**Introduction**



Secara umum tipe data terbagi menjadi 2, yaitu tipe data ***primitive*** dan ***non primitive***.

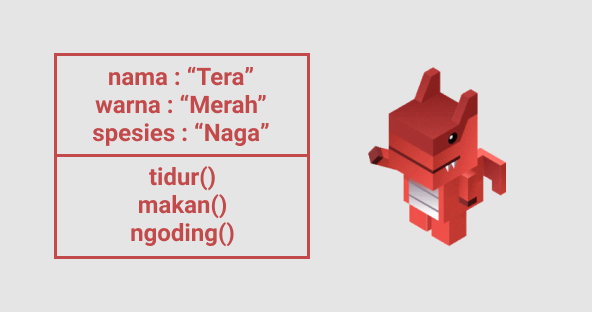
* Tipe data *primitive* adalah tipe data yang hanya dipakai untuk menyimpan data seperti: *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* dan *char*.
* Tipe data *non primitive* adalah tipe data yang selain dipakai untuk menyimpan data, tetapi juga mempunyai metode bawaan (*built-in method*) untuk dilakukan modifikasi dan operasi oleh *programmer*, seperti: *String*, *Array*, *Class* dan *Object*.

Berbicara terkait *Object*, dalam dunia pemrograman terdapat paradigma yang cukup populer yaitu **OOP** (*Object Oriented Programming*).

**Apa itu OOP?**

**OOP adalah suatu paradigma** atau konsep di dunia pemrograman yang **berorientasikan pada objek**. Biasanya objek memiliki 2 komponen, yaitu ciri-ciri dari objek tersebut (**properti**) dan kemampuan yang dapat dilakukan oleh objek tersebut (***method***).

Contohnya pada dunia nyata terdapat objek makhluk hidup seperti **manusia** yang memiliki **nama**, **umur**, dan\*\* jenis kelamin\*\*. Manusia juga dapat melakukan aktifitas seperti **tidur**, **belajar**, **bermain** dan sebagainya.



**Kenapa harus menggunakan OOP?**

Aplikasi yang dibuat semakin lama akan semakin besar dan baris kodenya pun semakin lama akan semakin banyak. Maka dari itu, kamu membutuhkan teknik baru untuk dapat mengatur manajemen kode yang kamu punya sehingga akan lebih mudah dikembangkan dan dilakukan *update*.

OOP memiliki 4 pilar utama yaitu:

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

**OOP juga banyak digunakan pada bahasa pemrograman lain** seperti Java, C++, C#, Kotlin, dan masih banyak lagi. Di topik selanjutnya kamu akan mempelajari pemahaman dasar dari OOP dan cara penerapannya pada bahasa pemrograman JavaScript.

📝**Catatan**:

JavaScript tidak sepenuhnya mendukung konsep OOP, tapi kamu akan mempelajari pemahaman dasar dari paradigma OOP.

**Class**

Objek pada JavaScript, dapat langsung dibuat dengan cara menggunakan objek literal seperti berikut ini:

let hewan = {

nama: "Kucing",

kaki: 4,

tidur: function () {

return `${this.nama} sedang tidur`;

},

};

Dengan cara di atas, kamu dapat langsung membuat properti ataupun *method* pada objek yang ingin dibuat. Ketika kamu ingin membuat objek yang serupa, pasti kamu membuatnya seperti ini:

let hewan = {

nama: "Kucing",

kaki: 4,

tidur: function () {

return `${this.nama} sedang tidur`;

},

};

let hewan2 = {

nama: "Ayam",

kaki: 2,

tidur: function () {

return `${this.nama} sedang tidur`;

},

};

Kode di atas mungkin masih terlihat sederhana karena objek yang dibuat masih sedikit. Tapi **bagaimana jika objek yang ingin dibuat ada banyak?** Pasti **kode** yang dimiliki **menjadi tidak efisien** karena ada banyak kode yang sama dibuat secara **berulang-ulang**.

**Bagaimana cara mengatasinya?**

Untuk mengatasi hal tersebut, **kamu dapat menggunakan fitur *class*** yang terdapat pada JavaScript.

Penulisan *class* hadir pada JavaScript ES6. **Dengan adanya *class*, kamu dapat membuat objek yang merupakan *instance* (perwujudan) dari sebuah *class***. Mudahnya, kamu akan membuat cetakannya terlebih dahulu sebelum kamu membuat objeknya. Berikut ini cara membuat suatu class:

class NamaClass {

constructor() {

// properti

}

// method

}

Contoh pembuatan class Hewan dengan properti nama dan kaki dilengkapi dengan *method* .tidur():

class Hewan {

constructor(nama, kaki) {

this.nama = nama; // properti

this.kaki = kaki; // properti

}

// Method

tidur() {

return `${this.nama} sedang tidur`;

}

}

Dari kode di atas, ada beberapa komponen yang perlu kamu perhatikan ketika membuat *class*:

* ***Constructor***, yaitu *method* yang wajib ada pada class jika ingn menyimpan suatu **properti**. Ketika suatu objek dibuat menggunakan class, maka *constructor*-nya juga ikut terpanggil. Parameter pada constructor berguna untuk memberikan data kepada properti.
* ***Properti***, yaitu data objek yang akan disimpan. Properti berada di dalam ***constructor*** dan diawali oleh kata kunci this. Kata kunci this mewakili kepemilikan dari *class* itu sendiri.
* ***Class method***, merupakan *method* yang dapat dimiliki dari sebuah objek. Biasanya dibuat di bawah *constructor*.

Setelah *class* selesai dibuat, yang perlu dilakukan adalah membuat *instance* dari *class* tersebut. **Objek dapat dibuat menggunakan kata kunci new kemudian dilanjutkan dengan nama *class*-nya.** Jangan lupa untuk memberi argumen pada *class* yang dipanggil jika ingin memberikan data secara langsung sesuai dengan parameter yang terdapat pada *constructor*.

let namaObjek = new NamaKelas(args1, args2);

Contoh membuat objek dari class Hewan yang telah kita buat di atas:

let hewan1 = new Hewan("Kucing", 4);

let hewan2 = new Hewan("Ayam", 2);

console.log(hewan1.nama); // Output: Kucing

console.log(hewan2.nama); // Output: Ayam

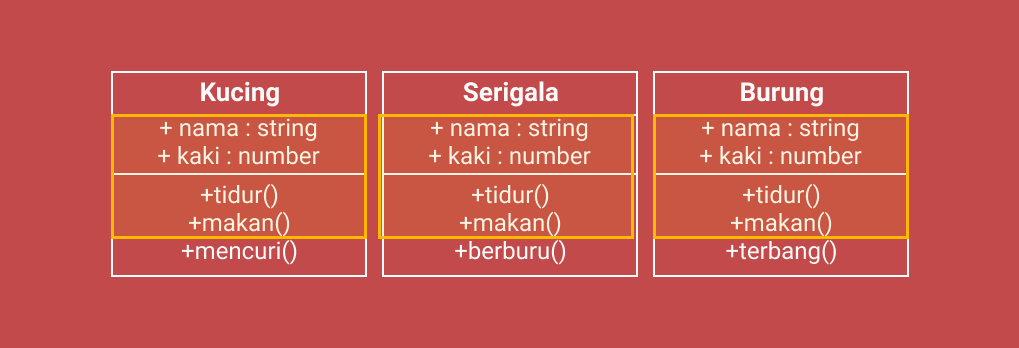
console.log(hewan1.tidur()); // Output: Kucing sedang tidur

console.log(hewan2.tidur()); // Output: Ayam sedang tidur

Setiap objek yang dibuat, akan memiliki properti dan *method* yang sama seperti milik *class*-nya. Dengan memberikan argumen pada *class* yang digunakan, objek akan memiliki nilai properti dan *method* berdasarkan argumen yang sudah diberikan.

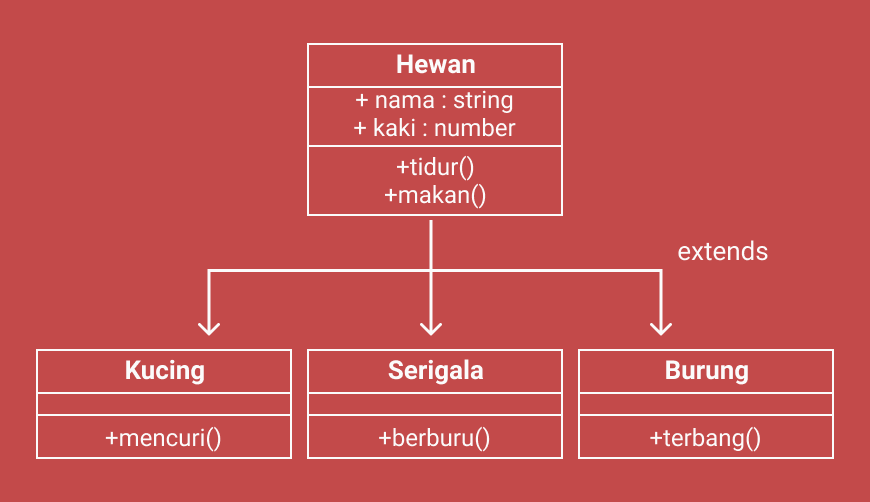
**Inheritance**

Pada topik sebelumnya, kamu sudah belajar membuat objek menjadi lebih efisien menggunakan *class*. Dari apa yang sudah kamu pelajari, ternyata kamu ingin membuat *class* yang serupa (memiliki properti dan *method* yang sama) tetapi ingin menambahkan suatu properti atau *method* tambahan. Mungkin pada awalnya kamu akan berfikir untuk membuatnya seperti ini:



Tapi ternyata berdasarkan gambar di atas, kamu hanya akan mengulangi kesalahan yang sama yaitu membuat kode menjadi tidak efisien. Ingat prinsip **DRY** (*Don't Repeat Yourself*).

Untuk mengatasi hal tersebut, pada topik ini kamu akan mempelajari caranya untuk **menurunkan/mewariskan properti dan *method* pada *class* lain.** Istilah ini biasa disebut dengan ***inheritance***. Sehingga nantinya, *class* yang akan dibuat terlihat seperti ini:



Pada gambar di atas terlihat bahwa, *class* Hewan akan menurunkan semua properti dan *method*-nya pada *class* Kucing, Serigala, dan Burung. Sehingga, *class* yang diturunkan hanya perlu menambahkan properti atau *method* yang diinginkan.

Sebelum mewariskan suatu *class*, pastikan kamu sudah memiliki:

1. ***Parent class***, yang akan menurunkan properti dan *method*-nya kepada *class* lain.
2. ***Child class***, yang akan menerima turunan dari *parent class*.

Untuk dapat menerima turunan dari ***parent class***, ***child class* harus menggunakan kata kunci extends dan diikuti oleh nama *parent class*-nya.**

class NamaClassB extends NamaKelasA {

// method

}

Jika kamu ingin menambahkan properti dan ingin memberikan datanya secara langsung ketika objek dibuat, kamu harus menggunakan method super() pada constructor ***child class*** agar dapat mengakses properti yang sama pada ***parent class***.

class NamaClassB extends NamaKelasA {

constructor() {

super();

// properti

}

// method

}

📝**Catatan**:

*Method* super() digunakan untuk memanggil *constructor* pada *parent class*

Contoh di bawah ini kita akan membuat sebuah *class* Hewan. Lalu kita akan membuat *class* Kucing yang mewarisi sifat dari *class* Hewan menggunakan extends:

class Hewan {

constructor(nama, kaki) {

this.nama = nama;

this.kaki = kaki;

}

tidur() {

return `${this.nama} sedang tidur`;

}

}

// class Hewan mewarisi class Kucing

class Kucing extends Hewan {

constructor(nama, kaki, makanan) {

super(nama, kaki);

this.makanan = makanan;

}

mencuri() {

return `${this.nama} mencuri ${this.makanan}`;

}

}

// membuat objek dari class Kucing

let kucing = new Kucing("Tom", 4, "ikan");

console.log(kucing.nama);

console.log(kucing.tidur());

console.log(kucing.mencuri());

// Output:

// Tom

// Tom sedang tidur

// Tom mencuri ikan

Pada contoh kode di atas, *class* Hewan mewarisi semua properti dan *method* miliknya kepada *class* Kucing dengan menggunakan kata kunci extends.

class Kucing extends Hewan

*Class* Kucing menggunakan *method* super() dan memberikan argumen berupa nama dan kaki. *Method* ini seolah memanggil *constructor* milik ***parent class*** serta memberikan parameter yg dibutuhkan.

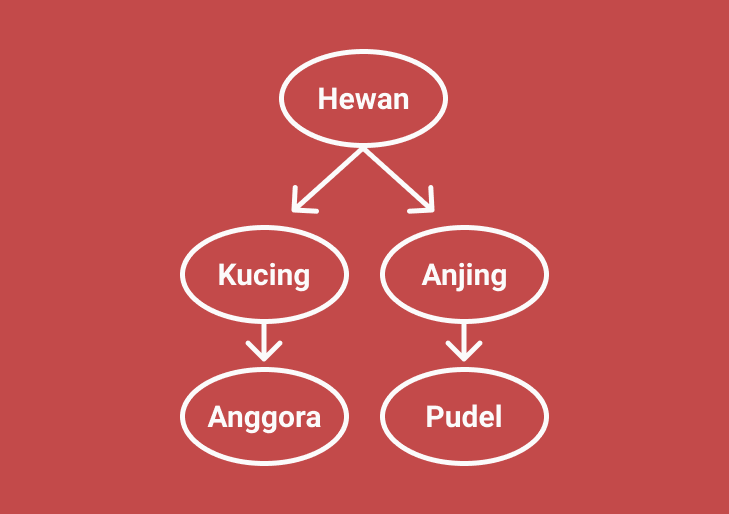
super(nama, kaki);

**Polymorphism**

Pada *child class*, selain mewarisi semua sifat dari *parent*-nya, *child class* juga bisa memiliki kemampuan yang berbeda dengan *parent*-nya. Dengan kata lain, *child class* memiliki metode yang sama dengan *parent*-nya namun bisa jadi berbeda *output* dengan *sibling* atau *parent*-nya, hal ini dikenal dengan sebutan ***polymorphism***.

Secara bahasa, ***Polymorphism*** berasal dari dua kata, yaitu *poly* yang berarti **banyak** dan *morphism* yang berarti **bentuk**.

*Polymorphism* ini juga dikenal pada konsep OOP yang berarti kemampuan dari suatu objek untuk memiliki banyak bentuk. Untuk memperjelas maksud di atas, coba perhatikan hirarki berikut:



* Kucing adalah Hewan
* Anjing adalah Hewan
* Anggora adalah Kucing
* Anggora adalah Hewan

Contoh gambar di atas menjelaskan bahwa, *child class* adalah bagian dari *parent class*. Tetapi *parent class* bukanlah bagian dari *child class*. Walaupun Kucing dan Anjing merupakan bagian dari Hewan, tetapi mereka memiliki karakteristik yang berbeda.

Pada topik sebelumnya, kamu sudah belajar caranya untuk mewarisi suatu *class*. Maka *child class* akan memiliki properti dan *method* dari *parent class*-nya. *Polymorphism* di sini mengarah pada *method* yang dimiliki oleh *child class*.

Perhatikan contoh kode berikut ini:

class Hewan {

constructor(nama) {

this.nama = nama;

}

suara() {

console.log("Ini suara hewan");

}

}

class Kucing extends Hewan {

constructor(nama) {

super(nama);

}

}

let kucing = new Kucing("TOM");

kucing.suara(); //Output: Ini suara hewan

Dari contoh kode di atas, objek yang dibuat dari *class* Kucing, dapat menggunakan *method* suara() yang dimiliki oleh *class* Hewan. Tapi ternyata *output* suara() yang dimiliki oleh class Kucing adalah Ini suara hewan, berbeda dengan yang diharapkan. Lalu bagaimana caranya agar *method* suara() pada *class* Kucing dapat menghasilkan suara miaaaw?

Untuk dapat melakukan hal tersebut, kamu perlu **membuat *method*** pada *child class* **dengan nama yang sama** pada *parent class* tetapi **isinya berbeda**. Cara seperti ini disebut dengan ***overriding***.

Sehingga kode yang seharusnya adalah seperti berikut ini:

class Hewan {

constructor(nama) {

this.nama = nama;

}

suara() {

console.log("suara hewan");

}

}

class Kucing extends Hewan {

constructor(nama) {

super(nama);

}

// Overriding

suara() {

console.log("miaaaw");

}

}

let kucing = new Kucing("TOM");

kucing.suara(); // Output: miaaaw

Dari kode di atas, di dalam *class* Kucing dibuat kembali *method* suara() dengan nama yang sama tetapi isinya berbeda. Sehingga ketika *method* itu dipanggil, maka akan menghasilkan *output* miaaaw.

**Encapsulation**

Objek yang kamu buat selama ini sudah menerapkan konsep yang bernama ***encapsulation***. Konsep ini dimaksudkan untuk **membungkus data variabel dan fungsi menjadi satu kesatuan**. Dengan membungkusnya, bisa menjadikannya sebuah objek yang memiliki properti dan *method*.

Tujuan lain dari *encapsulation* adalah membatasi hak akses pada properti yang dimiliki oleh suatu objek. Sehingga data tetap terjaga dan hanya bisa diakses melalui *method*.

📝**Catatan**:

JavaScript tidak memiliki fitur khusus untuk membatasi akses sebuah properti. Cara di bawah hanya bertujuan untuk memberi pemahaman terkait *encapsulation*.

Kamu dapat mencobanya dengan cara seperti ini:

class NamaClass {

constructor(nilai) {

var \_properti = nilai;

// getter

this.getProperti = function () {

return \_properti;

};

// setter

this.setProperti = function (nilaiBaru) {

\_properti = nilaiBaru;

};

}

// Method

}

Contoh penerapan *encapsulation* pada *class*:

class Hewan {

constructor(nama) {

let \_nama = nama;

this.getNama = function () {

return \_nama;

};

this.setNama = function (nama) {

\_nama = nama;

};

}

}

Penjelasan dari kode di atas adalah sebagai berikut:

* Deklarasikan properti seperti mendeklarasikan variabel biasa.
* Kemudian beri tanda \_ untuk membedakan antara properti dari parameter constructor
* *Method* *setter* dan *getter* untuk mengakses nilai properti.

Sekarang properti pada class Hewan sudah tidak bisa diakses secara langsung seperti ini:

let kucing = new Hewan("TOM");

console.log(kucing.nama) // Output: undefined

Untuk mengakses properti tersebut, kamu perlu menggunakan *getter* untuk mengambil nilai atau *setter* untuk mengubah nilai:

let kucing = new Hewan("TOM"); // membuat instance object

console.log(kucing.getNama()); // memanggil getter. Output: TOM

// Update nama hewan

kucing.setNama('Delta'); // memanggil setter

console.log(kucing.getNama()); // memanggil getter. Output: Delta

**Abstraction**

Pernahkan kamu menekan tombol *power* pada suatu mesin? Pasti pernah dong.

Nah yang kamu lakukan adalah menekan tombol dan seketika mesin tersebut menyala. Namun dibalik itu, terjadi suatu proses yang tidak kita ketahui sehingga mesin tersebut dapat menyala. Bisa dibilang mesin tersebut menerapkan konsep yang bernama ***abstraction***.

**Apa itu *Abstraction*?**

***Abstraction*** adalah konsep untuk **menyembunyikan suatu detil proses** yang tidak perlu diketahui oleh pengguna. ***Abstraction*** mengatur kompleksitas sistem dengan menyembunyikan detail dari internal sistem dan membuatnya ke dalam beberapa sistem kecil. Sehingga pengguna hanya perlu memberi suatu *input* untuk mendapatkan *output* yang diinginkan.

Sebenarnya kamu telah menerapkan konsep ini, seperti ketika kamu menggunakan *built-in method* pada JavaScript misalnya pada ***array***, kamu menggunakan *method* map() untuk iterasi atau pada ***string***, kamu menggunakan *method* indexOf() untuk mengetahui *index* pada sebuah *string*. Kamu tidak tahu kode seperti apa dibalik *method* tersebut. Yang kamu lakukan hanya menggunakannya untuk mendapatkan hasil yang diinginkan.

Untuk memperjelasnya, kamu bisa perhatikan contoh kode di bawah ini:

class Person {

constructor({firstName, lastName, job}) {

this.firstName = firstName;

this.lastName = lastName;

this.job = job;

}

fullName() {

return `${this.firstName} ${this.lastName}`;

}

}

class Job {

constructor(company, position, salary) {

this.company = company;

this.position = position;

this.salary = salary;

}

}

const salah = new Person({

firstName: 'Muhammad',

lastName: 'Salah',

job: new Job('Liverpool', 'striker', 200000)

});

const christ = new Person({

firstName: 'Christiano',

lastName: 'Ronaldo',

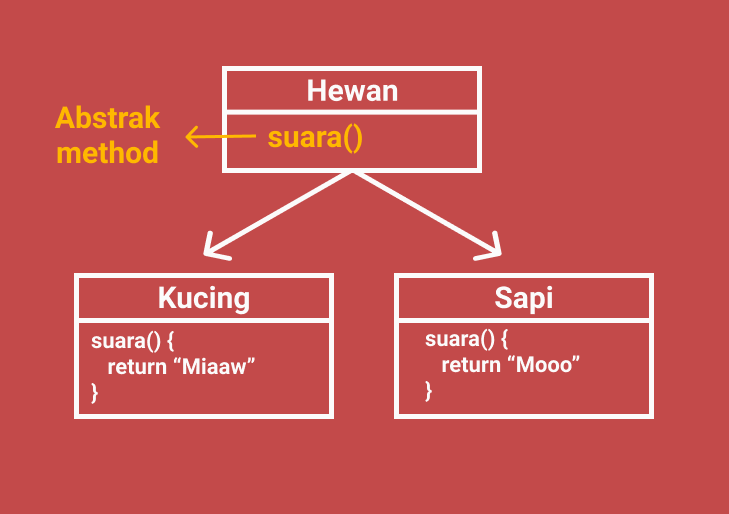
job: new Job('Juventus', 'striker', 1000000)

});

Dari contoh di atas, dapat kita lihat bahwa detil properti pada job disembunyikan ke dalam class Job. Dari sisi *programmer*, hal ini dilakukan agar kode terlihat lebih rapih dan terstruktur karena properti pada job dipisahkan ke dalam sistem yang lebih kecil, yaitu class Job. Namun dari segi pengguna, pengguna tidak dapat melihat dan tidak perlu mengetahui properti apa saja yang terdapat pada class Job.

**Pengertian lain dari *Abstraction***

Selain itu, ***abstraction*** dalam konteks yang berbeda memiliki arti lain yaitu **tidak mendefinisikan isi dari suatu *method***. Ini berarti *method* tersebut masih bersifat abstrak (tidak jelas). Konsep ini sering digunakan pada bahasa pemrograman yang kental OOP-nya.



Seperti contoh gambar di atas, *class* Hewan memiliki *method* suara() yang mana masih belum jelas suara dari hewan apa. Setelah diturunkan ke *class* Kucing dan Sapi, *method* tersebut sudah dapat didefinisikan berdasarkan suara dari hewan yang diturunkan.

📝**Catatan**:

JavaScript tidak memiliki fitur khusus untuk membuat *class* atau *method* menjadi abstrak.