

@arrowfunxtion

Object · Oriented · Programming

PHP OOP · MINI SERIAL

Episode 1: Class · Attribute · Method

Object Oriented Programming

Sebuah **paradigma** dalam pemrograman.

berdasarkan konsep
object (benda)



object kucing

- punya **data** dlm bentuk **fields**
(juga dikenal sebagai **attribute** atau **property**)
- punya **behaviour** / perilaku
dlm bentuk **method**

LIKE @arrowfunxtion

Object vs Class

Class



Class merupakan **blueprint** / rancangan mengenai sebuah **object** yang akan dibuat

Dalam kode, itu berarti sekumpulan definisi yang menjelaskan **fields / attributes** & **behaviour / methods**

LIKE @arrowfunxion

Object vs Class

Class Kucing



Object merupakan benda yang diwujudkan (instantiate) berdasarkan **class** / rancangan.

Object kucing 1



Object kucing 2



LIKE @arrowfunxion



kucing

Fields / attribute dari kucing ini:

Fields	Value
• warna	: merah
• jumlah kaki	: 4
• jenis ekor	: panjang
• makanan favorit	: ikan

LIKE @arrowfunxion

Class & Fields

```
class Kucing {  
    public $warna = "merah";  
    public $jumlah_kaki = 4;  
    public $jenis_ekor = "panjang";  
    public $makanan_fav = "ikan";  
}
```

field

value

LIKE @arrowfunxion



kucing

Behaviour / perilaku dari kucing ini:

- bersuara
- berlari
- berburu

LIKE @arrowfunxtion

Class & Methods

```
class Kucing {
```

```
  // ... fields ...
```

```
  public function bersuara(){  
    return "meowng";  
  }
```

```
  public function berburu(){  
    return "berburu ikan";  
  }
```

```
}
```

methods

LIKE @arrowfunxtion

@arrowfunxtion

Object · Oriented · Programming

PHP OOP · MINI SERIAL

Episode 2: Object

@arrowfunxtion

Kucing.php

```
class Kucing {  
    public $warna = "merah";  
    public $jumlah_kaki = 4;  
    public function berburu(){  
        return "berburu ikan";  
    }  
}
```

Class Kucing

Membuat Object Kucing

berdasarkan class Kucing

```
require("../Kucing.php");
```

```
$kucing = new Kucing;
```

object

class

Object Kucing

```
$kucing = new Kucing;
```

Object \$kucing

- punya semua attribute dari class Kucing
- punya methods dari class Kucing



Object Kucing

```
$kucing = new Kucing;
```

```
echo $kucing->warna; // "merah"
```

```
$kucing->berburu(); // "berburu ikan"
```

mengakses
attribute **warna**
pada object **\$kucing**

memanggil
method **berburu()**
pada object **\$kucing**

Mengubah Nilai Atribut

pada object **\$kucing**

```
$kucing = new Kucing;
```

```
$kucing->warna = "hijau";
```



Object · Oriented · Programming
Episode 3: Constructor
PHP OOP · MINI SERIAL

Ep.3

@arrowfunxtion

Kucing.php

Class Kucing

```
class Kucing {  
    public $warna = "merah";  
    public function berburu(){  
        return "berburu ikan";  
    }  
}
```


Sebelumnya kita mengubah atribut warna `$kucing` dari merah menjadi hijau

Hal itu kita lakukan Setelah *instantiate* object `$kucing`

membuat object

```
$kucing = new Kucing;  
$kucing->warna = "hijau";
```

Gimana kalau kita ingin mengubah atribut saat *instantiate* object?

misal

```
$kucing = new Kucing("Hijau");
```

Constructor method

Kucing.php

```
class Kucing {  
    public function __construct($warna){  
        $this->warna = $warna;  
    }  
    public $warna = "merah";  
    // ... fields & methods lainnya ...  
}
```

```
$kucing = new Kucing("hijau");
```

Pembuatan object
\$kucing

berdasarkan

Class Kucing

```
class Kucing {  
    public function __construct($warna){  
        $this->warna = $warna;  
    }  
    public $warna = "merah";  
    // ... fields & methods lainnya ...  
}
```

Nilai default dari
atribut warna

Penggunaan **Constructor**

membuat 3 object kucing berbeda

```
$kucing1 = new Kucing;
```



```
$kucing2 = new Kucing("hijau");
```



```
$kucing3 = new Kucing("biru");
```



```
echo $kucing1->warna; // "merah"
```



```
echo $kucing2->warna; // "hijau"
```



```
echo $kucing3->warna; // "biru"
```



Object · Oriented · Programming

Episode 4: Inheritance & Overriding

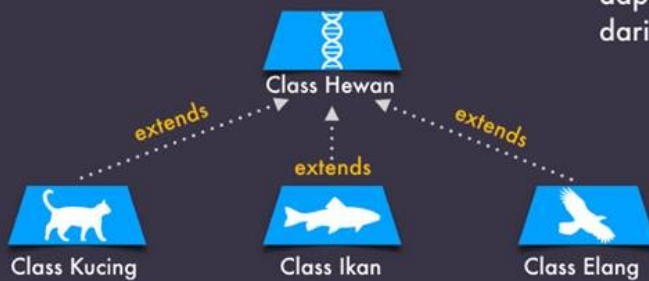
Ep.4

PHP OOP · MINI SERIAL

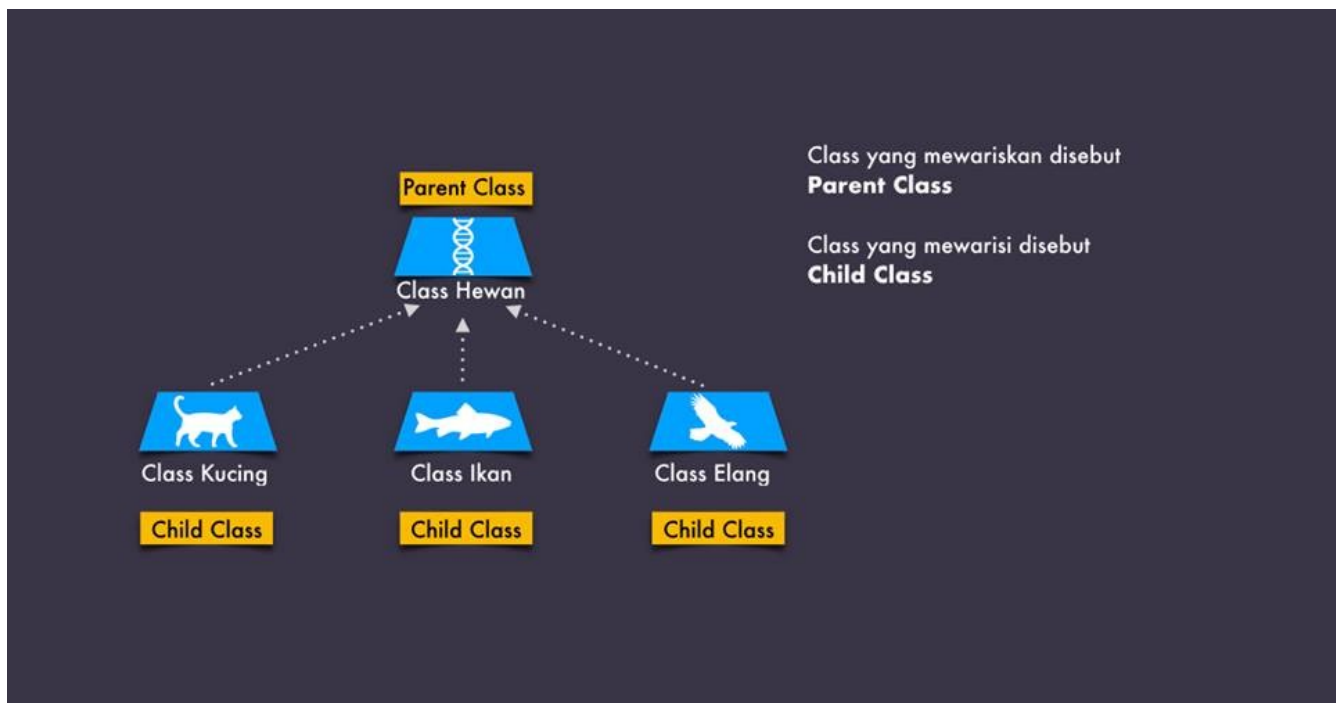
@arrowfunction


Inheritance

Berarti **pewarisan**, yaitu sebuah class dapat mewarisi **atribut & methods** dari class lain.



Class **Kucing**, **Ikan** & **Burung** semuanya mewarisi (*inherit*) dari Class **Hewan**





Class Hewan

atribut:

- warna
- umur

methods:

- bernafas()
- makan()

Hewan.php

```

class Hewan {
    public $warna = "coklat";
    public $umur = 2;
    public function bernafas(){
        echo "Saya bernafas";
    }
    public function makan(){
        echo "Saya makan";
    }
}
        
```



```
class Kucing extends Hewan {  
}
```

- class **Kucing** mewarisi class **Hewan**
- tidak perlu mendefinisikan atribut dan methods yg sudah didefinisikan di class **Hewan**

```
$kucing = new Kucing;
```

```
echo $kucing->warna; // "coklat" ✓
```

```
$kucing->makan(); // "saya makan" ✓
```



```
class Ikan extends Hewan {  
}
```

- class **Ikan** mewarisi class **Hewan**
- tidak perlu mendefinisikan atribut dan methods yg sudah didefinisikan di class **Hewan**

```
$ikan = new Ikan;
```

```
echo $ikan->warna; // "coklat" ✓
```

```
$ikan->makan(); // "saya makan" ✓
```

Ikan.php

```
class Ikan extends Hewan {  
    public function berenang(){  
        echo "Saya bisa berenang";  
    }  
}
```

Child Class juga bisa mendefinisikan **atribut & methods** yang tidak ada di **Parent Class**

Atribut & method class **Ikan**

atribut:

- warna
 - umur
- warisan dari class **Hewan**

methods:

- bernafas()
 - makan()
- warisan dari class **Hewan**
- berenang() method asli di class **Ikan**



Class Hewan

Saya makan



Class Kucing

Saya makan Ikan



Class Ikan

Saya makan Plankton



Class Elang

Saya makan Tikus

Setiap hewan memiliki jenis makanannya sendiri

Maka kita harus melakukan **Overriding** pada method **makan()** di masing-masing **Child Class**

Overriding

```
class Kucing extends Hewan {
    public function makan(){
        echo "Saya makan Ikan";
    }
}
```

```
class Ikan extends Hewan {
    public function makan(){
        echo "Saya makan Plankton";
    }
}
```

```
class Elang extends Hewan {
    public function makan(){
        echo "Saya makan Tikus";
    }
}
```

Untuk melakukan **Overriding** caranya cukup definisikan ulang **atribut / method** di **Child Class** dengan nama yang sama pada **Parent Class**, tapi dengan implementasi yang disesuaikan.

Sebelum overriding

```
$hewan = new Hewan;
$kucing = new Kucing;
$ikan = new Ikan;
$elang = new Elang;

$hewan->makan(); // "Saya makan"
$kucing->makan(); // "Saya makan"
$ikan->makan(); // "Saya makan"
$elang->makan(); // "Saya makan"
```

Sesudah overriding

```
$hewan = new Hewan;
$kucing = new Kucing;
$ikan = new Ikan;
$elang = new Elang;

$hewan->makan(); // "Saya makan"
$kucing->makan(); // "Saya makan Ikan"
$ikan->makan(); // "Saya makan Plankton"
$elang->makan(); // "Saya makan Tikus"
```


Object · Oriented · Programming
Episode 5: Method Chaining

Ep.5

PHP OOP · MINI SERIAL

@arrowfunction

Saat menggunakan PHP framework baik itu **Laravel**, **CakePHP** ataupun **Yii**, kita pasti sering melakukan hal semacam ini:



```
$kucing->berlari()->berburu()->makan();
```

atau

```
$kucing  
->berlari()  
->berburu()  
->makan();
```

Ini disebut **method chaining** yaitu pemanggilan methods secara berantai (*chain*)

```
$kucing->berlari()->berburu()->makan();
```

atau

```
$kucing  
->berlari()  
->berburu()  
->makan();
```

Seperti rantai kan?



```
class Kucing {  
  
    public function berburu($warna){  
        echo "Mulai berburu..";  
        return $this;  
    }  
  
    public function berlari($warna){  
        echo "Mulai berlari..";  
        return $this;  
    }  
  
    public function makan($warna){  
        echo "Mulai makan..";  
        return $this;  
    }  
  
}
```

Agar **method** bisa dipanggil secara berantai (*method chain*) setiap method harus mengembalikan (*return*)

\$this

\$this merujuk ke objek yang dipanggil.

\$this merujuk ke objek yang dipanggil.

```
class Kucing {  
  
    public function berburu($warna){  
        echo "Mulai berburu..";  
        return $this;  
    }  
  
    public function berlari($warna){  
        echo "Mulai berlari..";  
        return $this;  
    }  
  
    public function makan($warna){  
        echo "Mulai makan..";  
        return $this;  
    }  
  
}
```



`$kucing->berlari()->berburu()->makan();`

Setiap selesai memanggil method, maka object **\$kucing** akan dikembalikan, seperti ini:

```
$kucing  
->berlari() // return $kucing  
->berburu() // return $kucing  
->makan(); // return $kucing
```

Sehingga kode di atas sama seperti:

```
$kucing->berlari()  
$kucing->berburu()  
$kucing->makan();
```

```
$kucing->berlari()->berburu()->makan();
```

atau

```
$kucing  
->berlari()  
->berburu()  
->makan();
```

Contoh kode di samping kiri itu sama saja seperti kita memanggil method **berlari()**, **berburu()** & **makan()** secara berurutan seperti ini:

```
$kucing->berlari()  
$kucing->berburu()  
$kucing->makan();
```

Tanpa method chaining

Object · Oriented · Programming

Episode 6: Static & Self Keyword

Ep.6

PHP OOP · MINI SERIAL

@arrowfunction

sebelumnya, kita telah belajar bahwa **attribut & method** dapat diakses setelah kita membuat **object** dari **class** terlebih dahulu

```
// buat object
$kucing = new Kucing;

// akses attribut & method
$kucing->berlari();
$kucing->warna;
```

static keyword

static keyword dapat kita gunakan untuk menjadikan **atribut** atau **method** bisa diakses tanpa perlu membuat **object** terlebih dahulu.

Untuk menandai **atribut** atau **method** sebagai **static**, cukup tambahkan keyword **static** seperti contoh berikut:

```
class Kucing {  
    public static $nama_latin = "Felis Catus";  
    public static function getNamaLatin(){  
        // todo: return nama latin  
    }  
}
```

Nah, sekarang kita bisa mengakses **atribut & method** static tadi seperti ini:

Ga perlu buat **object** terlebih dahulu

Langsung menggunakan nama **class**

Menggunakan **::** bukan **->**

~~// buat object~~

~~\$kucing = new Kucing;~~

Kucing::getNamaLatin();

Kucing::\$nama_latin;

Kecuali untuk chaining, bisa pake **->**, misal Kucing::method1()->method2();

STATIC

```
// akses atribut & method
Kucing::getNamaLatin();
Kucing::$nama_latin;
```

NON STATIC

```
// buat object
$kucing = new Kucing;

// akses atribut & method
$kucing->berlari();
$kucing->warna;
```

self keyword

Mirip dengan konsep `$this` yang telah kita pelajari sebelumnya, tapi untuk lingkup static.

```
class Kucing {  
    public function berburu($warna){  
        echo "Mulai berburu..";  
        return $this;  
    }  
}
```



Sebelumnya, kita sudah belajar bahwa `$this` merujuk pada **object** di mana sebuah **method** dipanggil.

```

class Kucing {
    public function berburu($warna){
        echo "Mulai berburu..";
        return $this;
    }
}

```



Misalnya begini:

```

$kucing1 = new Kucing;
$kucing2 = new Kucing;

$kucing1->berburu();
// return $kucing1 ←..... $this

$kucing2->berburu();
// return $kucing2 ←..... $this

```

```

class Kucing {
    public static $nama_latin = "Felis Catus";

    public static function getNamaLatin(){
        // todo: return nama latin
    }
}

```



Bagaimana caranya agar **method getNamaLatin()** mengembalikan nilai dari **atribut \$nama_latin**?

```

return $this->nama_latin; ?

```



```

class Kucing {
    public static $nama_latin = "Felis Catus";

    public static function getNamaLatin(){
        return self::$nama_latin;
    }
}

```

jawabannya, adalah menggunakan **self** keyword, bukan **\$this**

`return $this->nama_latin;` ❌

Pada **static** method, kita tidak bisa mengakses **\$this**, karena kita tidak membuat **object** terlebih dahulu. Ingat **\$this** merujuk ke **object**

```

class Kucing {
    public static $nama_latin = "Felis Catus";

    public static function getNamaLatin(){
        return static::$nama_latin;
    }
}

```

Selain menggunakan **self** keyword, kita juga bisa menggunakan **static** keyword untuk mengakses **static attribute & method**



Tentu ada perbedaannya antara **self** & **static** keyword ketika digunakan untuk mengakses **static atribut** atau **method**. Tapi kita tidak bahas di sini, PR buat kamu untuk cari tahu sendiri ya!

Object · Oriented · Programming
Episode 7: Visibility

Ep.7

PHP OOP · MINI SERIAL

@arrowfunxtion

Sejauh ini kita sudah belajar membuat **atribut & method**, baik static maupun non-static. Tapi selalu kita menggunakan keyword **public**

public
public
public

MAKANAN
APA ITU?

```
class Kucing {  
    public $warna = "putih";  
    public function berlari($warna){  
        echo "Mulai berlari..";  
        return $this;  
    }  
    public function makan($warna){  
        echo "Mulai makan..";  
        return $this;  
    }  
}
```

LIKE @arrowfunxtion

visibility

Secara sederhana, **visibility** mengatur **siapa** yang bisa **mengakses method** atau **atribut** dari sebuah **class**

Apakah kita sedang membicarakan user role? access control? **Bukan!** Beda bahasan, itu tingkat aplikasi, kita lagi ngomong akses tingkat **class** ?

LIKE @arrowfunxtion

Keyword di depan setiap deklarasi **atribut** atau **method** ini disebut **visibility**

Selain **public**, PHP punya **visibility** lain, yaitu **private** & **protected**

```
class Kucing {  
    public $warna = "putih";  
    public function berlari($warna){  
        echo "Mulai berlari..";  
        return $this;  
    }  
    public function makan($warna){  
        echo "Mulai makan..";  
        return $this;  
    }  
}
```

LIKE @arrowfunxtion

public

Dengan visibility **public**, kita mengizinkan **atribut** atau **method** bisa diakses dari mana saja

- **class** itu sendiri ✓
- **class** turunannya (child class) ✓
- dari luar **class** (global) ✓

```
class Kucing {  
    public $warna = "putih";  
  
    public function berlari($warna){  
        echo "Mulai berlari..";  
        return $this;  
    }  
  
    public function makan($warna){  
        echo "Mulai makan..";  
        return $this;  
    }  
}
```

LIKE @arrowfunxtion

protected

Dengan visibility **protected**, kita hanya mengizinkan **atribut** atau **method** bisa diakses dari

- **class** itu sendiri ✓
- **class** turunannya (child class) ✓
- dari luar **class** (global) ✗

```
class Kucing {  
    protected $warna = "putih";  
  
    protected function berlari($warna){  
        echo "Mulai berlari..";  
        return $this;  
    }  
  
    protected function makan($warna){  
        echo "Mulai makan..";  
        return $this;  
    }  
}
```

LIKE @arrowfunxtion

private

Dengan visibility **private**, kita hanya mengizinkan **atribut** atau **method** bisa diakses dari

- **class** itu sendiri ✓
- **class** turunannya (child class) ✗
- dari luar **class** (global) ✗

```
class Kucing {  
    private $warna = "putih";  
    private function berlari($warna){  
        echo "Mulai berlari..";  
        return $this;  
    }  
    private function makan($warna){  
        echo "Mulai makan..";  
        return $this;  
    }  
}
```

LIKE @arrowfunxtion

Parent.php

```
class Parent {  
    public $public_attr = "public";  
    protected $protected_attr = "protected";  
    private $private_attr = "private";  
    // . . . methods . . .  
}
```

Akses dari dalam method di **class Parent** itu sendiri

```
class Parent {  
    // . . . atribut . . .  
    public function methodA(){  
        echo $this->public_attr; ✓  
        echo $this->protected_attr; ✓  
        echo $this->private_attr; ✓  
    }  
}
```

LIKE @arrowfunxtion

Parent.php

```
class Parent {  
    public $public_attr = "public";  
    protected $protected_attr = "protected";  
    private $private_attr = "private";  
    // . . . methods . . .  
}
```

Akses dari dalam method
di **class Child** turunan dari **class Parent**

```
class Child extends Parent {  
    // . . . atribut . . .  
    public function methodB(){  
        echo $this->public_attr; ✓  
        echo $this->protected_attr; ✓  
        echo $this->private_attr; ✗ error  
    }  
}
```

LIKE @arrowfunxtion

Parent.php

```
class Parent {  
    public $public_attr = "public";  
    protected $protected_attr = "protected";  
    private $private_attr = "private";  
    // . . . methods . . .  
}
```

Akses dari luar **Parent class & Child Class** (tapi dari global atau dari **class** yang bukan turunan **Parent Class**)

```
$parent = new Parent;  
$parent->public_attr; ✓  
$parent->protected_attr; ✗ error  
$parent->private_attr; ✗ error
```

LIKE @arrowfunxtion

Object · Oriented · Programming
Episode 8: Abstract

Ep.8

PHP OOP · MINI SERIAL

@arrowfunxtion



Sebuah **class** atau **method**
bisa kita jadikan sebagai
abstract

Jika kita menjadikan sebuah **method** sebagai **abstract method**, itu berarti implementasi dari **method** tersebut belum jelas (**abstract**)

implementasi dari **method** tersebut kita serahkan kepada **class** lain yang **mengextends** dari **abstract class** tersebut.

LIKE @arrowfunxtion



abstract class adalah perjanjian, bahwa semua **class** lain yang **extends** dari **abstract class** harus mengimplementasikan semua **abstract method** yang ada

abstract class:

- boleh mendefinisikan **constructor**
- boleh mendefinisikan **atribut**
- boleh mengubah visibility dari **atribut & method**
- **static** hanya bisa dipakai untuk **method** dengan implementasi (**concrete method**)

LIKE @arrowfunxtion

Saat sebuah **class** memiliki **abstract method**, maka **class** tersebut juga harus ditandai sebagai **abstract (abstract class)**

abstract method: Tidak ada implementasi kode dari method `caraMakan()`

```
abstract class Buah {  
  abstract protected function caraMakan();  
  public $warna = "kuning";  
}
```

abstract class boleh mendefinisikan atribut / property

LIKE @arrowfunxtion

abstract class boleh memiliki **constructor**

```
abstract class Buah {  
    public function __construct(){ ✓  
        // kode di sini  
    };  
}
```

LIKE @arrowfunxtion

```
abstract class Buah {  
    abstract protected function caraMakan();  
}
```



```
class Apel extends Buah {  
    protected function caraMakan(){  
        echo "langsung makan sama kulitnya";  
    }  
}
```

implementasi

abstract method: Tidak ada implementasi dari **method** caraMakan()

hanya **method signature** saja

class Apel harus mengimplementasikan **method** caraMakan() karena **extends** dari **abstract class** Buah;

Jika tidak maka akan **error** ✗

LIKE @arrowfunxtion

Terminologi

```
abstract class Buah {  
    abstract protected function caraMakan();  
}
```

abstract class
class Buah

abstract method
method tanpa implementasi

```
class Apel extends Buah {  
    protected function caraMakan(){  
        echo "langsung makan sama kulitnya";  
    }  
}
```

concrete class
class Apel

concrete method
method dengan implementasi

LIKE @arrowfunxtion

```
abstract class Buah {  
    abstract protected function caraMakan($param);  
}
```

abstract method sebagai acuan

```
class Apel extends Buah {  
    protected function caraMakan(){  
        echo "langsung makan sama kulitnya";  
    }  
}
```

concrete method harus punya jumlah parameter dan **type hint** yang sama dengan **abstract method** yang sedang diimplementasikan

```
class Nanas extends Buah {  
    protected function caraMakan($param){  
        echo "dikupas dulu baru dimakan";  
    }  
}
```

LIKE @arrowfunxtion

```
abstract class Buah {  
    abstract protected function caraMakan();  
}
```

abstract class tidak bisa diinstantiate seperti halnya concrete class

`$buah = new Buah;` ❌

```
class Apel extends Buah {  
    protected function caraMakan(){  
        echo "langsung makan sama kulitnya";  
    }  
}
```

jadi, abstract class harus dipakai oleh class lain sebagai parent class

`$buah = new Apel;` ✅

LIKE @arrowfunxtion

```
abstract class Buah {  
    abstract protected function caraMakan();  
    protected function getWarna($warna){  
        echo "warna = $warna";  
    }  
}
```

← abstract class juga boleh memiliki concrete method

```
class Apel extends Buah {  
    protected function caraMakan(){  
        echo "langsung makan sama kulitnya";  
    }  
}
```

LIKE @arrowfunxtion

Object · Oriented · Programming
Episode 9: Interface

Ep.9

PHP OOP · MINI SERIAL

@arrowfunxtion



interface adalah perjanjian, bahwa semua **class** lain yang **implement** sebuah **interface** harus mengimplementasikan semua **abstract method** yang ada sebagai **public**

interface:

- tidak boleh ada **method** yang memiliki implementasi
- tidak boleh mendefinisikan **property**
- semua method dianggap **public** sehingga tidak perlu mengetik keyword
- tidak boleh memiliki **constructor**
- bisa extends lebih dari 1 **interface** lainnya

LIKE @arrowfunxtion

Interface

menggunakan keyword **interface**
bukan **class**

```
interface MakhlukHidup {
```

```
  function bernafas();
```

```
}
```

hanya boleh mendefinisikan
abstract method.

Semua **method** dianggap
memiliki **visibility public**,
sehingga tidak perlu ditulis
visibility keyword nya

LIKE @arrowfunxtion

```
interface MakhlukHidup {
```

```
  function bernafas(){  
    // implementasi kode ❌  
  }
```

```
}
```

concrete method tidak
diperbolehkan

LIKE @arrowfunxtion

```
interface MakhlukHidup {  
    function bernafas();  
    public $warna = "putih";  
}
```



interface tidak boleh memiliki **atribut**

LIKE @arrowfunxtion

```
interface MakhlukHidup {  
    function __construct(){  
        // kode di sini  
    };  
}
```



interface tidak boleh memiliki **constructor**

LIKE @arrowfunxtion

```
interface MakhlukHidup {  
  function bernafas();  
}
```

```
interface Bersayap {  
  function terbang();  
}
```

```
interface Burung extends MakhlukHidup, PunyaSayap {  
}
```

interface boleh
mengextends lebih dari
satu **interface** lain



LIKE @arrowfunxtion

```
class Hewan {  
  function makan(){  
    echo "makan apa";  
  };  
}
```

```
interface Burung extends Hewan {  
}
```

interface tidak bisa
mengextends dari **class**



LIKE @arrowfunxtion

Implementasi Interface

```
interface MakhlukHidup {  
    function bernafas();  
}  
  
class Elang implements MakhlukHidup {  
}
```

• **class** mengimplementasikan **interface** dengan keyword **implements** bukan **extends**

LIKE @arrowfunxtion

```
interface MakhlukHidup {  
    function bernafas();  
    function tidur();  
}  
  
class Elang implements MakhlukHidup{  
    function bernafas(){  
        // implementasi  
    }  
    function tidur(){  
        // implementasi  
    }  
}
```

Karena **class** Elang mengimplementasikan **interface** MakhlukHidup, Maka **class** Elang harus juga mengimplementasikan **method** bernafas() & tidur() dari **interface** MakhlukHidup

LIKE @arrowfunxtion

	Abstract	Interface
constructor	✓	✗
concrete method	✓	✗
mengubah visibility	✓	✗
attribute	✓	✗
static method	boleh, hanya untuk concrete method	✗
cara pakai	extends	implement

LIKE @arrowfunxtion

Object · Oriented · Programming
Episode 10: Namespace

Ep.10

PHP OOP · MINI SERIAL

@arrowfunxtion



Ketiga **class** di samping
merepresentasikan **Hewan**

Kita ingin mengelompokkan **class**
tersebut menjadi 2 kelompok yaitu
hewan darat dan **hewan air**

LIKE @arrowfunxtion



LIKE @arrowfunxtion

PATH

NAMESPACE PATH

Hewan/Darat/Kucing> Hewan\Darat\Kucing

Hewan/Darat/Sapi> Hewan\Darat\Sapi

.....> backslash

Hewan/Air/Ikan> Hewan\Air\Ikan

LIKE @arrowfunxtion



Best practice nya ketika menulis **namespace** yaitu **namespace path** harus mereferensikan **path** sesungguhnya dalam file sistem seperti pada gambar sebelumnya

LIKE @arrowfunxtion

Hewan\Air\Ikan

namespace

nama class

LIKE @arrowfunxtion

Ikan.php

```
namespace Hewan\Air;
class Ikan {
    function makan(){
        echo "makan apa";
    }
}
```

LIKE @arrowfunxtion

ikan.php

```
namespace Hewan\Air;

class Ikan {

    function makan(){
        echo "makan apa";
    };

}
```

Cara Menggunakan Class

1. lengkap dengan **namespace** path + nama class

```
require_once("../Hewan/Air/Ikan.php");

$ikan = new \Hewan\Air\Ikan;
```

LIKE @arrowfunxtion

ikan.php

```
namespace Hewan\Air;

class Ikan {

    function makan(){
        echo "makan apa";
    };

}
```

Cara Menggunakan Class

2. menggunakan **use namespace**;

```
require_once("../Hewan/Air/Ikan.php");

use \Hewan\Air; ←..... use namespace

$ikan = new Ikan; ←..... langsung
                    nama class
```

LIKE @arrowfunxtion

```
// require_once("../Hewan/Air/Ikan.php");  
  
use \Hewan\Air;  
  
$ikan = new Ikan;
```

Jika menggunakan framework misalnya **Laravel**, kita tidak perlu melakukan **require** file, semua sudah di **Autoload**

Kita tinggal **use, use, use** saja. ^^

LIKE @arrowfunxtion

Sampai jumpa di episode berikutnya!

LIKE & SHARE!

@arrowfunxtion

disajikan oleh **Muhammad Azamuddin**
salah satu penulis buku-laravel-vue.com

