OOP

GAMBARAN UMUM MATERI

OOP
metodologi atau paradigma untuk merancang program menggunakan class dan object.

Class
Sebuat blueprint / cetakan untuk menciptakan object.

Object
entitas yang memiliki status dan behaviour (perilaku).

Encapsulation — Access Modifier

Abstraction — Inheritance

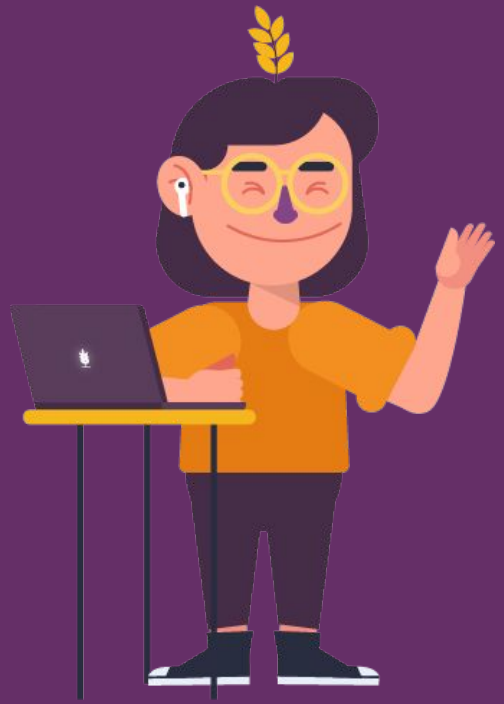
Constructor — Polymorphism

Overloading — Overriding

Properties
Setter | Getter

Sebelum berkenalan dengan konsep OOP, sebaiknya kamu membaca beberapa referensi tentang Paradigma Pemrograman berikut ini:

- https://www.geeksforgeeks.org/introduction-of-programming-paradigms/
- https://www.educba.com/functional-programming-vs-oop/
- https://medium.com/@LiliOuakninFelsen/functional-vs-object-oriented-vs-procedural-programming-a3d4585557f3
- https://medium.com/@sho.miyata.1/the-object-oriented-programming-vs-functional-programming-debate-in-a-beginner-friendly-nutshell-24fb6f8625cc
- https://medium.com/@adhywiranata/mengenal-paradigma-functional-programming-di-javascript-59d5eea7e2ac



Tenang...
Pada dasarnya Konsep OOP di semua bahasa pemrograman sama aja kok..
Kita hanya perlu penyesuaian aja..

The point is.. Gak Susah kok..!

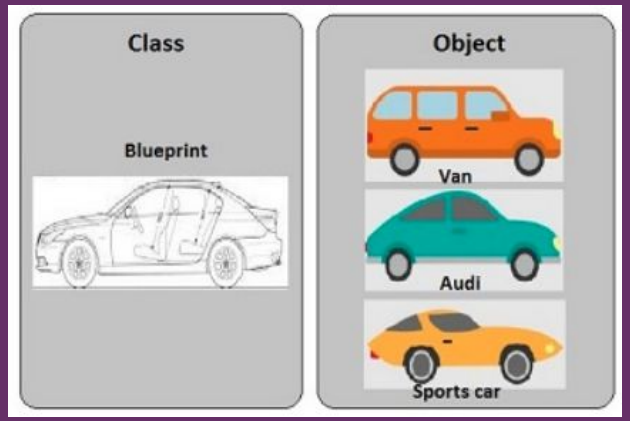
Apa itu OOP?

OOP

OOP (object Oriented Programming) adalah sebuah teknik pemrograman yang berorientasikan object.

Di OOP ini, Function dan Variable dibungkus sama object atau class yang saling berinteraksi, untuk membentuk satu program tertentu.

Hayo masih inget ga apa itu
object, class, constructor, property, method?



Object

```
let person1 = {
  name: "Isyana",
  address: "Jakarta
}

let person2 = {
  name: "Karina",
  address: "Bandung"
}

let person3 = {
  name: "Isyana Karina",
  address: "Batam"
  }
```

Gimana klo di object mau nambahin isMarried? Rempong ya edit satu-persatu objeknya.. Jadi, pakai Class aja..

Class

```
class Person {

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

}

let person1 = new Person("Isyana", "Jakarta", false);
/* This will create an object like this
   {
     name: "Isyana",
     address: "Jakarta",
     isMarried: false
   }
  */

let person2 = new Person("Isyana", "Jakarta", false);
let person3 = new Person("Karina", "Bandung", false);
```

Constructor Method

```
class Person {

  constructor(name, address) {
    this.name = name; // This will create a property called name
    this.address = address; // This will create a property called address
    this.isMarried = false; // This will create a property called isMarried
  }

}

let isyana = new Person("Isyana", "Jakarta")

console.log(isyana);
  // Output: Person { name: "Isyana", address: "Jakarta", isMarried: false }


/*
Object has property, to define the property of the object through Class, we need something called constructor method.
Notice that when you write new Person() it is actually calling the constructor Method, method == function.
In constructor method, it requires 2 params, name and address, so when you write new Person, it is expecting you to pass
the argument for that method.
*/
```

Property

Hanyalah data dari suatu objek biasa. Ada 2 jenis tipe properti, yaitu:

- Instance Property (disebut juga prototype)
  Properti Instance adalah properti yang dapat kita akses setelah instantiasi objek. Singkatnya, ini bisa dipanggil ketika kita menjalankan pernyataan *new Person*. Atau dalam contoh kehidupan nyata, contoh properti adalah nama, umur, alamat, dll. *Human* adalah *class*, dan Elon Musk adalah contoh/*instance* dari *Human*.

- Static Property
  Properti statis adalah properti yang dimiliki oleh kelas itu sendiri. Ini seperti berbicara tentang *Human* pada umumnya. Kita tidak perlu membuat instance objek ketika kita memanggil properti statis.

Instance Property

```
// Declaration
class Human {
  constructor(name, isAlive) {
    this.name = name; // This will create a property called name
    this.isAlive = isAlive; //
  }
}

// Instance 1
let mj = new Human("Michael Jackson", false)
// Instantiation of Human class.
It means we create a new object on a certain class.

// Instance 2
let km = new Human("Karl Marx", false) /
/ Instantiation of Human class.
It means we create a new object on a certain class.

console.log(mj.name)
// mj.name is an instance property of an instance from Human called mj.
console.log(km.name)
// km.name is an instance property of an instance from Human called km.
```

Static Property

```
class Human {

  static isLivingOnEarth = true;
  static group = "Vertebrate";

  constructor(name, isAlive) {
    this.name = name; // This will create a property called name
    this.isAlive = isAlive; //
  }

}

console.log(Human.group) // Output: Vertebrate
console.log(Human.isLivingOnEarth) // Output: true
```

Method

Adalah perilaku objek atau kelas. Sama seperti manusia, kita memiliki perilaku seperti bernapas, berjalan, berbicara, dll.
Ada dua jenis Method:

- Instance Method
  Sama seperti *instance property*, kita hanya bisa memanggil metode instance setelah kita instantiate objek kelas.

- Static Method
  Sama seperti *static property*.

Instance Method

```
class Human {

  static isLivingOnEarth = true;
  static group = "Vertebrate";

  constructor(name, isAlive) {
    this.name = name; // This will create a property called name
    this.isAlive = isAlive; //
  }

  // Instance method signature
  introduce() {
    console.log(`Hi, my name is ${this.name} and I'm ${this.isAlive ? "alive" : "dead"}`);

/*
`this` keyword refers to the instance object. In this case we have an instance named `mj`, so it will return this thing
Person { name: "Michael Jackson", isAlive: false } Which we can also call another method
*/
  }
}

// Instantiate
let mj = new Human("Michael Jackson", false);
mj.introduce() // Output: Hi, my name is Michael Jackson and I'm dead
```

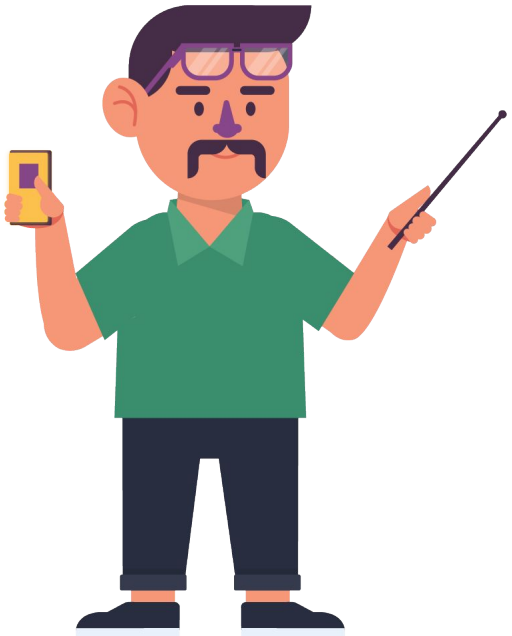
Static Method

```
class Human {
  static isLivingOnEarth = true;
  static group = "Vertebrate";
  constructor(name, isAlive) {
    this.name = name; // This will create a property called name
    this.isAlive = isAlive; //
  }
  // Instance method signature
  introduce() {
    console.log(`Hi, my name is ${this.name} and I'm ${this.isAlive ? "alive" : "dead"}`);

  static isEating(food) {
    let foods = ["plant", "animal"];
    return foods.includes(food.toLowerCase());
  }
}
console.log(Human.isEating("Plant")) // true
console.log(Human.isEating("Human")) // false
```

Modifying Class Method/Property on Fly

```
class Human {

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

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

// Add prototype/instance method
Human.protoype.greet = function(name) {
  console.log(`Hi, ${name}, I'm ${this.name}`)
}

// Add static method
Human.destroy = function(thing) {
  console.log(`Human is destroying ${thing}`)
}

let mj = new Person("Michael Jackson", "Isekai");
mj.greet("Donald Trump"); // Hi, Donald Trump, I'm Michael Jackson

Human.destroy("Amazon Forest") // Human is destroying Amazon Forest
```

Kenapa sih kita harus pakai OOP?



Kenapa sih pilih OOP?

1. OOP lebih cepat dan gampang buat dieksekusi.
2. OOP punya struktur yang jelas.
3. OOP bisa menjaga kode kita biar gak **DRY** alias **Don't Repeat Yourself**, jadi kodenya lebih gampang dikelola, dimodifikasi, dan di-*debug*.
4. OOP juga memungkinkan kamu buat bikin aplikasi yang bisa dipakai lagi dengan kode yang lebih sedikit dan waktu pengembangan yang lebih singkat.

Oke, biar ga bingung, kita mulai dari konsep-konsep dasar yang ada di OOP dulu ya!

1. INHERITANCE

***Inheritance* (pewarisan)** itu semacam konsep pewarisan yang kita terapkan di pemrograman 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

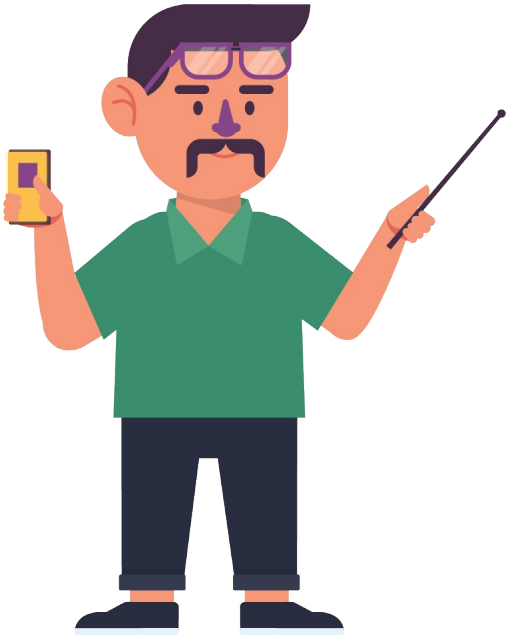
Perkenalkan, keluarga Suryo

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 tetapi juga ada 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

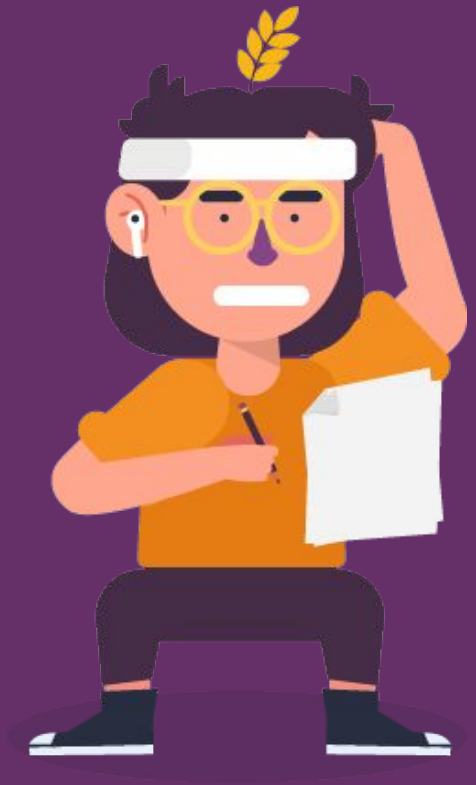
class yang semua fiturnya diwariskan kepada class turunannya.

Sub-class atau Child class

class turunan yang mewarisi semua fitur dari class lain.
Sub-class dapat menambah Field dan Methodnya 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 Field dan Method dari class yang udah ada.



Terus, gimana penerapan Inheritance dalam pemrograman 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:

- Attribut nama, alamat
- Default Constructor
- Constructor dengan parameter
- Terdapat method untuk menampilkan attribut

```
// 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 a programming using these languages`, 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
- Default Constructor
- Constructor dengan parameter
- Terdapat method untuk menampilkan attribute

OOP Concept - Inheritance

```
// 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 Karina", "Jakarta", ["Javascript", "Ruby", "Go", "Kotlin", "Python", "Elixir"]);
Isyana.introduce() // Hi, my name is Isyana; I can write a programming using these languages ["Javascript", "Ruby", "Go", "Kotlin", "Python", "Elixir"]
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
```

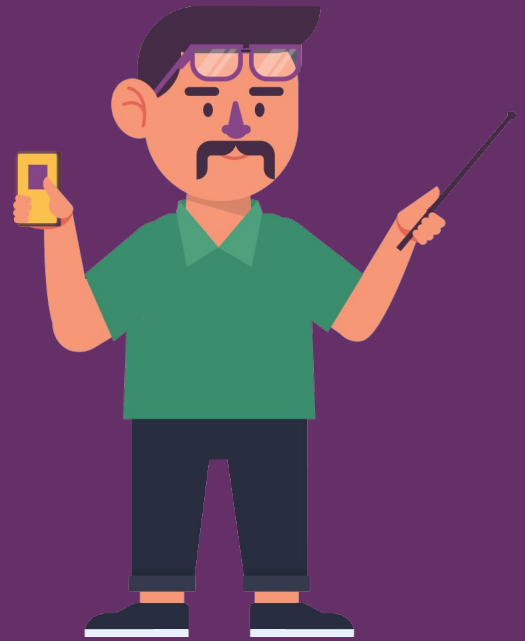
https://playcode.io/596704/

2. POLYMORPHISM

Polymorphism adalah sebuah prinsip yang memungkinkan class punya banyak bentuk method yang berbeda-beda meskipun namanya sama.

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

Biar kebayang, kita langsung bahas contohnya ya!



OOP Concept - Polymorphism

Overriding: https://playcode.io/596807/

```
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 a programming using these languages`, this.programmingLanguages);
  }

  code() {
    console.log(
      "Code some",
      this.programmingLanguages[
        Math.floor(Math.random * this.programmingLanguages.length)
      ]
    )
  }
}
```

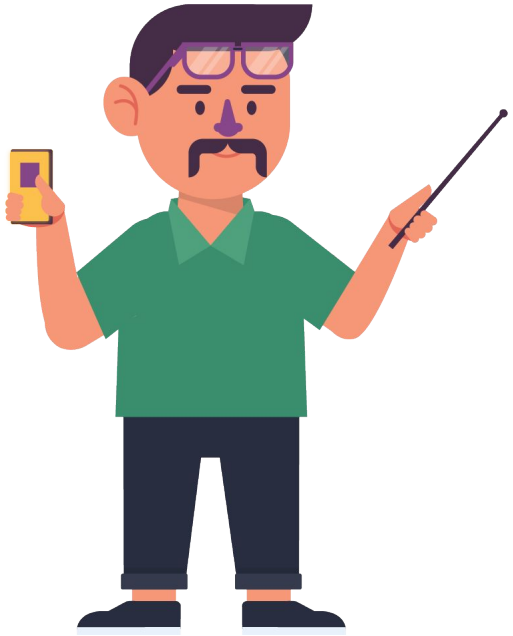
Overloading: https://playcode.io/596813/

```
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 a programming using these languages ${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 a programming using these languages ...
//Isyana.introduce("JavaScript") // Hi, my name is Isyana; Wrong Input
//Isyana.introduce(1) // Hi, my name is Isyana; Wrong Input
Isyana.code() //Code some ...
```

Contoh lain: https://playcode.io/596803/

3. ENCAPSULATION

Sebelum membahas **ENCAPSULATION,**
kita bakal belajar sebentar tentang **Access Modifier**

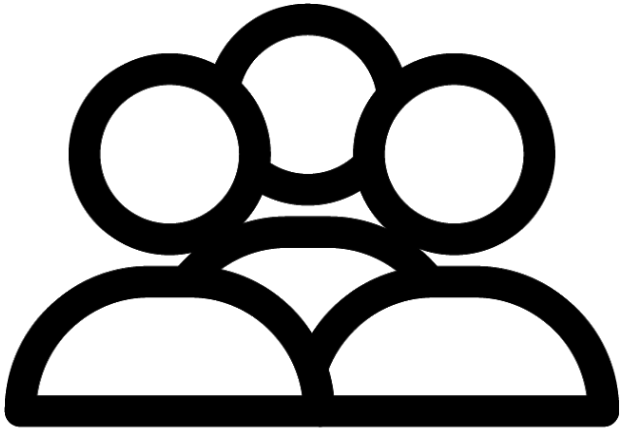


Apa itu Access Modifier?

Access Modifier adalah sebuah “hak akses” yang diberikan kepada variabel/method/class yang bertujuan buat menjaga integritas dari data tersebut ketika ingin diakses object lain.

BINAR ACADEMY

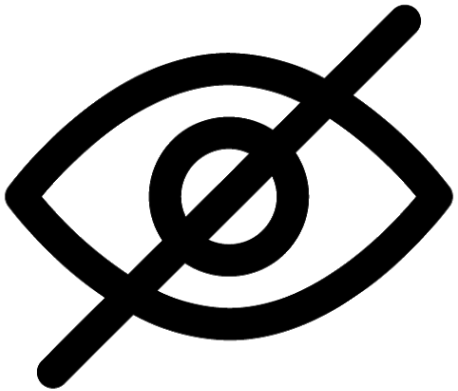
Terdapat tiga Access Modifier di JavaScript.



- **PUBLIC**
  Digunakan pada class/variable/method/constructor, biar bisa diakses sama semua class di dalam class yang sama atau di luar class yang berbeda.

  Modifier jenis ini punya tingkat akses yang sangat luas, jadi apa yang ada di dalam sebuah class bisa aja diakses sama class manapun tanpa batasan.

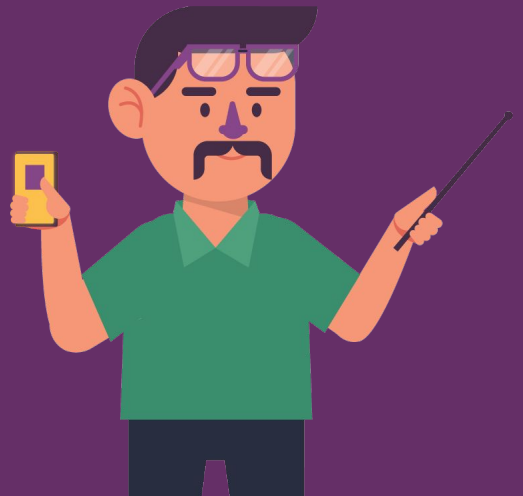
- **PROTECTED**
  Digunakan pada variable/method/constructor, yang diberikan modifier *protected* dapat diakses oleh sub-class atau class lain asalkan di dalam satu class yang sama.

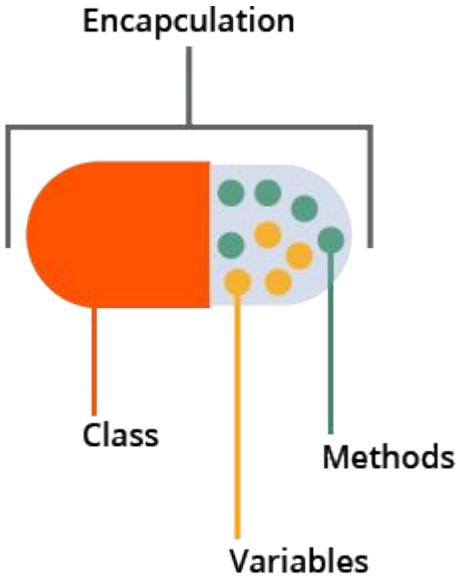


- **PRIVATE**
  Variable dan method yang diberikan *private* modifier cuma bisa diakses sama class itu sendiri. Data-data tersebut ga bisa diwariskan ke sub-class atau class lainnya.

Lanjuuuutt!

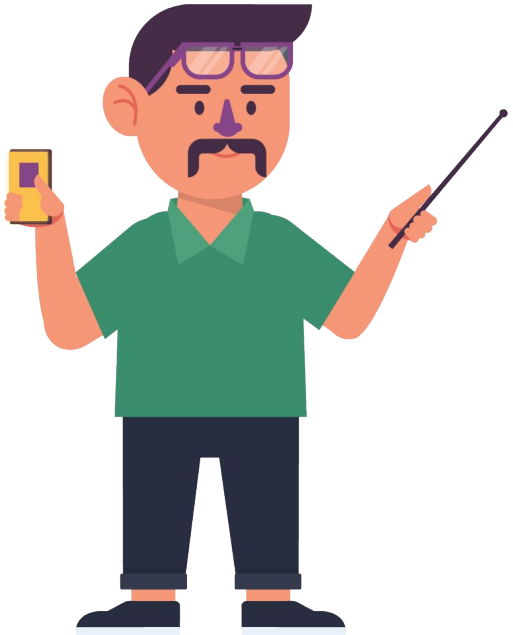
Nah, konsep Encapsulation ini identik sama salah satu *access modifier* tadi, yaitu Access Modifier PRIVATE.





Jadi, Encapsulation (pembungkusan) adalah Data yang dibungkus dengan *access modifier private* yang bertujuan biar method dan variable ga bisa diakses secara langsung dari luar class.

Kenapa sih data harus dibungkus?



Fungsi encapsulation :

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

BINAR ACADEMY

Masih bingung? Gampangnya gini ..

Saat kita pakai mesin ATM buat menarik atau menyetor uang, kita ga tahu proses yang ada di dalam mesin itu kan?

Kita tahunya ketika kita memasukkan kartu ATM, memasukkan pin, memilih nominal uang yang bakal diambil, uang akan keluar sesuai nominal yang kita pilih.

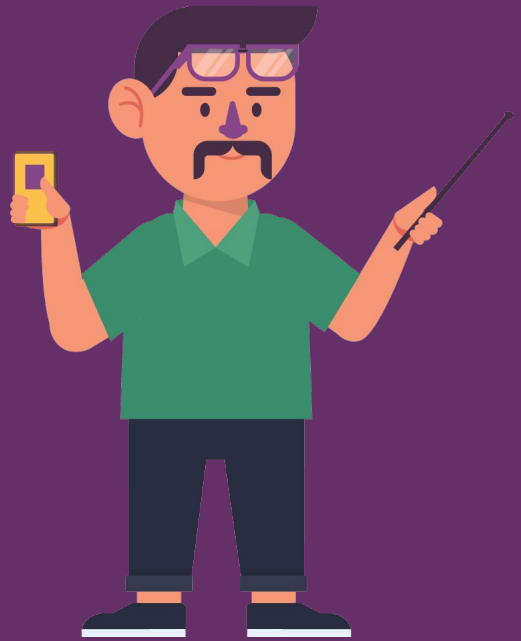
Nah, di dalam mesin ATM itu ada teknik enkapsulasi yang berjalan dan ga diketahui sama nasabah.

Nah, kira-kira itu yang terjadi di pemrograman Java dengan teknik Encapsulation.

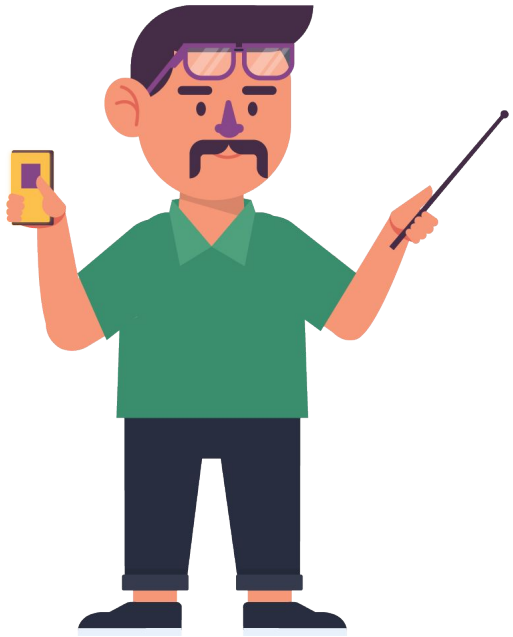
Method atau variable pakai model *private modifier* supaya class lain ga bisa mengakses variable atau method di dalam class tersebut.

Terus, gimana kalo ada class lain yang membutuhkan data dari class yang variable dan methodnya pake modifier private? Kan ga bisa diakses?

Nah, buat mengatasi kondisi kaya gitu, ada dua istilah lagi yang harus kamu pahami, namanya **SETTER** dan **GETTER**.

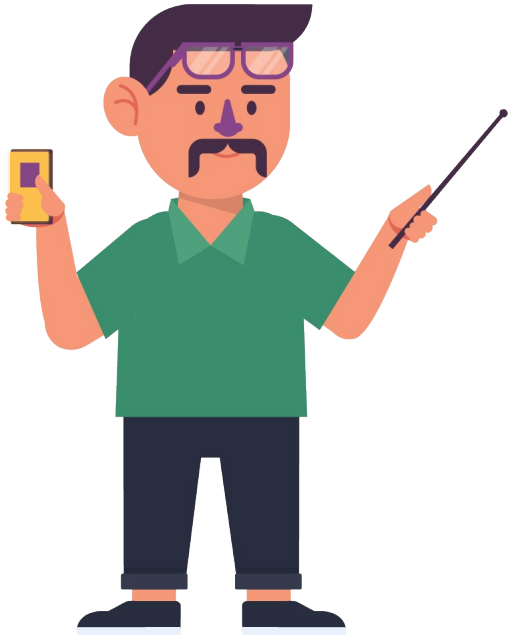


Apa itu Setter dan Getter?



Setter dan **Getter** itu termasuk jenis **method**.

Method Setter buat ngasih nilai
Method Getter buat dapetin nilai



Ada beberapa poin yang harus kamu perhatiin nih saat kamu mau bikin method Setter dan Getter :

- Method Setter dan Getter harus diberikan modifier public, karena method ini akan diakses dari luar class.
- Nama method Setter harus diawali dengan set.
- Nama method Getter harus diawali dengan get.

Perbedaan method Setter dan Getter ada pada nilai kembalian, parameter, dan isi method-nya.

- Method Setter ini ga punya nilai kembalian void (kosong). Karena tugasnya cuma buat isi data ke dalam attribute.
- Method Getter punya nilai kembalian sesuai dengan tipe data yang bakal diambil.

OOP Concept - Encapsulation

```
class Person {
    constructor({firstName, lastName, job}) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.job = job;
        this.skills = [];
        Person._amount = Person._amount || 0;
        Person._amount++;
    }

    static get amount() {
        return Person._amount;
    }

    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    }

    set fullName(fN) {
        if (/[A-Za-z]\s[A-Za-z]/.test(fN)) {
            [this.firstName, this.lastName] = fN.split(' ');
        } else {
            throw Error('Bad fullname');
        }
    }

    learn(skill) {
        return this.skills.push(skill);
    }
}
```

```
class Job {
    constructor(company, position, salary) {
        this.company = company;
        this.position = position;
        this.salary = salary;
    }
}

const john = new Person({
    firstName: 'John',
    lastName: 'Doe',
    job: new Job('Youtube', 'developer', 200000)
});

const roger = new Person({
    firstName: 'Roger',
    lastName: 'Federer',
    job: new Job('ATP', 'tennis', 1000000)
});

console.log(john.firstName) //output: John
console.log(john.lastName) //output: Doe

john.fullName = 'Mike Smith';
console.log(john.fullName) //output: Mike Smith
console.log(john.firstName) //output: Mike
console.log(john.lastName) //output: Smith

john.learn('es6');
john.learn('es7');
console.log(john.skills) //output: ["es6","es7"]

roger.learn('programming');
console.log(roger.skills) //output: ["programming"]
```

Contoh Setter & Getter:

https://playcode.io/596891/

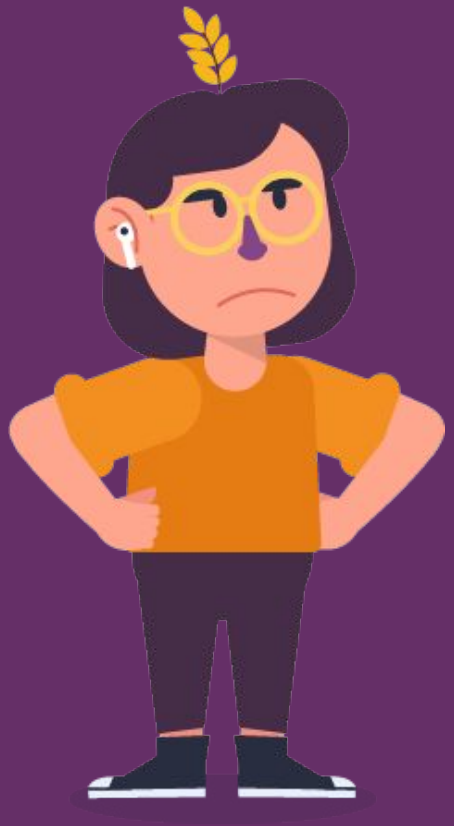
Oke, biar ga pusing, kita langsung aja bahas contohnya, ya!

OOP Concept - Encapsulation

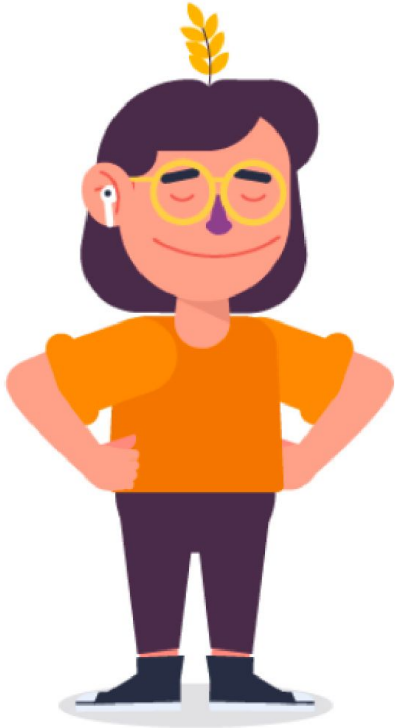
```
class User {
  constructor(props) {
    // props is object, because it is better that way
    let { email, password } = props; // Destruct
    this.email = email;
    this.encryptedPassword = this.#encrypt(password); // We won't save the plain password
  }

  // Private method
  #encrypt = (password) => {
    return `pretend-this-is-an-encrypted-version-of-${password}`
  }

  // Getter
  #decrypt = () => {
    return this.encryptedPassword.split(`pretend-this-is-an-encrypted-version-of-`)[1];
  }

  authenticate(password) {
    return this.#decrypt() === password; // Will return true or false
  }
}

let Isyana = new User({
  email: "IsyanaKarina@mail.com",
  password: "123456"
})

const isAuthenticated = Isyana.authenticate("123456");
console.log(isAuthenticated) // true
```

https://playcode.io/596794/

4. ABSTRACTION



Apa itu Abstraction?



First of all, apa yang ada di pikiranmu waktu denger kata “Orang”?

Mungkin kamu bakal jawab, polisi, hakim, tentara, atau dosen.

Sekarang apa yang kamu pikirkan saat mendengar kata,
“Orang berlari”?

Mungkin kamu bakal berpikir, “Pake apa larinya”, “Di mana larinya”, “Gimana cara larinya”, dst.

Ini yang kita sebut sebagai abstraksi.

Kata “Orang” sendiri masih bersifat abstrak, tapi kita bisa membayangkan konsep “Orang” itu kaya 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 udah pernah menggunakannya tanpa kita sadari.

Nah, di pemrograman JavaScript kita bisa bikin abstraksi ini.

Biar kebayang, kita lihat dulu yuk contohnya!

OOP Concept - Abstraction

```
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}`)
  }

}
```

OOP Concept - Abstraction

```
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

try {
  let Abstract = new Human({
    name: "Abstract",
    address: "Unknown"
  })
}

catch(err) {
  console.log(err.message) // Cannot instantiate from Abstract Class
}
```

https://playcode.io/596816/

BINAR

OOP
Object Oriented Programming, program yang struktur codenya berdasarkan Object

Inheritance
Pada Java dikenal dengan syntax extends. Digunakan untuk menggunakan kembali atribut ataupun method dari class induk.

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

Encapsulation
Pembungkusan object agar lebih aman.

Access Modifier
Hak akses yang bisa diberikan pada class, atribut, method.

Setter dan Getter
Method setter untuk menetapkan nilai, sedangkan getter untuk mendapatkan nilai pada suatu class.

Abstraction
Class yang tidak bisa langsung dibuat objectnya

Buatlah sebuah Class Student, yang memiliki atribut berikut:

- Name,
- Age,
- Date of Birth,
- Gender
- Student ID (bisa berupa angka atau teks), dan
- Hobbies (bisa menampung lebih dari 1 hobi).

Class tersebut juga bisa memanggil fungsi dengan proses sebagai berikut:

- SetName: mengubah nama student dengan mengirimkan satu parameter ke dalam fungsi berupa teks
- SetAge: mengubah umur student dengan mengirimkan satu parameter ke dalam fungsi berupa angka
- SetDateOfBirth: mengubah tanggal lahir student dengan mengirimkan satu parameter ke dalam fungsi berupa teks
- SetGender: mengubah gender student dengan mengirimkan satu parameter ke dalam fungsi berupa teks, dan hanya bisa menerima nilai `Male` atau `Female`
- addHobby: menambah hobi dengan mengirimkan satu parameter ke dalam fungsi berupa teks
- removeHobby: menghapus list hobi yang ada dengan mengirimkan satu parameter berupa teks, yang merupakan hobi apa yang akan dihapus
- getData: menampilkan seluruh data atribut murid