# Programming JavaScript

oleh Maulana Ifandika

Komentar
Variabel
Konstanta
Tipe Data
JavaScript Tentang Object
Bermain dengan Struktur Data
Fungsi
OOP (Object Oriented Programming)
Functional Programming

## |Komentar

Ini adalah expresei/baris kode yang tidak di eksekusi oleh Js, sebagai informasi tambahan.

Komentar single-line
// single-line

Komentar multi-line
/*
Komentar
Multi-line
*/

**[Kode]**

```
// Komentar
console.log("Komentar single-line");
// Komentar single-line/tunggal

/*
Komentar
multi-line/ganda
*/
console.log("Komentar multi-line");
```

**[Output]**

```
Komentar single-line
Komentar multi-line
```

## |Variabel

Sebuah data yang dapat menyimpan nilai dan nilai dapat diubah. Ada 2 jenis variabel

- let
- var

**[Kode]**
// Variabel

```
let age = 17;
var price = 5000;

console.log("age: "+age);
console.log("price: "+price);

age = 50;
price = 1000;

console.log("age: "+age);
console.log("price: "+price);
```

**[Keluaran]**
```
age: 17
price: 5000
age: 50
price: 1000
```

# Konstanta

Sebuah data yang dapat menyimpan nilai dan nilai tidak dapat diubah.

- const

**[Kode]**
```
// Konstanta

const gradute = true;
console.log("Graduate: "+gradute);

gradute = false;
console.log("Graduate: "+gradute);
```

**[Keluaran]**
```
Graduate: true
D:\Coding\JavaScript\konstanta.js:6
gradute = false;
      ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (D:\Coding\JavaScript\konstanta.js:6:9)
    at Module._compile (node:internal/modules/cjs/loader:1358:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1416:10)
    at Module.load (node:internal/modules/cjs/loader:1208:32)
    at Module._load (node:internal/modules/cjs/loader:1024:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:174:12)
    at node:internal/main/run_main_module:28:49

Node.js v20.14.0
```

# |Tipe Data

Tipe data yang ada pada JavaScript

**> Numbers**

Tipe data bilangan untuk menampung nilai bulat, pecahan, dll.

**[Kode]**

```
console.log(5);

let num_50 = new Number('50');
let num_1000 = new Number( 1000 );
let num_f = new Number(false);
let num_t = new Number(true);
let num_n = new Number(null);
// Nilai tidak bisa dikonversi = NaN/Tidak diketahui
let num_under = new Number(undefined);
let num_this = new Number('This number is 99');

// ininify
console.log('10 / 0: '+10 / 0);

// dengan string/kata
console.log('5' + 4);
console.log("2" + 2);
console.log("2" * 2);
console.log("2" - 2);
console.log("2" / 2);
console.log("two" / 2);
```

**[Keluaran]**

```
5
[Number: 50]
[Number: 1000]
[Number: 0]
[Number: 1]
[Number: 0]
[Number: NaN]
[Number: NaN]
10 / 0: Infinity
54
22
4
0
1
NaN
```

**> Strings**

Setiap rangkaian karakter—huruf, angka, simbol, dan sebagainya—di antara rangkaian tanda kutip ganda ("), tanda kutip tunggal ('), atau tanda petik terbalik (`) adalah primitif string. Anda telah melihat beberapa contoh string dalam kursus ini: contoh console.log dalam modul sebelumnya berisi primitif string.

[main.js]

```
// ============ String ================
console.log( 'Welcome Js' );
console.log( "Welcome Js" );
console.log( '"A string," I said.' );
console.log( '\"A string,\" I said.' );

// String object
let string_99 = String(99);
console.log(string_99);
console.log(typeof string_99);

let string_obj = new String("This string obj");
console.log(string_obj);
```

```
console.log(typeof string_obj);

// Concatenation
console.log( "My" + " string." );

// String literal & template
// const myString = "This is
// a string.";
// Uncaught SyntaxError: "" string literal contains an unescaped line break

const myString2 = `This is
a string.`;
console.log( myString2 );

console.log( "The result is " + ( 2 + 4 ) + "." );
console.log( `The result is ${ 2 + 4 }.` );

const myNoun = "template literal";
function myTagFunction( myStrings, myPlaceholder ) {
  const myInitialString = myStrings[ 0 ];
  console.log( `${ myInitialString }modified ${ myPlaceholder }.` );
}

myTagFunction`I'm a ${ myNoun }.`;
```

**Output:**

```
Welcome Js
Welcome Js
"A string," I said.
"A string," I said.
99
string
[String: 'This string obj']
object
My string.
This is
a string.
The result is 6.
The result is 6.
I'm a modified template literal.
```

**> Booleans**

Semua nilai dalam JavaScript secara implisit bernilai benar atau salah. Objek Boolean dapat digunakan untuk memaksa nilai menjadi boolean benar atau salah, berdasarkan status benar atau salah implisit dari nilai tersebut:

**[main.js]**

```
// ============ Boolean ================
let bol = Boolean( "A string literal" );
let bol2 = Boolean();
let bol3 = Boolean("");
console.log("bol: "+bol);
console.log("bol2: "+bol2);
console.log("bol3: "+bol3);

console.log("NaN: "+Boolean( NaN ));
console.log("-0: "+Boolean( -0 ));
console.log("5: "+Boolean( 5 ));
console.log("false: "+Boolean( "false" ));

// const
const falseBoolean = Boolean( 0 );
const falseObject = new Boolean( 0 );

console.log( "falseBoolean: "+falseBoolean  );
console.log( "falseObject: "+falseObject  );
console.log( "falseBoolean = false: "+(falseBoolean == false));
console.log( "!!falseObject: "+!!falseObject  );
falseObject.valueOf();
```

**Output:**

```
bol: true
bol2: false
bol3: false
NaN: false
-0: false
5: true
false: true
falseBoolean: false
falseObject: false
falseBoolean = false: true
!!falseObject: true
```

**> null & undefined**

**null** : Kata kunci null menunjukkan ketiadaan nilai yang didefinisikan secara sengaja. null adalah primitif, meskipun operator typeof mengembalikan bahwa null adalah objek. Ini adalah kesalahan yang terbawa dari JavaScript versi pertama dan sengaja tidak ditangani untuk menghindari pelanggaran perilaku yang diharapkan di seluruh web.

```
typeof null
// object
```

Anda dapat menetapkan variabel sebagai null dengan harapan bahwa variabel tersebut mencerminkan nilai yang ditetapkan padanya di beberapa titik dalam skrip atau nilai yang secara eksplisit tidak ada. Anda juga dapat menetapkan nilai null ke referensi yang ada untuk menghapus nilai sebelumnya.

**undefined** : undefined adalah nilai primitif yang ditetapkan ke variabel yang baru saja dideklarasikan, atau ke nilai hasil operasi yang tidak mengembalikan nilai yang berarti. Misalnya, hal ini dapat terjadi saat Anda mendeklarasikan fungsi di konsol pengembang browser:

```
function undefined() {}
// undefined
```

Suatu fungsi secara eksplisit mengembalikan undefined ketika pernyataan return-nya tidak mengembalikan nilai apa pun.

**[main.js]**

```
// ============ Null & Unfefined ================
let let_blank;
var var_blank;
console.log("let_blank: "+let_blank);
console.log("var_blank: "+var_blank);
console.log(null);
console.log(undefined);

// perbandingan
console.log("null == undefined: "+(null == undefined));
console.log("null === undefined: "+(null === undefined));
```

**Output:**

```
let_blank: undefined
var_blank: undefined
null
undefined
null == undefined: true
null === undefined: false
```

**> BigInt**

Primitif BigInt merupakan tambahan yang relatif baru pada JavaScript, yang memungkinkan operasi matematika pada angka di luar rentang yang diizinkan oleh Number. Untuk membuat BigInt, tambahkan n di akhir literal angka, atau berikan nilai integer atau string numerik ke fungsi BigInt().

**[main.js]**
```
// ============ BigInt ================
const myNumber = 9999999999999999;
const myBigInt = 9999999999999999n;

console.log(typeof myNumber);
console.log(typeof myBigInt);

console.log("myNumber: "+myNumber);
console.log("myBigInt: "+myBigInt);

const big_int = BigInt(99999999999999999999n);

console.log("BigInt: "+big_int);

// operasi
// 9999999999999999n + 5 (harus BigInt dengan BigInt)
// Uncaught TypeError: can't convert BigInt to number

console.log( 9999999999999999 + 10 );  // Dibulatkan
console.log( 9999999999999999n + 10n );
```

**Output:**
```
number
bigint
myNumber: 10000000000000000
myBigInt: 9999999999999999
BigInt: 999999999999999999999
10000000000000010
10000000000000009n
```

**> Symbol**
Simbol adalah primitif yang relatif baru yang diperkenalkan di ES6. Primitif simbol mewakili nilai unik yang tidak pernah bertabrakan dengan nilai lain, termasuk nilai primitif simbol lainnya. Dua primitif string yang terdiri dari karakter identik dievaluasi sebagai sama persis:

# | JavaScript Tentang Object
Bagaimana membuat object di JavaScript, menggunakannya, dsb.

**Membuat Object**

**[Kode]**
```
// Membuat Object

const user_blank = {};
const user_new = { name: "ifandika", age: 19 };

const user_new2 = {
  name: "ifandika",
```

```
  age: 19
};
```

**[Keluaran]**

```
user_blank: {}
user_new: {"name":"ifandika","age":19,"country":"Indonesia"}
user_new-name: ifandika
user_new-age: 19
```

**Mengakses Properti di Object**

**[Kode]**

```
const user_new = {
  name: "ifandika",
  age: 19,
  ‘country’: ‘Indonesia’
};

console.log("user_new: "+JSON.stringify(user_new));

// Akses dengan dot
console.log("user_new-name: "+user_new.name);
console.log("user_new-age: "+user_new.age);

//Akses dengan bracket
console.log("user_new-country: "+user_new.[‘country’]);

// Mengakses menggunakan object destructuring
const { name, country, isMale = true } = user_new;

console.log("user_new-name: "+name);
console.log("user_new-country: "+country);
```

**[Keluaran]**

```
user_new: {"name":"ifandika","age":19,"country":"Indonesia"}
user_new-name: ifandika
user_new-age: 19
user_new-country: Indonesia
user_new-name: ifandika
user_new-country: Indonesia
user_new-male: true
```

**Mengubah Nilai di Properti Object**

**[Kode]**

```
// Membuat Object
const user_new = { name: "ifandika", age: 19, 'country': 'Indonesia' };

console.log("user_new: "+JSON.stringify(user_new));

// Ubah data object
```

```
user_new.name = 'Kipli';
console.log("user_new: "+JSON.stringify(user_new));
```

**[Keluaran]**
```
user_new: {"name":"ifandika","age":19,"country":"Indonesia"}
user_new: {"name":"Kipli","age":19,"country":"Indonesia"}
```

**Menghapus Properti di Object**

**[Kode]**
```
// Membuat Object
const user_new = { name: "ifandika", age: 19, 'country': 'Indonesia' };

console.log("user_new: "+JSON.stringify(user_new));

// Hapus data object
delete user_new.name;
delete user_new['country'];
console.log("user_new: "+JSON.stringify(user_new));
```

**[Keluaran]**
```
user_new: {"name":"ifandika","age":19,"country":"Indonesia"}
user_new: {"age":19}
```

***Catatan**
Perlu dicatat bahwa mengakses properti yang tidak ada di dalam object akan menyebabkan error dan nilai kembaliannya adalah undefined yang mana hal ini jika tidak ditangani akan mengganggu program yang dibuat.

**[Kode]**
```
const user = { name: 'Dicoding' };
console.log(user.age);
```

**[Keluaran]**
```
undefined
```

## |Bermain dengan Struktur Data
Menggunakan struktur data pada pengkodean JavaScript akan membuat kode lebih mudah digunakan dan dibaca, beberapa struktur data seperi Array, Stack, dsb.

**Array**
Array adalah struktur data spesial yang dapat menyimpan kumpulan data yang terurut. Letak perbedaan array dengan object adalah data yang disimpan di dalam array terurut, sedangkan di object tidak terurut. Di array, kita bisa menambahkan nilai di antara data yang sudah ada. Data yang ada di array dapat diakses menggunakan pola indeks. Nilai yang disimpan di dalam array disebut dengan element.

**[Kode]**
```
// Struktur Data Array
```

```
const array = [1, 2, 3];
console.log('typeof array: '+typeof array);

// Menggunakan array literal
const fruits = ['apple', 'banana', 'cherry'];
console.log('fruits:', fruits);

// Membuat array
const student = new Array();
const student5 = new Array(5);
var student10 = new Array(10);
let student20 = new Array(20);

// Memasukan data
student.push('Joni');
console.log('student:',student);

// dengan Array.form()
const teacher = Array.from('teacher');
console.log('teacher:', teacher);

const users = new Array('John', 'Jane', 'Jack', 'Jill');
const customer = Array.from(users);
console.log('customer:', customer);

// Mengakses Element Array
console.log('akses user ke-0:', users[0]);

// Manipulasi Nilai Array
users[0] = 'Kipli';
console.log('akses user telah diubah ke-0:', users[0]);

// menghapus data pada array
delete users[0];
console.log('users:', users);

// hapus elemen dengan splice()
users.splice(0, 1);
console.log('users:', users);

// Terakhir, ada cara lainnya yaitu menggunakan method shift dan pop. Kekurangan
// dari kedua method ini adalah tidak sefleksibel delete dan splice karena shift
// hanya menghapus element pertama dan pop menghapus element terakhir.

users.shift();
console.log('after shift users:', users);

users.pop();
console.log('after pop users:', users);

// Array Destructuring
```

```
const introduction = ['Hello', 'Arsy'];
const [greeting, name] = introduction;
console.log('greeting, name:', greeting, name);

// Fungsi yang tersedia

// Reverse
const myArray = ['Android', 'Data Science', 'Web'];
console.log('myArray before:', myArray);
myArray.reverse();
console.log('myArray after:', myArray);

// Sort
const myArray2 = ['Web', 'Data Science', 'Android'];
console.log('myArray2 before:', myArray2);
myArray2.sort();
console.log('myArray2 after:', myArray2);
```

**[Keluaran]**

```
typeof array: object
fruits: [ 'apple', 'banana', 'cherry' ]
student: [ 'Joni' ]
teacher: [
  't', 'e', 'a',
  'c', 'h', 'e',
  'r'
]
customer: [ 'John', 'Jane', 'Jack', 'Jill' ]
akses user ke-0: John
akses user telah diubah ke-0: Kipli
users: [ <1 empty item>, 'Jane', 'Jack', 'Jill' ]
users: [ 'Jane', 'Jack', 'Jill' ]
after shift users: [ 'Jack', 'Jill' ]
after pop users: [ 'Jack' ]
greeting, name: Hello Arsy
myArray before: [ 'Android', 'Data Science', 'Web' ]
myArray after: [ 'Web', 'Data Science', 'Android' ]
myArray2 before: [ 'Web', 'Data Science', 'Android' ]
myArray2 after: [ 'Android', 'Data Science', 'Web' ]
```

## |Operator Spread

Spread, memiliki arti sesuai dengan namanya, yaitu menyebarkan. Spread operator digunakan untuk menyebarkan nilai yang ada pada object dan array. Spread operator yang ditandai dengan sintaks tiga titik (...) adalah fitur yang menarik dan membantu dalam pengelolaan object dan array. Dengan menggunakan spread operator, nilai object dan array dapat di-iterable menjadi beberapa element.

**[Kode]**

```
// Operator Spread
```

```
const obj1 = { name: 'Dicoding' };
const obj2 = { lastName: 'Indonesia', address: 'Jl. Batik Kumeli No 50' };
const newObj = { ...obj1, ...obj2 };

console.log('newObj:', newObj);

// 1 object
const originalObj = { name: 'Dicoding', age: 9 };
const copiedObj = { ...originalObj };

console.log('copiedObj:', copiedObj);

// Array
const array1 = ['Dicoding'];
const array2 = ['Indonesia', 'Jl. Batik Kumeli No 50'];
const newArray = [...array1, ...array2];

console.log('newArray:', newArray);
```

**[Keluaran]**

```
newObj: {
  name: 'Dicoding',
  lastName: 'Indonesia',
  address: 'Jl. Batik Kumeli No 50'
}
copiedObj: { name: 'Dicoding', age: 9 }
newArray: [ 'Dicoding', 'Indonesia', 'Jl. Batik Kumeli No 50' ]
```

## |Operator Rest

Ketika bekerja dengan function, sering kali function menerima argument yang kemudian menjadi parameter. Ketika argument-nya masih sedikit, seperti satu atau dua belum menimbulkan masalah. Masalah terjadi ketika argument-nya sudah melebihi dua karena terlalu banyak argument pada function membuat kode menjadi tidak bersih (tidak sesuai prinsip clean code) sehingga sulit untuk dibaca dan di-maintenance.

Solusinya adalah menggunakan rest operator. Rest operator memungkinkan function untuk menerima argument dalam bentuk array. Rest operator yang digunakan pada parameter fungsi sering disebut sebagai Rest Parameter. Cara menggunakan rest parameter adalah dengan menambahkan tiga titik (...) sebelum parameter terakhir.

**[Kode]**

```
// Operator Rest

function myFunc(...manyMoreArgs) {
  console.log('manyMoreArgs', manyMoreArgs);
}

myFunc('one', 'two', 'three');

function myFunc2(number, ...manyMoreArgs) {
  console.log('number', number);
```

```
  console.log('manyMoreArgs', manyMoreArgs);
}

myFunc2('one', 'two', 'three');

function myFunc3(...manyMoreArgs) {
  console.log(manyMoreArgs.length);
  console.log('manyMoreArgs', manyMoreArgs);
}

myFunc3('one', 'two', 'three');

const favorites = ['Nasi Goreng', 'Mie Goreng', 'Ayam Bakar', 'Tahu', 'Tempe'];
const [first, second, ...rest] = favorites;

console.log(first);
console.log(second);
console.log(rest);
```

**[Keluaran]**

```
manyMoreArgs [ 'one', 'two', 'three' ]
number one
manyMoreArgs [ 'two', 'three' ]
3
manyMoreArgs [ 'one', 'two', 'three' ]
Nasi Goreng
Mie Goreng
[ 'Ayam Bakar', 'Tahu', 'Tempe' ]
```

## | Error Handling

Bagaimana cara mengatasi eror pada sebuah program yang ada, memberikan pengecualian pada sebuah kondisi proram yang mengalami kesalahan, memberikan informasi tentang kesalahan tersebut.

**Throwing error**

Melempar eror/informasi eror tersebut.

**[Kode]**

```
// Error handling

// Throw error
const simple_error = new Error("Terjadi eror");
console.error(simple_error);
```

**[Keluaran]**

```
Error: Terjadi eror
    at Object.<anonymous> (D:\Coding\JavaScript\error_handling.js:4:22)
    at Module._compile (node:internal/modules/cjs/loader:1358:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1416:10)
    at Module.load (node:internal/modules/cjs/loader:1208:32)
    at Module._load (node:internal/modules/cjs/loader:1024:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:174:12)
    at node:internal/main/run_main_module:28:49
```

Pada contoh di atas, kita menggunakan built-in constructor milik JavaScript, Error. Kenapa kita perlu membangkitkan exception secara sengaja? Jawabannya adalah karena kita ingin program yang dibangun tidak mengalami crash ketika terjadi sesuatu di luar dugaan.

Misalnya, kita memiliki program yang menerima inputan pembayaran dari pembeli. Normalnya adalah jumlah yang dibayarkan harus lebih besar dari harga barang. Lalu, ada sebuah kasus dimana pembeli membayar lebih kecil dari harga barang. Hal ini akan menyebabkan error di program milik kita. Oleh karena itu, kita perlu throw error ketika pembayaran kurang dari harga barang seperti contoh berikut.

**[Kode]**
```
// Error handling

const price = 100;
const paid = 80;

if (paid < price) {
  throw new Error('Pembayaran kurang');
}
```

**[Keluaran]**
```
D:\Coding\JavaScript\error_handling.js:11
  throw new Error('Pembayaran kurang');
  ^

Error: Pembayaran kurang
    at Object.<anonymous> (D:\Coding\JavaScript\error_handling.js:11:9)
    at Module._compile (node:internal/modules/cjs/loader:1358:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1416:10)
    at Module.load (node:internal/modules/cjs/loader:1208:32)
    at Module._load (node:internal/modules/cjs/loader:1024:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:174:12)
    at node:internal/main/run_main_module:28:49

Node.js v20.14.0
```

**Catching Error**
Sebelumnya, Anda sudah tahu cara untuk membangkitkan error. Kini, saatnya Anda mengetahui cara untuk menangkap error yang dihasilkan oleh program JavaScript yang Anda tulis.

**Try-Catch**
Try-catch merupakan cara yang dimiliki JavaScript untuk menangani error. Try-catch memiliki dua blok utama yaitu try dan catch.

**[Kode]**
```
try {
  console.log('Memulai program');
  console.log('Mengakhiri program');
}
catch (err) {
  console.log('Karena tidak ada error, blok ini akan diabaikan');
}
```

**[Keluaran]**
```
Memulai program
Mengakhiri program
```

Jika ingin error di eksekusi maka berikan error.

**[Kode]**
```
try {
  console.log('Memulai program');
  throw new Error('Error: Program berhenti');
  console.log('Mengakhiri program');
}
catch (err) {
  console.log('Karena ada error, blok ini akan dieksekusi');
}
```

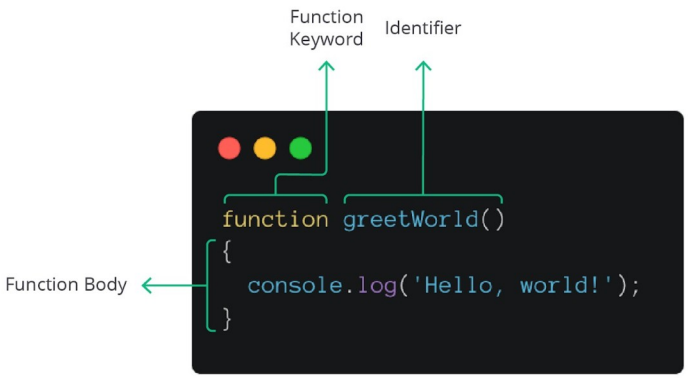
**[Keluaran]**
Memulai program
Karena ada error, blok ini akan dieksekusi

Untuk finally block yang pasti akan dieksekusi.

**[Kode]**
```
try {
  console.log('Ini try block');
  throw new Error('Error: Program berhenti');
}
catch (err) {
  console.log('Ini catch block');
}
finally {
  console.log('Ini finally block');
}
```

**[Keluaran]**
Ini try block
Ini catch block
Ini finally block

# | Fungsi
Membuat sebuah kode blok yang dapat digunakan berulang2, itulah fungsi. Dengan awalan function.




**[Kode]**

```
function convertCelsiusToFahrenheit(temperature) {
  const temperatureInFahrenheit = 9 / 5 * temperature + 32;
  console.log('Hasil konversi:', temperatureInFahrenheit);
}
console.log(convertCelsiusToFahrenheit);
convertCelsiusToFahrenheit(10);
```

**[Keluaran]**

```
[Function: convertCelsiusToFahrenheit]
Hasil konversi: 50
```

Ada hal yang perlu kita ketahui. Ia adalah fitur hoisting dalam JavaScript. Fitur ini memungkinkan kita menulis kode pemanggilan sebelum kode pendeklarasian function. Berikut contohnya.

**[Kode]**

```
// Memanggil terlebih dahulu
greetWorld();

function greetWorld() {
  console.log('Hello, world!');
}
```

**[Keluaran]**

```
Hello, world!
```

**Default/Bawaan Parameter**

Tahukah Anda bahwa argument dapat bernilai undefined jika kita tidak beri nilai apa pun dalam parentheses saat function dipanggil? Lalu, apa yang akan terjadi jika function body tetap dijalankan dalam keadaan seperti itu?

**[Kode]**

```
function convertCelsiusToFahrenheit(temperature) {
  const temperatureInFahrenheit = 9 / 5 * temperature + 32;
  console.log('Hasil konversi:', temperatureInFahrenheit);
}
convertCelsiusToFahrenheit();
```

**[Keluaran]**

```
Hasil konversi: NaN
```

Untuk mengatasi hal tersebut bisa dengan kondisi cek apakah undefined.

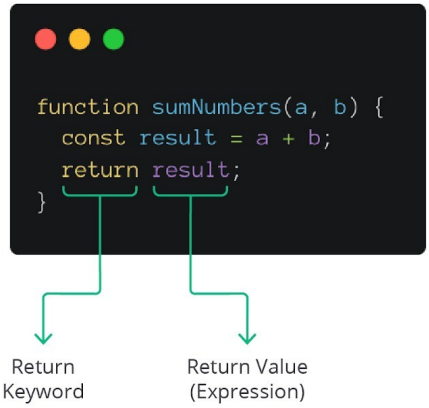
**[Kode]**

```
function convertCelsiusToFahrenheit(temperature) {
  if (temperature !== undefined) {
    const temperatureInFahrenheit = (9 / 5 * temperature + 32);
    console.log('Hasil konversi:', temperatureInFahrenheit);
  }
}
```

Atau dengan memberikan nilai bawaan/default.

**[Kode]**
```
function convertCelsiusToFahrenheit(temperature = 10) {
  const temperatureInFahrenheit = (9 / 5 * temperature + 32);
  console.log('Hasil konversi:', temperatureInFahrenheit);
}
```

**Nilai Kembalian/Return Value**
Untuk memberikan kemampuan function mengembalikan nilai (return statement), kita gunakan kata kunci return dan diikuti nilai kembaliannya. Perhatikan notasinya pada gambar berikut.




Untuk contoh sebagai berikut.

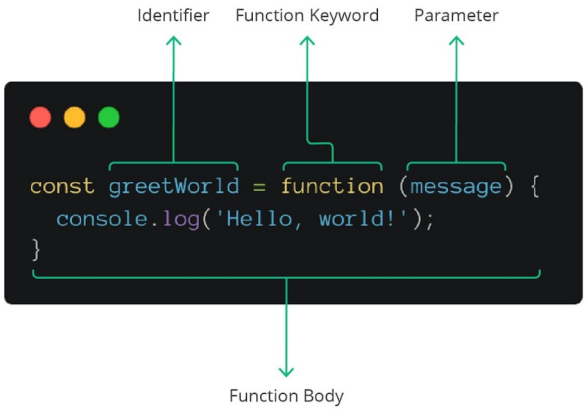
**[Kode]**
```
function sumNumbers(a, b) {
  const result = a + b;
  return result;
}
const result = sumNumbers(2, 4);
console.log('2 + 4:', result);
```

**[Keluaran]**
```
2 + 4: 6
```

# Function Expression

Function expression terdiri dari dua kata, yaitu "function" dan "expression". Kita sudah mengenal function, begitu juga dengan expression. Lantas, function expression dapat kita artikan sebagai kode yang mengembalikan nilai function. Bagaimana bentuknya, ya?

Kali ini, kita sudah mahir membuat function. Tata caranya adalah keyword function, identifier, parentheses untuk parameter, dan function body. Disadari ataupun tidak, sebetulnya kita sedang membuat, sebut saja, function statement. Artinya, kita memerintahkan JavaScript membuat function dengan statement tersebut. Tentunya, ini tidak akan mengembalikan nilai apa pun. Nah, kita dapat membuat function dengan gaya expression layaknya membuat variabel seperti kode di bawah ini.




Untuk contoh sebagai berikut

**[Kode]**

```
const convertCelsiusToFahrenheit = function (temperature) {
  const result = (9 / 5) * temperature + 32;
  return result;
};
const temperatureInFahrenheit = convertCelsiusToFahrenheit(90);
console.log('Hasil konversi:', temperatureInFahrenheit);
```

**[Keluaran]**

```
Hasil konversi: 194
```

***Catatan**

Perbedaan lainnya dari function expression ialah tidak memiliki hoisting padanya sehingga kita tidak dapat memanggil atau menjalankan function ini sebelum dideklarasikan.

**Menjadi First-Class Citizen**

Jika ada bahasa pemrograman yang mengatakan bahwa function dapat diperlakukan layaknya variabel, function tersebut dinyatakan sebagai first-class citizen. Apa maksud dari diperlakukan mirip variabel?

Pada JavaScript, function dapat kita jadikan sebagai nilai dan disimpan dalam variabel, nilai argumen function lain, mengembalikan nilai function dari suatu function, dsb. Mari kita lihat contohnya.

**[Kode]**

```
function multiply(a, b) {
  return a * b;
}
function calculate(operation, numA, numB) {
  return operation(numA, numB);
}
const result = calculate(multiply, 2, 4);
console.log(result);
```

**[Keluaran]**

```
8
```

# Function Expression

Function expression benar-benar bisa mengubah cara kita membuat function. Namun, ada alternatif sintaksis lain dalam JavaScript yang bisa lebih mengubah cara kita membuat function. Sintaksis tersebut bernama arrow function. Yap, ini cara baru yang bisa kita manfaatkan dalam membuat function.

Deklarasi arrow function.

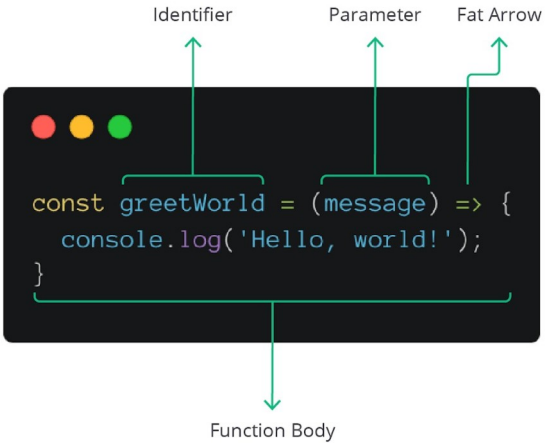
**[kode]**

```
// Arrow Function
let temperatureInFahrenheit = null;

// Deklarasi function dengan Regular Function
const convertCelsiusToFahrenheitUsingRegularFunction = function (temperature) {
  const result = (9 / 5) * temperature + 32;
  return result;
};

temperatureInFahrenheit = convertCelsiusToFahrenheitUsingRegularFunction(90);
console.log('Hasil konversi:', temperatureInFahrenheit);

// Deklarasi Function dengan Arrow Function
const convertCelsiusToFahrenheitUsingArrowFunction = (temperature) => {
  const result = (9 / 5) * temperature + 32;
  return result;
};

temperatureInFahrenheit = convertCelsiusToFahrenheitUsingArrowFunction(90);
console.log('Hasil konversi:', temperatureInFahrenheit);
```

**[Keluaran]**

```
Hasil konversi: 194
Hasil konversi: 194
```

Hanya mengganti kata function dengan kurung dan didalam adalah nilai parameter, untuk lebih lengkap dari notasi arrow fuction.




**Refactor**

Apakah Anda merasa arrow function belum dikatakan sebagai sintaksis function yang sederhana? Tenang, ternyata arrow function ini bisa lebih simpel lagi!

**[Kode]**

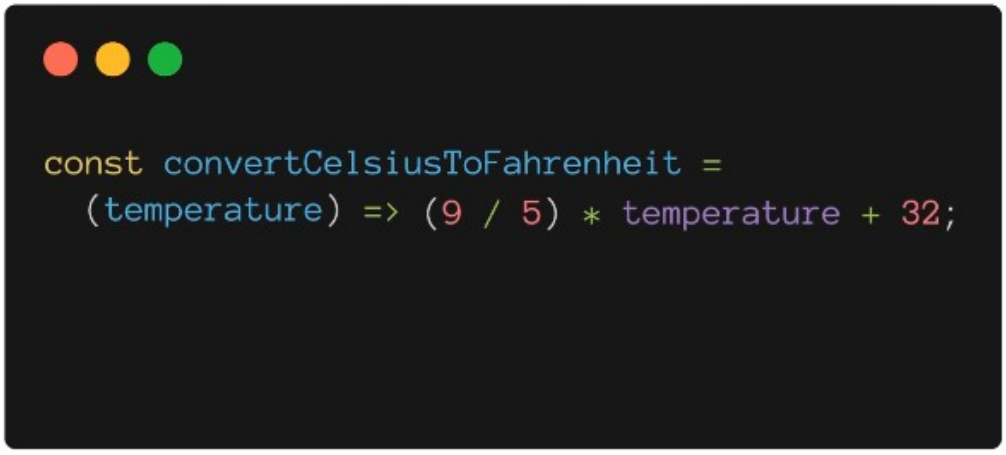
```
// Refactor
let temperatureInFahrenheit;

// Arrow function
const convertCelsiusToFahrenheit = (temperature) => {
```

```
  const result = (9 / 5) * temperature + 32;
  return result;
};

temperatureInFahrenheit = convertCelsiusToFahrenheit(90);
console.log('Hasil konversi:', temperatureInFahrenheit);

// Arrow function versi ringkas
const convertCelsiusToFahrenheitInConciseSyntax =
  (temperature) => (9 / 5) * temperature + 32;

temperatureInFahrenheit = convertCelsiusToFahrenheitInConciseSyntax(90);
console.log('Hasil konversi:', temperatureInFahrenheit);
```

**[Keluaran]**

```
Hasil konversi: 194
Hasil konversi: 194
```

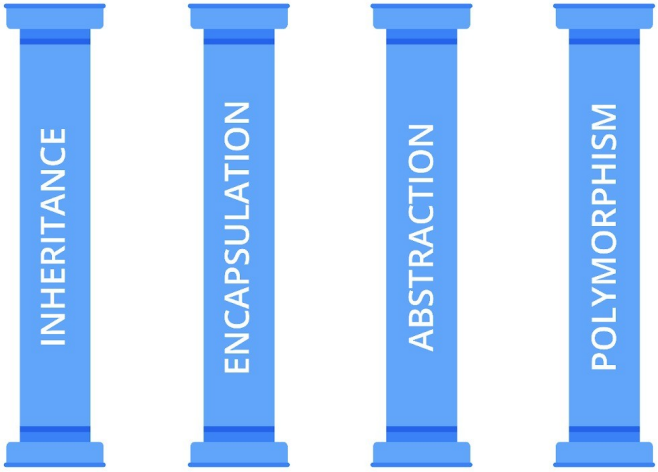
Untuk notasi lebih mudah dengan gambar.



# OOP (Object Oriented Programming)

Sesuai dengan namanya, di paradigma ini kita akan banyak berhubungan dengan objek. Object-oriented programming (OOP) adalah paradigma pemrograman yang memiliki pendekatan berbasis object. Object akan berinteraksi satu sama lain untuk menyelesaikan tugas sehingga membentuk keseluruhan program. Selain itu, object merupakan representasi dari entitas.

Object terdiri dari atribut informasi (property) dan perilaku (method). Property adalah informasi tentang objek tersebut seperti nama, warna, dan jenis, sedangkan method adalah aksi atau perilaku yang dapat dilakukan oleh objek seperti berjalan, berlari, dan terbang. Kita ambil contoh pada kehidupan sehari-hari, misalnya entitas kucing direpresentasikan menjadi object kucing dengan memiliki properti dan atribut. Property pada kucing adalah warna, jenis ras, nama, dan umur. Method pada kucing adalah berlari, tidur, makan, dan mencakar.




OOP sangat cocok digunakan pada program yang kompleks karena dapat mengelompokkan kode menjadi object dan class. Selain itu, kode yang kompleks akan menjadi lebih bersih dan ringkas karena bisa digunakan kembali melalui konsep inheritance (class dan inheritance akan dibahas lebih detail di materi berikutnya). OOP memiliki empat pilar yang membentuknya yaitu encapsulation, inheritance, polymorphism, dan abstraction.

## Inheritance

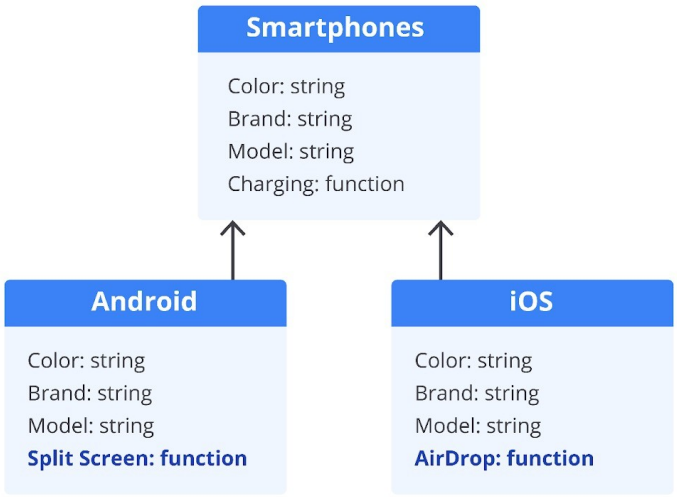
Pilar yang akan kita bahas pertama adalah inheritance. Inheritance jika diterjemahkan ke dalam bahasa Indonesia artinya adalah pewarisan. Sesuai dengan namanya, kita bisa mewariskan harta property dan method dari sebuah class ke class lain. Umumnya, properti dan method yang diwariskan berasal dari class (induk) dan digunakan oleh class baru (anak). Sama halnya di kehidupan sehari-hari, sedikit banyaknya sebagai anak, kita memperoleh sifat dan perilaku dari orang tua.

Di OOP, inheritance memungkinkan class untuk mewarisi property dan method yang dimilikinya sehingga membantu mengurangi penulisan kode secara berulang (mengurangi redundancy kode). Misalnya, ketika kita membuat sebuah class dengan property dan method, keduanya dapat digunakan kembali oleh class lainnya melalui inheritance. Berikut adalah contohnya.

**[Kode]**

```
class SuperClass { }
class SubClass extends SuperClass { }
```

Misalnya, Anda memiliki smartphones dengan jenis Android dan iOS, setiap smartphones tersebut pasti memiliki property color, brand, model, dan method-nya adalah charging. Dengan paradigma OOP, property dan method yang memiliki kesamaan bisa kita abstraksikan menjadi sebuah class baru bernama Smartphones. Kemudian kita bisa membuat dua class baru, yaitu Android dan iOS.




Android dan iOS akan mewariskan property dan method dari class Smartphones seperti yang ada pada gambar di atas. Dengan begitu, class Android dan iOS akan memiliki property color, brand, model dan method charging. Selain itu, di masing-masing class kita dapat menambahkan property yang hanya ada pada dirinya. Misalkan, class Android memungkinkan untuk memiliki method split screen, sedangkan class iOS memungkinkan untuk memiliki method AirDrop.


Jika contoh di atas kita terapkan pada kode JavaScript, kodenya akan menjadi seperti berikut ini.

**[Kode]**

```
// OOP Inheritance
class SmartPhones {
  constructor(color, brand, model) {
    this.color = color;
    this.brand = brand;
    this.model = model;
  }

  charging() {
    console.log(`Charging ${this.model}`);
  }
}

class iOS extends SmartPhones {
```

```
  airDrop() {
    console.log('iOS have a behavior AirDrop');
  }
}

class Android extends SmartPhones {
  splitScreen() {
    console.log('Android have a Split Screen');
  }
}

const ios = new iOS('black', 'A', '12 Pro Max');
const android = new Android('white', 'B', 'Galaxy S21');

ios.charging();
ios.airDrop();

android.charging();
android.splitScreen();
```

**[Keluaran]**

```
Charging 12 Pro Max
iOS have a behavior AirDrop
Charging Galaxy S21
Android have a Split Screen
```

## Encapsulation

Setelah kelar membahas pilar OOP pewarisan, berikutnya kita akan membahas pilar utama berikutnya dari OOP yaitu encapsulation. Encapsulation adalah proses untuk membungkus data di suatu wadah yang disebut dengan class. Menyembunyikan data adalah bagian kunci dari encapsulation.

Desain OOP yang baik adalah object hanya akan menampilkan data yang dibutuhkan oleh object lain. Data akan diisolasi dan tidak dapat diakses langsung dari luar. Secara sederhana, encapsulation adalah membuat data yang ada di class sebagai private

Contoh dengan kasus mesin kopi, dimana user mengubah nilai temperature secara manual.

**[Kode]**

```
// OOP Encapsulation
class CoffeeMachine {
  constructor(water) {
    this.waer = water;
    this.temperature = 90;
  }

  makeCoffee() {
    console.log("Make coffe with temperature:", this.temperature);
  }
}

const coffee = new CoffeeMachine();
// Ubah properti
coffee.temperature = 20;

coffee.makeCoffee();
```

**[Keluaran]**

Make coffe with temperature: 20

Getter terdiri dari method get. get adalah cara untuk mendapatkan nilai dari property, sedangkan setter terdiri dari method set. set adalah method untuk menetapkan nilai property. Dengan begitu, kita dapat mengatur akses ke property yang dimiliki oleh object. Perhatikan contoh berikut ini.

**[Kode]**

```
// OOP Encapsulation
class CoffeeMachine {
  constructor(waterAmount) {
    this.waterAmount = waterAmount;
    this._temperature = 90
  }

  set temperature(temperature) {
    console.log('you are not allowed to change the temperature');
  }

  get temperature() {
    return this._temperature;
  }
}

const coffee = new CoffeeMachine(10);
console.log('Sebelum diubah: ', coffee.temperature);
coffee.temperature = 20;
// akan mengambil get temperature() {}
console.log('Setelah diubah: ', coffee.temperature);
```

**[Keluaran]**

```
Sebelum diubah:  90
you are not allowed to change the temperature
Setelah diubah:  90
```

Penambahan underscore (_) di variable temperature untuk menandakan bahwa nilai temperature tidak dapat diubah. Namun, sebenarnya penggunaan tanda underscore tidak benar-benar membuat property temperature tidak dapat diubah, ia masih dapat diubah. Penggunaan underscore hanyalah code convention yang disepakati oleh komunitas JavaScript.

Untuk membuat nilainya benar-benar tidak dapat diubah, Anda dapat melakukannya dengan cara berikut.

**[Kode]**

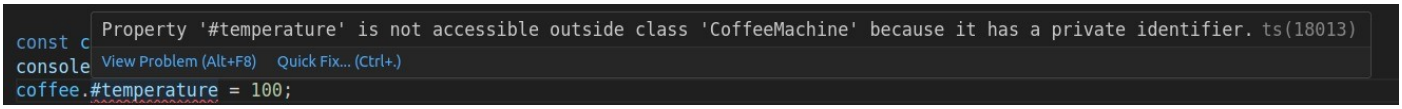
```
// OOP Encapsulation
class CoffeeMachine3 {
  #temperature = 90;

  constructor(waterAmount) {
    this.waterAmount = waterAmount;
    this.#temperature = this.#defaultTemperature();
  }

  set temperature(temperature) {
    console.log('you are not allowed to change the temperature');
  }

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

  #defaultTemperature() {
    return 90;
  }
}

const coffee3 = new CoffeeMachine3(10);
```

```
console.log('Sebelum diubah: ', coffee3.temperature);
coffee3.temperature = 20;
// akan mengambil get temperature() {}
console.log('Setelah diubah: ', coffee3.temperature);
```

**[Keluaran]**

```
Sebelum diubah:  90
you are not allowed to change the temperature
Setelah diubah:  90
```

Sejak JavaScript versi ES2022, kita dapat menggunakan tanda hashtag (#) untuk membuat hak akses private pada property dan method. Pada contoh di atas, kita menambahkan tanda hashtag di variable dan method yang bersifat private. Jika mencoba mengakses property yang bersifat private, Anda akan mendapatkan pesan error seperti berikut ini.

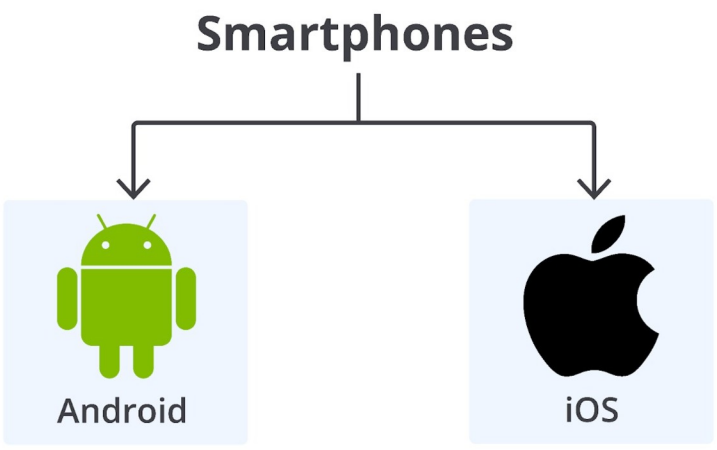



## Polymorphism

Seperti yang Anda ketahui sebelumnya bahwa kita dapat mewariskan property dan method ke class lainnya. Namun, apa yang terjadi jika SubClass ingin mengubah implementasi dari method yang diwariskan dari SuperClass? Layaknya kita sebagai anak, ingin mengubah suatu sifat atau perilaku dari orang tua yang kita mungkin tidak setuju atau butuhkan. Jangan khawatir, di OOP kita dapat mengubah implementasi method yang diturunkan dari SuperClass.

Bagaimana cara untuk mengubah implementasi yang diturunkan dari SuperClass? Caranya adalah menggunakan pilar utama lainnya dari OOP yaitu Polymorphism. Polymorphism berasal dari bahasa Yunani yang memiliki arti secara harfiah yaitu memiliki banyak bentuk. Polymorphism merupakan konsep di mana suatu entitas menjadi SuperClass untuk mewariskan property atau method ke SubClass.

Polymorphism berhubungan erat dengan pewarisan. Sebelumnya kita memiliki SuperClass Smartphones yang memiliki property color, brand, model dan method charging. Kemudian kita memiliki SubClass yang implementasinya berbeda tergantung dengan jenisnya seperti Android dan iOS.



Kini, bentuk implementasi dari Smartphones berbeda untuk setiap jenis. Inilah yang disebut dengan polymorphism. Lalu, bedanya apa dong dengan pewarisan? Bedanya terdapat pada implementasi method yang diubah. Untuk mengubah implementasi method tersebut, terdapat konsep yang disebut dengan overriding.

**Overriding**
OOP memiliki konsep overriding yang sangat erat kaitannya dengan pewarisan. Overriding adalah cara kita untuk membuat implementasi yang berbeda di SubClass untuk method yang diturunkan dari SuperClass. Overriding dapat diterapkan untuk membuat method yang lebih spesifik di SubClass. Selain itu, overriding juga dapat diterapkan untuk menambah properti baru di SubClass. Overriding dapat diterapkan pada constructor maupun pada method.

Contoh kita melakukan overriding constructor, dimana pada kelas Android menambah parameter devices.

**[Kode]**
```
// OOP Polymorphism

class SmartPhones {
  constructor(color, brand, model) {
    this.color = color;
    this.brand = brand;
    this.model = model;
  }

  charging() {
    console.log(`Charging ${this.model}`);
  }
}

class Android extends SmartPhones {
  constructor(color, brand, model, device) {
    super(color, brand, model);
    this.device = device;
  }

  splitScreen() {
    console.log('Android have a Split Screen');
  }
}

const android = new Android('white', 'B', 'Galaxy S21', 'smart TV');
android.charging();
```

**[Keluaran]**
```
Charging Galaxy S21
```

Lalu untuk overriding fungsi/method sebagai berikut.

**[Kode]**
```
// OOP Polymorphism

class SmartPhones {
  constructor(color, brand, model) {
    this.color = color;
    this.brand = brand;
    this.model = model;
  }

  charging() {
    console.log(`Charging ${this.model}`);
  }
}

class Android extends SmartPhones {
  constructor(color, brand, model, device) {
```

```
    super(color, brand, model);
    this.device = device;
  }

  charging() {
    console.log(`Charging ${this.model} with fast charger`);
  }

  splitScreen() {
    console.log('Android have a Split Screen');
  }
}

const android = new Android('white', 'B', 'Galaxy S21', 'smart TV');
android.charging();
```

**[Keluaran]**

```
Charging Galaxy S21 with fast charger
```

## Object Composition

Sebelumnya, Anda sudah menguasai pilar pewarisan di OOP, bahkan Anda sudah menguasai konsep overriding. Pewarisan memungkinkan kita untuk mengurangi menulis kode secara berulang (tidak efektif). Namun, apakah pewarisan mampu untuk memecahkan masalah kode yang kompleks? Apakah pewarisan hanya mampu untuk kasus sesederhana sebelumnya? Yuk, kita eksplorasi bersama.

Misalnya, Anda sedang mengembangkan sebuah video game. Video game tersebut memiliki banyak karakter seperti monster, wizard dan guardian. Setiap karakter memiliki kemampuan yang sama yaitu bergerak. Selain itu, setiap karakter memiliki kemampuan yang unik pada dirinya seperti menyerang, bertahan, dan mengeluarkan sihir. Jika skenario video game ini kita gambarkan dengan konsep OOP, karakter akan menjadi SuperClass, sedangkan monster, wizard, dan guardian akan menjadi SubClass seperti contoh berikut ini.

**[Kode]**

```
class Character {
  constructor(name, health, position) {
    this.name = name;
    this.health = health;
    this.position = position;
  }

  canMove() {
    console.log(`${this.name} moves to ${this.position}!`);
  }
```

```
}


class Monster extends Character {
  canAttack() {
    console.log(`${this.name} attacks with a weapon!`);
  }
}

class Guardian extends Character {
  canDefend() {
    console.log(`${this.name} defends with a shield!`);
  }
}

class Wizard extends Character {
  canCastSpell() {
    console.log(`${this.name} casts a magic spell!`);
  }
}
```

Oke, tidak ada yang salah dengan implementasi kode tersebut kan? Nah, timbul masalah ketika kita ingin menambahkan satu karakter lagi, misalnya karakter warrior. Warrior adalah karakter yang memiliki kekuatan super, ia bisa menyerang, bertahan, dan bergerak

dengan melakukan pewarisan dari SuperClass Character. Yup, hal itu benar karena memang itulah satu-satunya cara.

**[Kode]**

```
class Warrior extends Character {
  canAttack() {
    console.log(`${this.name} attacks with a weapon!`);
  }

  canDefend() {
    console.log(`${this.name} defends with a shield}!`);
  }
}
```

Namun, cara tersebut tidak efektif karena ketika kita mengubah implementasi salah satu method, kita perlu untuk mengubahnya di dua tempat. Katakanlah, kita mengubah method canAttack(), kita perlu untuk mengubahnya di SubClass Monster dan Warrior. Lantas, apa solusinya? Solusinya adalah mengubah pewarisan menjadi object composition.

Object composition dapat menjadi solusi untuk masalah pewarisan yang kompleks seperti di kasus polymorphism. Jika sebelumnya, pewarisan menggunakan pendekatan peran atau identitas dalam menstrukturkan kode, yakni Monster, Wizard, Warrior, dan Guardian. Ketika menggunakan object composition, pendekatan yang digunakan adalah berbasis kemampuan, bukanlah peran atau identitas.

Kode distrukturkan berdasarkan kemampuan, apakah ia bisa menyerang, bertahan atau mengeluarkan sihir seperti berikut ini.

**[Kode]**
```
function canAttack(character) {
  return {
    attack: () => {
      console.log(`${character} attacks with a weapon!`);
    }
  }
}

function canDefend(character) {
  return {
    defend: () => {
      console.log(`${character} defends with a shield!`);
    }
  }
}

function canCastSpell(character) {
  return {
    castSpell: () => {
      console.log(`${character} casts a spell!`);
    }
  }
}
```

Karena struktur kode sudah dipecah berdasarkan kemampuan, bukan peran atau identitas, ke depannya ketika ada karakter baru yang memiliki kombinasi kemampuan, akan lebih mudah untuk membuatnya. Untuk membuat object, kita dapat membuat function sebagai object creator dan mengomposisikan kemampuan-kemampuan tersebut.

Di JavaScript, kita dapat mengomposisikan objek secara mudah dengan menggunakan method Object.assign(). Object.assign() adalah method statis untuk menyalin semua properti dari satu atau lebih object ke objek target. Object.assign() akan mengembalikan objek target yang dimodifikasi.

**[Kode]**
```
function createMonster(name) {
  const character = new Character(name, 100, 0);
  return Object.assign(character, canAttack(name));
}

function createGuardian(name) {
  const character = new Character(name, 100, 0);
  return Object.assign(character, canDefend(name));
}

function createWizard(name) {
  const character = new Character(name, 100, 0);
  return Object.assign(character, canCastSpell(name));
}

function createWarrior(name) {
  const character = new Character(name, 100, 0);
  return Object.assign(character, canAttack(character), canDefend(character));
}
```

Lalu buat object Monster, Warrior, Wizard, Guardian-nya.

**[Kode]**
```
const monster = createMonster('Monster');
monster.canMove();
monster.attack();

const guardian = createGuardian('Guardian');
guardian.canMove();
guardian.defend();

const wizard = createWizard('Wizard');
wizard.canMove();
wizard.castSpell();

const warrior = createWarrior('Warrior');
warrior.canMove();
warrior.attack();
warrior.defend();
```

Maka kita membuat object dengan fungsi yang sudah disediakan untuk membuat object tersebut, untuk kode lengkapnya.

**[Kode]**
```
class Character {
  constructor(name, health, position) {
    this.name = name;
    this.health = health;
    this.position = position;
  }

  canMove() {
    console.log(`${this.name} moves to another position!`);
  }
}

function canAttack(character) {
  return {
    attack: () => {
      console.log(`${character.name} attacks with a weapon!`);
    }
  };
}

function canDefend(character) {
  return {
    defend: () => {
      console.log(`${character.name} defends with a shield!`);
    }
  };
}

function canCastSpell(character) {
  return {
    castSpell: () => {
      console.log(`${character.name} casts a spell!`);
    }
  };
}

function createMonster(name) {
  const character = new Character(name, 100, 0);
  return Object.assign(character, canAttack(character));
}

function createGuardian(name) {
  const character = new Character(name, 100, 0);
  return Object.assign(character, canDefend(character));
}

function createWizard(name) {
```

```
  const character = new Character(name, 100, 0);
  return Object.assign(character, canCastSpell(character));
}

function createWarrior(name) {
  const character = new Character(name, 100, 0);
  return Object.assign(character, canAttack(character), canDefend(character));
}

const monster = createMonster('Monster');
monster.canMove();
monster.attack();
console.log();

const guardian = createGuardian('Guardian');
guardian.canMove();
guardian.defend();
console.log();

const wizard = createWizard('Wizard');
wizard.canMove();
wizard.castSpell();
console.log();

const warrior = createWarrior('Warrior');
warrior.canMove();
warrior.attack();
warrior.defend();
console.log();
```

**[Keluaran]**

```
Monster moves to another position!
Monster attacks with a weapon!

Guardian moves to another position!
Guardian defends with a shield!

Wizard moves to another position!
Wizard casts a spell!

Warrior moves to another position!
Warrior attacks with a weapon!
Warrior defends with a shield!
```