do something );
names.forEach((name) => {
  console.log(`Hello, ${name}!`);
});

// Hello, Harry!
// Hello, Ron!
// Hello, Jeff!
// Hello, Thomas!
```

NOTE : Namun, ketahuilah bahwa ketika menggunakan `forEach`, kita tidak bisa menggunakan operator **break** atau **continue** pada proses perulangan (Anda bisa melakukannya pada perulangan `for`). Hal ini juga berlaku ketika pada fungsi `map` dan `filter`. Contoh :

```
const names = ['Harry', 'Ron', 'Jeff', 'Thomas'];

for (let i = 0; i < names.length; i++) {
  if (names[i] === 'Jeff') continue; // Bisa!

  console.log(`Hello, ${names[i]}!`);
}

names.forEach((name) => {
  if (name === 'Jeff') continue; // Tidak Bisa!
  console.log(`Hello, ${name}`);
});
```

32. JAVASCRIPT RUNTIME

Javascript Runtime adalah **Program yang memperluas mesin *JavaScript* dan menyediakan fungsionalitas tambahan, sehingga dapat berinteraksi dengan dunia luar.** Javascript runtime biasanya sudah tersedia di setiap browser, namun untuk menggunakannya diluar browser kita dapat menjalankan Javascript runtime menggunakan bantuan `node.js`. Node memberikan akses JavaScript ke seluruh sistem operasi,

memungkinkan **program JavaScript dapat membaca dan menulis file; mengirim dan menerima data melalui jaringan; serta membuat dan melayani permintaan HTTP.**

33. NODE JS

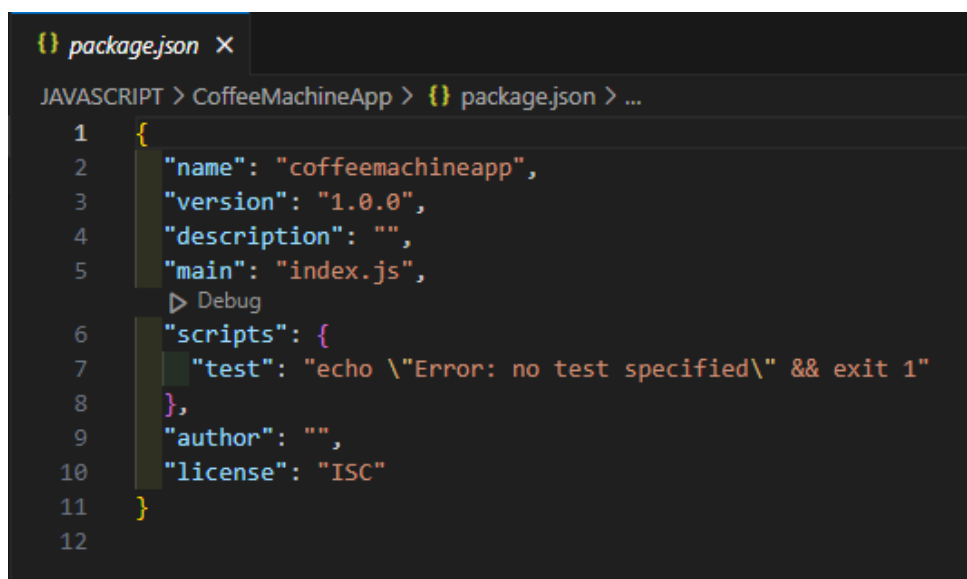
Node.js menjalankan V8 JavaScript engine (yang juga merupakan inti dari Google Chrome) di luar browser. Ini memungkinkan Node.js memiliki performa yang tinggi. Node.js juga menyediakan banyak library JavaScript yang membantu menyederhanakan pengembangan aplikasi web. Berikut ini adalah beberapa fitur penting dari Node.js yang menjadikannya pilihan utama dalam pengembangan aplikasi:

- a. **Asynchronous & Event-Driven** : Semua API dari Node.js bersifat **asynchronous, artinya tidak memblokir proses lain sembari menunggu satu proses selesai**. Server Node.js akan melanjutkan ke pemanggilan API berikutnya **lalu memanfaatkan mekanisme event notification untuk mendapatkan respon dari panggilan API sebelumnya**.
- b. **Very Fast** : Eksekusi kode dengan Node.js sangat cepat karena berjalan pada V8 JavaScript Engine dari Google Chrome.
- c. **Single Thread but Highly Scalable** : Node.js menggunakan model single thread dengan event looping. Mekanisme ini membantu server untuk merespon secara asynchronous dan menjadikan server lebih scalable dibandingkan server tradisional yang menggunakan banyak thread untuk menangani permintaan.

34. MEMBUAT PROJECT JAVASCRIPT dengan Node

Langkah-langkah pembuatan project Javascript :

- a. Ketik “ npm init “ pada terminal
- b. Masukkan data sesuai yang diinginkan, hingga muncul file baru, “ package.json ”



```
{ } package.json X
JAVASCRIPT > CoffeeMachineApp > { } package.json > ...
1  {
2    "name": "coffeemachineapp",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "author": "",
10   "license": "ISC"
11 }
12
```

- c. Buat File baru dengan nama “ index.js ” dengan beberapa code sederhana :

```
{ } package.json JS index.js X
JAVASCRIPT > CoffeeMachineApp > JS index.js
1 console.log("Menyalakan mesin kopi");
2 console.log("Menggiling biji kopi");
3 console.log("Memanaskan air");
4 console.log("Mencampurkan air dan kopi");
5 console.log("Menuangkan kopi ke dalam gelas");
6 console.log("Menuangkan susu ke dalam gelas");
7 console.log("Kopi Anda sudah siap!");
```

- d. Untuk menjalankannya, kita tinggal ketik “ node index.js ” di terminal

```
PS D:\0001x Kerja\00 Study\Belajar FrontEnd\JAVASCRIPT\CoffeeMachineApp> node index.js
Menyalakan mesin kopi
Menggiling biji kopi
Memanaskan air
Mencampurkan air dan kopi
Menuangkan kopi ke dalam gelas
Menuangkan susu ke dalam gelas
Kopi Anda sudah siap!
```

- e. **Run Scripts** : Biasa nya untuk tahap pengembangan, kita sering menggunakan script init yang berada di “ package.json ” .

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

- f. Untuk menjalankan object script, kita tinggal ketikkan “ npm run test ”, sesuai nama object scripts.

```
PS D:\0001x Kerja\00 Study\Belajar FrontEnd\JAVASCRIPT\CoffeeMachineApp> npm run test

> coffeemachineapp@1.0.0 test
> echo "Error: no test specified" && exit 1

"Error: no test specified"
```

- g. Kita tuliskan nama script sebagai properti. Kemudian tuliskan perintah yang akan dieksekusi sebagai nilai dari properti tersebut. Mari kita buat script baru untuk menjalankan kode dari berkas **index.js**.

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node index.js"
},
```

- h. Sekarang kita bisa menjalankan node index.js dengan command “ npm run start “

```
PS D:\0001x Kerja\00 Study\Belajar FrontEnd\JAVASCRIPT\CoffeeMachineApp> npm run start

> coffeemachineapp@1.0.0 start
> node index.js

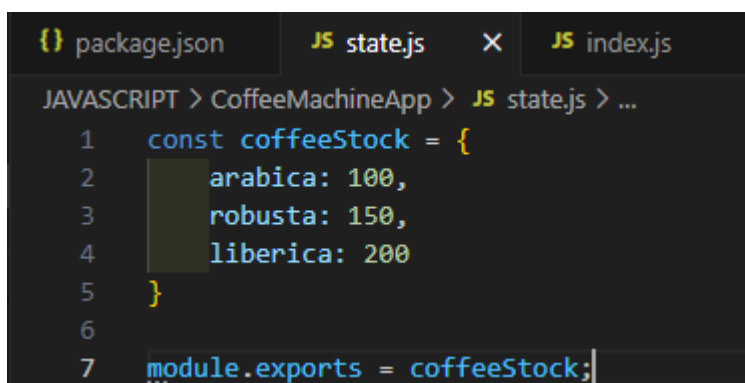
Menyalakan mesin kopi
Menggiling biji kopi
Memanaskan air
Mencampurkan air dan kopi
Menuangkan kopi ke dalam gelas
Menuangkan susu ke dalam gelas
Kopi Anda sudah siap!
```

35. MODULE EXPORT & IMPORT

Module merupakan bentuk code yang dipisah menjadi bagian tersendiri untuk memudahkan pengembangan aplikasi seperti mencari bug dan sebagainya. Sekarang kita akan membuat module yang akan saling berinteraksi.

- a. **Export & import** : Modul bekerja dengan cara exporting dan importing nilai. Baik itu **nilai variabel, fungsi, array, object, atau class** agar bisa digunakan pada berkas JavaScript lain. **Satu berkas JavaScript** terdiri dari **satu module** yang dapat **kita export menjadi lebih dari satu nilai**.

- Dalam environment Node.js, gunakan perintah **module.exports** untuk melakukan proses export module. Setiap berkas JavaScript yang berjalan pada Node, **memiliki objek module lokal yang memiliki properti exports**. Properti tersebut **digunakan untuk mendefinisikan nilai apa yang akan diekspor dari berkas tersebut**.
- Buatlah berkas baru bernama state.js dan masukkan code dibawah



```
{ } package.json JS state.js X JS index.js
JAVASCRIPT > CoffeeMachineApp > JS state.js > ...
1  const coffeeStock = {
2      arabica: 100,
3      robusta: 150,
4      liberica: 200
5  }
6
7  module.exports = coffeeStock;
```

- Kode **module.exports = coffeeStock** membuat object **coffeeStock** ditetapkan sebagai nilai dari **module.exports**. Nilai properti exports ini nantinya bisa di-import dan digunakan pada berkas JavaScript lain. Jika Anda mencoba melihat nilai module yang ada pada berkas **state.js** dengan menambahkan kode **console.log(module)** di akhir berkas, maka kita akan melihat object coffeeStock menjadi nilai dari properti exports.

- Jalankan " node state.js " dan sebelumnya tambahkan console.log(module)

```
PS D:\0001x Kerja\00 Study\Belajar FrontEnd\JAVASCRIPT\CoffeeMachineApp> node state.js
{
  id: '.',
  path: 'D:\\0001x Kerja\\00 Study\\Belajar FrontEnd\\JAVASCRIPT\\CoffeeMachineApp',
  exports: { arabica: 100, robusta: 150, liberica: 200 },
  filename: 'D:\\0001x Kerja\\00 Study\\Belajar FrontEnd\\JAVASCRIPT\\CoffeeMachineApp\\state.js',
}
```

- untuk melakukan import atau menggunakan object yang sudah di-export adalah menggunakan method `require()` pada `index.js`. Dan jalankan `npm run start`

```
{ } package.json JS state.js JS index.js X
JAVASCRIPT > CoffeeMachineApp > JS index.js > ...
1  const coffeeStock = require('./state');
2
3  console.log(coffeeStock);
```

```
PS D:\0001x Kerja\00 Study\Belajar FrontEnd\JAVASCRIPT\CoffeeMachineApp> npm run start

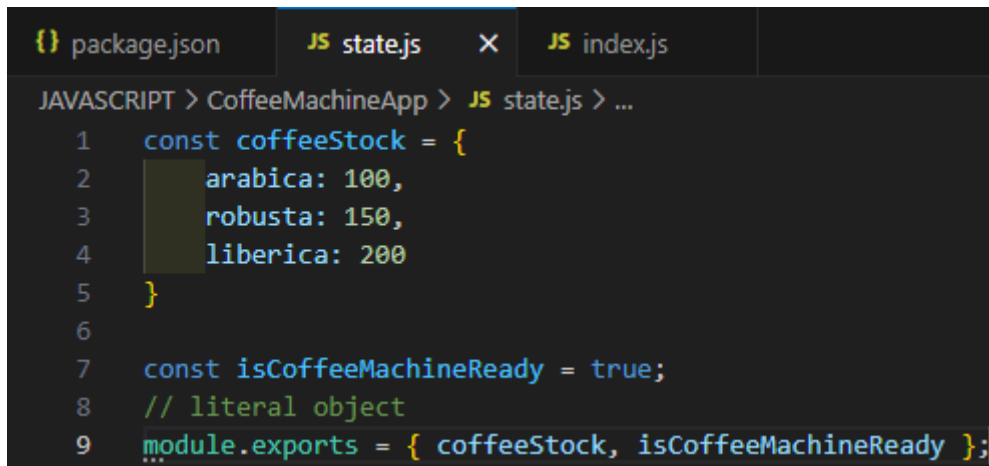
> coffeemachineapp@1.0.0 start
> node index.js

{ arabica: 100, robusta: 150, liberica: 200 }
```

- Setelah itu, kita juga dapat melakukan berbagai hal seperti membuat function (arrow)

```
{ } package.json JS state.js JS index.js X
JAVASCRIPT > CoffeeMachineApp > JS index.js > ...
1  const coffeeStock = require('./state');
2
3  console.log(coffeeStock);
4
5  const makeCoffee = (type, miligrams) => {
6    if (coffeeStock[type] >= miligrams) {
7      console.log("Kopi berhasil dibuat!");
8    } else {
9      console.log("Biji kopi habis!");
10   }
11 }
12
13 makeCoffee("robusta", 80);
14 // Output: Kopi berhasil dibuat!
```

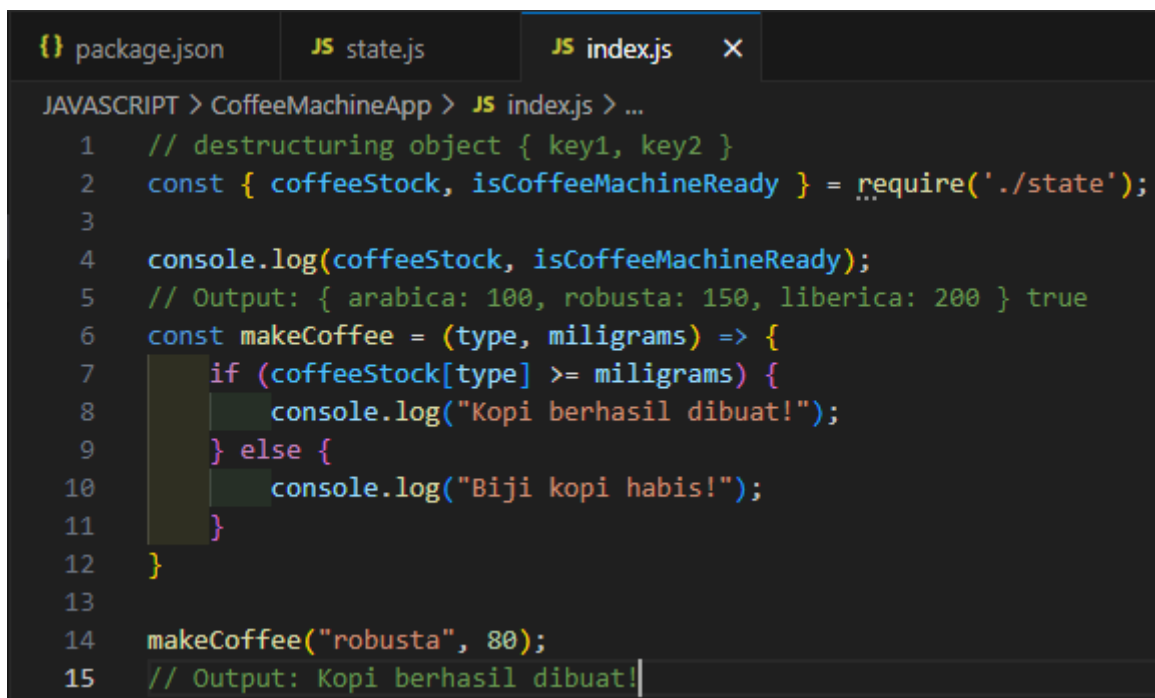

- Export Beberapa Nilai Sekaligus pada module, kita akan memanfaatkan **object literals** (`{}`).



```
package.json  JS state.js  X JS index.js

JAVASCRIPT > CoffeeMachineApp > JS state.js > ...
1  const coffeeStock = {
2    arabica: 100,
3    robusta: 150,
4    liberica: 200
5  }
6
7  const isCoffeeMachineReady = true;
8  // literal object
9  module.exports = { coffeeStock, isCoffeeMachineReady };
```

- Untuk mengakses kita dapat menggunakan destructuring object,



```
package.json  JS state.js  JS index.js  X

JAVASCRIPT > CoffeeMachineApp > JS index.js > ...
1  // destructuring object { key1, key2 }
2  const { coffeeStock, isCoffeeMachineReady } = require('./state');
3
4  console.log(coffeeStock, isCoffeeMachineReady);
5  // Output: { arabica: 100, robusta: 150, liberica: 200 } true
6  const makeCoffee = (type, miligrams) => {
7    if (coffeeStock[type] >= miligrams) {
8      console.log("Kopi berhasil dibuat!");
9    } else {
10     console.log("Biji kopi habis!");
11    }
12  }
13
14  makeCoffee("robusta", 80);
15  // Output: Kopi berhasil dibuat!
```

Note : Nama destructuring object harus sama dengan nama property, kalau tidak akan menghasilkan value undefined

36. ES6 MODULE EXPORT & IMPORT

JavaScript memiliki sistem modular secara native. Karena itu, sistem ini dapat dijalankan baik pada environment Node.js maupun browser. Pada Node.js sebelumnya **tidak ada perbedaan antara export satu atau beberapa nilai**. Semua nilai yang akan diekspor dijadikan nilai dari properti `module.exports`. Pada ES6 module, jika kita hanya mengekspor satu nilai pada sebuah berkas JavaScript baik itu primitive value, function, array, object, atau class, kita gunakan keyword **export default**. Bingung? Ni contohnya :

```
{ } package.json JS state.js X JS index.js
JAVASCRIPT > CoffeeMachineApp > JS state.js > [0] default
1  const coffeeStock = {
2      arabica: 100,
3      robusta: 150,
4      liberica: 200
5  };
6
7  export default coffeeStock;
```

Berbeda dengan gaya Node.js, kita gunakan keyword **import** ketika mendeklarasikan variabel yang di-import. Kita juga menggunakan keyword **from** untuk menentukan lokasi berkas JavaScript-nya. Ketika menggunakan **export default**, kita dapat menggunakan penamaan apa saja saat mendeklarasikan variabel untuk menyimpan nilai yang diimpor. Sebaiknya kita buat sama saja.

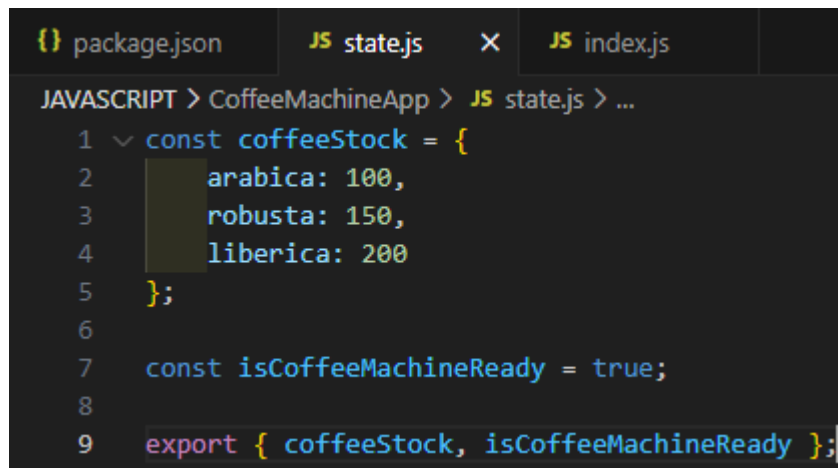
```
{ } package.json JS state.js JS index.js X
JAVASCRIPT > CoffeeMachineApp > JS index.js > ...
1  import coffeeStock from './state.js';
2
3  const displayStock = stock => {
4      for (const type in stock) {
5          console.log(type);
6      }
7  }
8
9  displayStock(coffeeStock);
```

NOTE : error akan muncul, disebabkan karena berkas JavaScript yang kita buat tidak dianggap sebagai module. Saat ini, fitur ES6 module tidak secara default diaktifkan. Pesan error di atas menyebutkan dua cara bagaimana mengaktifkan ES6 module. Dua cara tersebut adalah menambahkan properti pada **package.json** atau dengan mengubah ekstensi **.js** menjadi **.mjs**. Mari kita gunakan cara pertama yang lebih sederhana.

```
{ } package.json X JS state.js JS index.js
JAVASCRIPT > CoffeeMachineApp > { } package.json > ...
1  {
2      "name": "coffeemachineapp",
3      "version": "1.0.0",
4      "description": "",
5      "main": "index.js",
6      "type": "module", ←
7      "scripts": {
```

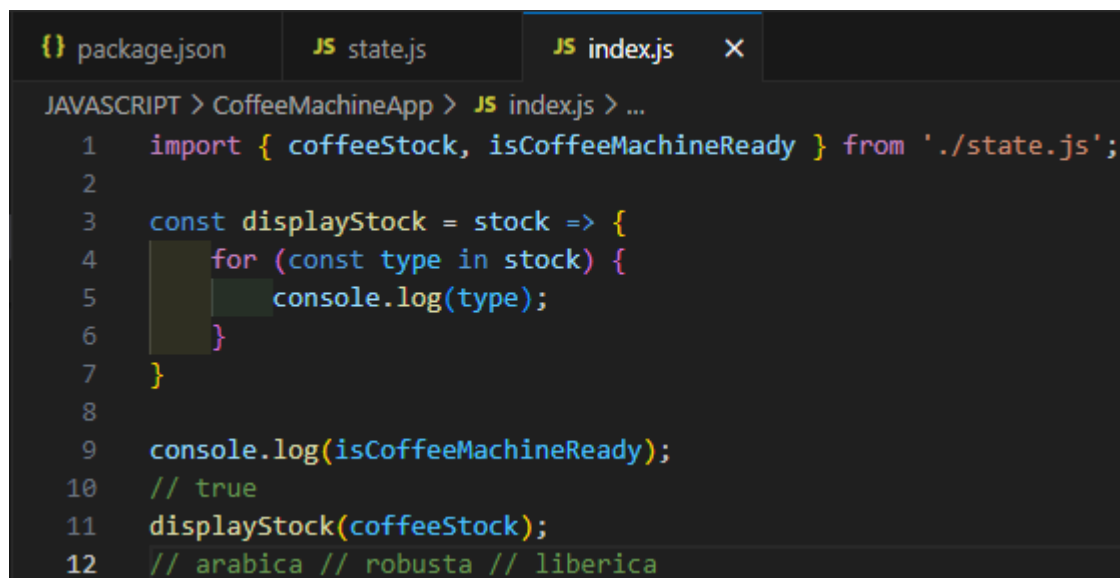
Selanjutnya, Export banyak nilai sekaligus :

- *Named export* digunakan untuk mengekspor banyak nilai dalam berkas JavaScript. Cara kerjanya mirip seperti pada Node.js. Nilai yang akan diekspor dituliskan di dalam object literals tanpa **default**,

A screenshot of a VS Code editor window with three tabs: package.json, state.js, and index.js. The state.js tab is active, showing a JavaScript file with the following code:

```
JAVASCRIPT > CoffeeMachineApp > JS state.js > ...
1  const coffeeStock = {
2    arabica: 100,
3    robusta: 150,
4    liberica: 200
5  };
6
7  const isCoffeeMachineReady = true;
8
9  export { coffeeStock, isCoffeeMachineReady };
```

- Untuk memanggilnya, juga hanya perlu menambahkan object literal

A screenshot of a VS Code editor window with three tabs: package.json, state.js, and index.js. The index.js tab is active, showing a JavaScript file with the following code:

```
JAVASCRIPT > CoffeeMachineApp > JS index.js > ...
1  import { coffeeStock, isCoffeeMachineReady } from './state.js';
2
3  const displayStock = stock => {
4    for (const type in stock) {
5      console.log(type);
6    }
7  }
8
9  console.log(isCoffeeMachineReady);
10 // true
11 displayStock(coffeeStock);
12 // arabica // robusta // liberica
```

NOTE : Karena **named import** menggunakan teknik **destructuring object** untuk mendapatkan nilai, maka **pastikan penamaan variabel** sesuai dengan nama variabel yang diekspor.

37. ERROR HANDLING, TRY & CATCH

Error Handling adalah cara untuk menangani permasalahan pada code yang bermasalah saat dijalankan.

- a. **Try and catch** : Jika terdapat code yang berkemungkinan memunculkan masalah di dalam blok try, sedangkan untuk menangkap error tersebut, kita tangani didalam blok catch. Jika tidak ada error, maka blok catch akan diabaikan. Sangat Powerful

```
try {  
  // kode  
} catch (error) {  
  // error handling  
}
```

Contoh catch diabaikan:

```
const books = [  
  { title: 'The Da Vinci Code', author: 'Dan Brown', sales: 5094805 },  
  { title: 'The Ghost', author: 'Robert Harris', sales: 807311 },  
  { title: 'White Teeth', author: 'Zadie Smith', sales: 815586 },  
  { title: 'Fifty Shades of Grey', author: 'E. L. James', sales: 3758936 },  
  { title: 'Jamie\'s Italy', author: 'Jamie Oliver', sales: 906968 },  
  { title: 'I Can Make You Thin', author: 'Paul McKenna', sales: 905086 },  
  { title: 'Harry Potter and the Deathly Hallows', author: 'J.K Rowling', sales: 4475152 },  
];  
  
const greatAuthors = books  
  .filter(item => item.sales > 1000000)  
  .map(item => `${item.author} adalah penulis buku ${item.title} yang sangat hebat!`);  
  
try {  
  console.log(greatAuthors);  
} catch (error) {  
  console.log('Error ditemukan!');  
}
```

Contoh catch aktif:

```
try {  
  console.log(greatAuthorss);  
} catch (error) {  
  console.log('Error ditemukan!');  
}  
// output : Error ditemukan!
```

Error diatas disebabkan oleh nama variable greatAuthorss tidak ditemukan, ternyata typo Authorss hanya 1 huruf s.

Untuk mengetahui jenis error, terdapat 3 property yaitu :

- **Name** : Nama error
- **Message** : Pesan detail error
- **Stack** : Informasi urutan kejadian yang menyebabkan error. Umumnya digunakan untuk debugging karena terdapat informasi baris mana yang menyebabkan error.

Contoh 3 property dan outputnya :

```
try {  
  console.log(greatAuthorss);  
} catch (error) {  
  console.log(error.name);  
  console.log(error.message);  
  console.log(error.stack);  
}  
// output : ReferenceError  
// output : greatAuthorss is not defined  
// output : ReferenceError: greatAuthorss is not defined
```

b. Try-catch-finally

Selain try dan catch, ada satu blok lagi yang ada dalam mekanisme error handling pada JavaScript, yaitu **finally**. Blok finally akan tetap dijalankan tanpa peduli apa pun hasil yang terjadi pada blok try-catch.

```
try {  
  console.log(greatAuthorss);  
} catch (error) {  
  console.log(error.name);  
  console.log(error.message);  
  console.log(error.stack);  
} finally {  
  console.log('Finally akan tetap dieksekusi');  
}  
// output : ReferenceError  
// output : greatAuthorss is not defined  
// output : ReferenceError: greatAuthorss is not defined  
// output : Finally akan tetap dieksekusi
```

38. THROWING ERROR

Throwing error adalah **implementasi try-catch** pada kasus yang lebih umum.

Contoh kasus parsing json tanpa ada error:

```
const json = '{ "name": "Yoda", "age": 20 }';

try {
  const user = JSON.parse(json);

  console.log(user.name);
  console.log(user.age);
} catch (error) {
  console.log(error.name);
  console.log(error.message);
}
```

Tambahan : fungsi `JSON.parse` akan melakukan *parsing* atau konversi dari variabel json (String) menjadi sebuah object. Skenario seperti di atas akan banyak kita temui ketika melakukan request ke API.

Contoh jika json tidak berformat variable json :

```
const json = '{ bad json }';

try {
  const user = JSON.parse(json);

  console.log(user.name);
  console.log(user.age);
} catch (error) {
  console.log(error.name);
  console.log(error.message);
}

// SyntaxError
// Expected property name or '}' in JSON at position 2 (line 1 column 3)
```

Sedangkan bagaimana jika nilai tersebut bernilai **Undefined**, Berarti **Tidak Error (secara code)** tapi **error secara logika**. Contohnya :

```
const json = '{ "age": 20 }';

try {
  const user = JSON.parse(json);

  console.log(user.name); // undefined
  console.log(user.age); // 20
} catch (error) {
  console.log(error.name); // diabaikan
  console.log(error.message); // diabaikan
}
```

Solusi, kita bisa menggunakan **throw**. Operator ini akan “melemparkan” eror pada program, sehingga eksekusi kode akan masuk pada blok catch :

```
const json = '{ "age": 20 }';

try {
  const user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("'name' is required.");
  }

  console.log(user.name); // undefined
  console.log(user.age); // 20
} catch (error) {
  console.log(`JSON Error: ${error.message}`);
}
// output : JSON Error: 'name' is required.
```

Sekarang anggaplah json sudah sesuai, **tetapi ternyata ada eror lain yang terjadi**, misalnya **karena variabel yang belum terdefinisi. Contoh :**

```
const json = '{ "name": "Yoda", "age": 20 }';

try {
  const user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("'name' is required.");
  }

  errorCode;

  console.log(user.name); // Yoda
  console.log(user.age); // 20
} catch (error) {
  console.log(`JSON Error: ${error.message}`);
}
// output : JSON Error: errorCode is not defined
```


Error berhasil ditangani, tetapi konsol tetap menampilkan pesan “JSON Error”, disini kita akan memanfaatkan if statement dengan operator `instanceOf`, kita bisa mendapatkan tipe dari error yang terjadi. Dari sana kita bisa membuat percabangan bagaimana cara menangani erornya :

```
const json = '{ "name": "Yoda", "age": 20 }';

try {
  const user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("'name' is required.");
  }

  errorCode;

  console.log(user.name); // Yoda
  console.log(user.age); // 20
} catch (error) {
  // Kalau syntax error, value undefined dsb
  if (error instanceof SyntaxError) {
    console.log(`JSON Error: ${error.message}`);
  } // kalau ReferenceError, berasal dari code error
  else if (error instanceof ReferenceError) {
    console.log(error.message);
  } // kalau error lainnya
  else {
    console.log(error.stack);
  }
}

// output : ReferenceError: errorCode is not defined
```

39. CUSTOM ERROR

Setelah menangani error, pada materi ini kita akan mempelajari bagaimana membuat error sendiri. Ketika mengembangkan suatu aplikasi, akan **ada banyak sekali kemungkinan munculnya error**. Seringkali, kita **membutuhkan kelas error sendiri untuk menunjukkan kesalahan yang spesifik dan tidak tersedia dalam kelas Error bawaan dari JavaScript**. Penggunaan yang lebih spesifik.

Untuk itu kita bisa membuat kelas Error kita sendiri dengan nama dan pesan yang lebih sesuai. Kelas ini merupakan turunan dari kelas Error yang sudah ada. Sebagai contoh, untuk mengecek validasi data dari json, kita bisa membuat kelas Error seperti ini:

```
// subclass dari Error
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}
```

Kelas ValidationError memiliki *parameter constructor* berupa `message` yang berisi pesan detail terkait erornya. Contoh Penerapan subclass error yaitu ValidationError:

```
// subclass dari Error
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

const json = '{ "age": 30 }';

try {
  const user = JSON.parse(json);

  if (!user.name) {
    throw new ValidationError("'name' is required.");
  }
  if (!user.age) {
    throw new ValidationError("'age' is required.");
  }

  console.log(user.name);
  console.log(user.age);
} catch (error) {
  if (error instanceof SyntaxError) {
    console.log(`JSON Syntax Error: ${error.message}`);
  } else if (error instanceof ValidationError) {
    console.log(`Invalid data: ${error.message}`);
  } else if (error instanceof ReferenceError) {
    console.log(error.message);
  } else {
    console.log(error.stack);
  }
}

// output : Invalid data: 'name' is required.
```

Karena sudah menggunakan statement if, dengan bantuan instanceof kita dapat menentukan jenis output pada catch agar sesuai dengan error yang ingin ditampilkan.

40. ASYNCHRONOUS PROCESS

Dalam pemrograman, tidak seluruh proses dapat berjalan dengan sangat cepat. Ada beberapa proses yang membutuhkan waktu tunggu, seperti baca tulis berkas, mendapatkan data dari internet, dan proses lainnya. **Agar tidak menghalangi proses lainnya**, kita harus mengetahui cara menjalankan proses yang lambat **secara asynchronous**. Dalam *asynchronous process*, kita bisa menjalankan proses yang berada di baris selanjutnya tanpa harus menunggu kode pada baris sebelumnya selesai dieksekusi. Arti lainnya, kita bisa melakukan lebih dari satu proses sekaligus dalam waktu yang bersamaan.

41. SET TIMEOUT FUNCTION

JavaScript merupakan bahasa pemrograman yang menerapkan pola *event-driven*, baik pada *environment* Node.js maupun browser. Seluruh proses yang berjalan dipicu oleh sebuah *event* atau kejadian, Dengan pola event-driven itu, artinya kode yang ditulis dengan JavaScript tidak harus dijalankan secara berurutan, tetapi kode dapat dijalankan berdasarkan event atau bahkan dijadwalkan. Fungsi tersebut menerima dua argumen dengan penjelasan berikut :

- Argumen pertama merupakan sebuah fungsi yang akan dipanggil secara terjadwal dan asynchronous.
- Argumen kedua merupakan delay waktu dalam satuan *milisecond* yang menentukan delay dari pemanggilan fungsi pada argumen pertama.

Contoh Asynchronous setTimeout, dimana baris console.log dibawah duluan di tampilkan :

```
console.log('Selamat datang!');

setTimeout(() => {
  console.log('Terima kasih sudah mampir, silakan datang kembali!');
}, 3000)

console.log('Ada yang bisa dibantu?');
// output : Selamat datang!
// output : Ada yang bisa dibantu?
// output : Terima kasih sudah mampir, silakan datang kembali!
```

NOTE : banyak fungsi asynchronous lainnya, seperti fetch(), dsb.

42. ASYNCHRONOUS HANDLING DENGAN CALLBACK

Perilaku ini membuat penanganan proses asynchronous berbeda dengan synchronous. Sebagai developer, kita harus tahu cara menangani proses asynchronous agar kelak kita bisa mendapatkan nilai yang dihasilkan dari proses tersebut. Yaitu Callback dan promise.

- a. **Pola Callback** : Callback merupakan sebuah fungsi yang disisipkan pada argumen fungsi asynchronous dan akan dipanggil ketika proses dinyatakan selesai. Fungsi callback akan membawa nilai-nilai yang dihasilkan dari proses asynchronous sehingga kita bisa memanfaatkan nilai tersebut.

Contoh :

```
function getUsers(callback) {  
  // simulate network delay  
  setTimeout(() => {  
    const users = ['John', 'Jack', 'Abigail'];  
  
    // invoke callback after 3 seconds  
    callback(users);  
  }, 3000);  
}  
  
// callback function  
function usersCallback(users) {  
  console.log(users);  
}  
  
// call function getUsers and pass usersCallback as parameter  
getUsers(usersCallback);
```

Proses yang dijalankan secara asynchronous, biasanya berpotensi menghasilkan error. Misalnya, sebuah fungsi yang mengambil data dari internet akan menghasilkan error ketika jaringan kita bermasalah. Contoh lainnya, ketika berkas tersebut tidak ditemukan, fungsi yang membacanya akan menghasilkan error. Dengan alasan tersebut, pada fungsi callback selain membawa argumen *data*, ia juga membawa argumen *error*. Argumen error hanya akan memiliki nilai ketika proses asynchronous gagal atau mengalami gangguan. Contohnya :

```
function getUsers(isOffline, callback) {
  // simulate network delay
  setTimeout(() => {
    const users = ['John', 'Jack', 'Abigail'];
    // kalau true, berarti akan dijalankan callback dengan parameter error
    if (isOffline) {
      callback(new Error('cannot retrieve users due offline'), null);
      return;
    }
    // kalau false, berarti akan dijalankan callback dengan parameter users
    callback(null, users);
  }, 3000);
}

function usersCallback(error, users) {
  if (error) {
    console.log('process failed:', error.message);
    return;
  }
  console.log('process success:', users);
}

getUsers(false, usersCallback); // process success: ['John', 'Jack', 'Abigail']
getUsers(true, usersCallback); // process failed: cannot retrieve users due offline
```

- b. **Callback Hell** : Seiring kompleksnya pengembangan aplikasi, kita akan semakin banyak menggunakan fungsi yang berjalan secara asynchronous. Tidak jarang kita perlu mengagregasikan banyak data dari berbagai proses asynchronous. Masalah akan timbul ketika sebuah proses asynchronous perlu dijalankan setelah tahapan serupa lainnya selesai. Masalah yang berhubungan dengan *readability code* adalah masalah yang serius. Saking seriusnya, masalah ini dikenal sebagai *callback hell* atau *pyramid of doom*. Anda bisa lihat betapa menyeramkannya lengkungan kode ketika kita banyak menggunakan callback, hingga bisa tampak seperti gambar di bawah ini. Solusinya adalah dengan PROMISE.



```
firstTask(data, function(err, result) {
  secondTask(data, function(err, result) {
    thirdTask(data, function(err, result) {
      fourthTask(data, function(err, result) {
        fifthTask(data, function(err, result) {
          // Code
        });
      });
    });
  });
});
```

43. ASYNCHRONOUS HANDLING DENGAN PROMISE

Semenjak ECMAScript 2015 (ES6), pola callback bukan menjadi satu-satunya cara dalam menangani proses asynchronous di JavaScript. Pada era ES6 atau kita sebut era JavaScript modern, hadir Promise yang menjadi fitur dasar dalam menjalankan operasi asynchronous. Saat ini, banyak sekali fungsi-fungsi di Node.js atau Browser API yang memanfaatkan Promise dibandingkan dengan pola callback dalam menangani proses asynchronous-nya. Ketahuilah bahwa penamaan ini sebenarnya cukup akurat dan memang Promise bisa dianalogikan sebagai sebuah janji.

Jika kita pikir secara mendalam, sebenarnya proses yang dijalankan secara asynchronous merupakan sebuah janji karena hasil dari proses tersebut tidak bisa langsung kita dapatkan, tetapi perlu ditunggu beberapa saat. Sama halnya dengan janji di dunia nyata yang butuh waktu untuk terpenuhi.

Selain sama-sama membutuhkan waktu, kesamaan lainnya terjadi pada hasil yang didapatkan. Di dunia nyata, janji bisa terpenuhi atau gagal. Contoh, jika teman Anda berjanji akan mengajak lari pada sore hari, bisa saja janji tersebut gagal terpenuhi karena hujan atau alasan lainnya. Promise di JavaScript pun memiliki konsep yang sama karena memiliki beberapa kondisi, yaitu *pending*, *fulfilled*, dan *rejected*.

- **Pending** : Keadaan Promise sedang berjalan
- **Fulfilled** : Keadaan Promise yang terpenuhi
- **Rejected** : Keadaan Promise gagal terpenuhi

Alih-alih fungsi `getUsers()` menerima `callback` sebagai argumen, dia mengembalikan objek Promise. Lalu, bagaimana dengan nilai yang belum dibawa oleh argumen callback? Nah, di sini Promise menawarkan penggantinya dengan memanfaatkan `resolve` dan `reject`. Dalam kasus yang menyebabkan proses asynchronous error, kita bisa bawa error tersebut menggunakan `reject`, sedangkan pada kasus proses asynchronous berjalan sukses, kita bisa bawa nilai tersebut dengan `resolve`. Contoh :

```
function getUsers(isOffline) {  
  // return a Promise object  
  return new Promise((resolve, reject) => {  
  
    // simulate network delay  
    setTimeout(() => {  
      const users = ['John', 'Jack', 'Abigail'];  
  
      if (isOffline) {  
        reject(new Error('cannot retrieve users due offline'));  
        return;  
      }  
  
      resolve(users);  
    }, 3000);  
  });  
}
```

NOTE : Objek Promise dibuat dengan cara memanggil constructor Promise, yakni `new Promise()`. Kemudian di dalam constructor, Anda wajib memberikan argumen berupa fungsi yang di dalamnya terdapat operasi asynchronous. Fungsi tersebut juga diberikan akses terhadap dua argumen, yaitu `resolve` dan `reject`. Kedua argumen ini bisa Anda manfaatkan dalam membawa hasil dari proses asynchronous berupa data ataupun error.

Setelah fungsi `getUsers()` diubah dari callback menjadi *Promise-based*, cara penggunaan fungsinya pun akan berubah. Fungsi yang mengembalikan objek Promise akan memiliki fungsi `.then` dan `.catch`. Fungsi tersebut digunakan untuk mengambil nilai yang dibawa oleh `resolve` dan `reject`. Gambarannya seperti ini.

```
function getUsers(isOffline) {
  // return a promise object
  return new Promise((resolve, reject) => {

    // simulate network delay
    setTimeout(() => {
      const users = ['John', 'Jack', 'Abigail'];

      if (isOffline) {
        reject(new Error('cannot retrieve users due offline'));
        return;
      }

      resolve(users);
    }, 3000);
  });
}

getUsers(false)
  .then(users => console.log(users))
  .catch(err => console.log(err.message));

// output : ['John', 'Jack', 'Abigail']
```

44. MENGUBAH CALLBACK MENJADI PROMISE DENGAN PROMISIFY

kita sudah belajar cara mengubah callback menjadi Promise. Sekarang kita akan coba mengubah callback menjadi Promise dengan pendekatan yang berbeda. Di Node.js, terdapat sebuah fungsi dari core module `util` yang dapat mengubah fungsi asynchronous *callback-based* menjadi *Promise-based* dengan mudah, ia bernama `promisify`.

```
function getUsers(isOffline, callback) {
  // simulate network delay
  setTimeout(() => {
    const users = ['John', 'Jack', 'Abigail'];
    // kalau true, berarti akan dijalankan callback dengan parameter error
    if (isOffline) {
      callback(new Error('cannot retrieve users due offline'), null);
      return;
    }
    // kalau false, berarti akan dijalankan callback dengan parameter users
    callback(null, users);
  }, 3000);
}
```

Sebelumnya, kita sudah mengubahnya menjadi *Promise-based* dengan me-refactor kode fungsi tersebut. Namun, dalam skenario nyata, sering kali kita tidak memiliki akses untuk me-refactor sebuah fungsi. Contohnya ketika menggunakan library pihak ketiga. Ketika menghadapi kasus tersebut, bagaimana cara termudah dalam mengubah fungsi tersebut menjadi *Promise-based*?

Node.js menawarkan solusi, yaitu menggunakan fungsi `promisify` dari core module `util`. Dengannya, kita bisa secara mudah membuat fungsi baru yang mengimplementasikan Promise berdasarkan fungsi *callback-based*, contohnya seperti ini.

```
const { promisify } = require('util');

function getUsers(isOffline, callback) {
  // simulate network delay
  setTimeout(() => {
    const users = ['John', 'Jack', 'Abigail'];
    if (isOffline) {
      callback(new Error('cannot retrieve users due offline'), null);
      return;
    }
    callback(null, users);
  }, 3000);
}

// create a Promise version of getUsers
const getUsersPromise = promisify(getUsers);
```

Sekarang, kita memiliki fungsi `getUsers()`, tetapi dapat menerapkan Promise tanpa perlu mengubahnya lagi.

Kita dapat menggunakan pemanggilan promise seperti berikut :

```
const { promisify } = require('util');

function getUsers(isOffline, callback) {
  // simulate network delay
  setTimeout(() => {
    const users = ['John', 'Jack', 'Abigail'];
    if (isOffline) {
      callback(new Error('cannot retrieve users due offline'), null);
      return;
    }
    callback(null, users);
  }, 3000);
}

// create a Promise version of getUsers
const getUsersPromise = promisify(getUsers);

getUsersPromise(false)
  .then(users => console.log(users)) // ['John', 'Jack', 'Abigail']
  .catch(err => console.log(err.message));

getUsersPromise(true)
  .then(users => console.log(users))
  .catch(err => console.log(err.message)); // cannot retrieve users due offline
```

NOTE : beberapa catatan ketika Anda hendak mengubah fungsi asynchronous callback-based menjadi Promise-based menggunakan `util.promisify()` :

- Promisify selalu menganggap callback berada pada argumen paling akhir sebuah fungsi asynchronous. Dengan begitu, Anda tidak dapat menggunakan promisify jika callback berada di posisi awal ataupun tengah argumen. (Default node, callback selalu diakhir argumen)
- Promisify akan bekerja dengan baik jika callback memiliki struktur argumen `callback(error, data)`. Jika callback memiliki struktur argumen di luar aturan tersebut, nilai yang dibawa oleh Promise ketika fulfilled dan rejected bisa tertukar.
- Singkatnya, promisify hanya dapat digunakan pada fungsi yang mengikuti aturan Node.js callback. Inilah salah satu alasan perlunya mengikuti aturan yang standar.

Dalam kasus nyata, Anda bisa gunakan `promisify` dalam mengubah berbagai fungsi yang disediakan Node.js menjadi Promise-based, contohnya fungsi `fs.readFile()` yang digunakan untuk membaca berkas secara asynchronous.

```
const fs = require('fs');
const { promisify } = require('util');

//deklarasi fungsi readFilePromise dengan promisify
const readFilePromise = promisify(fs.readFile);

// './data.txt' adalah berkas terpisah
// dalam direktori yang sama
readFilePromise('./data.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => console.log(err.message));
```

45. PROMISE BERANTAI

Masih ingat tentang masalah *readability code* yang disebabkan *callback hell*? Masalah tersebut kerap terjadi ketika proses agregasi data dari berbagai proses asynchronous yang saling bergantung antara satu dan yang lainnya. Dengan hadirnya Promise, kita bisa terbebas dari *callback hell*.

Contoh Cerita : Dalam kehidupan sehari-hari, kita mungkin pernah melakukan *chaining Promise* tanpa disadari. Misalkan ketika ingin menonton film di bioskop, kita perlu melakukan beberapa tahap sebelum akhirnya bisa menikmati film tersebut. Misalnya, tahap pertama kita perlu **menarik uang** untuk nantinya membeli tiket bioskop. Tahap ini bisa direpresentasikan menjadi sebuah fungsi bernama `withdrawMoney()`. Fungsi tersebut akan mengembalikan sejumlah uang jika saldonya mencukupi, dalam arti lain Promise-nya menjadi *fulfilled* (**resolve**). Selain itu, fungsi akan menampilkan pesan error jika saldonya kurang, yang artinya *rejected* (**reject**). Jika Promise-nya fulfilled, tahap selanjutnya akan dijalankan.

Tahap kedua adalah membeli tiket bioskop yang dapat direpresentasikan dengan fungsi bernama `buyCinemaTicket()`. Fungsi tersebut menerima argumen uang yang sudah kita tarik. Jika uang yang diberikan pada argumen kurang dari harga tiketnya, Promise akan di-reject dengan pesan “not enough money to buy ticket”. Jika nilai argumen cukup, Promise akan mengembalikan nilai “tiket-1”.

Tahap ketiga adalah masuk ke bioskop, aktivitas ini bisa digambarkan dengan fungsi `goInsideCinema()` yang menerima satu argumen berupa tiket. Jika tidak ada tiket, Promise akan di-reject dengan pesan “no ticket”. Jika ada, Promise akan di-resolve dengan pesan “enjoy the movie”.

Penerapan dalam bentuk promise berantai :

```
1  function withDrawMoney(amount) {
2      return new Promise((resolve, reject) => {
3          setTimeout(() => {
4              if (amount > 100) {
5                  reject(new Error('Not enough money to withdraw'));
6              }
7
8              resolve(amount);
9          }, 1000);
10     });
11 }
12
13 function buyCinemaTicket(money) {
14     return new Promise((resolve, reject) => {
15         setTimeout(() => {
16             if (money < 10) {
17                 reject(new Error('not enough money to buy ticket'));
18             }
19
20             resolve('ticket-1');
21         }, 1000);
22     });
23 }
24
25 function goInsideCinema(ticket) {
26     return new Promise((resolve, reject) => {
27         setTimeout(() => {
28             if (!ticket) {
29                 reject(new Error('no ticket'));
30             }
31
32             resolve('enjoy the movie');
33         }, 1000);
34     });
35 }
36
37 function watchMovie() {
38     withDrawMoney(10)
39         .then((money) => {
40             return buyCinemaTicket(money);
41         })
42         .then((ticket) => {
43             return goInsideCinema(ticket);
44         })
45         .then((result) => {
46             console.log(result);
47         })
48         .catch((error) => {
49             console.log(error.message);
50         });
51 }
52
53 watchMovie();
```

NOTE : Ada tips dalam melakukan Promise chaining di atas. Anda bisa membuat kode tampak lebih bersih dan singkat dengan memanfaatkan arrow function implisit `return`. Namun, tips ini hanya dapat digunakan jika arrow function hanya memiliki satu baris kode saja.

```
function watchMovie() {  
  withDrawMoney(10)  
    .then((money) => buyCinemaTicket(money))  
    .then((ticket) => goInsideCinema(ticket))  
    .then((result) => console.log(result))  
    .catch((error) => console.log(error.message));  
}
```

46. PROMISE STATIC METHOD

Promise memiliki beberapa static method yang dapat digunakan untuk mempermudah penggunaan dalam berbagai kasus. Yaitu :

- a. **Promise.all** : digunakan untuk mengeksekusi banyak Promise secara paralel. Method ini menerima sebuah array yang berisi beberapa Promise sebagai argumen. Fungsi ini akan mengembalikan sebuah Promise baru dan membawa nilai yang di-resolve dalam masing-masing Promise yang diletakkan pada array argumen.

```
const promise1 = new Promise((resolve) => setTimeout(() => resolve(1), 1000));  
const promise2 = new Promise((resolve) => setTimeout(() => resolve(2), 2000));  
const promise3 = new Promise((resolve) => setTimeout(() => resolve(3), 3000));  
  
Promise.all([promise1, promise2, promise3]).then((values) => console.log(values));  
// [1, 2, 3] setelah 3 detik
```

kita menggunakan `Promise.all()` untuk mengeksekusi ketiga Promise tersebut secara paralel. Ketika semua selesai dieksekusi, `Promise.all()` akan mengembalikan Promise baru. Ia membawa nilai array berisi nilai-nilai yang dikembalikan oleh ketiga Promise tersebut. Jika terdapat rejection pada salah satu Promise, `Promise.all()` akan langsung menghasilkan rejected tanpa mengembalikan nilai Promise lain yang statusnya *fulfilled*. Berikut contohnya.

```
const promise1 = new Promise((resolve) => setTimeout(() => resolve(1), 1000));  
const promise2 = new Promise((resolve, reject) => setTimeout(() => reject(new Error('ups')), 2000));  
const promise3 = new Promise((resolve) => setTimeout(() => resolve(3), 3000));  
  
Promise.all([promise1, promise2, promise3])  
  .then((values) => console.log(values))  
  .catch((error) => console.log(error.message)); // ups, karena paralel
```

- b. **Promise.race** : digunakan untuk mengeksekusi beberapa Promise secara paralel seperti **Promise.all()**. Namun, ia hanya mengembalikan nilai Promise yang paling cepat menyelesaikan eksekusinya.

```
const promise1 = new Promise((resolve) => setTimeout(() => resolve(1), 1000));
const promise2 = new Promise((resolve) => setTimeout(() => resolve(2), 2000));
const promise3 = new Promise((resolve) => setTimeout(() => resolve(3), 3000));

Promise.race([promise1, promise2, promise3])
  .then((value) => console.log(value)); // 1 setelah 1 detik
```

kita membuat tiga Promise yang masing-masing membutuhkan waktu selesai berbeda-beda. Kemudian, kita menggunakan **Promise.race()** untuk mengeksekusi ketiganya secara paralel. **Ingat! Promise.race()** hanya mengembalikan nilai Promise yang prosesnya paling cepat. Jadi, ia hanya mengembalikan nilai dari **promise1**.

Lalu, bagaimana jika salah satu Promise ada yang **rejected**? Jika proses rejection merupakan proses yang paling cepat, ia akan mengembalikan rejection tersebut. **Jika rejection tidak kalah cepat dengan proses lain yang nilainya fulfilled, ia akan tetap mengembalikan nilai resolved tercepat.**

```
const promise1 = new Promise((resolve, reject) => setTimeout(() => reject(new Error('ups')), 1000));
const promise2 = new Promise((resolve) => setTimeout(() => resolve(2), 2000));
const promise3 = new Promise((resolve) => setTimeout(() => resolve(3), 3000));

Promise.race([promise1, promise2, promise3])
  .then((value) => console.log(value))
  .catch((error) => console.log(error.message)); // Ups
```

- c. **Promise.allSettled** : bekerja mirip seperti **Promise.all()**. Namun, **Promise.allSettled()** mengembalikan seluruh hasil dari Promise yang dieksekusi dalam bentuk array of object dengan format berikut.

```
{
  status: 'fulfilled' | 'rejected',
  value: 'value of the Promise',
  reason: 'error of the Promise',
}
```

Dari struktur objek yang dihasilkan, Anda bisa melihat bahwa **Promise.allSettled()** akan mengembalikan seluruh nilai Promise yang dijalankan, baik yang statusnya *fulfilled* ataupun *rejected*.

Contoh :

```
const promise1 = new Promise((resolve) => setTimeout(() => resolve(1), 1000));
const promise2 = new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error")), 2000));
const promise3 = new Promise((resolve) => setTimeout(() => resolve(3), 3000));

Promise.allSettled([promise1, promise2, promise3])
  .then((results) => console.log(results));
// [{status: "fulfilled", value: 1}, {status: "rejected", reason: Error}, {status: "fulfilled", value: 3}] setelah 3 detik
```

Kita menggunakan **Promise.allSettled()** untuk menjalankan seluruh Promise secara paralel. **Promise.allSettled()** mengembalikan array of object seperti yang sudah dijelaskan sebelumnya.

- d. **Promise.any** : Cara kerja method ini mirip seperti **Promise.race()**, tetapi hanya **mengembalikan nilai tercepat yang berstatus fulfilled**. Jika seluruh Promise berstatus rejected, method ini akan mengembalikannya dengan pesan "All Promises were rejected".

```
// fulfilled sample
const promiseResolve1 = new Promise((resolve, reject) => setTimeout(() => resolve("1"), 1000));
const promiseResolve2 = new Promise((resolve, reject) => setTimeout(() => resolve("2"), 2000));
const promiseResolve3 = new Promise((resolve, reject) => setTimeout(() => resolve("3"), 3000));

Promise.any([promiseResolve1, promiseResolve2, promiseResolve3])
  .then((value) => console.log(value)) // 1
  .catch((error) => console.log(error.message));

// rejected sample
const promiseReject1 = new Promise((resolve, reject) => setTimeout(() => reject(new Error('1')), 1000));
const promiseReject2 = new Promise((resolve, reject) => setTimeout(() => reject(new Error('2')), 2000));
const promiseReject3 = new Promise((resolve, reject) => setTimeout(() => reject(new Error('3')), 3000));

Promise.any([promiseReject1, promiseReject2, promiseReject3])
  .then((value) => console.log(value))
  .catch((error) => console.log(error.message)); // All Promises were rejected
```

47. ASYNCHRONOUS HANDLING DENGAN SYNTAKS ASYNC DAN AWAIT

Bagaimana jika kita bisa menangani proses asynchronous layaknya proses synchronous? Tentunya, ini akan membuat kode kita jauh lebih ringkas dan mudah dipahami karena **tidak perlu menggunakan .then() dan .catch()** untuk mendapatkan nilainya. Untunglah harapan tersebut kini sudah terealisasi dengan hadirnya fitur **async** dan **await** di JavaScript versi ECMAScript 2017.

Sintaks **async** dan **await** memungkinkan kita untuk menulis kode dalam menangani proses asynchronous *Promise-based* dengan cara yang sama seperti kode synchronous. Dalam penanganan *error-nya (rejection)* pun kita bisa menggunakan **try** dan **catch** layaknya error yang dihasilkan oleh proses synchronous. Mari kita lihat contohnya dengan mengangkat kembali kasus menonton film di bioskop secara asynchronous.

```
function watchMovie() {
  withdrawMoney(10)
    .then((money) => buyCinemaTicket(money))
    .then((ticket) => goInsideCinema(ticket))
    .then((result) => console.log(result))
    .catch((error) => console.log(error.message));
}
watchMovie();
```

Perubahan yang didapatkan Ketika menggunakan async dan await :

```
async function watchMovie() {
  try {
    const money = await withdrawMoney(10);
    const ticket = await buyCinemaTicket(money);
    const result = await goInsideCinema(ticket);

    console.log(result);
  } catch (error) {
    console.log(error.message);
  }
}

watchMovie();
```

Untuk menggunakan fitur async dan await, kita perlu mengubah sebuah fungsi agar berjalan secara asynchronous dengan menambahkan kata kunci `async` pada awal fungsi. Setelah itu, di dalam fungsi tersebut, kita bisa mengambil data yang dihasilkan oleh Promise dengan menggunakan sintaks `await`. Sintaks `await` akan memberi tahu JavaScript untuk menunggu proses Promise selesai sebelum mengeksekusi kode baris selanjutnya. Jadi, kita bisa melakukan Promise berantai tanpa perlu menggunakan `.then()`. Simak juga cara penanganan error-nya. Kita sudah tidak lagi menggunakan `.catch()`, tetapi `try` dan `catch` layaknya proses synchronous.

Ketahuiilah bahwa setiap fungsi yang didefinisikan dengan kata `async` akan selalu mengembalikan sebuah Promise. Contoh pada kode di atas, kita bisa menggunakan fungsi `.then()` setelah pemanggilan fungsi `watchMovie()` seperti ini.

```
watchMovie().then(() => console.log('done'));
// enjoy the movie
// done
```

Jika fungsi yang diberikan kata “async” mengembalikan Promise, bagaimana caranya mengontrol status Promise tersebut agar bernilai *fulfilled* atau *rejected*? Kata `async` secara implisit menggunakan `return` untuk mengubah status Promise menjadi fulfilled dan menggunakan `throw` untuk mengubah status Promise menjadi rejected. Mari kita buktikan dengan mengubah fungsi `watchMovie()` dengan mengembalikan nilai dan *throw error*.

```
async function watchMovie(amount) {
  try {
    const money = await withdrawMoney(amount);
    const ticket = await buyCinemaTicket(money);
    const result = await goInsideCinema(ticket);

    return result;
  } catch (error) {
    throw error;
  }
}

watchMovie(10)
  .then((result) => console.log(result)) // enjoy the movie
  .catch((error) => console.log(error.message));

watchMovie(5)
  .then((result) => console.log(result))
  .catch((error) => console.log(error.message)); // not enough money to buy ticket
```


Sintaks async dan await memiliki beberapa keuntungan dibandingkan dengan penggunaan callback dan Promise secara langsung :

- Lebih mudah dipahami dan ditulis
- Terhindar dari callback
- Lebih mudah dalam mengelola error

48. NPM / NODE PACKAGE MANAGER

Package merupakan **sebuah kode yang dibuat untuk menyelesaikan suatu masalah**. Contohnya ketika aplikasi yang kita buat membutuhkan fitur kalender sementara fitur tersebut tidak didukung secara default oleh JavaScript. Alih-alih membuat fitur kalender dari nol, kita dapat menggunakan package yang telah dibuat oleh developer lain. Waktu pembuatan fitur menjadi lebih singkat. Package manager merupakan tools yang dapat membantu pengelolaan package pada proyek agar lebih mudah. NPM adalah salah satu package manager yang banyak digunakan oleh JavaScript developer dalam mengelola package, selain NPM ada juga Yarn Package Manager. Bahkan, melalui NPM kita dapat menggunakannya pada proyek yang berbeda. Dalam arti lain, package yang tersedia pada NPM adalah sebuah module.

49. INSTALLING, USING, & UNINSTALL PACKAGE

a. Installing package

- menambahkan package eksternal ke dalam project aplikasi kita. Anda dapat membuat project baru atau menggunakan project node yang telah dibuat sebelumnya. Pastikan di dalam folder project terdapat berkas **package.json**, jika belum, jalankan perintah **npm init** pada project Anda.
 - a) **Local installation** : Local package akan dipasang di dalam direktori atau folder yang sama dengan project kita. Package ini akan diletakkan di dalam folder **node_modules**.
 - b) **Global installation** : Sementara global package dipasang di satu tempat pada sistem perangkat kita (tergantung pengaturan pada saat instalasi node/npm).
 - c) Umumnya, semua package harus diinstal secara lokal. Ini memastikan setiap project atau aplikasi di komputer kita memiliki package dan versi yang sesuai dengan kebutuhan. Untuk menginstal package secara lokal caranya sama seperti yang telah kita bahas sebelumnya, yaitu dengan perintah **npm install**. “ **npm install <package-name>** ”. Package harus diinstal secara global hanya saat ia **menyediakan perintah yang dapat dieksekusi dari CLI dan digunakan kembali di semua project**. Npm, nodemon, mocha
- Sebagai contoh, kita akan menggunakan *library lodash*. Ketik “ **npm i lodash** ”. Jika berhasil, akan muncul **sebuah dependencies** di **package.json**

```
"dependencies": {  
  "lodash": "^4.17.21"  
}
```

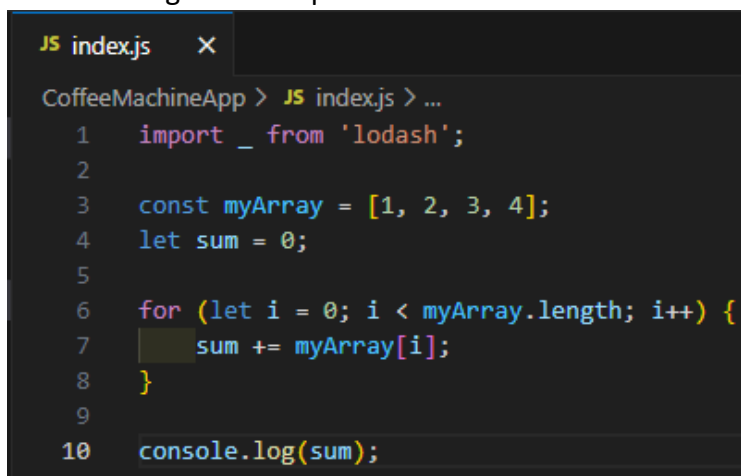

- Sebenarnya terdapat dua tipe object dependencies dalam berkas **package.json**. Yang pertama object **dependencies** dan yang kedua object **devDependencies**. Apa perbedaannya? Objek dependencies merupakan objek yang menampung package yang kita gunakan untuk membuat aplikasi. Sedangkan objek devDependencies digunakan untuk package yang berkaitan hanya saat **proses pengembangan aplikasi yang digunakan untuk testing**. Package seperti jest ini hanya digunakan saat proses pengembangan aplikasi. **Ia tidak digunakan lagi ketika aplikasi rilis atau digunakan langsung oleh pengguna.**

b. Using package : Perlu kita pahami kembali bahwa package yang kita tambahkan ke dalam project sebenarnya adalah module. Itulah kenapa di dalam project kita akan muncul juga folder **node_modules**. Di dalamnya berisi kode-kode JavaScript yang menyusun suatu package. Jika Anda “berani”, Anda dapat melihat seperti apa kode di dalam package lodash.

- Karena merupakan module, kita bisa menambahkan kode dari package menggunakan keyword **import** seperti yang telah dipelajari pada materi Module. Syntax : *import variableName from 'package-name';*

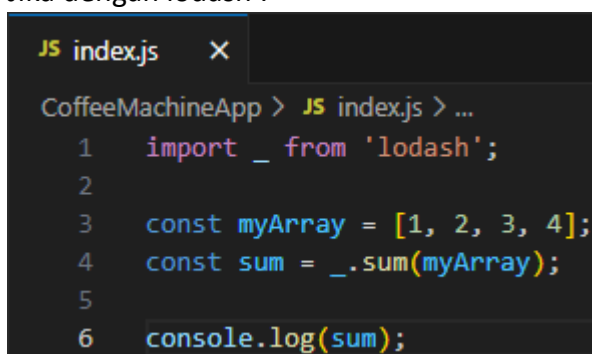
```
import _ from 'lodash';  
// standar lodash pakai _ underscore
```

- Di dalam dokumentasinya, lodash menyebutkan bahwa mereka menyediakan utilitas untuk membuat JavaScript lebih mudah dengan menghilangkan kerumitan ketika menggunakan array, number, object, string, dll. Misalnya, untuk menjumlahkan setiap nilai number di dalam array, lakukan dengan cara seperti berikut.



```
JS index.js X  
CoffeeMachineApp > JS index.js > ...  
1  import _ from 'lodash';  
2  
3  const myArray = [1, 2, 3, 4];  
4  let sum = 0;  
5  
6  for (let i = 0; i < myArray.length; i++) {  
7    sum += myArray[i];  
8  }  
9  
10 console.log(sum);
```

Jika dengan lodash :



```
JS index.js X  
CoffeeMachineApp > JS index.js > ...  
1  import _ from 'lodash';  
2  
3  const myArray = [1, 2, 3, 4];  
4  const sum = _.sum(myArray);  
5  
6  console.log(sum);
```

-

c. **Uninstall package** : Untuk melakukannya cukup mudah. Jika package berada pada objek dependencies, kita dapat menghapusnya menggunakan perintah:

- ***npm uninstall <package-name>*** : npm uninstall lodash
- Atau jika di dalam devDependencies : ***npm uninstall <package-name> --save-dev***

NOTE : Selain menghapus package, perintah tersebut juga menghapus package node_modules

50. JAVASCRIPT TESTING

Sebelum sebuah program dipublikasikan, seharusnya sebuah program harus melalui tahap pengujian terlebih dahulu. Proses pengujian tersebut digunakan untuk melakukan kontrol kualitas dari kode yang kita tulis, memastikan bahwa software yang akan dipublikasikan ke pengguna minim adanya kecacatan atau bugs. Oleh sebab itu, sebagai seorang pemrogram **kita harus melakukan testing saat proses pengembangan perangkat lunak.**

a. **Manual** : Proses pengujian secara manual oleh seorang yang ditunjuk sebagai test, atau sebagian pengguna yang memang mendapatkan akses untuk pengujian *pre-release*. Proses ini biasanya berkaitan dengan usability, accessibility dari sebuah aplikasi.

b. **Otomatis** : Proses pengujian secara otomatis dilakukan oleh komputer dengan menuliskan script khusus, biasanya dilakukan oleh software engineer langsung ataupun oleh seorang QA Engineer. Proses ini biasanya berkaitan dengan fungsionalitas dari sebuah aplikasi.

c. **Jenis-jenis pengujian** :

- **Static test** : Memastikan tidak adanya **typo** (naming convention yang standar) dan memastikan tidak ada error types.
- **Unit test** : Memastikan bahwa setiap unit kode yang kita tulis sudah bisa bekerja sesuai harapan. Unit sendiri berarti komponen terkecil yang **dapat diuji secara terisolasi dalam perangkat lunak** yang kita buat, dapat berupa fungsi bahkan kelas jika kita menggunakan paradigma OOP. Proses ini juga dapat diautomasikan.
- **Integration test** : Memastikan beberapa serangkaian fungsi yang saling ketergantungan satu sama lain berjalan semestinya. Proses pengujian ini dapat dilakukan secara otomatis dengan menuliskan script test.
- **End-to-end test** : Proses pengujian sebuah aplikasi untuk menguji flow dari awal hingga akhir, layaknya seorang user saat menggunakan aplikasi. Memastikan bahwasanya aplikasi berfungsi selayaknya. Biasanya proses ini dapat dilakukan secara otomatis maupun manual oleh tester.

51. JEST

Dalam penulisan kode pengujian, kita dapat menggunakan library tambahan untuk mempermudah penulisan kode pengujian. Jest merupakan salah satu framework testing paling populer untuk menuliskan kode pengujian pada bahasa pemrograman JavaScript. Jest dapat digunakan untuk menuliskan script testing pada aplikasi backend maupun frontend.

52. MENULIS & MELAKUKAN PENGUJIAN KODE JEST

To be Continued, next time