



Object Oriented JavaScript

Silver- Chapter 3 - Topic 1

**Selamat datang di Chapter 3 Topic 1 pada course
React Native dari Binar Academy!**



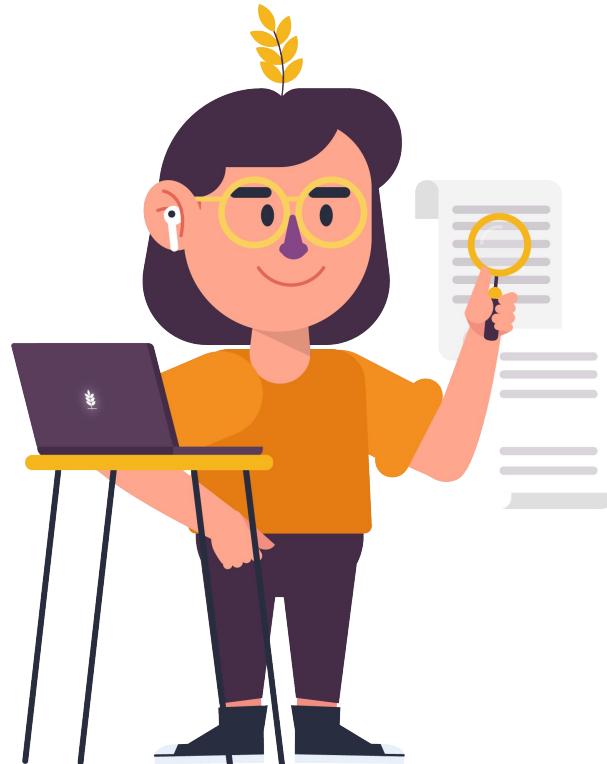


Haii teman-teman 

Selamat datang di Chapter 3 React Native.

Pada chapter ini, kamu udah mulai bisa bikin prototype aplikasi mobile nih. Di sesi pertama chapter ini, kita akan belajar tentang **OOP (Object Oriented Programming)** dan **cara kerja konsep OOP**.

Ayo kita mulai!





Detailnya, kita bakal bahas hal-hal berikut ini:

- Mengenal OOP dan Class Object
- Cara kerja konsep Inheritance
- Cara kerja konsep Encapsulation
- Cara kerja konsep Abstraction
- Cara kerja konsep Polymorphism



Guys, kamu sudah tahu belum kalau di dunia pemrograman ada yang disebut dengan **paradigma pemrograman?**

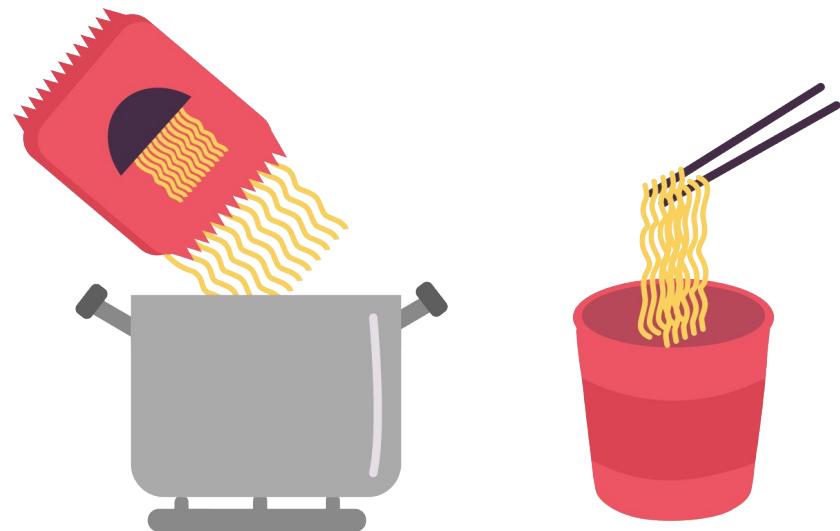
Apa itu paradigma pemrograman?



Biar lebih gampang dipahami, kita pakai analogi aja yuk~

Untuk mengolah mie instan, ada banyak cara yang bisa digunakan.

Ada yang direbus di air mendidih, ada yang diseduh, bahkan ada yang dimakan langsung. 😊

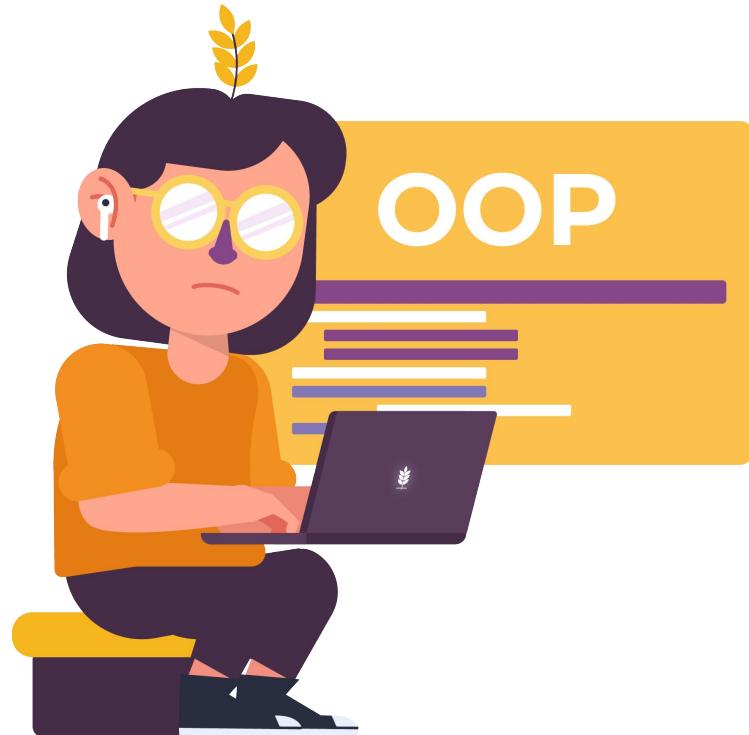


Kaitannya dengan paradigma pemrograman kayak gini ya~

Untuk memecahkan sebuah masalah dengan bahasa pemrograman juga memiliki banyak cara dan metodenya, guys!

Nah, berbagai cara dan metode itu lah yang dikatakan sebagai **paradigma pemrograman**.

Salah satu paradigma pemrograman yang sangat umum digunakan oleh para programmer adalah OOP atau Object Oriented Programming.

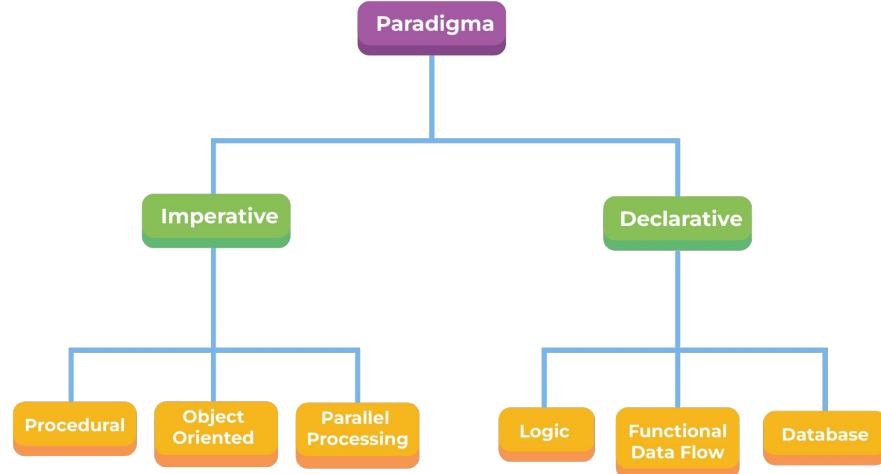


Kayak masak mie instan, nulis program metodenya juga banyak

Jadi bukan cuma OOP ya, guys. Ada banyak paradigma pemrograman lain, tapi yang sering dipakai di antaranya:

- Procedural Programming
- Parallel Processing Approach
- Logic Programming
- Functional Programming
- Database Processing Approach

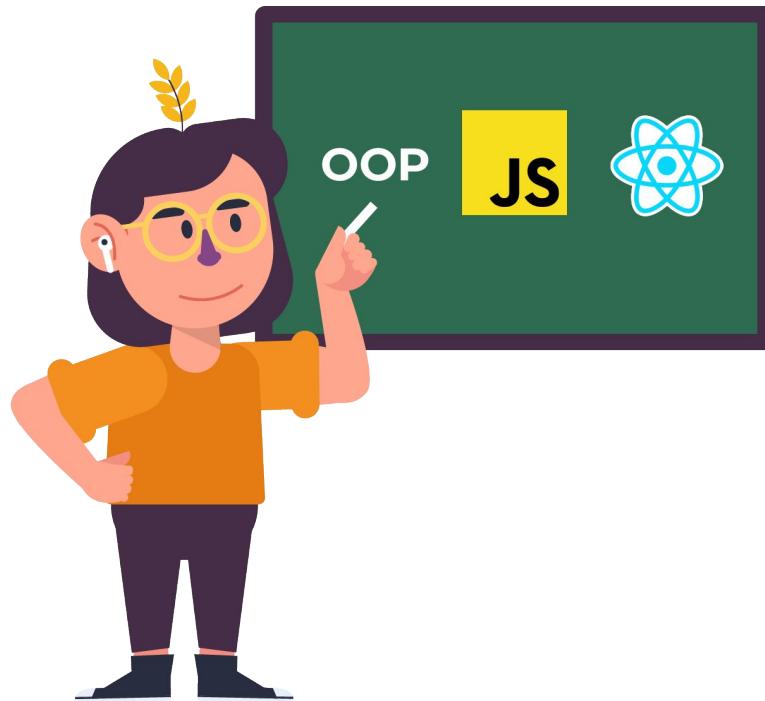
Untuk tahu lebih banyak tentang paradigma pemrograman yang nggak akan kita bahas di materi ini, kamu bisa baca di [link ini](#) ya!





Tapi, di sesi ini kita hanya fokus mempelajari OOP aja ya! Kenapa OOP? 🤔

Karena paradigma ini bisa dipakai di JavaScript, yang bakal berguna saat kamu mau bikin project dengan menggunakan library JavaScript yang kita pelajari di course ini, yaitu **React**.

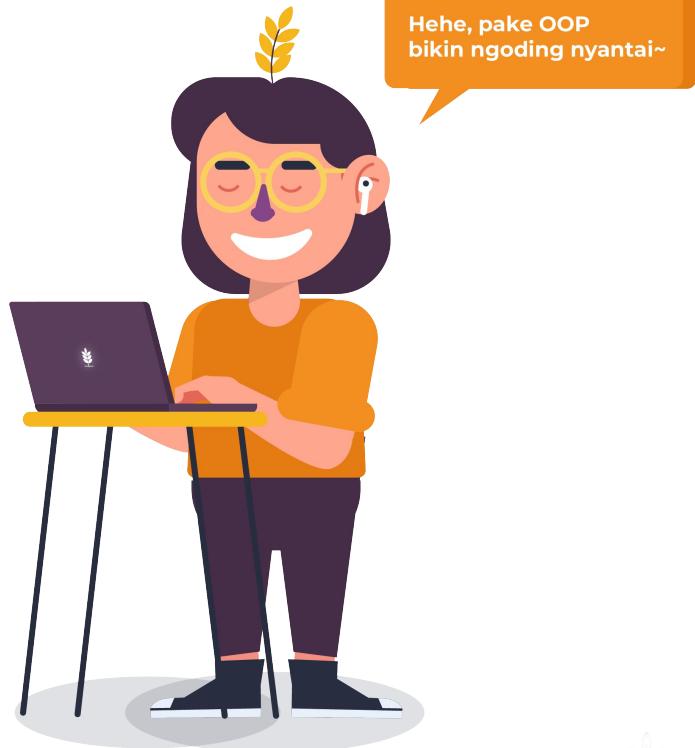




Ada lagi nih alasan lainnya!

OOP punya struktur yang jelas, jadi akan memudahkan kamu dalam penulisan kode.

OOP juga membantu kamu supaya nggak perlu terus mengulang penulisan kode. Dengan begitu kodenya jadi lebih gampang dikelola, dimodifikasi, dan di-debug.



Guys, OOP ini juga bisa didefinisikan sebagai paradigma pemrograman yang mengandalkan konsep Class dan Object.

Makanya, kita perlu kenalan dengan Class dan Object dalam OOP!





Di Chapter 1 kamu sudah belajar cara membuat sebuah data dengan tipe data object, masih ingat nggak nih?

Yuk kita review lagi!



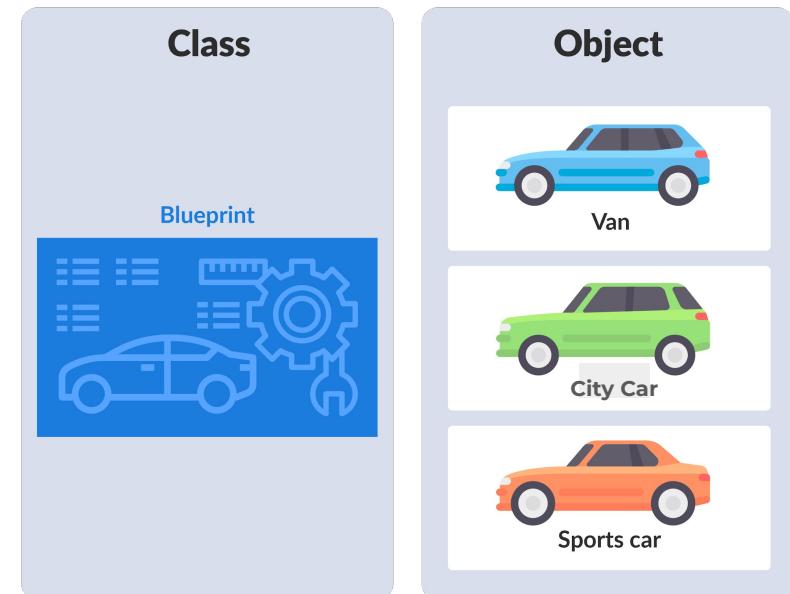


Kita mulai dari perbedaan class dan object, serta hubungan keduanya

Coba perhatikan gambar di samping!

Misalnya, kamu punya blueprint untuk merancang mobil. Nah, **class adalah suatu blueprint atau acuan** untuk membuat object mobil. Dari blueprint ini, kamu bisa tahu kalau mobil itu punya 4 roda, ada pintunya dan lain-lain.

Dengan mengandalkan blueprint mobil ini, kamu pun jadi bisa mewujudkan berbagai jenis mobil, misalnya mobil van, city car, dan mobil sport.



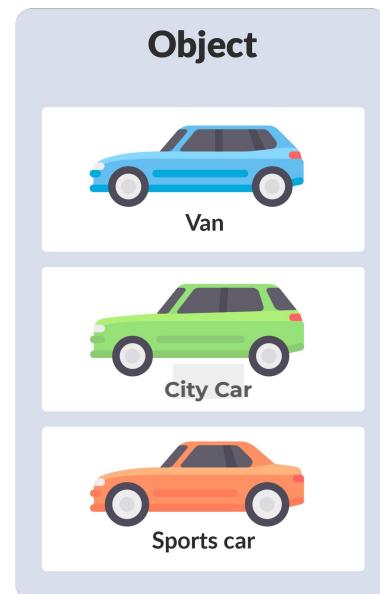
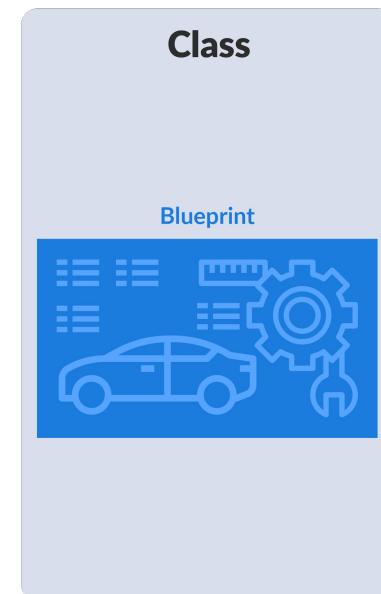


Sedangkan object adalah hasil dari class

Dengan mengandalkan blueprint mobil tadi, kamu bisa mewujudkan berbagai jenis mobil, misalnya mobil van, city car, dan mobil sport.

Berarti **jenis-jenis mobil yang bisa dibuat dari blueprint (class) tadi adalah object?**

Yaps! betul banget!





Oke! Kita udah paham tentang Class dan Object.

Lalu di OOP ini, kenapa harus pakai Class untuk membuat object? Kan bikin object dengan cara biasa (literal) juga bisa? 🤔





Iya sih, bisa aja, tapi kamu bakalan repot kalau mau mengubah atau menambahkan property baru di object tersebut. Kenapa?

Karena **kamu harus menulis ulang object satu per satu.**
Itu kalau jumlah objeknya cuma puluhan, bayangan kalau jumlah objeknya ratusan dan seterusnya!

Capek dehh~





Kita bayangin pakai ilustrasi coding deh..

Misalnya kamu mau bikin objek mobil dengan struktur property object yang sama. Seperti kode di samping, kamu buat 3 objek dengan property yang sama, tapi value nya berbeda-beda.

```
● ● ●  
const mobilAnna = {  
  tipe: 'Lamborgini',  
  warna: 'Kuning/Gold',  
  apakahMobileBalap: true,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Lamborgini warna Kuning/Gold dihidupkan'  
  },  
};  
const mobilJason = {  
  tipe: 'Ferrari',  
  warna: 'Merah',  
  apakahMobileBalap: true,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Ferrari warna Merah dihidupkan'  
  },  
};  
const mobilGerald = {  
  tipe: 'Kijang Innova',  
  warna: 'Silver',  
  apakahMobileBalap: false,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Kijang Innova warna Silver dihidupkan'  
  },  
};
```



Pake cara biasa beneran bikin capek~

Nah bayangin betapa ribetnya kalau kamu mau membuat 20 objek atau lebih, berapa banyak kode yang harus kamu tulis atau copas? Yang ada jari kamu keriting guys 😱

```
● ● ●  
const mobilAnna = {  
  tipe: 'Lamborgini',  
  warna: 'Kuning/Gold',  
  apakahMobileBalap: true,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Lamborgini warna Kuning/Gold dihidupkan'  
  },  
};  
const mobilJason = {  
  tipe: 'Ferrari',  
  warna: 'Merah',  
  apakahMobileBalap: true,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Ferrari warna Merah dihidupkan'  
  },  
};  
const mobilGerald = {  
  tipe: 'Kijang Innova',  
  warna: 'Silver',  
  apakahMobileBalap: false,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Kijang Innova warna Silver dihidupkan'  
  },  
};
```



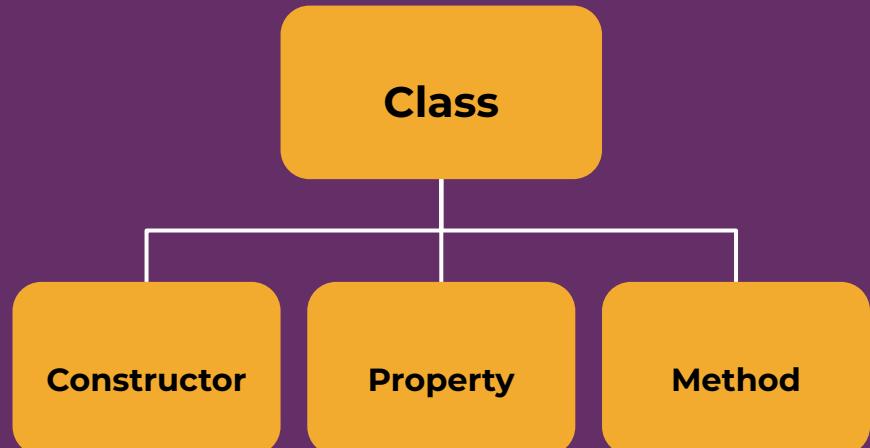
Nah, makanya untuk masalah kayak gini, kamu bisa **pakai class untuk membuat sebuah objek.**

Ya okelah, kita langsung aja belajar cara membuat class. Cuss ke slide selanjutnya.





Sebelum mulai bikin Class, kamu harus tahu dulu nih bagian-bagian dari sebuah Class, yuk cekidot~





Constructor

Constructor adalah sebuah fungsi yang membuat instance/turunan dari sebuah class, yaitu object. Jadi, bisa dibilang constructor ini adalah fungsi yang digunakan untuk membuat object pada sebuah class.

Di JavaScript, constructor yang akan dipanggil ketika kamu mendeklarasikan object menggunakan **kata kunci “new”**.

```
class BlueprintMobile {  
    constructor(tipe, warna, apakahMobileBalap) {  
        this.tipe = tipe;  
        this.warna = warna;  
        this.apakahMobileBalap = apakahMobileBalap;  
    }  
  
    const data = new MyClass();  
  
    console.log(data);  
    // Output => {namaDepan: 'Cristiano', namaBelakang: 'Ronaldo'}  
  
    console.log(data.namaDepan);  
    // Output => 'Cristiano'
```



Penggunaan constructor ada tujuannya, guys!

Constructor digunakan **untuk membuat object dan menetapkan nilainya jika terdapat property di dalam object.**

Coba kamu perhatikan contoh di samping

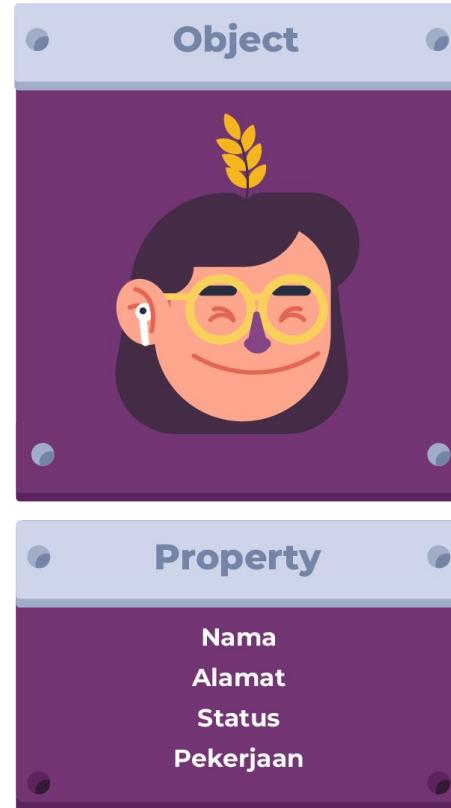
```
class BlueprintMobile {  
    constructor(tipe, warna, apakahMobileBalap) {  
        this.tipe = tipe;  
        this.warna = warna;  
        this.apakahMobileBalap = apakahMobileBalap;  
    }  
  
    const data = new MyClass();  
  
    console.log(data);  
    // Output => {namaDepan: 'Cristiano', namaBelakang: 'Ronaldo'}  
  
    console.log(data.namaDepan);  
    // Output => 'Cristiano'
```



Property

Property adalah data dari suatu object atau variabel-variabel yang ada di object, biasanya disebut juga dengan **attribute**.

Setiap object memiliki property, misalnya manusia bernama Sabrina adalah object. Maka dia punya beberapa property, seperti nama, alamat, dsb.



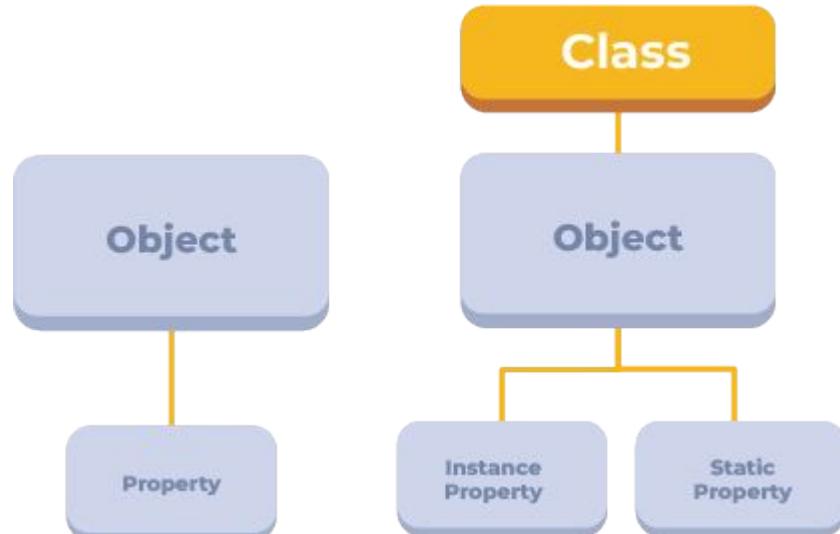


Ini salah satu perbedaan object biasa dengan object pada class di OOP!

Kalau kamu bikin object tanpa class, maka object itu cuma punya satu tipe property.

Sedangkan kalau kamu membuat object dengan class, maka akan ada dua tipe property, yaitu **instance property dan static property**.

Yuk kita bahas satu per satu~





Tipe yang pertama adalah Instance Property

Yaitu sebuah property yang dapat diakses setelah si object kamu instantiate (dibuat melalui keyword new).

Jadi, kalau ada nilai pada property yang ingin kamu ubah, dia masih bisa diakses.

Instance property bisa kamu gunakan untuk nilai/value dari sebuah property yang sifatnya berubah-ubah.

```
// Membuat Class
class MyClass {
  // Menggunakan constructor method
  constructor() {
    // Properti yang kita tulis dengan 'this'
    // nantinya akan tercreate menjadi sebuah properti pada object

    this.namaDepan = 'Cristiano';
    this.namaBelakang = 'Ronaldo';
  }
}

const data = new MyClass();

console.log(data);
// Output => {namaDepan: 'Cristiano', namaBelakang: 'Ronaldo'}

console.log(data.namaDepan);
// Output => 'Cristiano'
```



Begini penjelasan penerapan instance property

Properti namaDepan **didapatkan dari hasil pemanggilan new MyClass()**, inilah yang dimaksud instance property, property yang dihasilkan dari setiap pemanggilan sebuah class dengan menggunakan keyword new.

```
// Membuat Class
class MyClass {
    // Menggunakan constructor method
    constructor() {
        // Properti yang kita tulis dengan 'this'
        // nantinya akan tercreate menjadi sebuah properti pada object
        this.namaDepan = 'Cristiano';
        this.namaBelakang = 'Ronaldo';
    }
}

const data = new MyClass();

console.log(data);
// Output => {namaDepan: 'Cristiano', namaBelakang: 'Ronaldo'}

console.log(data.namaDepan);
// Output => 'Cristiano'
```



Tipe yang kedua adalah Static Property

Yaitu sebuah property yang nilainya akan selalu sama di semua instance dari class tersebut.

Tipe property ini bisa kamu gunakan untuk nilai/value dari sebuah property yang sifatnya tetap..

```
// Membuat class
class MyClass {
    // Membuat static property
    static myStaticProperty = 'Hello'
}

// Akses static properti "myStaticProperty" pada class "MyClass"
console.log(MyClass.myStaticProperty)
// Output:
// 'Hello'
```



Begini penjelasan penerapan static property

Jadi, hasil dari syntax pada gambar di samping akan selalu sama, yaitu kata 'Hello'.

```
// Membuat class
class MyClass {
    // Membuat static property
    static myStaticProperty = 'Hello'
}

// Akses static properti "myStaticProperty" pada class "MyClass"
console.log(MyClass.myStaticProperty)
// Output
// 'Hello'
```



Method

Method adalah suatu **fungsi atau aksi** dari suatu object.

Misalnya kamu punya class “Manusia”, maka method ibarat aksi (perilaku) yang bisa dilakukan oleh manusia itu, seperti berjalan, berbicara, bekerja dan sebagainya..

Di dalam method, kamu bisa **mengakses property dari object dengan keyword “This”**, yang artinya adalah memanggil object itu sendiri.

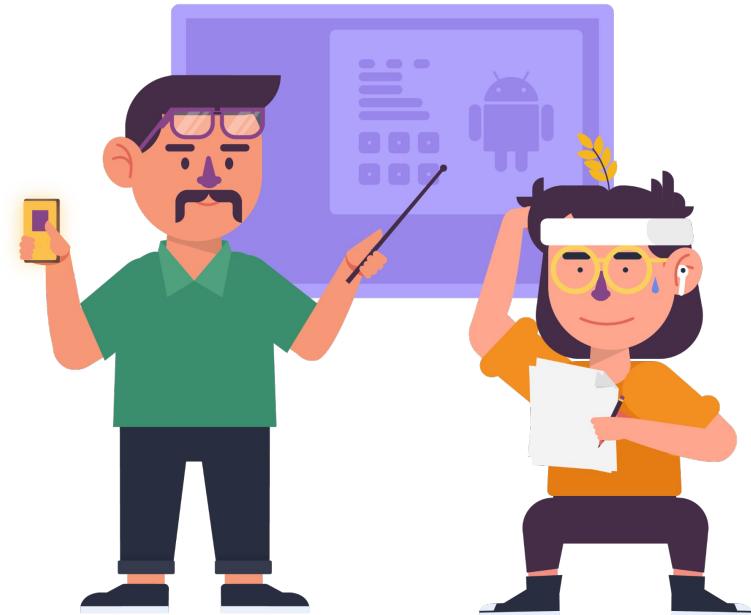




Method juga ada tipe-tipenya, guys!

Sama kayak property, method punya dua tipe dalam hal aksesnya yaitu **Instance method (prototype)** dan **Static method**.

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



Ini contoh coding Instance Method dan Static Method

Instance Method

```
class BlueprintMobil {
  constructor(tipe, warna, apakahMobileBalap) {
    this.tipe = tipe;
    this.warna = warna;
    this.apakahMobileBalap = apakahMobileBalap;
  }

  ubahWarnaMobil(warnaBaru) {
    this.warna = warnaBaru;
    console.log('ganti warnaa');
  }
}

const mobilAndi = new BlueprintMobil('Ferari', 'Biru', false);
mobilAndi.ubahWarnaMobil('Jingga');
console.log(mobilAndi);
// { tipe: 'Ferari', warna: 'Jingga', apakahMobileBalap: false }
```

Static Method

```
class BlueprintMobil {
  constructor(tipe, warna, apakahMobileBalap) {
    this.tipe = tipe;
    this.warna = warna;
    this.apakahMobileBalap = apakahMobileBalap;
  }

  ubahWarnaMobil(warnaBaru) {
    this.warna = warnaBaru;
    console.log('ganti warnaa');
  }

  static hidupinMobil() {
    console.log('Mobil hidup ... ');
  }
}

const mobilAndi = new BlueprintMobil('Ferari', 'Biru', false);
mobilAndi.ubahWarnaMobil('Jingga');
console.log(mobilAndi);

BlueprintMobil.hidupinMobil();
```



Cukup teorinya. Sekarang mari kita coba
buat class yuk. Practice makes perfect!





```
class BlueprintMobile {  
    constructor(tipe, warna, apakahMobileBalap) {  
        this.tipe = tipe;  
        this.warna = warna;  
        this.apakahMobileBalap = apakahMobileBalap;  
    }  
}
```

Nggak sesulit yang kamu bayangin kok, ini langkah-langkah bikin Class

Untuk membuat sebuah class, kamu bisa pakai **keyword Class**, lalu diikuti dengan nama kelasnya.

Oke, ayo kita coba buat sebuah class, dengan nama class “BlueprintMobil”, seperti contoh di samping. Class “BlueprintMobil” akan mencetak/menghasilkan sebuah data berupa objek.



```
// Buat class
class BlueprintMobile {
    constructor(tipe, warna, apakahMobileBalap) {
        this.tipe = tipe;
        this.warna = warna;
        this.apakahMobileBalap = apakahMobileBalap;
    }

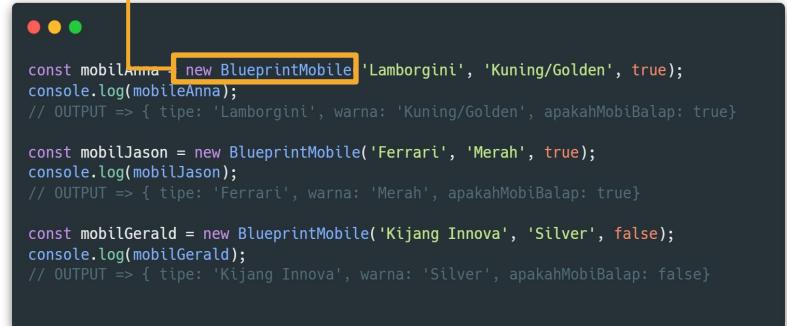
    hidupinMesin() {
        console.log(`Mobile ${this.tipe} warna ${this.warna} dihidupkan`);
    }
}
```

Di simulasi ini, kamu akan membuat objek berbagai macam jenis mobil menggunakan **class “BlueprintMobile”**



Pemanggilan class dengan menggunakan keyword 'new'

bertujuan untuk membuat instance atau turunan dari kerangka class yang telah kamu buat, kemudian turunan itu akan dibuat menjadi bentuk berupa sebuah object.



```
const mobilAnna = new BlueprintMobile('Lamborgini', 'Kuning/Golden', true);
console.log(mobilAnna);
// OUTPUT => { tipe: 'Lamborgini', warna: 'Kuning/Golden', apakahMobiBalap: true}

const mobilJason = new BlueprintMobile('Ferrari', 'Merah', true);
console.log(mobilJason);
// OUTPUT => { tipe: 'Ferrari', warna: 'Merah', apakahMobiBalap: true}

const mobilGerald = new BlueprintMobile('Kijang Innova', 'Silver', false);
console.log(mobilGerald);
// OUTPUT => { tipe: 'Kijang Innova', warna: 'Silver', apakahMobiBalap: false}
```



Nah, sekarang kode kamu jadi lebih efektif dan lebih rapi. Coba lihat perbandingan object biasa dan object dengan class di bawah ini!

Membuat object secara manual/object literal

```
● ● ●  
const mobilAnna = {  
  tipe: 'Lamborghini',  
  warna: 'Kuning/Gold',  
  apakahMobileBalap: true,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Lamborghini warna Kuning/Gold dihidupkan'  
  },  
};  
const mobilJason = {  
  tipe: 'Ferrari',  
  warna: 'Merah',  
  apakahMobileBalap: true,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Ferrari warna Merah dihidupkan'  
  },  
};  
const mobilGerald = {  
  tipe: 'Kijang Innova',  
  warna: 'Silver',  
  apakahMobileBalap: false,  
  hidupinMesin: function () {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
    // Output console => 'Mobil Kijang Innova warna Silver dihidupkan'  
  },  
};
```

Membuat object dengan class

```
● ● ●  
class BlueprintMobil {  
  constructor(tipe, warna, apakahMobilBalap) {  
    this.tipe = tipe;  
    this.warna = warna;  
    this.apakahMobilBalap = apakahMobilBalap;  
  }  
  
  hidupinMesin() {  
    console.log(`Mobil ${this.tipe} warna ${this.warna} dihidupkan`);  
  }  
}  
  
const mobilAnna = new BlueprintMobil('Lamborghini', 'Kuning/Golden', true);  
const mobilJason = new BlueprintMobil('Ferrari', 'Merah', true);  
const mobilGerald = new BlueprintMobil('Kijang Innova', 'Silver', false);
```



**Habis belajar bikin class, sekarang kamu
akan belajar menerapkan konsep OOP
di JavaScript.**

Let's go!





Oke, hal pertama yang akan kita bahas dalam konsep OOP adalah **inheritance**.

Sesuai artinya, inheritance ini semacam konsep pewarisan yang akan kamu terapkan di OOP, guys.



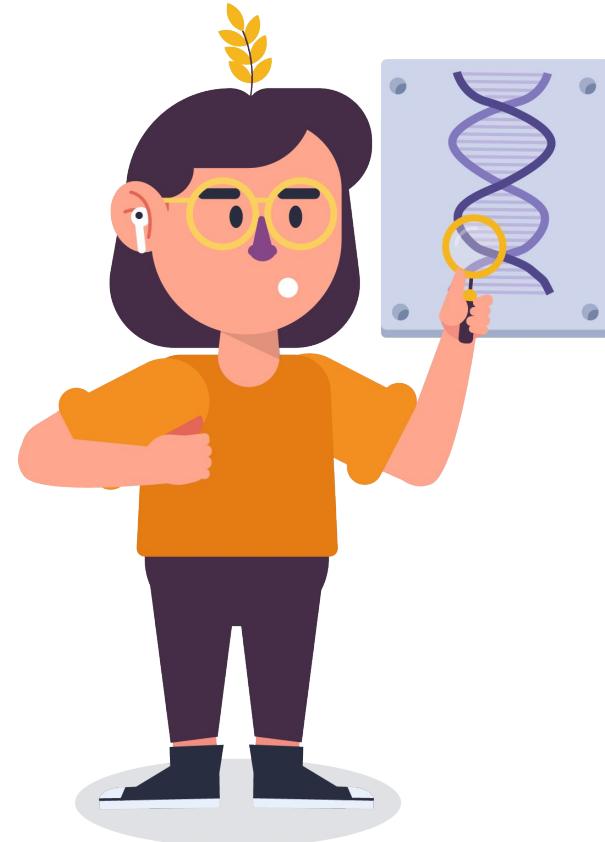


Begini cara kerja inheritance~

Cara kerja inheritance sama kayak DNA yang diwariskan orangtua ke anak-anaknya, termasuk ciri dan keunikannya.

Fungsi inheritance adalah agar kamu nggak capek ngulang-ngulang penulisan kode.

Biar kamu lebih gampang memahami konsep ini, kita coba jabarkan pakai analogi aja yuk!





Ini adalah keluarga Pak Suryo dengan ciri-ciri nya masing-masing



Pak Suryo

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

Bu Arum

- Kulit putih
- Mata bulat
- Berambut cokelat muda
- Suka pakai rok berwarna abu
- Suka dengan baju berwarna pink

Kevin

- **Kulit putih**
- **Mata bulat**
- **Berambut cokelat muda**
- Suka pakai celana berwarna biru
- Suka pakai sepatu warna hitam

Jessica

- Kulit coklat
- Mata bulat
- Berambut cokelat tua
- Suka pakai baju warna pink
- Suka mengikat rambut



Sekarang kita bahas soal ciri-ciri anak-anak Pak Suryo dan Bu Arum ya~

Pak Suryo adalah suami dan ayah, Bu Arum adalah istri dan ibu, Kevin adalah anak laki-laki, dan Jessica adalah anak perempuan.

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





Lanjutannya~

Ciri Kevin dan Jessica yang ditandai **font tebal adalah ciri yang diwariskan** dari Pak Suryo dan Bu Arum sebagai orang tua.

Namun, ada ciri pada Kevin dan Jessica yang nggak dimiliki oleh Pak Suryo dan Bu Arum (ditandai dengan font yang tidak tebal).



Kevin

- Kulit putih
- Mata bulat
- **Berambut cokelat muda**
- Suka pakai celana berwarna biru
- Suka pakai sepatu warna hitam

Jessica

- Kulit coklat
- Mata bulat
- **Berambut cokelat tua**
- Suka pakai baju warna pink
- Suka mengikat rambut



Begini kalau analoginya diterapkan pada class~

Nah, parent class ini ibarat Pak Suryo dan Bu Arum, sementara child class / sub class ini ibarat Kevin dan Jessica.

Parent class mewariskan semua property-nya kepada child class, tetapi ada beberapa property pada child class yang bukan berasal dari parent class nya.

Parent Class

```
//Buat class dengan nama 'Manusia'  
class Manusia {  
    constructor(nama, alamat) {  
        this.nama = nama;  
        this.alamat = alamat;  
    }  
  
    sapa() {  
        console.log(`Hallo, nama saya ${this.nama}`);  
    }  
  
    kerja() {  
        console.log('Lagi kerja nih...');  
    }  
}
```

Child Class

Property yang
bukan
diwariskan dari
parent class

```
class Guru extends Manusia {  
    constructor(namaGuru, alamatGuru, bidangMataPelajaran) {  
        super(namaGuru, alamatGuru);  
        this.bidangMataPelajaran = bidangMataPelajaran;  
    }  
  
    sapaMurid() {  
        super.sapa();  
    }  
}
```



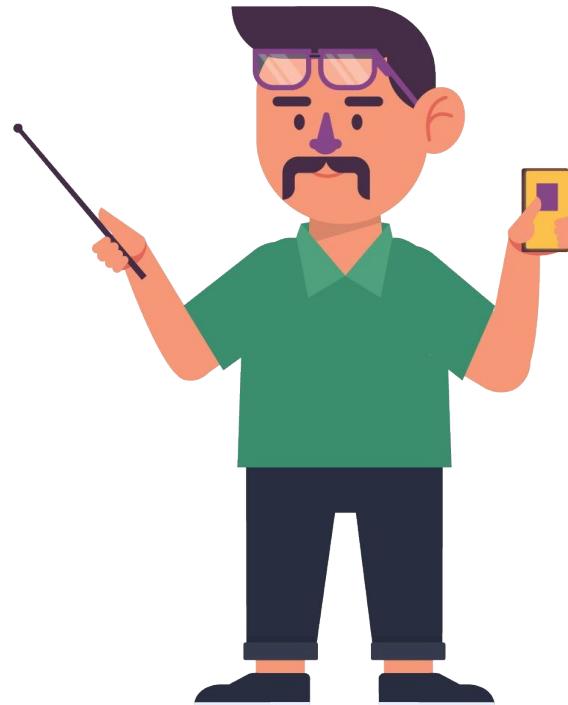
Kalau kamu masih bingung, ini penjelasan tentang parent class dan child class:

Super class atau Parent class

Adalah class yang semua fiturnya diwariskan kepada class turunannya.

Sub-class atau Child class

Adalah 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.





Nah, salah satu keuntungan dari inheritance adalah reusability!

Misalnya kamu ingin membuat class baru, tapi ternyata sebelumnya sudah ada class yang berisi kode yang kita butuhkan di class baru tersebut.

Di kasus ini, kamu bisa menurunkan class baru itu dari class yang sudah ada tadi.

Dengan begitu, berarti kamu menggunakan kembali fitur dari class yang sudah ada, misalnya fitur metodenya.





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

Begini nih..





Kita akan buat class ‘Manusia’ sebagai Super Class/Parent Class

Class ‘Manusia’ memiliki:

- Property: nama, alamat
- Constructor dengan parameter
- Method untuk menampilkan property

```
//Buat class dengan nama 'Manusia'  
class Manusia {  
    constructor(nama, alamat) {  
        this.nama = nama;  
        this.alamat = alamat;  
    }  
  
    sapa() {  
        console.log(`Hallo, nama saya ${this.nama}`);  
    }  
  
    kerja() {  
        console.log('Lagi kerja nih...');  
    }  
}
```



Sementara Class ‘Guru’ adalah Child Class/Sub Class

Syntax extends berarti class Manusia merupakan class Parent dari class Guru.

Class ‘Guru’ memiliki:

- Property: namaGuru, alamatGuru, bidangMataPelajaran
- Constructor dengan parameter
- Method untuk menampilkan property

```
/*Buat class baru dengan nama 'Guru', tapi tambahkan keyword 'extends'. Setelah keyword 'extends' tambahkan nama Super Class/Parent Class yang telah kita buat sebelumnya yaitu class 'Manusia'
```

```
class Guru extends Manusia {  
    constructor(namaGuru, alamatGuru, bidangMataPelajaran) {  
        super(namaGuru, alamatGuru);  
        this.bidangMataPelajaran = bidangMataPelajaran;  
    }  
    sapaMurid() {  
        super.sapa();  
    }  
}
```



Begini contoh simulasi inheritance

Keyword “extends” berfungsi untuk menurunkan semua property dan method yang berada di parent class (di contoh gambar adalah class ‘Manusia’) agar semua property pada parent class/super class bisa di akses pada class ‘Guru’



```
//Buat class untuk Manusia
class Manusia {
  constructor(nama, alamat) {
    this.nama = nama;
    this.alamat = alamat;
  }

  sapa() {
    console.log(`Hallo, nama saya ${this.nama}`);
  }

  kerja() {
    console.log('Lagi kerja nih...');
  }
}
```

```
// Buat sub class dengan nama 'Guru'
// kemudian extend class 'Guru' ke parent class 'Manusia'
class Guru extends Manusia {
  constructor(namaGuru, alamatGuru, bidangMataPelajaran) {
    super(namaGuru, alamatGuru);
    this.bidangMataPelajaran = bidangMataPelajaran;
  }

  sapaMurid() {
    super.sapa();
  }
}

const guruHendra = new Guru(
  'Hendra Majareng',
  'Jl. Rajawali, Pontianak',
  'Matematika'
);
guruHendra.sapaMurid();
// => Output: 'Hallo, nama saya Hendra Majareng'

console.log(guruHendra);
// => Output: { nama : 'Hendra Majareng', alamat: 'Jl. Rajawali, Pontianak', bidangMataPelajaran: 'Matematika' }
```

Lanjutan contoh simulasi inheritance

Keyword “super” berfungsi untuk memanggil constructor pada parent class. Dengan kata lain ketika kamu panggil super (namaGuru, alamatGuru) pada class ‘Guru’ itu sama dengan kamu memanggil new Manusia (namaGuru, alamatGuru)

```
● ○ ●

//Buat class untuk Manusia
class Manusia {
  constructor(nama, alamat) {
    this.nama = nama;
    this.alamat = alamat;
  }

  sapa() {
    console.log(`Hallo, nama saya ${this.nama}`);
  }

  kerja() {
    console.log('Lagi kerja nih...');
  }
}
```

```
● ○ ●

// Buat sub class dengan nama 'Guru'
// kemudian extends class 'Guru' ke parent class 'Manusia'
class Guru extends Manusia {
  constructor(namaGuru, alamatGuru, bidangMataPelajaran) {
    super(namaGuru, alamatGuru);
    this.bidangMataPelajaran = bidangMataPelajaran;
  }
  sapaMurid() {
    super.sapa();
  }
}

const guruHendra = new Guru(
  'Hendra Majarenta',
  'Jl. Rajawali, Pontianak',
  'Matematika'
);
guruHendra.sapaMurid();
// => Output: 'Hallo, nama saya Hendra Majarenta'

console.log(guruHendra);
// => Output: { nama : 'Hendra Majarenta', alamat: 'Jl. Rajawali, Pontianak', bidangMataPelajaran: 'Matematika' }
```



Konsep inheritance menggunakan keyword “super”

Dengan menggunakan variable **super** yang terdapat pada **Child Class**, kamu bisa memanggil semua method yang terdapat pada Parent Class. Ini lah yang dinamakan konsep **Inheritance** atau **penurunan**. Yaitu penurunan method dan property dari Parent Class ke Child Class.

```
//Buat class untuk Manusia
class Manusia {
  constructor(nama, alamat) {
    this.nama = nama;
    this.alamat = alamat;
  }
  sapa() {
    console.log(`Hallo, nama saya ${this.nama}`);
  }
  kerja() {
    console.log('Lagi kerja nih...');
  }
}
```

```
// Buat sub class dengan nama 'Guru'
// kemudian extends class 'Guru' ke parent class 'Manusia'
class Guru extends Manusia {
  constructor(namaGuru, alamatGuru, bidangMataPelajaran) {
    super(namaGuru, alamatGuru);
    this.bidangMataPelajaran = bidangMataPelajaran;
  }
  sapaMurid() {
    super.sapa();
  }
}

const guruHendra = new Guru(
  'Hendra Majarenta',
  'Jl. Rajawali, Pontianak',
  'Matematika'
);
guruHendra.sapaMurid();
// => Output: 'Hallo, nama saya Hendra Majarenta'

console.log(guruHendra);
// => Output: { nama : 'Hendra Majarenta', alamat: 'Jl. Rajawali, Pontianak, bidangMataPelajaran: 'Matematika' }
```



Ini adalah hasil dari penerapan inheritance!

Ketika method sapaMurid() dipanggil, **method sapaMurid() akan memanggil method super.sapa()**. Nah, method sapa() ini terdapat pada Class Manusia yang merupakan Parent Class dari Class Guru.

Dengan menggunakan variable super yang terdapat pada Child Class, kamu bisa memanggil semua method yang terdapat pada Parent Class.

```
//Buat class untuk Manusia
class Manusia {
  constructor(nama, alamat) {
    this.nama = nama;
    this.alamat = alamat;
  }

  sapa() {
    console.log(`Hallo, nama saya ${this.nama}`);
  }

  kerja() {
    console.log('Lagi kerja nih...');
  }
}
```

```
// Buat sub class dengan nama 'Guru'
// kemudian extends class 'Guru' ke parent class 'Manusia'
class Guru extends Manusia {
  constructor(namaGuru, alamatGuru, bidangMataPelajaran) {
    super(namaGuru, alamatGuru);
    this.bidangMataPelajaran = bidangMataPelajaran;
  }

  sapaMurid() {
    super.sapa();
  }
}

const guruHendra = new Guru(
  'Hendra Majarenta',
  'Jl. Rajawali, Pontianak',
  'Matematika'
);
guruHendra.sapaMurid();
// => Output: 'Hallo, nama saya Hendra Majarenta'

console.log(guruHendra);
// => Output: { nama : 'Hendra Majarenta', alamat: 'Jl. Rajawali, Pontianak, bidangMataPelajaran: 'Matematika' }
```



Oke, kita lanjutkan yaa!

Setelah inheritance, dalam konsep OOP juga ada encapsulation.

FYI, saat membahas **encapsulation**, kita juga bakal belajar beberapa jenis **visibility** dalam deklarasi kelas.





Kita pakai analogi dulu yuk biar kamu lebih gampang memahami encapsulation.

Misalnya kamu pakai mesin ATM buat ambil atau setor uang. Kamu nggak tahu proses yang ada di dalam mesin itu, kan?

Pasti kamu tahu ny cuma harus masukin kartu ATM dan PIN, pilih jumlah nominal yang kamu butuh, lalu uang otomatis keluar.

Padahal nih ya, di dalam mesin ATM sebenarnya ada teknik enkapsulasi yang berjalan dan nggak diketahui olehmu sebagai nasabah.





Nah, kira-kira itulah yang terjadi di JavaScript dengan teknik encapsulation.

Dari analogi di atas, nasabah nggak tahu cara kerja ATM karena mereka nggak bisa lihat cara kerjanya. Sama kayak method atau variabel yang menggunakan visibility private agar class lain nggak bisa akses variabel atau method yang ada di dalam class tersebut.



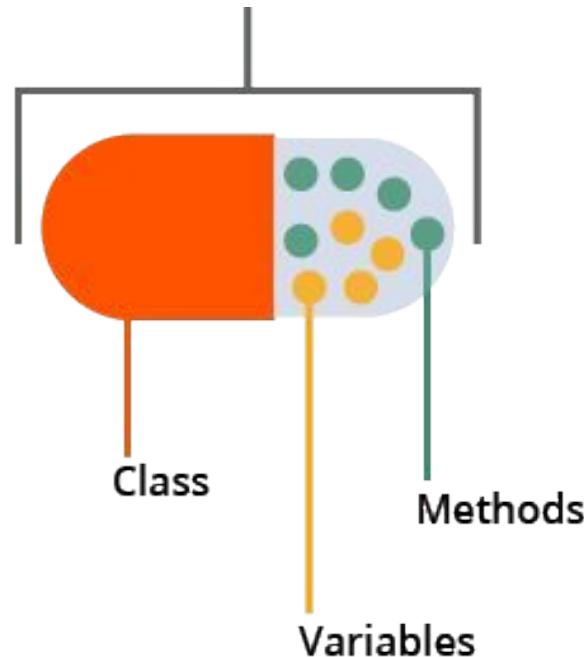


Jadi apa sih encapsulation itu?

Encapsulation (pembungkusan) itu ibaratnya kayak kapsul. Lihat ilustrasi contoh pada gambar di samping.

Jadi dengan encapsulation, data bisa disembunyikan dengan suatu cara yang dinamakan '**visibility**', **tujuannya agar method dan variable nggak bisa diakses secara langsung dari luar class.**

Encapsulation



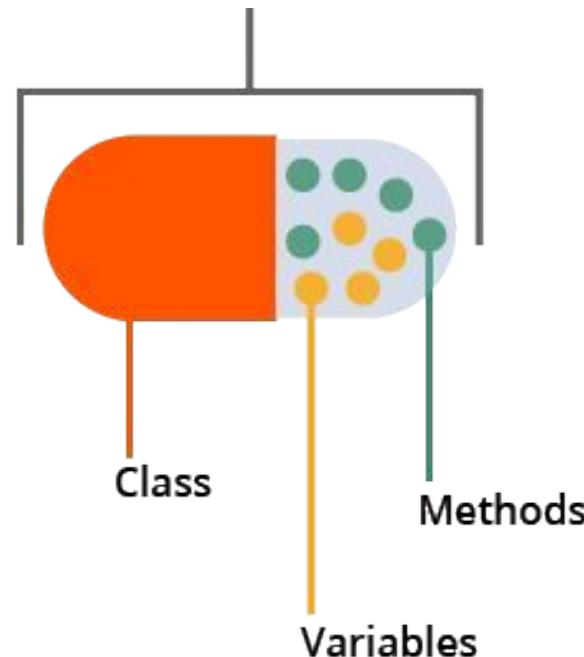


Lalu apa saja yang bisa kamu sembunyikan dalam encapsulation?

Yang kamu sembunyikan bisa data berupa prosedur atau property yang sifatnya sensitif dan nggak boleh diubah-ubah seenaknya.

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

Encapsulation





Ngomongin tentang visibility, ada 2 visibility yang akan kita pelajari, yaitu **private** dan **public**.





Public

Adalah suatu visibility level di mana kamu mendefinisikan suatu method atau property secara publik.

Artinya, method/property tersebut bisa dipanggil di luar deklarasi class.

Semua property objek dan method dari instance Class 'Barang' bisa diakses tanpa ada error

```
class Barang {  
    constructor(namaBarang, hargaBarang, beratBarang, warna) {  
        this.namaBarang = namaBarang;  
        this.beratBarang = beratBarang;  
        this.hargaBarang = hargaBarang;  
        this.warna = warna;  
    }  
  
    getDetailTipeBarang() {  
        return `${this.namaBarang} - ${this.warna}`;  
    }  
}  
  
const laptop = new Barang('Lenovo', 300000, 3, 'Black Gold')  
  
console.log(laptop.namaBarang); // Output: 'Lenovo'  
console.log(laptop.beratBarang); // Output: 3  
console.log(laptop.hargaBarang); // Output: 300000  
console.log(laptop.warna); // Output: 'Black Gold'  
console.log(laptop.getDetailTipeBarang()); // Output 'Lenovo - Black Gold'
```



Private

Suatu **method/property yang nggak bisa diakses di luar blok class.** Ini berarti kamu nggak bisa akses method/property dari luar kurung kurawal class/scope dari kelas tersebut.

Untuk mendeklarasikan suatu private method atau private property, kamu bisa pakai tanda pagar (#) sebelum nama class.

```
class Barang {
    // Deklarasi private properti disini, menggunakan tanda #
    #beratBarang;
    #hargaBarang;
    #warna;

    constructor(namaBarang, hargaBarang, beratBarang, warna) {
        this.namaBarang = namaBarang;
        this.#beratBarang = beratBarang;
        this.#hargaBarang = hargaBarang;
        this.#warna = warna;
    }

    getDetailTipeBarang() {
        return `${this.namaBarang} - ${this.#warna}`;
    }
}

const laptop = new Barang('Lenovo', 300000, 3, 'Black Gold');

console.log(laptop.namaBarang); // Output: 'Lenovo'
console.log(laptop.beratBarang); // Output: undefined => karena beratBarang private properti
console.log(laptop.hargaBarang); // Output: undefined => karena beratBarang private properti
console.log(laptop.warna); // Output: undefined => karena beratBarang private properti
console.log(laptop.#warna); // Error: Private field '#warna' must be declared in an enclosing class
console.log(laptop.getDetailTipeBarang()); // Output 'Lenovo - Black Gold'
```



Private method juga nggak bisa jalan dalam class yang mewarisi class itu

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

```
class Barang {  
    // Deklarasi private properti disini, menggunakan tanda #  
    #beratBarang;  
    #hargaBarang;  
    #warna;  
  
    constructor(namaBarang, hargaBarang, beratBarang, warna) {  
        this.namaBarang = namaBarang;  
        this.#beratBarang = beratBarang;  
        this.#hargaBarang = hargaBarang;  
        this.#warna = warna;  
    }  
  
    getDetailTipeBarang() {  
        return `${this.namaBarang} - ${this.#warna}`;  
    }  
}  
  
const laptop = new Barang('Lenovo', 300000, 3, 'Black Gold');  
  
console.log(laptop.namaBarang); // Output: 'Lenovo'  
console.log(laptop.beratBarang); // Output: undefined => karena beratBarang private properti  
console.log(laptop.hargaBarang); // Output: undefined => karena beratBarang private properti  
console.log(laptop.warna); // Output: undefined => karena beratBarang private properti  
console.log(laptop.#warna); // Error: Private field '#warna' must be declared in an enclosing class  
console.log(laptop.getDetailTipeBarang()); // Output 'Lenovo - Black Gold'
```



Begini penjelasan private visibility di dalam sebuah coding

Semua property objek dan method dari instance Class 'Barang' bisa diakses tanpa ada error

Assign value private property

Private property dapat diakses di dalam blok Class

Apabila private property diakses di luar blok Class, maka akan error

```
class Barang {  
    // Deklarasi private properti disini, menggunakan tanda #  
    #beratBarang;  
    #hargaBarang;  
    #warna;  
    constructor(namaBarang, hargaBarang, beratBarang, warna) {  
        this.namaBarang = namaBarang;  
        this.#beratBarang = beratBarang;  
        this.#hargaBarang = hargaBarang;  
        this.#warna = warna;  
    }  
  
    getDetailTipeBarang() {  
        return `${this.namaBarang} - ${this.#warna}`;  
    }  
}  
  
const laptop = new Barang('Lenovo', 300000, 3, 'Black Gold');  
  
console.log(laptop.namaBarang); // Output: 'Lenovo'  
console.log(laptop.beratBarang); // Output: undefined => karena beratBarang private properti  
console.log(laptop.hargaBarang); // Output: undefined => karena beratBarang private properti  
console.log(laptop.warna); // Output: undefined => karena beratBarang private properti  
console.log(laptop.#warna); // Error: Private field '#warna' must be declared in an enclosing class  
console.log(laptop.getDetailTipeBarang()); // Output 'Lenovo - Black Gold'
```



Kesimpulannya, **fungsi encapsulation** atau alasan datamu harus dibungkus antara lain:

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





Inheritance, sudah. Capsulation sudah.
Selanjutnya, masih ada abstraction.

Kayak apa ya cara kerja abstraction?





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

Mungkin saat dengar kata orang yang kamu bayangkan adalah polisi, hakim, tentara, atau dosen killer.





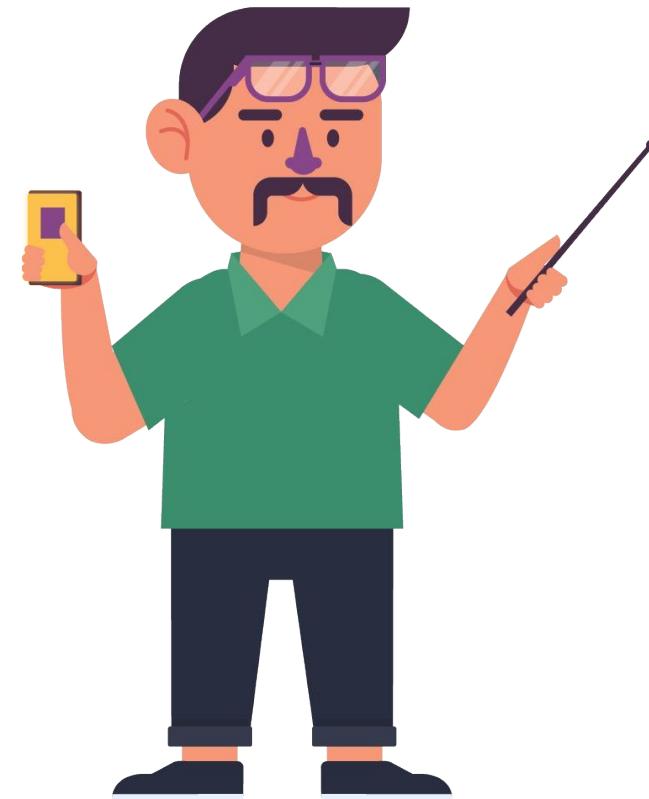
Nah, bayanganmu yang muncul tentang ‘orang’ itulah yang disebut abstraksi.

Yups, kata “orang” sendiri masih bersifat abstrak, tapi kamu bisa membayangkan konsep “orang” itu seperti apa.

Polisi, hakim, tentara, dan dosen masih lebih nyata dibandingkan dengan sekadar kata “orang”.

Prinsip abstraksi ini juga ada dalam OOP dan kita sebenarnya sering menggunakannya di kehidupan tanpa disadari.

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





Biar kebayang, lihat contoh kode yang pakai konsep abstraksi yuk

Pada contoh di samping, ada dua class yaitu **“Human” dan “Police”**.

Kita sengaja bikin kondisi untuk abstract di class Human, agar memberikan pesan error kalau kita nggak sengaja meng-instance class Human.

```
class Hzman { 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 kamu coba melakukan instansiasi (bikin objek) dari **class Human yang sudah punya kondisi untuk abstraction**, maka kamu 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 janji, ini adalah poin terakhir dalam konsep OOP yang bakalan kita bahas, yaitu tentang konsep **Polymorphism**.

Langsung aja kita bahas..

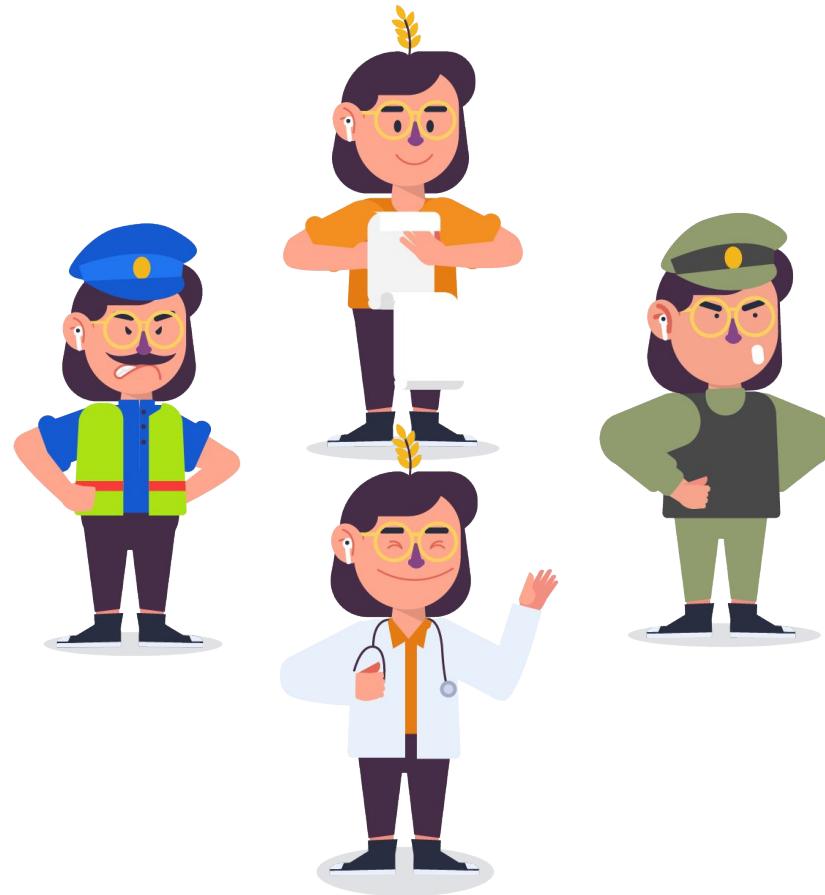




Yuk kenalan sama Polymorphism~

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

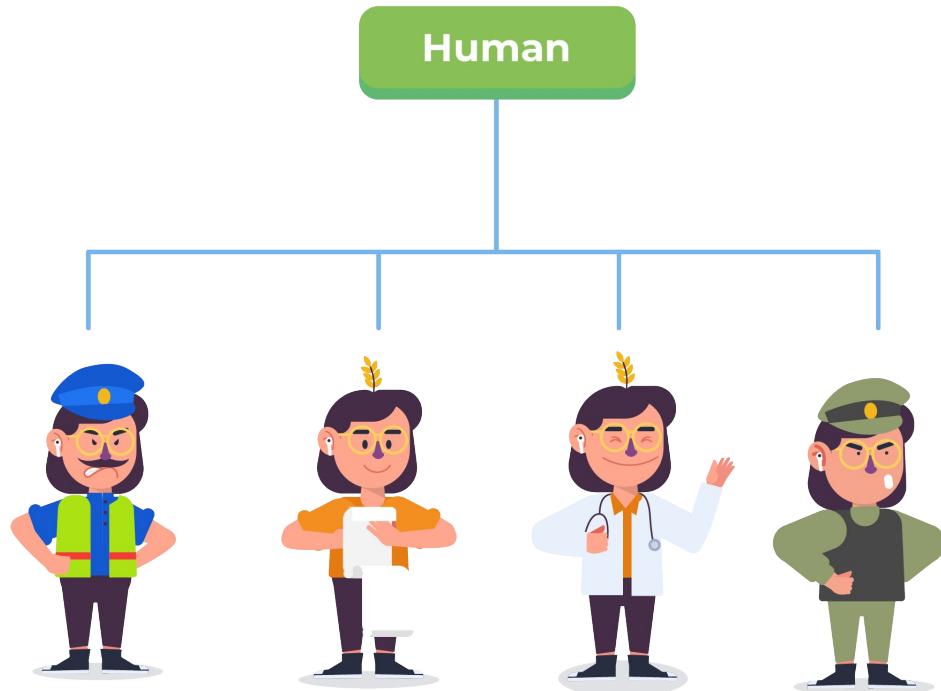
Prinsip ini berlaku ketika kamu punya banyak class yang terkait satu sama lain melalui inheritance.





Misalnya, kamu punya 4 *class*, yaitu Human (*abstract*), Doctor, Police, Writer, and Army, maka 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 punya method *save* untuk menyelamatkan orang lain.





Kayak gini contoh kode berkonsep polymorphism dari ketentuan itu:

Pertama, kamu buat class Human sebagai parent class.

Kemudian, kamu buat module/helper untuk public server dan military.

Nah, untuk implementasi module/helper ini kita pakai konsep **mix-ins**.

```
● ● ●

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!" )
}
}
```



Kenapa harus pakai konsep Mix-ins? Yuk simak baik-baik~

Mix-ins atau abstract subclasses ini ibarat suatu template untuk class.

Kamu harus pakai konsep mix-ins ini karena **class di ECMAScript cuma bisa memiliki satu superclass**, sedangkan kamu 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!")
}
}
```



Selanjutnya, kamu buat class yang lain.

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

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

```
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  
  }  
}
```



Begini pula untuk class Army dan Writer seperti contoh berikut ini.

Di dalam class tersebut, kamu 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();
    }
}
```



Kemudian coba lakukan instantiate, maksudnya kamu membuat objek baru dari masing-masing class.

```
/* 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"
})
```



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

Nahhh, seperti itulah cara kerja 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!
```

Saatnya kita Quiz!





1. Apa output yang akan muncul apabila kode di samping dieksekusi?

- A. 'green'
- B. 'orange'
- C. TypeError

•••

```
class Chameleon {  
    static colorChange(newColor) {  
        this.newColor = newColor;  
        return this.newColor;  
    }  
  
    constructor({ newColor = 'green' } = {}) {  
        this.newColor = newColor;  
    }  
  
    const freddie = new Chameleon({ newColor: 'purple' });  
    console.log(freddie.colorChange('orange'));
```



1. Apa output yang akan muncul apabila kode di samping dieksekusi?

- A. 'green'
- B. 'orange'
- C. TypeError

```
•••  
class Chameleon {  
    static colorChange(newColor) {  
        this.newColor = newColor;  
        return this.newColor;  
    }  
  
    constructor({ newColor = 'green' } = {}) {  
        this.newColor = newColor;  
    }  
  
    const freddie = new Chameleon({ newColor: 'purple' });  
    console.log(freddie.colorChange('orange'));
```

Fungsi `colorChange` adalah statis. Metode statis dirancang hanya dapat aktif pada konstruktor dimana fungsi itu dibuat, dan tidak bisa dibawa ke-turunannya. Kita tahu bahwa `freddie` adalah sebuah turunan, maka fungsi itu tidak bisa turun, dan tidak tersedia pada instance `freddie`: sebuah pesan `TypeError` akan dikembalikan.



2. Apa output yang akan muncul apabila kode di samping dieksekusi?

- A. Lydia
- B. Sarah
- C. Error: cannot redeclare Person



```
class Person {  
    constructor() {  
        this.name = 'Lydia';  
    }  
}  
  
Person = class AnotherPerson {  
    constructor() {  
        this.name = 'Sarah';  
    }  
};  
  
const member = new Person();  
console.log(member.name);
```



2. Apa output yang akan muncul apabila kode di samping dieksekusi?

- A. Lydia
- B. Sarah
- C. Error: cannot redeclare Person

● ● ●

```
class Person {
    constructor() {
        this.name = 'Lydia';
    }
}

Person = class AnotherPerson {
    constructor() {
        this.name = 'Sarah';
    }
};

const member = new Person();
console.log(member.name);
```

Kita dapat mengatur kelas yang sama dengan kelas / fungsi konstruktor lainnya. Dalam kasus ini, kita mengatur Person sama dengan AnotherPerson. Nama pada konstruktor ini adalah Sarah, jadi nama properti yang baru pada Person instance member adalah "Sarah".



3. Apa output yang akan muncul apabila kode di samping dieksekusi?

- A. 1
- B. 2
- C. 3



```
1 class Counter {  
2   constructor() {  
3     this.count = 0;  
4   }  
5  
6   increment() {  
7     this.count++;  
8   }  
9 }  
10  
11 const counterOne = new Counter();  
12 counterOne.increment();  
13 counterOne.increment();  
14  
15 const counterTwo = counterOne;  
16 counterTwo.increment();  
17  
18 console.log(counterOne.count);
```



3. Apa output yang akan muncul apabila kode di samping dieksekusi?

- A. 1
- B. 2
- C. 3

```
● ● ●  
1 class Counter {  
2   constructor() {  
3     this.count = 0;  
4   }  
5  
6   increment() {  
7     this.count++;  
8   }  
9 }  
10  
11 const counterOne = new Counter();  
12 counterOne.increment();  
13 counterOne.increment();  
14  
15 const counterTwo = counterOne;  
16 counterTwo.increment();  
17  
18 console.log(counterOne.count);
```

counterOne adalah instance/bentuk dari Class Counter. Class Counter mempunyai property dengan nama count yang mana value awalnya 0. Dan didalam Class Counter ada method yang Bernama increment(), method increment ini setiap kali dipanggil bakal menambahkan nilai count menjadi satu. Nah pada baris kode ke 12 dan 13, kita telah memanggil method increment() sebanyak dua kali, sehingga nilai dari property count saat ini adalah 2.

Setelah itu, kita membuat sebuah variable dengan nama counterTwo, dan mengeset variable tersebut sama dengan variable counterOne, nah dengan melakukan ini, kita membuat variable counterTwo menjadi bagian dari Class Counter.



4. Syntax kode diatas merupakan cerminan salah satu konsep OOP yaitu..

- A. Encapsulation
- B. Inheritance
- C. Polymorphism

```
● ● ●  
1 class Counter {  
2     #number = 10  
3  
4     increment() {  
5         this.#number++  
6     }  
7  
8     getNum() {  
9         return this.#number  
10    }  
11 }
```



4. Syntax kode diatas merupakan cerminan salah satu konsep OOP yaitu..

- A. Encapsulation
- B. Inheritance
- C. Polymorphism

```
 1 class Counter {  
 2     #number = 10  
 3  
 4     increment() {  
 5         this.#number++  
 6     }  
 7  
 8     getNum() {  
 9         return this.#number  
10    }  
11 }
```

Private property merupakan konsep encapsulation yaitu cara membatasi akses langsung ke property atau method diluar blok class.



5. Apa output yang akan muncul apabila kode diatas di eksekusi?

- A. I'm pink. *
- B. I'm pink. * I'm a bird. *
- C. I'm a bird. * I'm pink. *



```
1 class Bird {  
2     constructor() {  
3         console.log("I'm a bird. 🦢");  
4     }  
5 }  
6  
7 class Flamingo extends Bird {  
8     constructor() {  
9         console.log("I'm pink. 🌸");  
10    super();  
11 }  
12 }  
13  
14 const pet = new Flamingo();  
15
```



5. Apa output yang akan muncul apabila kode diatas di eksekusi?

- A. I'm pink. *
- B. I'm pink. * I'm a bird. *
- C. I'm a bird. * I'm pink. *

```
 1 class Bird {  
 2   constructor() {  
 3     console.log("I'm a bird. 🦢");  
 4   }  
 5 }  
 6  
 7 class Flamingo extends Bird {  
 8   constructor() {  
 9     console.log("I'm pink. 🌸");  
10     super();  
11   }  
12 }  
13  
14 const pet = new Flamingo();  
15
```

Kita membuat variabel pet yang merupakan turunan dari class Flamingo. Saat kita membuat turunan, constructor pada Flamingo dipanggil. Pertama, "I'm pink. 🌸" ditampilkan, setelah itu kita memanggil super(). super() memanggil konstruktor class induk, Bird. Constructor pada Bird dipanggil, dan menampilkan "I'm a bird. 🦢".

Terima Kasih!



Next Topic

loading...