



OOP di JavaScript

Gold - chapter 4 - Topic 1

**Selamat datang di Chapter 4 Topik 1 online
course Fullstack Web dari Binar Academy!**





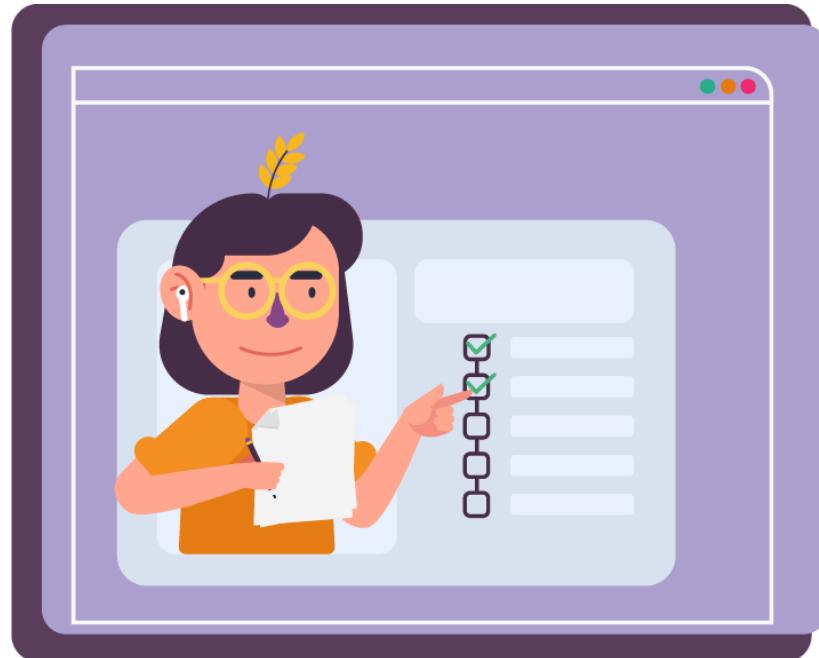
Sebagai pembuka course, chapter 4 bakal ngajak kamu buat **memahami cara menerapkan OOP dan DOM dalam pengembangan website**, mulai dari pengenalan OOP, DOM, Node.JS, dan penggunaan HTTP server

Di topik pertama ini, kita bakal bahas dulu tentang **konsep dan penggunaan OOP**. Cus langsung aja kita kepoin~



Detailnya, kita bakal bahas hal-hal berikut ini:

- Mengenal konsep OOP
- Memahami cara menggunakan function, class dan method di JavaScript
- Menerapkan 4 pilar OOP yaitu, Inheritance, Encapsulation, Abstraction, dan Polymorphism





Sebenarnya OOP itu apa ya?

Kenapa kita harus menguasai konsep ini?

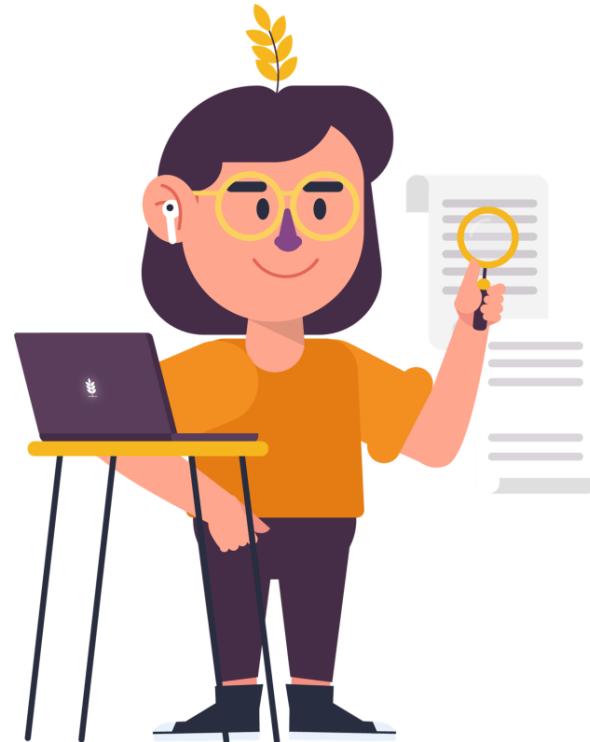
Apa cuma ada konsep OOP aja di dunia pemrograman?



Hmmm.. pasti diantara kamu kepikiran bertanya-tanya begitu ya?

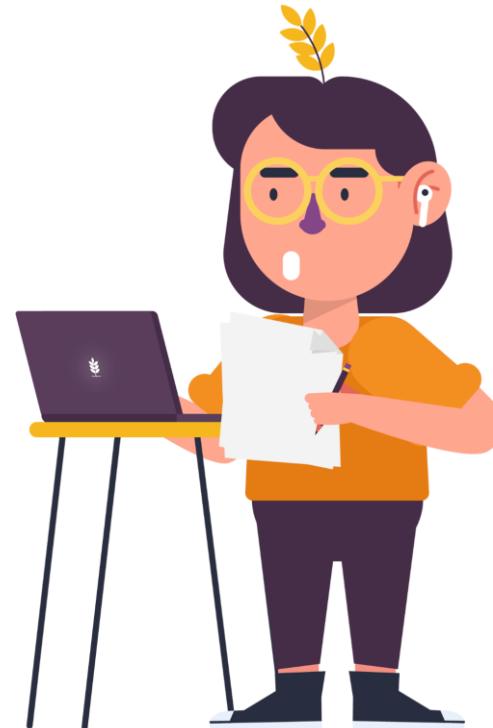
Bagusss! Kita emang perlu tahu dulu sih OOP itu apa, ada apa aja paradigma di pemrograman, sampe akhirnya kita jadi tahu alasannya kenapa kita perlu belajar OOP.

Paradigma pemrograman bisa dibilang sebagai **konsep atau kerangka berpikir atau style yang bisa kita gunakan untuk menyelesaikan masalah, melalui bahasa pemrograman**. Nulis program itu sama aja kayak kita lagi nulis puisi, jadi gayanya bisa beda-beda.



OOP ini termasuk jenis paradigma pemrograman yang banyak dipakai di JavaScript, karena gaya penulisannya yang berorientasi objek.

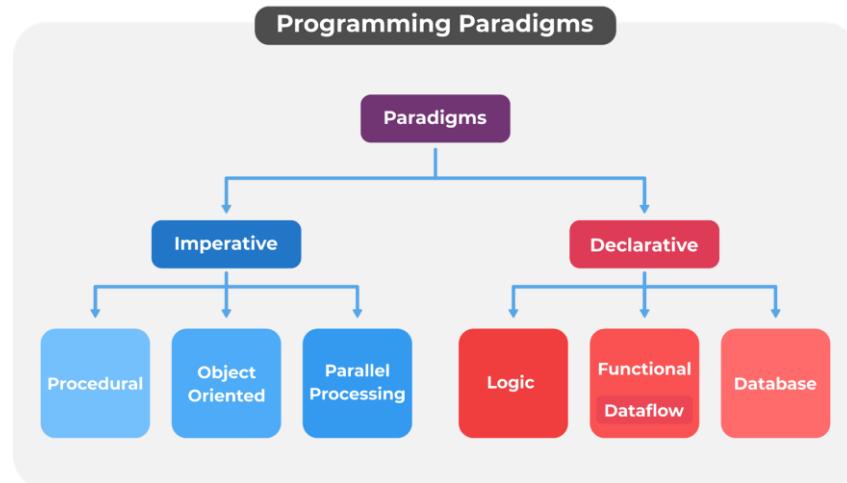
Sesuai sama namanya, yaitu Object Oriented Programming.



Eits, paradigma pemrograman nggak cuma ada OOP aja, lho~

Yap, jadi sebenarnya paradigma pemrograman itu nggak cuma ada OOP aja, tapi ada juga paradigma lain yang sering dipake di pemrograman, yaitu:

- Procedural Programming
- Parallel Processing Approach
- Logic Programming
- Functional Programming
- Database Processing Approach
- dan masih banyak lagi lainnya...





Tapi, di sesi ini kita cuma bakal fokus buat belajar OOP aja karena paradigma ini dipake di JavaScript, terutama pas nanti kita bikin project sama library JavaScript, yaitu React.

Kalau kita tulis kode pake paradigma OOP, itu bakal bikin kita gampang lho~

Karena strukturnya bakal lebih jelas dan ngejaga kode kita biar gak **DRY** alias **Don't Repeat Yourself**, jadiii kode kita bisa lebih gampang buat dikelola, dimodifikasi, dan di-debug.

Makin semangat kan belajar OOP-nya? Gasss dong!





Sebelum bahas OOP di Javascript, kita harus tahu dulu cara pake **function, **class** dan **method**.**

Di topik sebelumnya kita udah bahas gimana cara pake variabel dan object. Kamu masih ingat?

Kita ulas tipis-tipis yaaa~





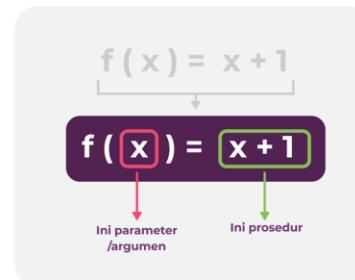
Masih ingat nggak waktu dulu kita pernah belajar matematika dasar?

Kita udah belajar yang namanya fungsi kan? Biar kamu ingat, coba lihat contoh fungsi di samping dulu deh~

Fungsi di dalam bahasa pemrograman juga nggak jauh beda sama fungsi di pelajaran matematika dasar.

Fungsi terdiri dari 3 hal, yaitu:

- Parameter
- Procedure
- Return value (hasil)




$$f(x) = x + 1$$
$$f(2) = 2 + 1$$

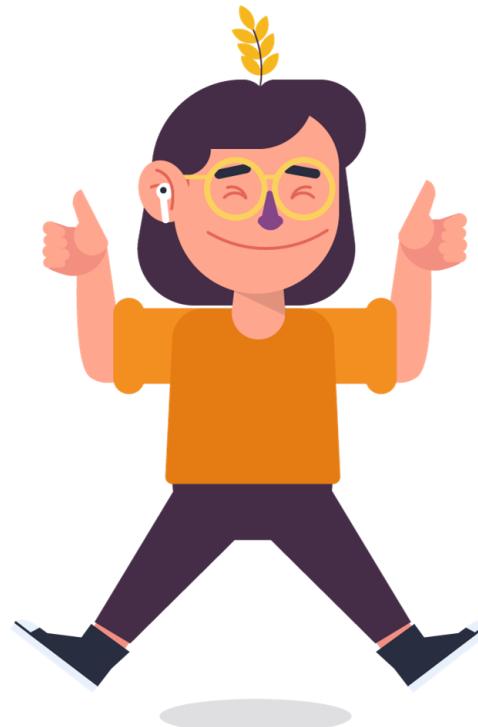

$$y = f(2)$$
$$y = \boxed{3}$$

Ini hasil fungsi

Dari bentuk matematika tadi bisa disimpulkan, kalau $f(x)$ adalah sebuah fungsi yang bakal minta x sebagai parameter.

Nah, x ini ditugaskan buat jadi variabel dari parameter.

Maksudnya, pas kita manggil fungsi tersebut, maka nilai x bisa punya nilai berapapun dan bakal dimasukkan ke dalam fungsi sebagai parameter. Sampai akhirnya ngasih hasil berupa nilai tertentu.





Oke, sekarang kita masuk cara untuk menulis fungsi di JavaScript, ya~

Jadi, bakal ada dua kata kunci yang kita pake, yaitu :

- **Function**

Buat mendeklarasikan fungsi dan di dalam scope function yang pasti bakal ada **kode/perintah yang bakal dieksekusi**.

Kalau di contoh, diskon dijadikan sebagai nama fungsi. Buat kode yang bakal dieksekusi, siap buat ditampung ke dalam variabel let musimPandemik.



```
function diskon(x) {  
  let musimPandemik = (x * 30)/100  
  return musimPandemik  
}  
  
let sale = diskon(20000)  
console.log(sale) // Output: 6000
```



- **Return**

Buat **memberikan/mengembalikan hasil dari fungsi** tersebut.

Kalau setelah keyword return kosong, berarti hasil dari fungsi tersebut bakal dianggap undefined.



```
function diskon(x) {  
  let musimPandemik = (x * 30)/100  
  return musimPandemik  
}  
  
let sale = diskon(20000)  
console.log(sale) // Output: 6000
```



Ada hal yang harus kita perhatikan nih tentang **penulisan parameter**. Kita **bisa input tipe data apapun, tapi syaratnya data yang kita pake harus relevan sama fungsinya**.

Misalnya, prosedur fungsi kita bakal memproses parameter yang dianggap sebagai String, jadi kita wajib masukin nilai dengan tipe data String juga.

Tapiii, kalau nilai yang kita input tipe datanya nggak sama, yaaa jadinya bakalan error. So, hati-hati ya!



```
function sayHiTo(name) {  
  let halo = `Hai  
${name.toUpperCase()}!`  
  return halo  
}  
  
let test1 = sayHiTo("everything")  
console.log(test1)  
// Output: Hai EVERYTHING!  
let test2 = sayHiTo(100)  
console.log(test2)  
// Output: TypeError:  
name.toUpperCase is not a function
```



Ohya, kita bisa mendeklarasikan fungsi pake cara yang berbeda lho!

Biasanya kan kita mendeklarasikan sebuah fungsi pake keyword function (**function declaration**), ternyata keyword function juga bisa dipake buat mendefinisikan fungsi di dalam ekspresi, artinya fungsi bakal disimpan di dalam sebuah variabel. Fungsi ini disebut sebagai fungsi anonim (**anonymous function**).

Kita juga bisa pake syntax ES6 yaitu **arrow function** biar fungsi jadi lebih sederhana. Terutama kalau cuma satu baris, jadi kita nggak perlu nulis keyword return.



```
// Function Declaration (ES5)
function volTabung(r, t) {
  return 3.14 * r**2 * t
}
console.log('Volume Tabung:', volTabung(10, 4))
// Volume Tabung: 1256

// Function Expression
const volTabung = function(r, t) {
  return 3.14 * r**2 * t
}
console.log('Volume Tabung:', volTabung(10, 4))
// Volume Tabung: 1256

// Arrow Function (ES6)
const volTabung = (r, t) => 3.14 * r**2 * t
console.log('Volume Tabung:', volTabung(10, 4))
// Volume Tabung: 1256
```



Ada yang menarik lagi nih mengenai fungsi!

Fungsi juga bisa nerima fungsi sebagai parameternya,
atau dikenal dengan istilah **Higher Order Function**.

Contohnya adalah method **forEach** yang kita pake buat ngelakuin loop di dalam array.

Fungsi forEach ini nerima fungsi sebagai parameter,
secara spesifiknya di parameter yang namanya callback.



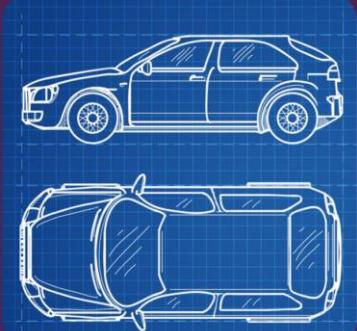
```
const strArray = ['JavaScript', 'Java', 'C'];
function forEach(array, callback) {
  const newArray = [];
  for(let i = 0; i < array.length; i++) {
    newArray.push(callback(array[i]));
  }
  return newArray;
}

const lenArray = forEach(strArray, (item) => {
  return item.length;
});
console.log(lenArray);
// Output: [ 10, 4, 1 ]
```



CLASS

BLUEPRINT



OBJECT

VAN



AUDI



SPORT CAR



Setelah memahami cara pake fungsi, kita bakal memahami **class** dan **object**, dan hubungan antara keduanya.

Misalnya, kita punya blueprint mobil. **Class adalah suatu blueprint atau acuan untuk membuat object** mobil. Dari blueprint ini, kita bisa tahu kalau mobil itu punya 4 roda, ada pintunya, dll.

Dengan mengandalkan si blueprint mobil ini, kita bisa bikin berbagai jenis mobil. Ada yang akan kita buat jadi mobil van, mobil audi, dan mobil sport.

Keren ya! Sebuah class bisa membuat banyak object!



Buat mendeklarasikan suatu class, kita bisa pake keyword class terus diikuti sama nama kelasnya.

Oke, kita coba deklarasi sebuah class, yuk!

Kita kasih nama jadi class Person, kayak contoh disamping~



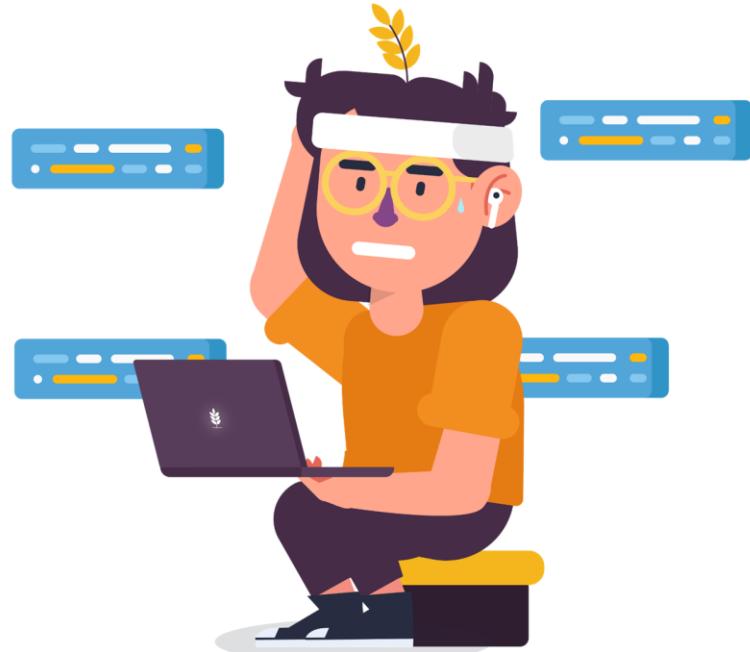
```
class Person {  
    constructor(name, address) {  
        this.name = name;  
        this.address = address;  
    }  
}
```

Bikin object pake class gampang banget, kan?!

Tentunya ini bakal memudahkan kita, terutama kalau kita mau bikin banyak object.

Sebenarnya, bikin object pake cara biasa atau object literal itu bisa-bisa aja lho. Tapi, kita bakalan repot kalau mau mengubah atau menambahkan property baru di object tersebut, soalnya kita harus nulis ulang object tersebut satu per satu.

Bayangan kalau jumlah objeknya ada ratusan bahkan ribuan! Weleh-weleh... capek deh ~



Sebenarnya class kita pake buat bikin tipe data baru yang nantinya data itu bakal berupa object.

Ada beberapa hal yang perlu kita tahu tentang penggunaan class nih, yaitu:

- **Constructor**, buat mendefinisikan property pada suatu object.
- **Property**, data dari suatu object atau variabel-variabel yang ada di object. Biasanya disebut juga **atribut**. Setiap object pasti punya property, misalnya manusia bernama Sabrina adalah object, berarti dia punya beberapa property, kayak nama, alamat, dsb.





Ohyah, kalau kita bikin object tanpa class, berarti object itu cuma punya satu tipe property aja. Sedangkan kalau kita bikin pake class, berarti bakal ada dua tipe property, yaitu:

1. **Instance property**, sebuah property yang bisa kita akses setelah object kita instantiate (dibuat lewat keyword new).
2. **Static property**, nilainya bakal selalu sama di semua instance dari class tersebut.



- **Method**, fungsi atau aksi dari suatu object. Maksudnya, kalau kita punya class Human, berarti method itu ibarat aksi yang bisa dilakukan manusia, kayak jalan, berbicara, dsb.

Di dalam method, kita bisa akses property dari object pake keyword this, yang artinya adalah manggil object itu sendiri. Sama kayak property, method punya dua tipe dalam hal aksesnya yaitu **Instance method** (prototype) dan **Static method**.





Untuk lebih jelasnya bisa lihat contoh pembuatan class dan cara aksesnya dibawah ini ya~

```
● ● ●  
class Human {  
    // Add static property  
    static isLivingOnEarth = true;  
  
    // Add constructor method  
    constructor(name, address) {  
        this.name = name;  
        this.address = address;  
    }  
  
    // Add instance method signature  
    introduce() {  
        console.log(`Hi, my name is  
        ${this.name}`)  
    }  
  
    console.log(Human.isLivingOnEarth)  
    // Output static property: true
```

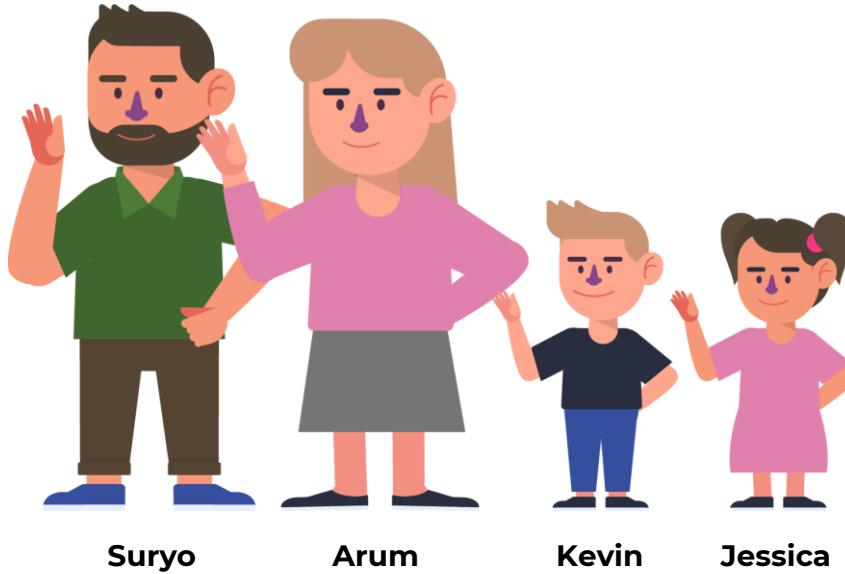
```
● ● ●  
// Add prototype-instance method  
Human.prototype.greet = function(name) {  
    console.log(`Hi, ${name}, I'm ${this.name}`)  
}  
  
// Add static method  
Human.destroy = function(thing) {  
    console.log(`Human is destroying ${thing}`)  
}  
  
// Instantiation of Human class, we create a new object.  
let mj = new Human("Michael Jackson", "Isekai");  
console.log(mj);  
// Output: Human {name: "Michael Jackson", address: "Isekai"}  
// Checking instance of class  
console.log(mj instanceof Human) // true  
console.log(mj.introduce())  
// Hi, my name is Michael Jackson  
console.log(mj.greet("Donald Trump"));  
// Hi, Donald Trump, I'm Michael Jackson  
console.log(Human.destroy("Amazon Forest"))  
// Human is destroying Amazon Forest
```



Setelah kita belajar function, class dan object, sekarang kita melangkah ke jenjang berikutnya, yaitu OOP!

Kita bakal bahas tentang Inheritance, Encapsulation, Abstraction, dan Polymorphism.

Yuk, kita gass ~



Inheritance

Inheritance itu semacam konsep pewarisan yang kita terapkan di OOP. Lihat dulu analogi ini yuk !

Perkenalkan, ini keluarga Suryo.

Suryo adalah seorang suami dan ayah.

Arum adalah seorang istri dan ibu

Kevin adalah anak laki-laki

Jessica adalah anak perempuan



Suryo

Arum

Kevin

Jessica

Suryo

- Kulit coklat
- Mata bulat
- Rambut warna coklat tua
- Suka pakai celana berwarna coklat
- Suka pakai baju berwarna hijau

Arum

- Kulit putih
- Mata bulat
- Rambut warna coklat muda
- Suka pakai rok berwarna abu
- Suka dengan baju berwarna pink

Kevin

- **Kulit putih**
- **Mata bulat**
- **Rambut warna coklat muda**
- Suka pakai celana berwarna biru
- Suka pakai sepatu warna hitam

Jessica

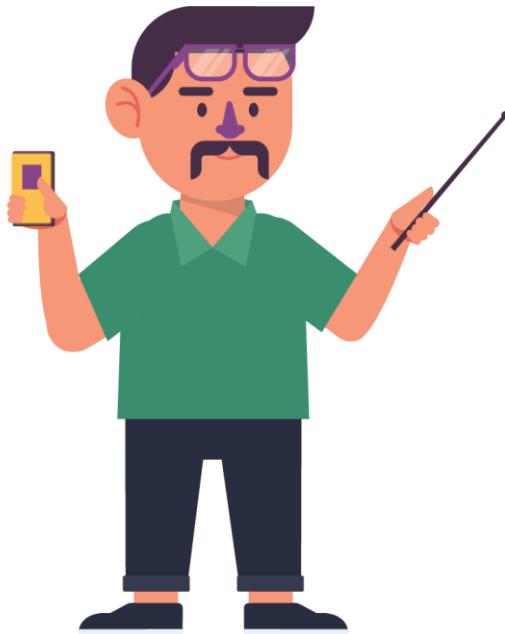
- **Kulit coklat**
- **Mata bulat**
- **Rambut warna coklat tua**
- Suka pakai baju berwarna pink
- Suka mengikat rambut



Kevin dan Jessica masing-masing punya ciri-ciri yang diwariskan dari orang tua, tapi ada juga ciri-ciri yang muncul karena keunikan mereka masing-masing.

Contohnya nih,

Ciri Kevin dan Jessica yang bermata bulat dan berambut coklat merupakan ciri yang diwariskan dari Suryo dan Arum sebagai orang tua mereka. Tapi ada juga ciri dari Kevin dan Jessica yang nggak dimiliki sama Suryo dan Arum, yaitu gimana cara mereka berpakaian.



Analogi tadi nyambung banget nih sama konsep terminologi pada inheritance!

- **Super class atau Parent class**

Class yang semua fiturnya diwariskan pada class turunannya.

- **Sub-class atau Child class**

Class turunan yang mewarisi semua fitur dari class lain. Sub-class bisa menambah field dan method-nya sendiri sebagai tambahan dari class yang ngasih warisan.

- **Reusability**

Pas kita mau bikin class baru dan udah ada class yang berisi kode yang kita mau, kita bisa kok menurunkan class baru itu dari class yang udah ada. Jadi, kita pake lagi fitur dari class yang udah ada, misalnya method.



```
● ● ●

class Human {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }

  introduce() {
    console.log(`Hi, my name is ${this.name}`)
  }

  work() {
    console.log("Work!")
  }
}
```

Class Human sebagai parent class atau super class

Memiliki:

- Attribute : nama, alamat
- Constructor dengan parameter
- Ada method buat menampilkan attribute



```
● ● ●

// Create a child class from Human
class Programmer extends Human {
  constructor(name, address, programmingLanguages) {
    super(name, address)
    /* Call the super/parent class constructor,
    in this case Person.constructor; */
    this.programmingLanguages = programmingLanguages;
  }
  introduce() {
    super.introduce();
    // Call the super class introduce instance method.
    console.log(`I can write `,
    this.programmingLanguages);
  }
  code() {
    console.log(
      "Code some",
      this.programmingLanguages[
        Math.floor(Math.random() *
      this.programmingLanguages.length)
      ]
    )
  }
}
```

Class Programmer sebagai child class atau sub class

Syntax **extends** nunjukin bahwa **class Human merupakan class Parent** dari class Programmer.

Memiliki:

- Attribute : programmingLanguage
- Constructor dengan parameter
- Ada method buat menampilkan attribute



Sekarang kita coba lakukan instance atau membuat objek baru dari kedua class tersebut dan pake method yang tersedia di class tersebut.

```
● ● ●

// Initiate from Human directly
let Obama = new Human("Barrack Obama", "Washington DC");
Obama.introduce() // Hi, my name is Barack Obama

let Isyana = new Programmer("Isyana", "Jakarta", ["Javascript", "Kotlin", "Python"]);
Isyana.introduce() // Hi, my name is Isyana; I can write ["Javascript", "Kotlin", "Python"]
Isyana.code() // Code some Javascript/Ruby/...
Isyana.work() // Call super class method that isn't overrided or overloaded

try {
  // Obama can't code since Obama is an direct instance of Human, which don't have code method
  Obama.code() // Error: Undefined method "code"
}
catch(err) {
  console.log(err.message)
}

console.log(Isyana instanceof Human) // true
console.log(Isyana instanceof Programmer) // true
```

Setelah tahu konsep Inheritance, kita perlu tahu juga nih kalau sebenarnya ada dua metode untuk implementasinya

1. Overriding method

Overriding method artinya mengesampingkan atau mengabaikan. Mirip-mirip kan sama kamu yang suka nyuekin dia?~

Tapi, kalau yang ini maksudnya, pas kita ganti method dari super class buat diimplementasi ulang di sub class, hal itu nggak bakal mengubah parameter yang udah didefinisikan sama super class-nya. Jadi, yaa diabaikan gitu deh~



1. Overloading method

Overloading method ini sama kayak overriding method, tapi di dalam overload ini, kita mengubah definisi parameter dari super class.

Maksudnya, nama method yang kita pake sama dengan nama method di super class-nya, tapi parameter yang ada di subclass-nya berbeda. Begitu, bosque~





```
● ● ●
class Person {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  introduce() {
    console.log(`Hi, my name is ${this.name}`)
  }
}
// Create a child class from Person
class Programmer extends Person {
  constructor(name, address, programmingLanguages) {
    super(name, address)
    // Call the super/parent class constructor, in this case Person.constructor;
    this.programmingLanguages = programmingLanguages;
  }
  // Override the Introduce Method
  introduce() {
    super.introduce(); // Call the super class introduce instance method.
    console.log('I can write ', this.programmingLanguages);
  }
  code() {
    console.log("Code some",this.programmingLanguages[Math.floor(Math.random * this.programmingLanguages.length)])
  }
}
let Isyana = new Programmer("Isyana Karina", "Jakarta", ["Javascript", "Python"]);
Isyana.introduce()
// Hi, my name is Isyana; I can write ["Javascript", "Python"]
```

Disamping adalah **contoh kode overriding method** ya!

Class Programmer bisa melakukan override method dari class Person. Jadi, kalau kita manggil method introduce dari class Programmer, berarti yang bakal terpanggil adalah method introduce dari class Programmer, bukan dari class Person.

Soalnya, method introduce dari class Person udah di-override sama method introduce dari class Programmer.



Untuk **contoh kode overloading method**, bisa dilihat kalau kedua class punya nama method yang sama, yaitu introduce.

Tapi, method introduce di class Programmer punya parameter withDetail, sedangkan method introduce di class Person nggak punya parameter.

```
lass Person {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  introduce() {
    console.log(`Hi, my name is ${this.name}`)
  }
}
// Create a child class from Person
class Programmer extends Person {
  constructor(name, address, programmingLanguages) {
    super(name, address)
    // Call the super/parent class constructor, in this case Person.constructor;
    this.programmingLanguages = programmingLanguages;
  }
  // Overload the Introduce Method
  introduce(withDetail) {
    super.introduce(); // Call the super class introduce instance method.
    (Array.isArray(withDetail)) ?
      console.log(`I can write ${this.programmingLanguages}`) : console.log("Wrong input")
  }
  code() {
    let acak = Math.floor(Math.random() * this.programmingLanguages.length)
    console.log("Code some", this.programmingLanguages[acak])
  }
}
let Isyana = new Programmer("Isyana Karina", "Jakarta", ["JavaScript", "Kotlin"]);
Isyana.introduce(["JavaScript"])
// Hi, my name is Isyana; I can write ...
//Isyana.introduce("JavaScript") // Hi, my name is Isyana; Wrong Input
//Isyana.introduce(1) // Hi, my name is Isyana; Wrong Input
Isyana.code() //Code some ...
```

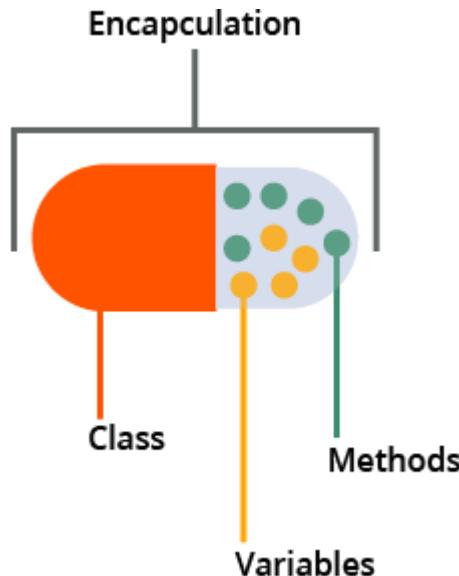


Encapsulation

Misalnya gini, pas kita pake mesin ATM buat narik atau setor uang, kita nggak tahu proses yang ada di dalam mesin itu kan?

Kita taunya pas kita memasukkan kartu ATM, PIN, memilih nominal uang yang bakal diambil, terus uang bakal keluar sesuai sama nominal yang kita pilih. Di dalam mesin ATM itu sebenarnya ada teknik enkapsulasi yang berjalan dan nggak diketahui sama nasabah.

Kira-kira itulah yang terjadi di JavaScript dengan teknik encapsulation. **Method atau variable pakai visibility yang private supaya class lain nggak bisa akses variable atau method di dalam class tersebut.**



Encapsulation itu ibaratnya kayak kapsul. Jadi, **data kita bisa disembunyikan dengan suatu cara yang namanya visibility**, tujuannya biar method dan variable nggak bisa diakses secara langsung dari luar class.

Yang kita sembunyikan itu bisa berupa prosedur atau property yang sifatnya sensitif, jadi nggak boleh diubah seenaknya. Encapsulation bisa meminimalisir terjadinya bug karena kita secara eksplisit bilang bahwa method/ property ini nggak boleh dipanggil di luar kelas deklarasi.

Oh iya, visibility itu ada 3 macam ya, **public, private, and protected**.



Public

Suatu visibility level, dimana kalau kita mendefinisikan suatu method atau property secara publik, artinya **method/ property itu bisa dipanggil di luar deklarasi kelas.**

Kamu bisa liat contohnya disamping ini yaa~



```
class Human {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  // This is public instance method
  introduce() {
    console.log(`Hello, my name is ${this.name}`);
  }
  // This is public static method
  static isEating(food) {
    let foods = ["plant", "animal"];
    return foods.includes(food.toLowerCase());
  }
}

let mj = new Human("Michael Jackson", "Isekai");
console.log(mj)
// Output: Human {name: "Michael Jackson", address: "Isekai"}
console.log(mj.introduce());
console.log(Human.isEating("Plant")) // true
console.log(Human.isEating("Human")) // false
```



```
● ● ●  
class Human {  
  constructor(name, address) {  
    this.name = name;  
    this.address = address;  
  }  
  #doGossip = () => {  
    console.log(`My address will become viral ${this.address}`)  
  }  
  talk() {  
    console.log(this.#doGossip()); // Call the private method  
  }  
  static #isHidingArea = true;  
}  
  
let mj = new Human("Michael Jackson", "Isekai");  
console.log(mj.talk()) // Will run, won't return error!  
// Output: My address will become viral Isekai  
try {  
  Human.#isHidingArea // Will return an error!  
  mj.#doGossip() // Won't run, will return error!  
}  
catch(err) {  
  console.error(err)  
}  
// Private field '#isHidingArea' must be declared in an  
enclosing class
```

Private

Suatu **method /property yang nggak bisa diakses di luar deklarasi class**. Artinya, kita nggak bisa akses method/property dari luar kurung kurawal class/scope dari kelas tersebut.

Sejak ES8 hadir di JavaScript buat mendeklarasikan suatu private method atau private property, kita bisa pake tanda pagar (#) sebelum nama class.

Ohya, private method ini juga nggak bisa berjalan di dalam class yang mewarisi class tersebut lho. Maksudnya, kalau kita bikin class Programmer extends dari class Human, berarti method/property private yang ada di class Programmer nggak bakal bisa berjalan.



Protected

Protected visibility ini **bisa kita akses di dalam sub class**. Ini yang jadi pembeda antara private dengan protected.

Walaupun sebenarnya belum ada implementasi secara spesifik buat protected visibility di JavaScript sekarang, tapi kita bisa melakukan duck typing buat ini.

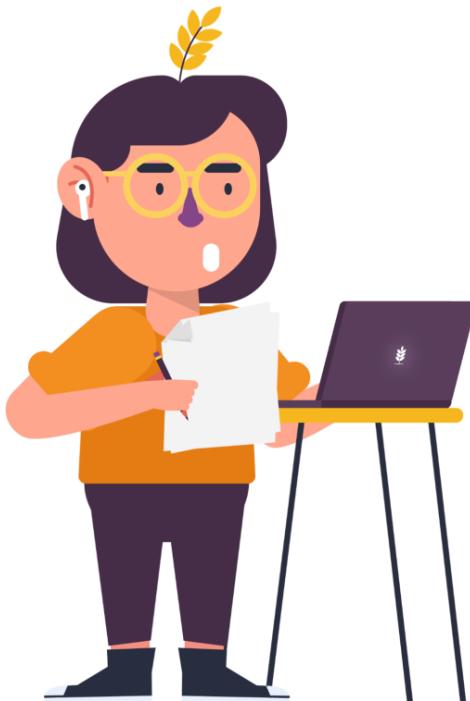
Kita bisa nambahin tanda “_” sebelum nama method/property buat ngasih tahu bahwa itu protected buat developer lain.

```
● ● ●

class Human {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  _call() {
    console.log(`Call me as a ${this.name}`)
  }
}

class Programmer extends Human {
  constructor(name, address, task, salary) {
    super(name, address);
    this.task = task;
    this.salary = salary;
  }
  doCall() {
    super._call() // Will run
  }
}

let sb = new Human("Sabrina", "Jakarta");
let job = new Programmer("Developer", "$1000");
console.log(sb._call()) // Call me as a Sabrina
/*Meskipun ini gak error ketika kita panggil protected secara public. Tapi, kita harus paham method ini protected, yang semestinya hanya boleh dipanggil di dalam class declaration atau sub-classnya.*/
console.log(job.doCall()) // Call me as a Developer
```



Sekarang kebayang kan kalau konsep **encapsulation ini identik sama visibility private?**

Hmmm... emang kenapa ya kita perlu “membungkus” data kita?

Kira-kira ini loh guys alasannya ~

- Meningkatkan **keamanan data**.
- Lebih **mudah mengontrol attribute dan method**.
- **Class bisa kita buat read-only atau write-only**.
- **Fleksibel**, maksudnya programmer bisa ganti sebagian dari kode tanpa harus takut berdampak pada kode yang lain.



Contoh disamping kita coba **implementasi Encapsulation buat menyembunyikan method #encrypt dan #decrypt.**

Kita nggak mau ada orang lain yang pake method tersebut di luar kelas deklarasi, karena hal itu berbahaya.

Waspadalah, Waspadalah~

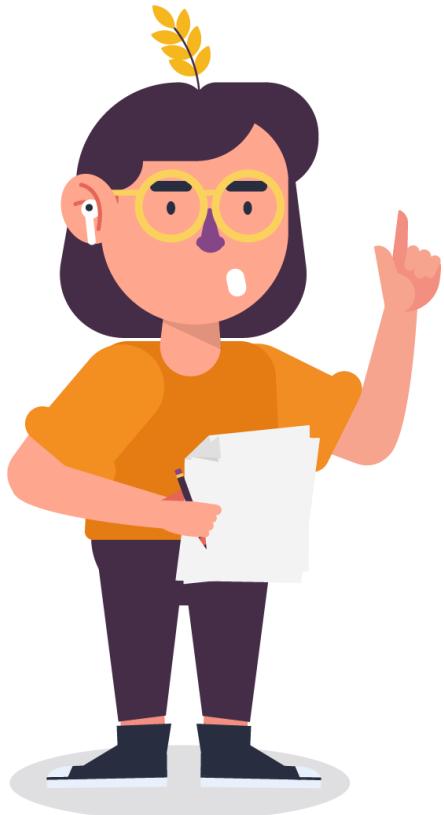
```
class User {
  constructor(props) {
    // props is object, because it is better that way
    let { email, password } = props; // Destruct
    this.email = email;
    this.encryptedPassword = this.#encrypt(password);
    // We won't save the plain password
  }
  // Private method
  #encrypt = (password) => {
    return `encrypted-version-of-${password}`
  }
  // Getter
  #decrypt = () => {
    return this.encryptedPassword.split(`encrypted-version-of-`)[1];
  }
  authenticate(password) {
    return this.#decrypt() === password; // Will return true or false
  }
}
let Bot = new User({
  email: "bot@mail.com",
  password: "123456"
})
const isAuthenticated = Bot.authenticate("123456");
console.log(isAuthenticated) // true
```



Abstraction

First of all, apa yang kamu bayangin pas dengar kata "orang"?

Mungkin dari sebagian jawaban, orang itu bisa dibayangkan sebagai polisi, hakim, tentara, dosen, atau bahkan yang jawaban lainnya.



Itulah yang bisa kita sebut sebagai abstraksi.

Kata “orang” sendiri masih bersifat abstrak, tapi kita bisa bayanganin konsep “orang” itu gimana. Kata “polisi, hakim, tentara, dan dosen” lebih konkret atau nyata kalau dibandingin sama “orang” yang masih abstrak banget.

Prinsip abstraksi ini juga ada dalam OOP dan kita sebenarnya sering menggunakaninya tanpa kita sadari.

Di JavaScript kita bisa bikin konsep OOP abstraksi ini.



Biar kebayang, kita lihat dulu contoh kode yang pake konsep abstraksi

Dicontoh ini ada dua class, yaitu Human dan Police.

Kita sengaja bikin kondisi buat abstract di class Human, biar ngasih pesan error kalo kita nggak sengaja menginstansiasi class Human.

```
● ● ●

class Human {
  constructor(props) {
    if (this.constructor === Human) {
      throw new Error("Cannot instantiate from
Abstract Class")
      // Because it's abstract
    }
    let { name, address } = props;
    this.name = name; // Every human has name
    this.address = address; // Every human has address
    this.profession = this.constructor.name;
    // Every human has profession, and let the child
    class to define it.
  }
  // Yes, every human can work
  work() {
    console.log("Working...")
  }
  // Every human can introduce
  introduce() {
    console.log(`Hello, my name is ${name}`)
  }
}
```



```
● ● ●  
  
class Police extends Human {  
  constructor(props) {  
    super(props);  
    this.rank = props.rank; // Add new property, rank.  
  }  
  
  work() {  
    console.log("Go to the police station");  
    super.work();  
  }  
  
  const Wiranto = new Police({  
    name: "Wiranto",  
    address: "Unknown",  
    rank: "General"  
  })  
  console.log(Wiranto.profession); // Police
```

Nah, ini terbukti ya!

Kalau kita coba melakukan instansiasi class Human, yang emang udah ada kondisi buat abstraction, berarti kita bakal dapet pesan **error**.

```
● ● ●  
  
try {  
  let Abstract = new Human({  
    name: "Abstract",  
    address: "Unknown"  
  })  
}  
catch(err) {  
  console.log(err.message)  
  // Cannot instantiate from Abstract Class  
}
```



Polymorphism

Polymorphism berarti bahwa **satu class bisa punya banyak wujud dari sub class-nya**. Biasanya sub class-nya punya perilaku yang beda banget dari super class-nya.

Prinsip ini berlaku pas kita punya banyak class yang terkait satu sama lain lewat inheritance.

Misalnya, kita punya 4 class, yaitu Human (abstract), Doctor, Police, Writer, and Army. Class tersebut punya aturan sebagai berikut:

- Doctor, Police, dan Army adalah sub class dari Human.
- Army dan Police punya method yang namanya **shoot**, tapi Dokter nggak.
- Doctor, Army, dan Police sama-sama punya method save buat menyelamatkan orang lain.





Biar kebayang, kita lihat dulu contoh kode yang pakai konsep polymorphism dari ketentuan sebelumnya, ya~

Pertama, kita buat class Human sebagai parent class. Terus, kita bikin module/helper buat public server dan military.

Nah, buat implementasi module/helper ini kita pake konsep **mix-ins**.

Mix-ins atau abstract subclasses ini ibarat suatu template buat class. Kita pake konsep mix-ins ini karena class di ECMAScript cuma bisa punya satu superclass. Sedangkan kita butuh fungsi dengan superclass sebagai input dan subclass yang memperluas superclass sebagai output.



```
class Human {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  introduce() {
    console.log(`Hi, my name is ${this.name}`)
  }
  work() {
    console.log(`${this.constructor.name}: "Working!"`)
  }
}

// Public Server Module/Helper
const PublicServer = Base => class extends Base {
  save() {
    console.log("SFX: Thank You!")
  }
}
// Military Module/Helper
const Military = Base => class extends Base {
  shoot() {
    console.log("DOR!")
  }
}
```



```
● ● ●

class Doctor extends PublicServer(Human) {
  constructor(props) {
    super(props);
  }
  work() {
    super.work(); // From Human Class
    super.save(); // From Public Server Class
  }
}

class Police extends PublicServer(Military(Human)) {
  static workplace = "Police Station";

  constructor(props) {
    super(props);
    this.rank = props.rank;
  }

  work() {
    super.work();
    super.shoot(); // From Military class
    super.save(); // From Public Server Class
  }
}
```

Terusssss, kita buat class yang lain.

Contoh kode buat class Doctor dan Police, kayak contoh berikut ini.

Di dalam class tersebut, kita coba panggil method dari class: Public Server dan Military.



Begini pula buat class Army dan Writer kayak contoh berikut ini.

Di dalam class tersebut, kita coba panggil method dari class: Public Server dan Military.

```
class Army extends PublicServer(Military(Human)) {
    static workplace = "Police Station";

    constructor(props) {
        super(props);
        this.rank = props.rank;
    }

    work() {
        super.work();
        super.shoot(); // From Military class
        super.save(); // From Public Server Class
    }
}

class Writer extends Human {
    work() {
        console.log("Write books");
        super.work();
    }
}
```



```
/* Instantiate Military Based Class */
const Wiranto = new Police({
  name: "Wiranto",
  address: "Unknown",
  rank: "General"
})

const Prabowo = new Army({
  name: "Prabowo",
  address: "Undefined",
  rank: "General"
}

/* -----Instantiate Doctor----- */
const Boyke = new Doctor({
  name: "Boyke",
  address: "Jakarta"
}

/* -----Instantiate Writer----- */
const Dee = new Writer({
  name: "Dee",
  address: "Bandung"
})
```

Terus kita coba lakukan instantiate, maksudnya kita bikin objek baru dari masing-masing class.



Terakhir, kita coba tes setiap objek dari empat class yang udah kita bikin pake beberapa method buat menghasilkan keluaran/output berdasarkan fungsinya masing-masing.

Beginilah konsep polymorphism!~

```
Wiranto.shoot(); // DOR!
Prabowo.shoot(); // DOR!

Wiranto.save() // SFX: Thank You!
Prabowo.save() // SFX: Thank You!
Boyke.save() // SFX: Thank You!

Wiranto.work()
// Police: Working! DOR! SFX: Thank You!
Prabowo.work()
// Army: Working! DOR! SFX: Thank You!
Boyke.work()
// Doctor: Working! SFX: Thank You!
Dee.work()
// Write books. Writer: Working!
```

Nah, biar kalian makin manteb dapet gambaran tentang OOP, Sabrina udah nyiapin repository yang mana di dalem situ ada implementasi OOP-nya. Cekidot <https://github.com/fnurhidayat/stunning-tribble>

Object Oriented Programming



Pemrograman berorientasi objek merupakan paradigma pemrograman berdasarkan konsep "objek", yang dapat berisi data, dalam bentuk field atau dikenal juga sebagai atribut; serta kode, dalam bentuk fungsi/prosedur atau dikenal juga sebagai method.

Jangan lupa buka konsol hehe. 😊

Buka main.js

Buka post.js

Buka user.js

Buka record.js



Saatnya kita Quiz!





1. Keuntungan penulisan kode dengan paradigma OOP adalah.

- A. Kode lebih mudah dimodifikasi
- B. Kode terproteksi agar tidak bisa di-debug
- C. Kode mudah diatur berdasarkan keinginan *user*



1. Keuntungan penulisan kode dengan paradigma OOP adalah

- A. Kode lebih mudah dimodifikasi
- B. Kode terproteksi agar tidak bisa di-debug
- C. Kode mudah diatur berdasarkan keinginan user

Keuntungan penulisan kode dengan paradigma OOP adalah kode kita menjadi lebih gampang dikelola, dimodifikasi, dan di-debug.



2. Di bawah ini adalah cara yang tepat untuk mendeklarasikan sebuah fungsi

A.

```
const luasLapangan = (p, l) {  
    return p * l  
}
```



B.

```
function luasLapangan(p, l) {  
    return p * l  
}
```



C.



```
const luasLapangan (p, l) => { return p * l }
```



2. Di bawah ini adalah cara yang tepat untuk mendeklarasikan sebuah fungsi.

A.

```
const luasLapangan = (p, l) {  
    return p * l  
}
```

B.

```
function luasLapangan(p, l) {  
    return p * l  
}
```

C.



```
const luasLapangan (p, l) => { return p * l }
```

Jawabannya adalah B.

Fungsi tersebut dideklarasikan dengan keyword *function* ES5 (*function declaration*).



3. Konsep OOP apa yang diterapkan pada kode ini

- A. Abstraction
- B. Encapsulation
- C. Inheritance

```
class Person {  
    constructor(name) {  
        this.name = name;  
    }  
    introduce() {  
        console.log(`Hi, my name is ${this.name}`)  
    }  
}  
  
class Programmer extends Person {  
    constructor(name, programmingLanguages) {  
        super(name)  
        this.programmingLanguages = programmingLanguages;  
    }  
  
    introduce() {  
        super.introduce();  
        console.log(`I can write ` + this.programmingLanguages);  
    }  
}  
let Isyana = new Programmer("Isyana Karina", ["Javascript", "Python"]);  
Isyana.introduce()  
// Hi, my name is Isyana; I can write ["Javascript", "Python"]
```



3. Konsep OOP apa yang diterapkan pada kode ini

- A. Abstraction
- B. Encapsulation
- C. Inheritance

Kode pada gambar tersebut menerapkan konsep OOP Inheritance: *overriding method* karena kita menggunakan method yang sama dengan *super class*-nya, tanpa mengubah parameternya.

```
class Person {  
    constructor(name) {  
        this.name = name;  
    }  
    introduce() {  
        console.log(`Hi, my name is ${this.name}`)  
    }  
}  
  
class Programmer extends Person {  
    constructor(name, programmingLanguages) {  
        super(name)  
        this.programmingLanguages = programmingLanguages;  
    }  
  
    introduce() {  
        super.introduce();  
        console.log(`I can write ` + this.programmingLanguages);  
    }  
}  
  
let Isyana = new Programmer("Isyana Karina", ["Javascript", "Python"]);  
Isyana.introduce()  
// Hi, my name is Isyana; I can write ["Javascript", "Python"]
```



4. Konsep OOP apa yang diterapkan pada kode ini

- A. Polymorphism
- B. Abstraction
- C. Inheritance

```
class Doctor extends PublicServer(Human) {  
    constructor(props) {  
        super(props);  
    }  
    work() {  
        super.work(); // From Human Class  
        super.save(); // From Public Server Class  
    }  
  
class Police extends PublicServer(Military(Human)) {  
    static workplace = "Police Station";  
  
    constructor(props) {  
        super(props);  
        this.rank = props.rank;  
    }  
  
    work() {  
        super.work();  
        super.shoot(); // From Military class  
        super.save(); // From Public Server Class  
    }  
}
```



4. Konsep OOP apa yang diterapkan pada kode ini

- A. Polymorphism
- B. Abstraction
- C. Inheritance

Kode pada gambar tersebut menerapkan OOP Polymorphism. Yang berarti bahwa satu class dapat memiliki banyak wujud dari sub class-nya. Dimana doctor dan police merupakan extensi dari publicserver superclass

```
class Doctor extends PublicServer(Human) {  
    constructor(props) {  
        super(props);  
    }  
    work() {  
        super.work(); // From Human Class  
        super.save(); // From Public Server Class  
    }  
  
class Police extends PublicServer(Military(Human)) {  
    static workplace = "Police Station";  
  
    constructor(props) {  
        super(props);  
        this.rank = props.rank;  
    }  
  
    work() {  
        super.work();  
        super.shoot(); // From Military class  
        super.save(); // From Public Server Class  
    }  
}
```



5. Dari pilihan berikut, manakah yang bukan merupakan macam dari visibility

- A. Protected
- B. Prioritize
- C. Public



5. Dari pilihan berikut, manakah yang bukan merupakan macam dari visibility

- A. Protected
- B. Prioritize
- C. Public

Visibility itu memiliki 3 macam yaitu, public, private, and protected.

Referensi dan bacaan lebih lanjut~

1. https://www.w3schools.com/js/js_objects.asp
2. https://www.w3schools.com/js/js_object_constructors.asp
3. https://www.w3schools.com/js/js_object_methods.asp
4. <https://medium.com/easyread/penerapan-oop-dalam-javascript-part-1-98ed3a427e77>
5. <https://medium.com/easyread/penerapan-oop-dalam-javascript-part-2-822e6c4c53c8>
6. <https://javascriptissexy.com/oop-in-javascript-what-you-need-to-know/>





Nah, selesai sudah pembahasan kita di chapter 4 Topic 1 ini. Selanjutnya, kita akan belajar tentang DOM!



Terima Kasih!



Next Topic

loading...