Fundamental Javascript #5

Sesi 7

Map, Reduce & Filter

Memahami Map, Filter & Reduce

Kita akan belajar dan memahami cara menggunakan method **.map(), .filter(),** dan **.reduce()**, untuk membantu mempermudah loop atau perulangan.

Map, **reduce**, dan **filter** adalah metode-metode array dalam JavaScript.

Setiap metode akan mengiterasi melalui array dan melakukan transformasi atau perhitungan. Setiap metode akan mengembalikan array baru berdasarkan hasil fungsi yang diberikan. Dalam artikel ini, Anda akan belajar mengapa dan bagaimana menggunakan masing-masing metode ini.

Memahami Map

Method .map() digunakan ketika kamu ingin membuat array baru dari array yang existing, baik setiap elemennya melalui suatu proses maupun tidak. **Perhatikan contoh kode dibawah ini (belum menggunakan Map):**

```
const harga = [19.99, 4.95, 25, 3.5];
let hargaSekarang = [];
for (let i = 0; i < harga.length; i++) {
  hargaSekarang.push(harga[i] * 1.06);
}

console.log(hargaSekarang);
// [ 21.1894, 5.247000000000001, 26.5, 3.71 ]
```

Memahami Map

Dengan .map(), kamu bisa mensimplifikasi kode yang sebelum menjadi seperti ini:

```
const harga = [19.99, 4.95, 25, 3.5];
let hargaSekarang = harga.map((h) => {
  return h * 1.06;
});

console.log(hargaSekarang);
// [ 21.1894, 5.247000000000001, 26.5, 3.71 ]
```

Memahami Filter

Filter merupakan method array di JavaScript yang berfungsi untuk mencari semua elemen di dalam array yang sesuai dengan kriteria atau kondisi tertentu.

Contoh kode dibawah ini tanpa menggunakan filter()

```
const bilangan = [1, 2, 3, 4, 5, 6, 7, 8];
let ganjil = [];
for (let i = 0; i < bilangan.length; i++) {
  if (bilangan[i] % 2 == 1) {
    ganjil.push(bilangan[i]);
  }
}

console.log(ganjil);
// [ 1, 3, 5, 7 ]
```

Memahami Filter

Dari iterasi menggunakan looping, kita bisa mensimplifikasi kode sebelumnya dengan menggunakan filter

```
const bilangan = [1, 2, 3, 4, 5, 6, 7, 8];
let ganjil = bilangan.filter((num) => num % 2);

console.log(ganjil);
// [ 1, 3, 5, 7 ]
```

Memahami Reduce

Berbeda dengan .map() dan .filter(), callback method .reduce() memerlukan dua parameter: **accumulator** dan **current value**. **Accumulator** akan menjadi parameter pertama dan merupakan nilai “pass it down”.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce(function (result, item) {
  return result + item;
}, 0);
console.log(sum); // 10
```

Memahami Reduce

Contoh tanpa reduce()

```
const sumbangan = [1000, 2000, 1000, 8000, 7500];
let total = 0;
for (let i = 0; i < sumbangan.length; i++) {
  total += sumbangan[i];
}

console.log(total);
// 19500
```

Memahami Reduce

Dengan reduce()

```
const sumbangan = [1000, 2000, 1000, 8000, 7500];
let total = sumbangan.reduce((total, sumbang) => {
  return total + sumbang;
});

console.log(total);
// 19500
```

“Use strict”

Use Strict

"use strict" adalah sebuah directive dalam JavaScript yang memungkinkan Anda memilih untuk menggunakan versi "ketat" dari JavaScript dalam fungsi atau file tertentu.
Ini diperkenalkan di ECMAScript 5 (ES5) untuk membantu programmer menulis kode yang lebih aman dan lebih efisien.

```
"use strict";

var v = "Hello World"; // OK

n = "Hello again"; // Error, because n is not declared
```

Pada contoh di atas, "use strict" akan memaksa Anda untuk mendeklarasikan variabel sebelum menggunakannya. Jika tidak, JavaScript akan melempar error. Ini dapat membantu Anda menemukan dan mencegah kesalahan dalam kode Anda.

Kegunaan Use Strict

Tentu, berikut beberapa contoh lain dari penggunaan "use strict" dalam JavaScript:

1. **Mencegah pembuatan variabel global secara tidak sengaja**

Tanpa "use strict", jika Anda lupa menulis var, let, atau const saat mendeklarasikan variabel, variabel tersebut akan menjadi variabel global. Namun, dengan "use strict", hal ini akan menyebabkan error.

```
"use strict";
x = 3.14; // This will cause an error because x is not declared
```

Kegunaan Use Strict

2. Mencegah penggunaan kata yang dicadangkan sebagai nama variabel

Anda tidak diizinkan menggunakan kata-kata tertentu (kata-kata yang dicadangkan dalam JavaScript) sebagai nama variabel atau fungsi. "use strict" akan memastikan bahwa Anda mendapatkan error jika Anda mencoba melakukannya.

```
"use strict";
var let = 10; // This will cause an error because 'let' is a reserved word
```

Kegunaan Use Strict

3. Mencegah penghapusan variabel, fungsi, atau fungsi argumen

Anda tidak diizinkan menghapus variabel, fungsi, atau argumen fungsi. "use strict" akan memastikan bahwa Anda mendapatkan error jika Anda mencoba melakukannya.

```
"use strict";
var x = 3.14;
delete x; // This will cause an error
```

Kegunaan Use Strict

4. Mencegah duplikasi parameter

Dalam mode "use strict", fungsi dengan parameter duplikat akan melemparkan error.

```
"use strict";
function x(p1, p1) {}; // This will cause an error
```

Modular Function

(Single Responsibility Principle)

Modular Function

Modular Function disini menggunakan Single Responsibility Principle (SRP) sebagai prinsip, Dimana SRP merupakan salah satu prinsip desain perangkat lunak yang mengatakan bahwa sebuah kelas atau modul harus memiliki satu tujuan.

Prinsip ini merupakan bagian dari lima prinsip desain yang dikenal sebagai SOLID, yang bertujuan untuk meningkatkan pemahaman, fleksibilitas, dan maintainability (kemudahan perawatan) dari kode.

Dengan menerapkan prinsip SRP, setiap bagian dari program memiliki satu tujuan yang jelas dan terpisah dari tugas lain, sehingga perubahan atau penambahan fitur pada satu bagian tidak akan mempengaruhi bagian lain.

Modular Function

Masih ingat dengan function? Function adalah salah satu fitur yang ada di hampir seluruh bahasa pemrograman, gunanya adalah untuk menulis code yang dapat digunakan berulang kali tanpa harus menulis ulang code-nya (reusable). Jika fitur ini digunakan dengan bijak, tentu saja code kita akan menjadi lebih rapi dan tertata.

Perlu diketahui bahwa function kita bisa saja terlalu banyak meng-handle proses yang kita butuhkan. Idealnya, kita membuat satu function hanya untuk satu tujuan dan tidak lebih.

Contoh:

Buatlah sebuah program untuk menghitung total harga barang yang diskon

Modular Function

Contoh Kasus: Menghitung total harga barang yang diskon

```
// Fungsi untuk menghitung total diskon
function hitungTotalDiskon(harga, diskon) {
  return harga - (harga * diskon);
}

// Fungsi untuk menghitung total harga setelah diskon
function hitungTotalHarga(harga, diskon) {
  const totalDiskon = hitungTotalDiskon(harga, diskon);
  return totalDiskon + (totalDiskon * 0.1); // Ditambahkan 10% pajak
}

// Penggunaan fungsi
const hargaBarang = 100;
const diskon = 0.2;

const totalHarga = hitungTotalHarga(hargaBarang, diskon);
console.log(`Total harga setelah diskon: ${totalHarga}`);
```

Seperti yang dilihat pada kode disamping, setiap fungsi memiliki tugasnya masing-masing.

1. Fungsi hitungTotalDiskon: mempunyai tugas untuk menghitung harga barang setelah diskon.

2. Fungsi hitungTotalHarga: mempunyai tugas untuk menghitung total harga barang akhir.

Modular Function

Mari coba contoh lainnya.

Encrypted Password

Buatlah sebuah program untuk meng-encrypt input dari user untuk dijadikan password.

- Hapus semua spasi dari input
- reverse input, jadi jika user memasukkan "abcde" kita putar jadi "edcba"
- Ganti huruf vokal menjadi satu huruf setelahnya (A menjadi B, I menjadi H dan seterusnya)

Modular Function

Best practice-nya, kamu bisa buat 3 function. Jadi setiap step masalah akan diselesaikan dengan 1 function. Bisa saja kita membuat satu function untuk menyelesaikan semuanya, tapi akan lebih sulit untuk di-debug (mencari kesalahan logika/code).

```
1   function removeSpaces (text) {
2       //code to remove spaces from text
3   }
4
5   function reverseText (text) {
6       //code to reverse the text
7   }
8
9   function updateVowels (text) {
10      //code to update vowels
11  }
12
13
14  var password = 'hacktiv 8';
15  var noSpaces = removeSpaces(password);
16  var reversed = reverseText(noSpaces);
17  var encryptedPassword = updateVowels(reversed);
18
19  console.log(encryptedPassword);
```

Callback, Promise, Async

Callback Concept

Callback sebenarnya adalah sebuah function yang bedanya dengan function pada umumnya adalah cara meng-eksekusinya. Jika function pada umumnya di eksekusi berurutan dari atas ke bawah maka callback di eksekusi pada titik tertentu, itu sebabnya di sebut callback.

Callback disebut juga dengan high-order function. Jika function pada umumnya di eksekusi secara langsung sedangkan callback di eksekusi dalam function lain melalui parameter.

```
function main(param1,param2,callBack){
    console.log(param1, param2)
    callBack()
}

function myCallback(){
    console.log ('hello callback')
}

main(1,2,myCallback)

/* ====================
Output :
 1 2
 hello callback
 */
```

Callback Concept

Kenapa function bisa dijadikan sebagai parameter?

Function dalam javascript adalah object atau sering disebut first-class object, yang artinya :

- Function bisa di jadikan parameter
- Function dapat disimpan ke dalam variabel
- Seperti object pada umumnya, function bisa memiliki property dan method
- Function dapat mengembalikan value dalam bentuk function

Callback Concept

Kapan Callback digunakan ?

Callback dapat digunakan untuk proses synchronous maupun asynchronous. Beberapa contoh implementasi callback adalah :

- Injeksi atau modifikasi hasil eksekusi sebuah function
- Event listener
- Menangani proses asynchronous

Callback Concept

Callback sebagai Injeksi sebuah function

```
function calculate(x,y){
  result = x + y
  return result
}
calculate(3,2) // 5
```

Kode diatas cukup sederhana yaitu untuk melakukan operasi penjumlahan. Berikut tantangannya :

- Buatlah function diatas agar bisa melakukan operasi matematika yg lain seperti kurang, bagi, kali dan lain sebagainya.
- Output dari function di atas harus bisa di format ke dalam mata uang

Callback Concept

Callback sebagai Injeksi sebuah function

Dengan cara umum kita bisa menyelesaikanya dengan bantuan if atau switch untuk menguji operatornya. Tapi ini akan membuat code lebih panjang dan kurang dinamis.

Dengan callback kita dapat membuat function diatas menjadi lebih dinamis.

```
function calculate(param1,param2,callback){
  //default operation
  result = param1 + param2;

  // callback is function ?
  if (typeof callback == 'function'){
    result= callback(param1,param2);
  }

  return result;
}

  //execute
  var a = calculate(2000,4000, function(x,y){return "$ " + (x + y) });
  var b = calculate(7000,2000, function(x,y){return "Rp " + (x * y) });
  console.log(a); // $ 6000
  console.log(b); // $ 14000
```

Promise Concept

Untuk membuat promise cukup dengan memanggil constructor nya :

```
var janjian = new Promises();
console.log(janjian);
```

Sampai disini output dari code atas adalah Promise { <pending> }.

Promise Concept

Lalu bagaimana untuk mengatur state Fullfilled dan Reject ?

Untuk state ini gunakan salah satu listener, *resolve()* atau *reject()*

```
var janjian = new Promise(
(resolve,reject) => {
    //salah satu dari 2 callback berikut
    //resolve
    //reject
}
);
```

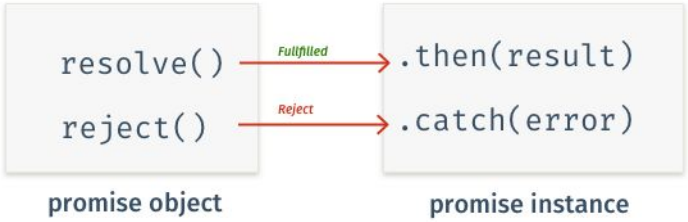
Promise Concept

Untuk menggunakan promise pada gambar dihalaman sebelumnya, gunakan method then dan catch.

```
janjian
    .then((result) => { console.log(result) })
    .catch((error) => { console.log(error) })
```

Output dari code diatas ada 2 kemungkinan,

- Jika comment pada resolve( ) di hapus maka hasilnya “berhasil”
- Jika comment pada reject( ) di hapus maka hasilnya “Janji di batalkan”



Asynchronous Concept

Ada banyak sekali implementasi asynchronous dalam javascript seperti event, timer, request ajax, listener, interaksi user dan masih banyak lagi.

Dalam dunia programming kedua istilah ini digunakan untuk membedakan tentang cara urutan eksekusi perintah-perintah yang ada dalam kode anda.

Synchronous adalah yang paling umum dan mudah di mengerti. Setiap perintah di eksekusi satu persatu sesuai urutan kode yang anda tuliskan. Contoh :

```
console.log('Hello');
console.log('Javascript');
console.log('Coder');

/*
Output:
Hello
Javascript
Coder
*/
```

Asynchronous Concept

```
console.log('Hello');
console.log('Javascript');
console.log('Coder');

/*
Output:
Hello
Javascript
Coder
*/
```

Output dari kode diatas dijamin akan sesuai urutan, karena setiap perintah harus menunggu perintah sebelumnya selesai. Proses seperti ini disebut 'blocking'.

Dalam dunia nyata, ini mirip seperti antrian di BANK. Jika anda berada antrian nomor 4, maka anda akan dilayani setelah antrian 1–3 sampai selesai.

Sedangkan Asynchronous hasil eksekusi atau output tidak selalu berdasarkan urutan kode, tetapi berdasarkan waktu proses. Eksekusi dengan asynchronous tidak akan membloking atau menunggu suatu perintah sampai selesai.

Daripada menunggu, asynchronous akan mengeksekusi perintah selanjutnya.

Asynchronous Concept

```
console.log('Hello');
setTimeout(()=>
  {console.log('Javascript')},
100); //tunda selama 100 miliseconds
console.log('Coder');

/*
Output:
Hello
Coder
Javascript
*/
```

Catatan :

Pada baris ke 2 **setTimeout** digunakan untuk menunda eksekusi dalam satuan milisecond dalam hal ini untuk simulasi prosess async.

Perhatikan bahwa outputnya tidak berurutan sesuai input (kode). Karena cara kerja **asynchronous** adalah berdasarkan waktu proses. Jika ada salah satu eksekusi membutuhkan proses yang agak lama, maka sembari menunggu prosess tersebut Javascript mengeksekusi perintah selanjutnya.

Question

Apakah javascript secara default mengeksekusi perintah dengan metode synchronous atau asynchronous ?

Javascript secara default mengeksekusi perintah secara synchronous, kecuali untuk beberapa hal seperti : ajax,websocket, worker, file, database, animasi dan beberapa hal lainya.

Bisakah kita membuat proses asynchronous ?

Kita tidak bisa membuat proses asynchronous murni. Tapi untuk membuat simulasi proses asynchronous, maka kita bisa menggunakan fungsi setInternal dan setTimeout.

Dalam kasus apa teknik asynchronous digunakan ?

Teknik asynchronous paling banyak digunakan untuk mengelola komunikasi yang tidak mungkin sinkron atau harus menunggu seperti proses request ajax, operasi file, koneksi ke database, websocket, real time communication seperti pada aplikasi chating dan masih banyak lagi.

Extras

Callback Pada Asynchronous

Proses asynchronous identik dengan delay/jeda, dimana hasil dari proses tersebut membutuhkan selang waktu tertentu untuk menghasilkan output.

Kita akan menemukan proses asynchronous pada proses Ajax, komunikasi HTTP, Operasi file, timer, dan lain-lain.

Pada synchronous output di prosess berdasarkan urutan kode seperti pada gambar di sebelah kanan.

```
function p1(){
  console.log('p1 done');
}

function p2() {
  console.log('p2 done');
}

function p3(){
  console.log('p3 done');
}

p1();
p2();
p3();

/* Output:
p1 done
p2 done
p3 done
*/
```

Extras

Callback Pada Asynchronous

Tetapi pada proses asynchronous output dari kode yang tuliskan tidak selalu berurutan. Hasilnya tergantung yang mana yang lebih dulu selesai.

Perhatikan kode pada gambar di sebelah kanan.

Catatan : setTimeout digunakan untuk simulasi asynchronous. Karena sebenarnya kita tidak bisa membuat proses asynchronous murni.

Perhatikan output dari kode di sebelah kanan tidak lagi berurutan. Kerena javascript mengerjakan mana yang lebih dulu selesai.

```
function p1(){
  console.log('p1 done');
}

function p2(){
  //Set TimeOut or delay for async simulation
  setTimeout(
      function(){
          console.log('p2 done');
      },100
  )
}

function p3(){
  console.log('p3 done');
}

p1();
p2();
p3();

/* Output:
p1 done
p3 done
p2 done
*/
```

Extras

Callback Hell

Callback hell adalah istilah ketika membuat beberapa callback bercabang atau callback di dalam callback. Sebagai contoh menggabung beberapa file ke dalam satu file.

```
var a = readFileContent("a.md");
var b = readFileContent("b.md");
var c = readFileContent("c.md");
writeFileContent("result.md", a + b + c);
console.log("we are done");
```

Karena *readFileContent( )* adalah proses asynchronous maka di kelola dengan callback seperti berikut :

```
readFileContent("a.md", function (a){
    readFileContent("b.md", function (b){
      readFileContent("b.md", function (b){
          writeFileContent("result.md", a + b + c, function(){
              console.log("we are done");
          });
        });
    });
  });
```

Extras

Apa masalah pada kode di halaman sebelumnya?

Sebenarnya dari segi output tidak ada problem, yang menjadi problem adalah

1. Kode sulit dibaca, dalam kasus tertentu piramida kode bisa menjadi lebih panjang dan sulit untuk di kelola.
2. Tidak ada error handling, jika salah satu proses readFile error sulit di debug bagian yang mana yang error.

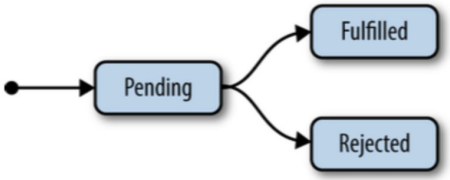
Extras

Bagaimana Solusinya?

Solusi pertama adalah dengan membuat kode yang lebih modular agar lebih mudah dibaca. Tapi solusi yang lebih mudah adalah menggunakan promise.

Promise adalah salah fitur terbaru dari ES6. Jika anda sebelumnya sudah pernah menggunakan method **.then**, maka anda sudah menggunakan promise. Mari kita mulai dari analogi sederhana. Anda melakukan perjanjian unutuk bertemu dengan salah satu kolega anda, tiba-tiba kolega tersebut bertanya anda sudah dimana ? Ada beberapa kemungkinan jawaban disini : dalam perjalanan, sudah sampai atau janjinya di batalkan.

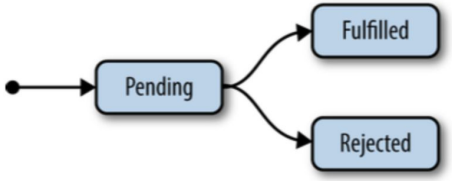
Dalam dunia promise analogi di atas juga sama, ketika melakukan request asynchronous seperti Ajax, maka ada 3 kemungkinan state :



https://miro.medium.com/max/952/0*-Jok-C_yIXm3xb_t.png

Extras

Bagaimana Solusinya?



https://miro.medium.com/max/952/0*-Jok-C_yIXm3xb_t.png

- Pending (sedang dalam proses)
- Fulfilled (berhasil)
- Rejected (gagal)

Lalu bagaimana implementasinya dalam Javascript ? Untuk sekarang ingat saja bahwa promise itu adalah object. Object yang merepresentasikan state diatas.

Extras

Async/Await

Async/await adalah fitur yang hadir sejak ES2017. Fitur ini mempermudah kita dalam menangani proses asynchronous.

Ada 2 kata kunci disini yaitu async dan await, mari kita lihat contohnya :

```
async function hello(){
    result = await doAsync();
    console.log(result);
}
```

Keterangan :

async → mengubah function menjadi asynchronous.

await → menunda eksekusi hingga proses asynchronous selesai, dari kode di atas berarti console.log(result) tidak akan di eksekusi sebelum proses doAsync( ) selesai, dan await juga bisa digunakan berkali-kali di dalam function.

Extras

Pattern

```
// 1. Anonymous Async Function
let main = (async function() {
  let value = await doAsync();
})();


// 2. Async Function Declaration
let main = async function() {
  let value = await doAsync();
};

//3. Async Function Assignment
let main = async () => {
  let value = await doAsync();
};

//4. Async Function as Argument
document.body.addEventListener('click', async function() {
  let value = await doAsync();
});


//5 Object & Class Methods
let obj = {
  async method() {
    let value = await doAsync();
  }
};

class MyClass {
  async myMethod() {
    let value = await doAsync();
  }
}
```

Extras

Serial & Pararel

Pada saat mengeksekusi beberapa proses asynchronous, ada kalanya kita harus memilih eksekusi secara serial atau parallel. Serial biasanya digunakan jika kita ingin mengeksekusi proses asynchronous secara berurutan. Sedangkan paralel jika ingin di eksekusi secara bersamaan, dalam hal ini urutan tidak menjadi prioritas tapi hasil dan performa.

```
const firstPromise= () => (new Promise((resolve,reject) => {
  setTimeout(() =>{ resolve('first Promise')},1000);
}))

const secondPromise = () => ( new Promise((resolve,reject) =>{
  setTimeout(() =>{ resolve('second Promise')},1000);
}))

const thirdPromise = () => ( new Promise((resolve,reject) =>{
  setTimeout(() =>{ resolve('third Promise')},1000);
}))

async function asyncParalel() {
  let a = firstPromise();
  let b = secondPromise();
  let c = thirdPromise();
  console.log('done');
}

async function asyncSerial() {
   let a = await firstPromise();
   let b = await secondPromise();
   let c = await thirdPromise();
   console.log('done');
}
```