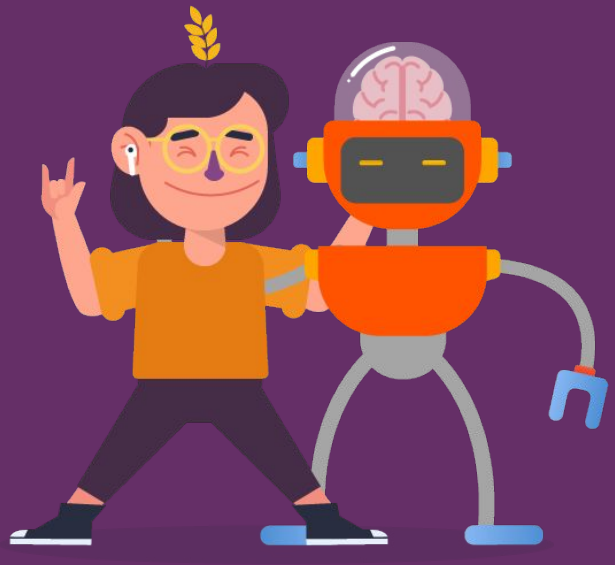
OOP di JavaScript

Gold - Chapter 4 - Topic 2

Ini adalah Topik Kedua di Chapter 4 *online course* Full-Stack Web dari Binar Academy! *Let's rock*!



Di **Chapter 4 Topic 2** ini kita akan mempelajari tentang paradigma ***Object Oriented Programming*** **(OOP)** dan gimana menerapkannya di Javascript.

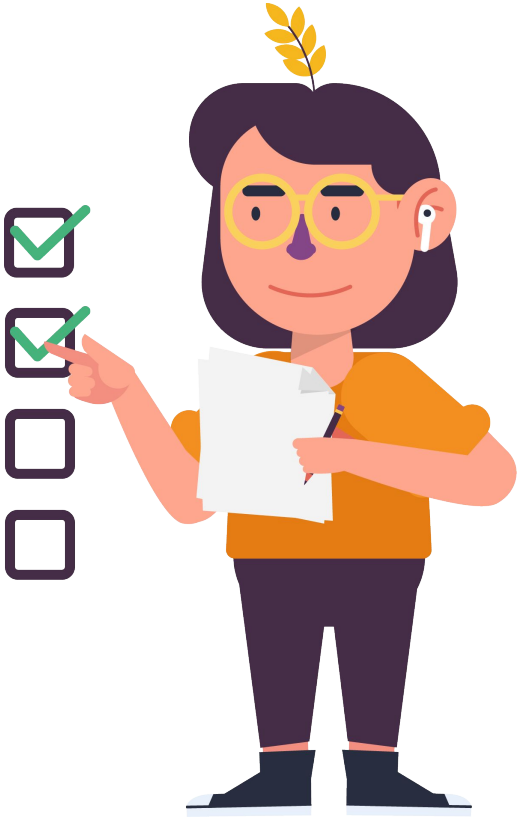
Tapi, sebelum belajar tentang OOP, ada tiga hal yang perlu kita bahas dulu nih, yaitu: ***function***, ***class*** dan ***method***.

Nah, setelah memahami ketiga hal itu nanti kita akan lebih mudah menerapkan 4 pilar atau konsep OOP antara lain:

1. ***Inheritance***
2. ***Encapsulation***
3. ***Abstraction***
4. ***Polymorphism***

Dari sesi ini, harapannya kamu bisa mendapatkan beberapa hal yang di antaranya sebagai berikut!

1. Memahami konsep paradigma pemrograman OOP di JavaScript
2. Memahami cara menggunakan *function*, *class* dan *method* di JavaScript
3. Menerapkan konsep OOP di JavaScript, seperti *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?

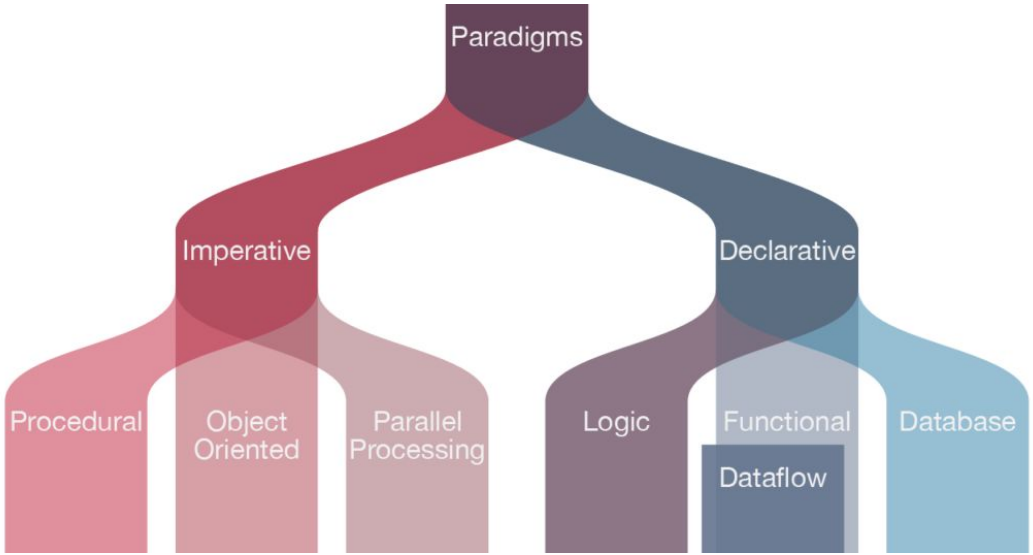
Hmm... pasti di antara kalian ada yang bertanya-tanya begitu ya?

Bagusss! Kita memang perlu kenalan lebih jauh dengan konsep OOP dan cari tahu konsep paradigma di pemrograman~

Menulis program itu sama aja seperti kalau kita menulis puisi, gayanya bisa berbeda-beda.

Paradigma Pemrograman itu bisa dikatakan suatu konsep atau kerangka berpikir yang dapat kita gunakan untuk menyelesaikan masalah dengan menggunakan bahasa pemrograman.

Nah, OOP ini termasuk jenis paradigma pemrograman yang umum dipakai di JavaScript. **Kenapa?** Karena ia adalah ***Object Oriented Programming,*** atau paradigma dengan gaya penulisannya yang berorientasi objek.

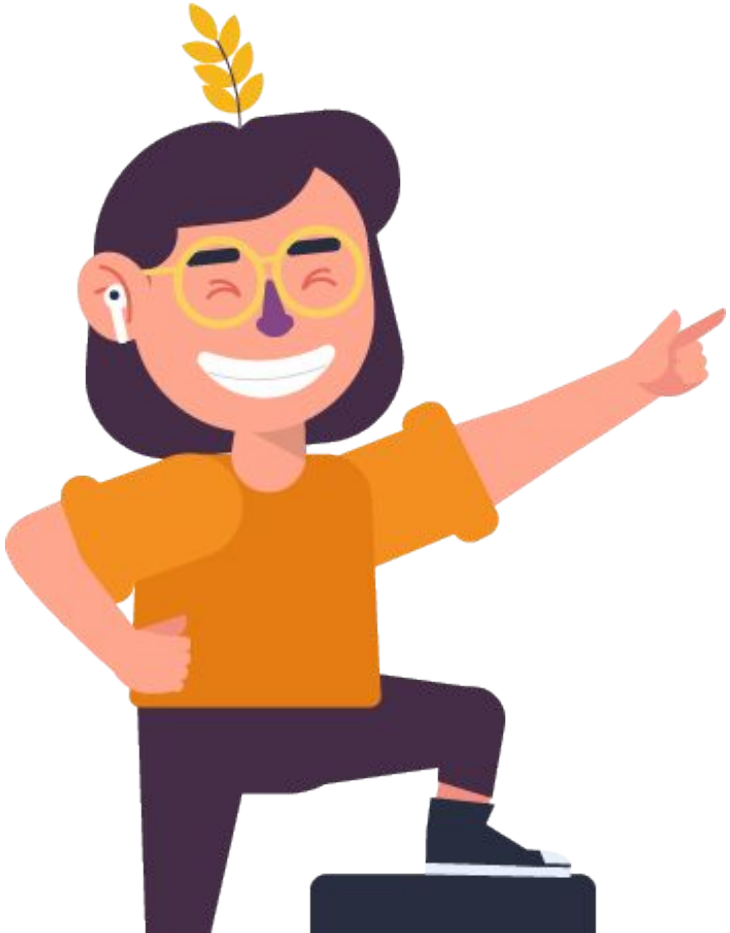


Sebenarnya paradigma pemrograman itu gak cuma OOP.

Paradigma lain yang sering digunakan dalam pemrograman, yaitu:

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

Tapi, di sesi ini kita hanya akan fokus mempelajari OOP saja.



Apa alasannya? Karena paradigma ini dipakai di JavaScript. Ini bakal berguna ketika kita ingin membuat *project* dengan *library* JavaScript, yaitu **React**.

Selain itu, penulisan kode dengan paradigma OOP ini juga akan memudahkan kita. **Mengapa?** karena strukturnya jelas dan bisa menjaga kode kita biar gak **DRY** alias ***Don't Repeat Yourself***.

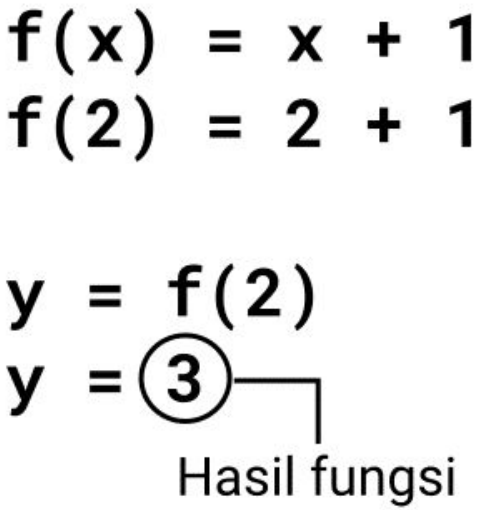
Dengan begitu, kode kita jadi lebih gampang dikelola, dimodifikasi, dan di-*debug*.

Sebelum bahas OOP di JavaScript,
kita harus tahu dulu cara pakai
function, class dan ***method***

Di topik sebelumnya,
kita udah bahas gimana cara pakai variabel dan *object*,
masih ingat kan?

Yuk, kita coba *flashback* dulu!

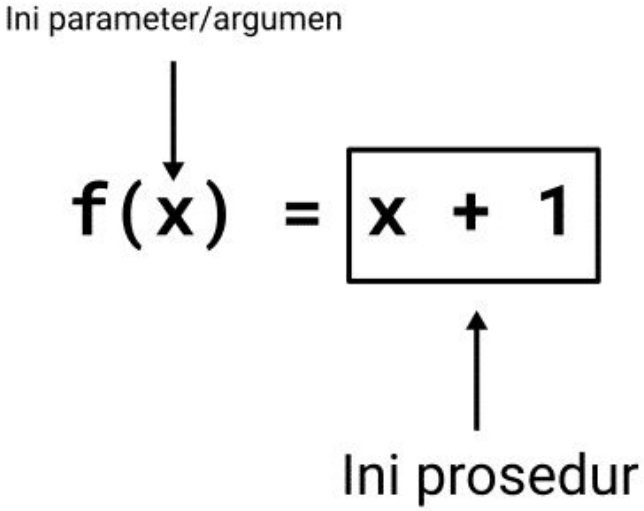
Masih ingat gak waktu zaman dulu kita belajar matematika dasar? Kita udah belajar yang namanya fungsi kan? Untuk menyegarkan ingatan kalian, lihat contoh fungsi di samping deh...



Nah, fungsi di dalam bahasa pemrograman juga nggak jauh beda dengan fungsi di pelajaran matematika.

Pada dasarnya, fungsi terdiri dari 3 hal, yaitu:

- Parameter
- Procedure
- Return value (hasil)



Dari contoh fungsi matematika yang sebelumnya, kalau kita jabarkan jadi fungsi di bahasa pemrograman, maka akan jadi seperti berikut ini.

f(x) adalah sebuah fungsi yang akan meminta x sebagai parameter. Nah, **x** di sini ditugaskan untuk menjadi variabel dari parameter.

Maksudnya, ketika kita memanggil fungsi tersebut, maka **x** bisa kita ganti dengan berbagai nilai, dan akan dimasukkan ke dalam fungsi sebagai parameter.

Sekarang kalo kita mau menulis fungsi di JavaScript, maka ada dua *keywords* yang akan kita pakai, yaitu:

- ***function***: untuk mendeklarasikan fungsi dan di dalam *scope function,* tentunya akan ada kode/perintah yang akan dieksekusi.

  Pada contoh, “diskon” sebagai nama fungsi untuk kode yang akan dieksekusi dan siap ditampung ke dalam variabel *let* “musimPandemik”.

- ***return***: untuk memberikan/mengembalikan hasil dari fungsi tersebut. Jika setelah *keyword return* kosong, maka hasil dari fungsi tersebut akan dianggap *undefined*.

```
function diskon(x) {
 let musimPandemik = (x * 30)/100
 return musimPandemik
}

let sale = diskon(20000)
console.log(sale) // Output: 6000
```

Nah, ada hal yang harus kita perhatikan nih mengenai penulisan parameter. Kita bisa *input* tipe data apa pun. Syarat, data yang kita pakai **harus relevan** dengan fungsinya.

Misalnya, prosedur fungsi kita akan memproses parameter yang dianggap sebagai *String*, maka kita wajib memasukkan nilai dengan tipe data *String* juga.

Kalau nilai yang kita *input* tipe datanya nggak sama, jadinya bakalan *error*. Jadi, 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
```

Oh, iya! kita dapat mendeklarasikan fungsi dengan cara yang berbeda, lho!

Biasanya kan kita mendeklarasikan sebuah fungsi dengan *keyword function* (*function declaration*).

Nah, *keyword function* juga bisa digunakan untuk mendefinisikan fungsi di dalam ekspresi dan nantinya disimpan di dalam sebuah variabel. Fungsi ini sebenarnya adalah **fungsi anonim** (*anonymous function*).

Kita juga bisa menggunakan *syntax* ES6 yaitu *arrow function* agar fungsi jadi lebih simpel, terutama jika hanya satu baris kita gak perlu menuliskan *keyword return*.

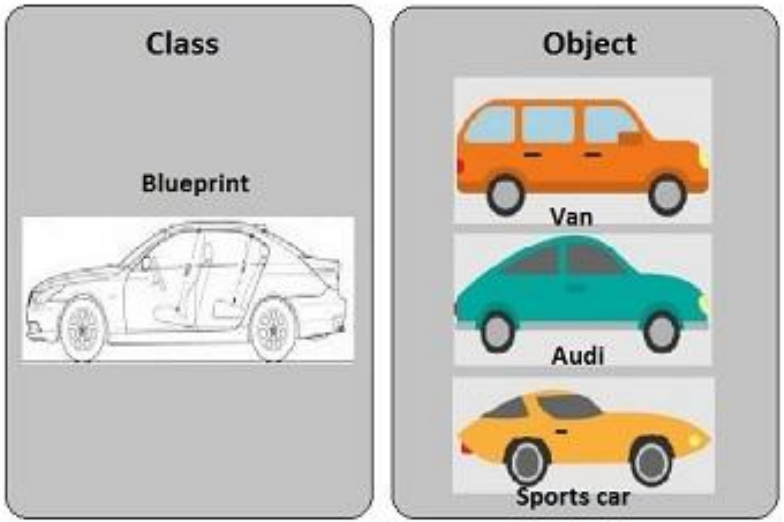
```
// Function Declaration (ES5)
function volTabung(r, t) {
 return 3.14 * r**2 * t
}
console.log('Volume Tabung:', volTabung(10, 4))
// Volume Tabung: 1256

// 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
```

```
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 ]
```

Sebuah fungsi juga bisa menerima fungsi lain sebagai parameternya, yang dikenal dengan istilah ***Higher Order Function***.

Sebagai contoh adalah method ***forEach*** yang kita pakai untuk melakukan *loop* di dalam *array*.

Nah, fungsi *forEach* ini menerima fungsi sebagai parameter, secara spesifik di parameter yang bernama *callback*.



Setelah memahami cara penggunaan fungsi, kita akan memahami ***class*** dan ***object*** serta hubungan antara keduanya. Nah, bisa lihat dulu contoh gambar di samping ini, ya!

Misalnya, kita punya *blueprint* mobil. Nah, *class* ini 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 *blueprint* mobil ini, kita bisa mewujudkan berbagai jenis mobil. Ada yang akan kita buat jadi mobil van, mobil audi, dan mobil *sport*.

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

Untuk mendeklarasikan suatu *class*, kita bisa pakai *keyword class* lalu diikuti dengan nama kelasnya.

Oke, kita coba deklarasi sebuah *class*, yuk! Kita beri nama jadi class “Person”, seperti contoh berikut ini.

Mungkin kalian ada yang bingung, **“bikin *object* dengan cara biasa atau *object literal* kan juga bisa?”**

Iya sih, bisa-bisa aja. Tapi, kita bakalan repot kalau mau mengubah atau menambahkan *property* baru di *object* tersebut. **Mengapa?** karena kita harus menulis ulang *object* tersebut satu per satu.

Bayangkan jika jumlah objeknya ada ratusan dan seterusnya! Weleh-weleh... capek deh... hehe...

Sebenarnya *class* kita gunakan untuk membuat tipe data baru, yang nantinya data ini berupa *object*.

Nah, ada beberapa hal yang perlu kita ketahui tentang penggunaan *class* nih. Langsung maju ke slide berikutnya yaa~

Constructor

Constructor adalah sebuah fungsi yang membuat *instance* dari sebuah *class,* yang biasanya disebut "*object*".

Di JavaScript, *constructor* dipanggil saat kita mendeklarasikan *object* menggunakan kata kunci ***new***. Tujuan menggunakan *constructor* ini untuk membuat *object* dan menetapkan nilai jika terdapat *object property*.

```
constructor(){

}
```

Ini adalah cara yang rapi untuk membuat *object* karena kita gak perlu secara eksplisit menyatakan apa yang harus dikembalikan sebagai fungsi *constructor*, secara *default* akan mengembalikan *object* yang dibuat di dalamnya.

Kalau kita gak menentukan *constructor* apa pun, *constructor default* dibuat otomatis tanpa memiliki parameter, seperti contoh ini:

Property

Hanyalah data dari suatu *object* atau variabel-variabel yang ada di *object*, biasanya disebut juga dengan atribut. Setiap *object* tentunya memiliki *property*, misalnya manusia bernama Sabrina adalah *object*, maka dia punya beberapa *property,* seperti nama, alamat, dsb.

Oh, iya! Jika kita buat object tanpa class, maka object itu kan hanya memiliki satu tipe property aja. Sedangkan kalau kita membuat dengan class, maka akan ada dua tipe property, yaitu:

- *Instance property,* yang berarti sebuah *property* dapat kita akses setelah *object* kita *instantiate* (dibuat melalui *keyword new*).
- *Static property*, yang berarti nilainya akan selalu sama di semua *instance* dari *class* tersebut.

Method

Suatu fungsi atau aksi dari suatu object. Kalau kita punya class Human, maka method itu ibarat aksi yang bisa dilakukan manusia seperti berjalan, berbicara, dsb.

Di dalam method, kita bisa mengakses property dari object dengan keyword this, yang artinya adalah memanggil object itu sendiri. Sama seperti property, method memiliki dua tipe dalam hal aksesnya yaitu Instance method (prototype) dan Static method.

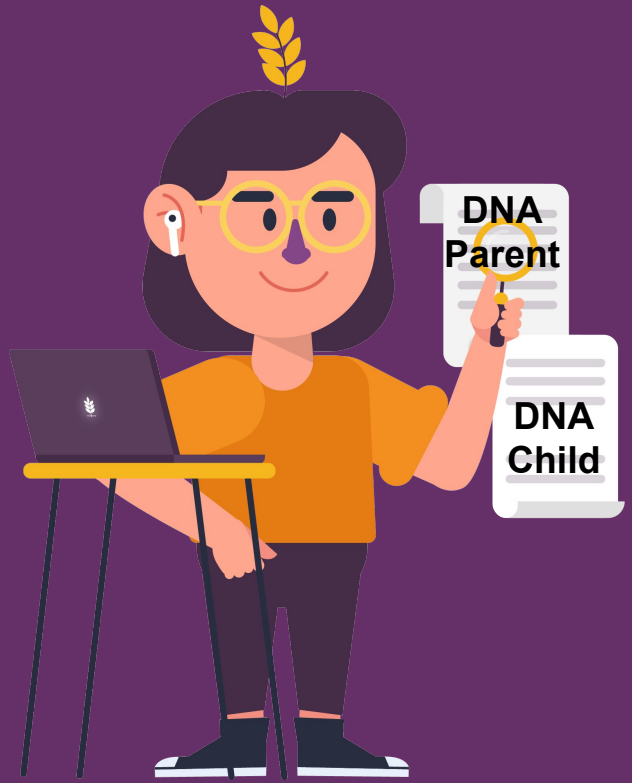
- *Instance method, yang berarti kita hanya bisa memanggil method instance setelah object kita instantiate (dibuat melalui keyword new).*
- *Static method, yang berarti nilainya akan selalu sama di semua instance dari class tersebut.*

Untuk lebih jelasnya bisa lihat contoh pembuatan *class* dan cara aksesnya.

```
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
```

Mari kita mulai menerapkan
konsep OOP di JavaScript



Oke, pertama kita akan bahas tentang *inheritance*.

Inheritance itu semacam konsep pewarisan yang kita terapkan di OOP.

Cara kerja *inheritance* sama kaya DNA yang diwariskan dari orang tua ke anak-anaknya.

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

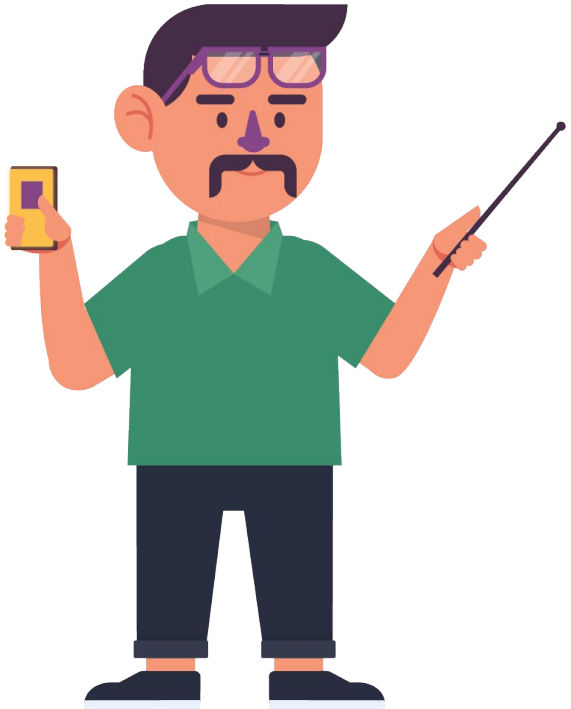
Jessica

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

Perkenalkan, ini keluarga Suryo.
Suryo sebagai suami dan ayah
Arum sebagai istri dan ibu
Kevin sebagai anak laki-laki
Jessica sebagai anak perempuan

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.

Sebagai contoh:
Ciri Kevin dan Jessica yang warna kuning adalah ciri yang diwariskan dari Suryo dan Arum sebagai orang tua mereka. Tetapi ada ciri Kevin dan Jessica yang ga dimiliki sama Suryo dan Arum.



Kamu perlu tahu beberapa terminologi yang ada di *Inheritance*:

Super class* atau *Parent class

class yang semua fiturnya diwariskan kepada *class* turunannya.

Sub-class* atau *Child class

class turunan yang mewarisi semua fitur dari *class* lain. *Sub-class* dapat menambah *field* dan *method*-nya sendiri sebagai tambahan dari *class* yang memberi warisan.

Reusability

Ketika kita ingin membuat *class* baru dan udah ada *class* yang berisi kode yang kita inginkan, kita bisa kok menurunkan *class* baru itu dari *class* yang udah ada. Dengan begitu, kita menggunakan kembali fitur dari *class* yang udah ada, misalnya *method*.

Terus, gimana penerapan ***inheritance*** di JavaScript?

```
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
- Terdapat *method* untuk 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 berarti *class* Human merupakan *class* Parent dari *class* Programmer.

Memiliki:

- *Attribute:* programmingLanguage
- *Constructor* dengan parameter
- Terdapat *method* untuk menampilkan *attribute*

Sekarang kita coba lakukan *instance* atau membuat objek baru dari kedua *class* tersebut dan menggunakan *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 kita mengetahui konsep inheritance, kita perlu mengetahui juga nih kalo sebenarnya ada dua method untuk konsep ini, yaitu:

Overriding method

Overriding method ini dari kata override yang artinya **mengesampingkan** atau **mengabaikan.**

Maksudnya, saat kita *replace* atau mengubah *method* dari *super class* untuk diimplementasi ulang di dalam *subclass*-nya, berarti nggak mengubah parameter yang sudah didefinisikan oleh super class-nya.

Overloading method

Overloading method ini sama seperti ***overriding method***. Tapi, di dalam overload ini, kita mengubah definisi parameter dari *super class*.

Maksudnya, nama method yang kita gunakan sama dengan nama method di *super class*-nya, tapi parameter yang ada di *subclass*-nya berbeda.

```
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"]
```

Dapat dilihat dari contoh kode ***Overriding method***, bahwa *class* Programmer bisa meng-*override method* dari *class* Person.

Sehingga jika kita memanggil *method* ***introduce*** dari class Programmer, akan terpanggil *method* ***introduce*** dari class Programmer, **bukan** dari *class* Person.

Karena method **introduce** dari class **Person** sudah di-override oleh method **introduce** dari class **Programmer.**

Nah, untuk contoh kode ***Overloading method***, bisa dilihat bahwa kedua *class* memiliki nama *method* yang sama, yaitu ***introduce***.

Namun, method ***introduce*** di *class* **Programmer** memiliki parameter ***withDetail***. Sedangkan, *method* ***introduce*** di *class* **Person** tidak memiliki parameter**.**

```
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;
  }
  // 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 ...
```

Oke, kita lanjutkan ya!

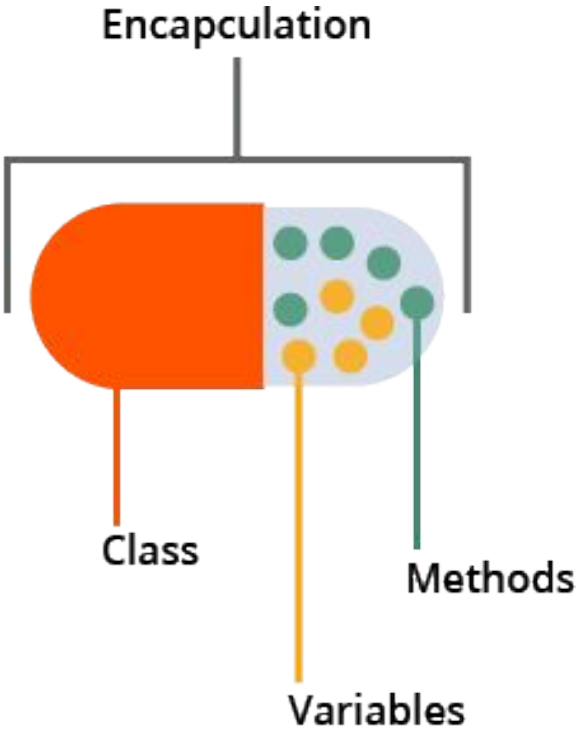
Tapi, ketika kita membahas **ENCAPSULATION**, kita juga bakal belajar tentang beberapa jenis ***visibility*** dalam deklarasi kelas, nih.

Maksudnya apa coba yaa?

Misalnya gini, saat kita pakai mesin ATM buat menarik atau menyetor uang, kita nggak tahu proses yang ada di dalam mesin itu, kan?

Kita hanya tahu memasukkan kartu ATM, memasukkan PIN, memilih nominal uang yang bakal diambil, uang akan keluar sesuai nominal yang kita pilih. Padahal, di dalam mesin ATM itu sebenarnya ada teknik enkapsulasi yang berjalan dan gak diketahui sama nasabah.

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



Dengan analogi yang sederhana, *encapsulation* (pembungkusan) itu ibaratnya seperti kapsul. Lihat contoh pada gambar.

Maksudnya *encapsulation*, data kita bisa disembunyikan dengan suatu cara yang dinamakan *visibility*, yang bertujuan biar *method* dan *variable* gak bisa diakses secara langsung dari luar *class*.

Yang kita sembunyikan itu bisa berupa prosedur atau *property* yang sifatnya sensitif, gak boleh diubah-ubah seenaknya.

Encapsulation bisa meminimalisir terjadinya *bug* karena kita secara eksplisit bilang bahwa *method*/*property* ini gak boleh dipanggil di luar kelas deklarasi.

Nah, jenis *visibility* yang dimaksud ada 3 macam, yaitu **public, private,** dan **protected**.

Kita bahas satu per satu yaa~

PUBLIC

Suatu *visibility level* di mana bila kita mendefinisikan suatu *method* atau *property* secara publik.

Artinya, *method/property* itu bisa dipanggil di luar deklarasi kelas.

```
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*. Ini berarti kita nggak bisa akses *method*/*property* dari luar kurung kurawal *class*/*scope* dari kelas tersebut.

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

Oh, iya! *Private method* ini juga nggak bisa berjalan di dalam *class* yang mewarisi *class* tersebut lho!

Maksudnya, kalo kita bikin *class* Programmer *extends* dari *class* Human, maka *method*/*property* *private* yang ada di *class* Programmer nggak bisa berjalan.

PROTECTED

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

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

Kita bisa menambahkan tanda _ sebelum nama *method/property* untuk memberi tahu bahwa itu *protected* untuk developer lain.

Oya, tanda "_" itu memang udah kaidahnya begitu (*convention*) untuk *protected* ya.

```
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
```

Nah, konsep *encapsulation* ini identik sama salah satu *visibility* yang sudah dibahas tadi, yaitu **PRIVATE**.

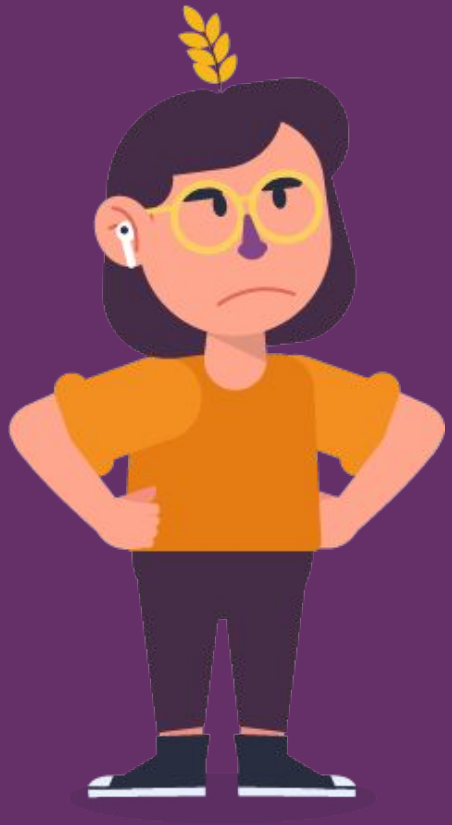
Jadi, **fungsi encapsulation** atau alasan data kita harus dibungkus antara lain:

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

Nah, pada contoh ini kita coba implementasi *Encapsulation* untuk menyembunyikan *method #encrypt* dan *#decrypt*.

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

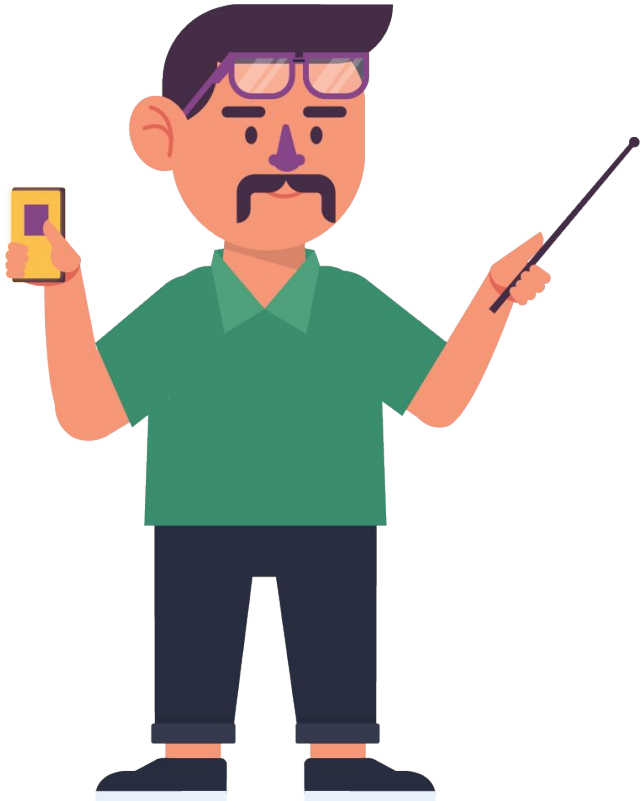
```
class User {
 constructor(props) {
   // props is object, because it is better that way
   let { email, password } = props; // Destruct
   this.email = email;
   this.encryptedPassword = this.#encrypt(password);
   // We won't save the plain password
 }
 // Private method
 #encrypt = (password) => {
   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
```



Selanjutnya, apa sih yang dimaksud dengan **Abstraction?**

First of all, apa yang kamu bayangkan pas dengar kata “orang”?

Mungkin kalian ada yang jawab, orang itu bisa dibayangkan sebagai polisi, hakim, tentara, atau dosen.



Ini yang kita sebut sebagai abstraksi.

Kata “orang” sendiri masih bersifat abstrak, tapi kita bisa membayangkan konsep “orang” itu gimana.

Polisi, hakim, tentara, dan dosen lebih konkrit atau nyata kalo dibandingin sama “orang” yang masih abstrak banget.

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

Nah, di JavaScript kita bisa bikin konsep OOP abstraksi ini.

Biar kebayang, kita lihat dulu contoh kode yang pakai konsep abstraksi.

Pada contoh ini ada dua *class*, yaitu "*Human*" dan "*Police*".

Kita sengaja buat kondisi untuk *abstract* di *class Human*, agar memberikan pesan *error* kalo kita gak 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!

Jika kita coba melakukan instansiasi *class Human*, yang mana sudah ada kondisi untuk *abstraction*, maka kita akan dapat pesan *error*.

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

Oke, terakhir kita bahas tentang konsep ***Polymorphism***.

Polymorphism memiliki arti bahwa satu *class* dapat memiliki banyak wujud dari *sub class*-nya. Biasanya *sub class*-nya memiliki perilaku yang sangat berbeda dari *super class*-nya.

Prinsip ini berlaku ketika kita punya banyak *class* yang terkait satu sama lain melalui *inheritance*.

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

- Doctor, Police dan Army adalah *sub class* dari Human.
- Army dan Police memiliki *method* bernama ***shoot***, tapi Dokter tidak.
- Doctor, Army dan Police sama-sama memiliki method *save* untuk menyelamatkan orang lain.

Biar kebayang, kita lihat dulu contoh kode yang pakai konsep *polymorhphism* dari ketentuan sebelumnya.

Pertama, kita buat *class* Human sebagai *parent class*. Kemudian, kita buat *module/helper* untuk *public server* dan *military*. Nah, untuk implementasi *module/helper* ini kita menggunakan konsep ***mix-ins***.

Mix-ins atau *abstract subclasses* ini ibarat suatu *template* untuk *class*. Kita pakai konsep *mix-ins* ini karena *class* di ECMAScript hanya dapat memiliki 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
 }
}
```

Selanjutnya, kita buat *class* yang lain.

Contoh kode untuk *class* Doctor dan Police seperti contoh berikut ini.

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

Begitu pula untuk *class* Army dan Writer seperti contoh berikut ini.

Di dalam *class* tersebut, kita coba memanggil *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"
})
```

Kemudian kita coba melakukan *instantiate*, maksudnya kita membuat objek baru dari masing-masing *class*.

Terakhir, kita coba tes setiap objek dari empat *class* yang sudah kita buat dengan menggunakan beberapa *method* untuk menghasilkan keluaran/*output* berdasarkan fungsinya masing-masing.

Begitulah 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!
```

Rangkuman

Paradigma Pemrograman
Suatu konsep atau kerangka berpikir atau *style* yang dapat kita gunakan untuk menyelesaikan masalah dengan menggunakan bahasa pemrograman.

OOP
Object Oriented Programming, program yang struktur kodenya berdasarkan *Object.*

Class
Class ini adalah suatu *blueprint* atau acuan untuk membuat suatu *object*.

Inheritance
Pada JavaScript dikenal dengan *syntax extends*. Digunakan untuk menggunakan kembali atribut ataupun *method* dari *super class*.

Encapsulation
Pembungkusan *object* agar lebih aman.

Abstraction
Class yang tidak bisa langsung dibuat *object*-nya.

Polymorphism
Memungkinkan *class* punya banyak bentuk *method* dengan nama yang sama. Juga memungkinkan instansiasi objek yang ditujukan pada *super class*.

BINAR

Saatnya

QUIZ

BINAR

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*

BINAR

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 }
```

BINAR

3

Kode pada gambar ini menerapkan konsep OOP?

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"]
```

BINAR

4

Kita bisa menyembunyikan *method #encrypt* dan *#decrypt* dengan konsep OOP ...

A. Polymorphism

B. Abstraction

C. Encapsulation

BINAR

5

```
let mj = new Human("Michael Jackson", "Isekai");
```

Kode di atas di dalam konsep OOP digunakan untuk ...

A. Inheritance

B. Instance method

C. Instantiation class

BINAR

Pembahasan

Quiz

BINAR

1

Keuntungan penulisan kode dengan paradigma OOP adalah ...

A. Kode lebih mudah dimodifikasi

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

BINAR

2

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

B

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

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

BINAR

3

Kode pada gambar menerapkan konsep OOP?

C. Inheritance

Kode pada gambar ini 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"]
```

BINAR

4

Kita bisa menyembunyikan *method #encrypt* dan *#decrypt* dengan konsep OOP ...

C. Encapsulation

Konsep encapsulation itu digunakan karena kita nggak mau ada orang lain yang menggunakan *method* tersebut di luar kelas deklarasi, karena hal itu bisa berbahaya.

BINAR

5

```
let mj = new Human("Michael Jackson", "Isekai");
```

Kode di atas di dalam konsep OOP digunakan untuk ...

C. Instantiation class

Ciri khas instantiation class adalah ketika kita ingin mendeklarasikan *object* menggunakan kata kunci ***new***.

Referensi

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