**Declaring Function**

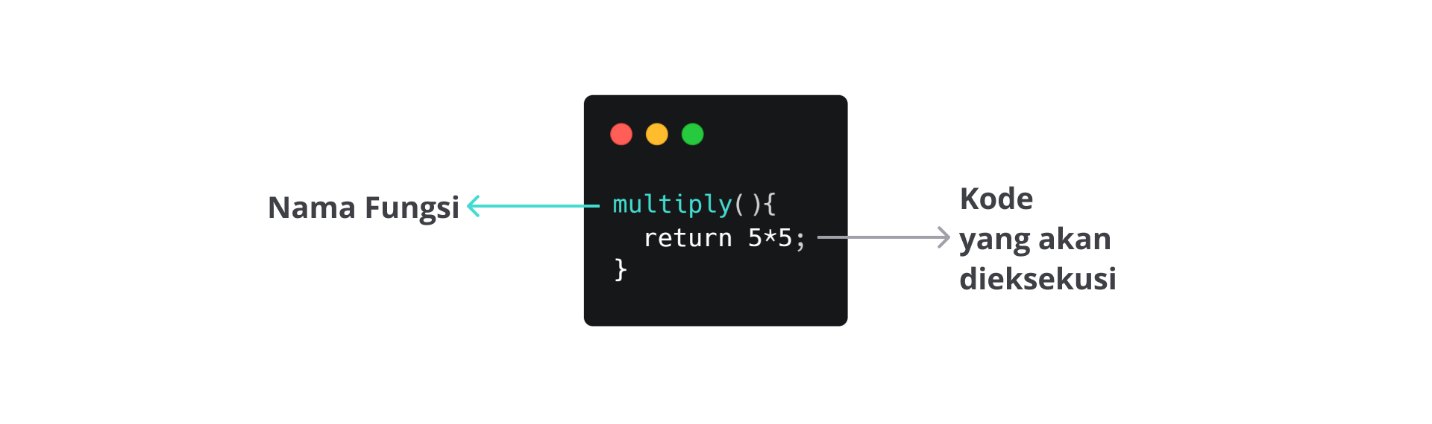
Fungsi merupakan bagian penting dalam bahasa pemrograman. Tanpa sadar, sebenarnya kita sudah menggunakan sebuah fungsi pada contoh kode yang ada sebelumnya. log() pada console.log() merupakan sebuah function yang berguna untuk menampilkan data pada konsol. Tapi sebenarnya apa itu *function*? Bagaimana ia bisa bekerja?

Mirip dengan fungsi pada matematika, fungsi dalam pemrograman juga digunakan untuk menghasilkan output berdasarkan input tertentu.

[](https://www.dicoding.com/academies/256/tutorials/13917?from=15327)

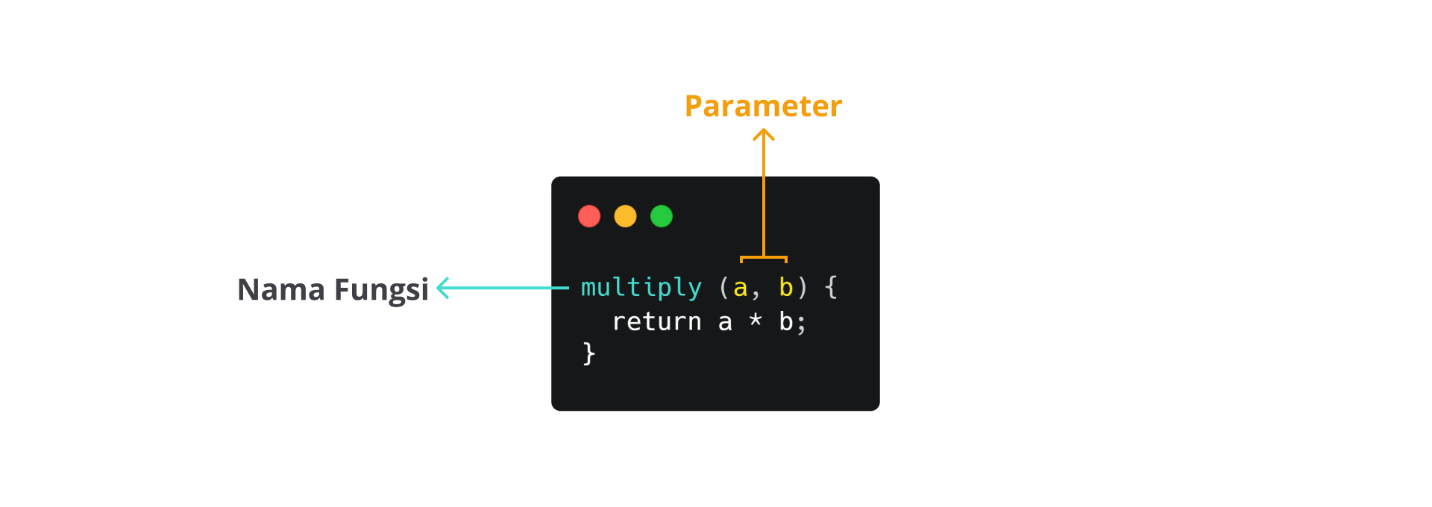
Namun, fungsi juga bisa digunakan sebagai blok kode atau prosedur yang dapat digunakan secara berulang. Dalam arti lain, kita dapat berpikir bahwa function merupakan sebuah variabel yang berisi blok logika. Blok logika tersebut akan dieksekusi ketika variabelnya dipanggil.

Semua fungsi memiliki struktur yang sama. Fungsi dideklarasikan dengan *keyword* function dan nama fungsinya. Nama fungsi selalu diikuti dengan tanda kurung (*parentheses*) tanpa spasi, lalu terdapat sepasang kurung kurawal yang berisi logika dari fungsi tersebut.

[](https://www.dicoding.com/academies/256/tutorials/13917?from=15327)

Terkadang di dalam tanda kurung kita membutuhkan sebuah informasi tambahan yang disebut dengan parameter. Parameter merupakan data yang digunakan pada fungsi untuk diproses di dalamnya. Sebagai contoh, fungsi console.log() dapat menerima *argument* berupa string atau data lain untuk ditampilkan ke konsol.

Berikut merupakan ilustrasi dari struktur fungsi dengan parameter:

[](https://www.dicoding.com/academies/256/tutorials/13917?from=15327)

**Parameter & Argument**

Di dalam fungsi kita akan banyak bertemu istilah *parameter* & *argument*. Penggunaan istilah ini sering kali tertukar, bahkan di kalangan developer.

 Perbedaan mendasar antara keduanya antara lain:

Parameter merupakan variabel yang didefinisikan sebagai inputan dari sebuah fungsi. Contoh:

function multiply(**a, b**) {

    return a \* b;

}

Argument merupakan nilai atau *expression* yang dimasukkan ke dalam fungsi. Contohnya:

multiply(**3, 4**);

Setelah membuat fungsi kita dapat memanggilnya dengan menuliskan nama fungsi diikuti tanda kurung dan memasukkan argumen di dalamnya (jika ada).

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13917?from=15327#run1-editor1)

function greeting() {

console.log("Good Morning!")

}

​

greeting();

​

/\* output

Good Morning!

\*/

Tetapi jika sebuah fungsi hanya menjalankan baris kode yang sama dirasa kurang fungsional, bukan? Kita dapat membuat fungsi tersebut untuk menerima dan memanfaatkan parameter untuk mengubah perilaku dari fungsinya.

Untuk menambahkan parameter pada fungsi, tambahkan variabel di dalam tanda kurung fungsi. Namun, variabel tersebut tidak memerlukan keyword var, let, ataupun const. Kita juga bisa menambahkan lebih dari satu parameter dengan memberikan tanda koma antar variabel parameternya. Contohnya fungsi greeting akan kita tambahkan parameter name dan language seperti ini:

function greeting(name, language) {

    if(language === "English") {

        console.log(`Good Morning ${name}!`);

    } else if (language === "French") {

        console.log(`Bonjour ${name}!`);

    } else {

        console.log(`Selamat Pagi ${name}!`);

    }

}

Sehingga dalam memanggilnya pun kita perlu mengirimkan dua buah nilainya sebagai argumen:

1

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13917?from=15327#run2-editor1)

function greeting(name, language) {

if(language === "English") {

console.log(`Good Morning ${name}!`);

} else if (language === "French") {

console.log(`Bonjour ${name}!`);

} else {

console.log(`Selamat Pagi ${name}!`);

}

}

​

greeting("Harry", "French");

​

/\* output

Bonjour Harry!

\*/

Satu hal lagi, function dapat menghasilkan output atau mengembalikan sebuah nilai. Dengan nilai kembalian, kita dapat membuat function yang berfungsi untuk melakukan perhitungan matematika dan hasilnya dapat kita masukkan ke dalam sebuah variabel. Contohnya seperti ini:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13917?from=15327#run3-editor1)

function multiply(a, b) {

return a \* b;

}

let result = multiply(10, 2)

console.log(result)

​

/\* output

20

\*/

Agar fungsi bisa mengembalikan nilai, gunakan *keyword* return diikuti dengan nilai yang akan dikembalikan. Nilai kembalian tidak hanya number, bisa juga berupa string, boolean, object, array, atau tipe yang lain. Seperti inilah fungsi greeting() jika kita ubah agar mengembalikan nilai string:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13917?from=15327#run4-editor1)

function greeting(name, language) {

if(language === "English") {

return `Good Morning ${name}!`

} else if (language === "French") {

return `Bonjour ${name}!`;

} else {

return `Selamat Pagi ${name}!`;

}

}

​

let greetingMessage = greeting("Harry", "French");

console.log(greetingMessage);

​

/\* output

Bonjour Harry!

\*/

Yang perlu kita perhatikan lagi, ketika statement return tereksekusi, maka fungsi akan langsung terhenti dan mengembalikan nilai.

**Expression Function**

Cara lain untuk membuat sebuah fungsi pada JavaScript adalah *expression function*. Ingat kembali bahwa *expression* adalah kode atau instruksi yang mengembalikan nilai, sehingga expression function bisa disimpan dalam sebuah variabel.

Pada expression function umumnya kita tidak perlu menuliskan nama fungsinya. Fungsi yang tidak bernama juga dikenal dengan *anonymous function*. Berikut ini merupakan contoh penulisan expression function:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13917?from=15327#run5-editor1)

const greeting = function(name, language) {

if(language === "English") {

return "Good Morning " + name + "!";

} else if (language === "French") {

return "Bonjour " + name + "!";

} else {

return "Selamat Pagi " + name + "!";

}

}

​

console.log(greeting('Ron', 'English'));

​

/\* output

Good Morning Ron!

\*/

**Function Parameter**

Pada materi sebelumnya, kita telah berkenalan dengan fungsi yang merupakan suatu blok kode yang dapat menerima input dan menghasilkan output tertentu. Input ke dalam sebuah fungsi dikirimkan melalui parameter.

Di dalam fungsi kita akan banyak bertemu istilah parameter & argument. Penggunaan istilah ini sering kali tertukar, bahkan di kalangan developer.

Perbedaan mendasar antara keduanya antara lain:

Parameter merupakan variabel yang didefinisikan sebagai inputan dari sebuah fungsi. Contoh:

function multiply(a, b) {

    return a \* b;

}

Argument merupakan nilai atau expression yang dimasukkan ke dalam fungsi. Contohnya:

multiply(3, 4);

Parameter dari fungsi dapat berupa tipe data apa pun, mulai dari string, number, object, bahkan fungsi lain.

Jika parameter dari fungsi adalah sebuah object, kita juga bisa memanfaatkan destructuring object untuk mendapatkan nilainya. Contohnya seperti berikut:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/17272?from=13917#run1-editor1)

const user = {

id: 24,

displayName: 'kren',

fullName: 'Kylo Ren',

};

​

function introduce({displayName, fullName}) {

console.log(`${displayName} is ${fullName}`);

}

​

introduce(user);

​

/\* output

kren is Kylo Ren

\*/

**Default Parameters**

Fungsi pada JavaScript tidak melakukan pengecekan terhadap jumlah maupun tipe argumen yang dimasukkan ke dalam parameter. Ini berarti kita bisa memasukkan argumen meskipun tidak sesuai dengan parameter yang telah didefinisikan.

[**main.js**](https://www.dicoding.com/academies/256/tutorials/17272?from=13917#run2-editor1)

function exponentsFormula(baseNumber, exponent) {

let result = baseNumber \*\* exponent;

console.log(`${baseNumber}^${exponent} = ${result}`);

}

​

exponentsFormula(2);

​

/\* output

2^undefined = NaN

\*/

Seperti yang kita lihat pada contoh kode di atas, ketika argumen lebih sedikit dari parameternya, maka parameter yang tidak terdefinisi akan bernilai *undefined*. Sebagai solusi jika memungkinkan, kita bisa memberikan nilai default pada parameter. Nilai ini akan digunakan jika kita tidak memasukkan parameter.

[**main.js**](https://www.dicoding.com/academies/256/tutorials/17272?from=13917#run3-editor1)

function exponentsFormula(baseNumber, exponent = 2) {

let result = baseNumber \*\* exponent;

console.log(`${baseNumber}^${exponent} = ${result}`);

}

​

exponentsFormula(3);

​

/\* output

3^2 = 9

\*/

**Rest Parameter**

Masih ingat dengan spread operator (...)? Jika spread operator menyebarkan array menjadi beberapa elemen berbeda, rest parameter ini adalah kebalikan dari operator tersebut. Penasaran?

Rest parameter juga dituliskan menggunakan three consecutive dots (...). Dengan rest parameter, kita dapat menggabungkan beberapa elemen menjadi satu array. Tentu teknik ini sangat bermanfaat ketika kita hendak membuat sebuah fungsi dengan parameter yang tidak pasti.

Sebagai contoh adalah fungsi yang menjumlahkan seluruh nilai argumen seperti berikut:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/17272?from=13917#run4-editor1)

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

\*/

**Arrow Function**

ES6 memperkenalkan fungsi baru yang dinamakan *arrow function expression* atau lebih dikenal sebagai *arrow function*. Arrow function mirip seperti regular function secara perilaku, tetapi berbeda dalam penulisannya. Sesuai namanya, fungsi didefinisikan menggunakan tanda panah atau fat arrow ( => ). Tentunya penulisan arrow function ini akan lebih singkat.

Selain perbedaan sintaksis, terdapat perbedaan perilaku antara arrow function dan regular function. Regular function dapat berupa function declaration dan function expression. Namun, arrow function hanya berupa expression function saja. Itu sebabnya arrow function memiliki nama lengkap “arrow function expression”.

[**Arrow function**](https://www.dicoding.com/academies/256/tutorials/13922?from=17272#tab1-code2)

// function expression

const sayHello = (greet) => {

    console.log(`${greet}!`)

}

const sayName = (name) => {

    console.log(`Nama saya ${name}`)

}

Pada arrow function 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.

const sayName = (name) => {

    console.log(`Nama saya ${name}`)

}

Apabila fungsi hanya memiliki satu parameter, maka kita bisa menghapuskan tanda kurung seperti berikut:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13922?from=17272#run1-editor1)

const sayName = name => {

console.log(`Nama saya ${name}`)

}

​

sayName("Leia");

​

/\* output

Nama saya Leia

\*/

Namun, jika kita sama sekali tidak membutuhkan parameter, maka kita tetap menuliskan tanda kurung namun kosong seperti ini:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13922?from=17272#run2-editor1)

const sayHello = () => {

console.log("Selamat pagi semuanya!")

};

​

sayHello();

​

/\* output

Selamat pagi semuanya!

\*/

Satu hal yang menarik, ketika *body* dari function hanya terdiri dari satu baris, kita bisa menghapus tanda kurung kurawal. Tentunya ini akan menghemat baris kode yang kita tulis.

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13922?from=17272#run3-editor1)

const sayName = name => console.log(`Nama saya ${name}`);

sayName("Leia");

​

const sayHello = () => console.log("Selamat pagi semuanya!");

sayHello();

​

/\* output

Nama saya Leia

Selamat pagi semuanya!

\*/

Ketika sebuah fungsi perlu mengembalikan nilai, kita tidak perlu lagi menuliskan return (hanya bekerja untuk fungsi satu baris).

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13922?from=17272#run4-editor1)

const multiply = (a, b) => a \* b;

console.log(multiply(3, 4));

​

/\* output

12

\*/

**Variable Scope**

Sejauh ini kita sudah mengenal *function*. Setelah kita memisahkan kode ke dalam blok atau fungsi terpisah, ada satu hal penting yang perlu kita tahu, yaitu *variable scoping*. Ada banyak 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.

Variabel yang dapat diakses dari seluruh *script* disebut dengan “globally scoped”, sementara variabel yang hanya diakses hanya pada fungsi tertentu disebut dengan “locally scoped”.

Variabel JavaScript menggunakan fungsi untuk mengelola cakupannya. Jika variabel didefinisikan di luar fungsi, maka variabel tersebut bersifat global. Jika variabel didefinisikan di dalam fungsi, maka variabel bersifat lokal dan cakupannya hanya pada fungsi tersebut beserta turunannya.

Berikut ini merupakan contoh *scoping* dalam kode:

// global variable, dapat diakses pada parent() dan child()

const a = 'a';

function parent() {

    // local variable, dapat diakses pada parent() dan child(), tetapi tidak dapat diakses di luar dari fungsi tersebut.

    const b = 'b';

    function child() {

        // local variable, dapat diakses hanya pada fungsi child().

        const c = 'c';

    }

}

Kita harus berhati-hati dalam mendefinisikan variabel di dalam fungsi. Pasalnya, kita bisa mendapatkan hasil yang tidak diperkirakan, contohnya seperti berikut:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13927?from=13922#run1-editor1)

function multiply(num) {

total = num \* num;

return total;

}

​

let total = 9;

let number = multiply(20);

​

console.log(total)

​

/\* output

400

\*/

Mungkin kita berharap nilai total akan tetap 9, mengingat variabel total pada fungsi multiply seharusnya tidak akan berpengaruh untuk kode di luar dari fungsi tersebut. Hal ini bisa terjadi karena pada fungsi multiply() kita tidak menetapkan variabel total sebagai cakupan lokal. Kita tidak menggunakan *keyword* const atau let ketika mendeklarasikan variabel total pada fungsi multiply() sehingga variabel total menjadi global.

Perlu kita perhatikan, jika kita lupa menuliskan keyword let, const, atau var pada script ketika membuat sebuah variabel, maka variabel tersebut akan menjadi global.

Sebisa mungkin kita harus menghindari pembuatan variabel global, karena variabel global dapat diakses pada seluruh script yang kita tuliskan. Semakin banyak variabel global yang kita tuliskan, semakin tinggi kemungkinan tabrakan (*collision*) terjadi.

**Closure**

Setelah mempelajari tentang scope pada materi sebelumnya, kali ini kita akan membahas seputar *closure*. Sebelumnya kita telah tahu bahwa fungsi dapat didefinisikan dalam lingkup global atau di dalam fungsi lain. Suatu fungsi yang dapat mengakses variabel di dalam *lexical scope*-nya disebut dengan closure. *Lexical scope* berarti pada sebuah fungsi bersarang, fungsi yang berada di dalam memiliki akses ke variabel di lingkup induknya.

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13928?from=13927#run1-editor1)

function init() {

var name = 'Obi Wan'; // Variabel lokal di dalam scope fungsi init

function greet() { // Inner function, merupakan contoh closure

console.log(`Halo, ${name}`); // Memanggil variabel yang dideklarasikan di parent function

}

​

greet();

}

​

init();

​

/\* output

Halo, Obi Wan

\*/

Fungsi init() memiliki variabel lokal name dan fungsi greet(). Fungsi greet() adalah *inner function* yang didefinisikan di dalam init() dan hanya bisa diakses dari dalam fungsi init(). Perhatikan bahwa fungsi greet() tidak memiliki variabel lokal. Namun, karena *inner function* memiliki akses ke variabel di *parent function*-nya, sehingga greet() dapat mengakses variabel name. Itulah yang dimaksud dengan *lexical scope*.

Sekarang perhatikan contoh kode berikut:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13928?from=13927#run2-editor1)

function init() {

var name = 'Obi Wan';

​

function greet() {

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

}

​

return greet;

}

​

let myFunction = init();

myFunction();

​

/\* output

Halo, Obi Wan

\*/

Kode di atas akan menghasilkan output yang sama. Perbedaannya adalah fungsi greet() dikembalikan (*return*) dari *outer function*-nya sebelum dieksekusi. Karena variabel name berada dalam scope init(), maka umumnya variabel tersebut akan hilang atau dihapus ketika fungsinya selesai dijalankan. Namun, pada kasus di atas fungsi greet() yang diakses melalui fungsi MyFunction() masih memiliki referensi atau akses ke variabel name. Variabel pada mekanisme di atas telah tertutup (*close covered*), yang berarti variabel tersebut berada di dalam *closure*.

Memang di awal cukup sulit untuk memahami closure. Jadi, mari kita lihat langsung untuk apa closure ini digunakan pada suatu program yang nyata.

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 variabel. Perhatikan contoh berikut:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13928?from=13927#run3-editor1)

let counter = 0;

​

let add = () => {

return ++counter;

}

​

console.log(add());

console.log(add());

counter = 23;

console.log(add());

​

/\* output

1

2

24

\*/

Nilai counter akan bertambah ketika kita memanggil fungsi add(). Namun, kita juga bisa mengubah nilai counter secara langsung dengan *assignment operator*. Pada contoh program yang lebih kompleks, sebaiknya hal ini dihindari karena perubahan langsung pada nilai counter bisa saja memunculkan *bug*.

Closure memungkinkan kita membuat fungsi dan variabel seolah menjadi *private*. Seperti inilah contoh program counter yang dibuat dengan closure:

[**main.js**](https://www.dicoding.com/academies/256/tutorials/13928?from=13927#run4-editor1)

let add = () => {

let counter = 0;

return () => {

return ++counter;

};

}

7

​

let addCounter = add();

​

console.log(addCounter());

console.log(addCounter());

console.log(addCounter());

​

/\* output

1

2

3

\*/