
{
Pemrograman
Berbasis
Objek Modern
Dengan PHP
}
<?php (Muhammad_Surya_Iksanudin); ?>

Table of Contents

Pendahuluan

Judul 1.1

Kata Pengantar 1.2

Tentang Penulis 1.3

Tentang Buku Ini 1.4

Lisensi 1.5

Testimonial 1.6

Persiapan dan Setting

I. Pengenalan PHP 2.1

- Sejarah PHP 2.1.1
- Kelebihan PHP 2.1.2
- Kekurangan PHP 2.1.3
- Market Share PHP 2.1.4
- PHP 7 - The Next Generation 2.1.5
- Perbandingan PHP 7 2.1.6

II. Minimum Requirement Environment 2.2

- Versi PHP Minimum 2.2.1
- Menjalankan PHP Melalui Command Line 2.2.2

Dasar-Dasar OOP

III. Pemrograman Berbasis Objek 3.1

Apa itu Pemrograman Berbasis Objek 3.1.1

Kelebihan Pemrograman Berbasis Objek 3.1.2

Kekurangan Pemrograman Berbasis Objek 3.1.3

Kenapa harus belajar OOP 3.1.4

Fitur Dasar Yang Ada pada OOP 3.1.5

IV. Class dan Object 3.2

Apa itu Class 3.2.1

Contoh Class pada PHP 3.2.2

Anatomi Class 3.2.3

Pembuatan Object (Instansiasi) 3.2.4

V. Property dan Method 3.3

Apa itu Property 3.3.1

Apa itu Method 3.3.2

Urutan Parameter pada Method 3.3.3

VI. Visibilitas 3.4

Apa itu Visibilitas 3.4.1

Tingkatan Visibilitas pada PHP 3.4.2

Tips Visibilitas 3.4.3

VII. Konsep Statis dan Konstanta 3.5

Apa itu Konsep Statis 3.5.1

Contoh Penerapan Konsep Statis 3.5.2

Visibilitas pada Konsep Statis 3.5.3

Apa itu Konstanta 3.5.4
Contoh Penerapan Konsep Konstanta 3.5.5
Visibilitas pada Konsep Konstanta 3.5.6
VIII. Keyword $this dan self 3.6
Apa itu $this 3.6.1
Apa itu self 3.6.2
IX. Return Value 3.7
Apa itu Return Value 3.7.1
Contoh Penggunaan Return Value 3.7.2
X. Constructor dan Destructor 3.8
Apa itu Constructor 3.8.1
Contoh Penggunaan Constructor 3.8.2
Apa itu Desctructor 3.8.3
Contoh Penggunaan Desctructor 3.8.4
Jangan Bilang Siapa-Siapa 3.8.5
XI. Enkapsulasi 3.9
Apa itu Enkapsulasi 3.9.1
Penerapan Enkapsulasi 3.9.2
XII. Pewarisan 3.10
Apa itu Pewarisan 3.10.1
Penerapan Pewarisan 3.10.2

OOP Lanjutan

XIII. Overloading dan Overriding 4.1

Keyword parent 4.1.1
Apa itu Overloading dan Overriding 4.1.2
Penerapan Overloading dan Overriding 4.1.3
XIV. Abstract Class dan Abstract Method 4.2
Apa itu Abstract Class 4.2.1
Apa itu Abstract Method 4.2.2
Kegunaan Abstract Class dan Abstract Method 4.2.3
Penerapan Abstract Class dan Abstract Method 4.2.4
XV. Interface 4.3
Apa itu Interface 4.3.1
Contoh Penggunaan Interface 4.3.2
Membuka Wawasan 4.3.3
XVI. Method Chaining 4.4
Apa itu Method Chaining 4.4.1
Cara Pembuatan Method Chaining 4.4.2
XVII. Pengelompokan Berkas 4.5
Pengelompokan Berkas pada OOP PHP 4.5.1
Keyword namespace dan use 4.5.2
Contoh Penggunaan namespace dan use 4.5.3
Memberikan alias dengan keyword as 4.5.4
XVIII. Parameter Casting dan Return Type Declaration 4.6
Apa itu Parameter Casting 4.6.1
Scalar Type Hinting 4.6.2
Object Type Hinting 4.6.3
Nullable Type Hinting 4.6.4

Apa itu Return Type Declaration 4.6.5
Cara Penggunaan Return Type Declaration 4.6.6
XIX. Recursive Function 4.7
Apa itu Recursive Function 4.7.1
Contoh Penggunaan Recursive Function 4.7.2

OOP Expert

XX. Late Static Bindings 5.1
Apa itu Late Static Bindings 5.1.1
Contoh Late Static Bindings 5.1.2
XXI. Trait 5.2
Apa itu Trait 5.2.1
Cara Penggunaan Trait 5.2.2
Best Practice pada Trait 5.2.3
XXII. Coding Standard 5.3
Apa itu FIG dan PSR 5.3.1
Kenapa harus menerapkan PSR 5.3.2
Aturan Penulisan Syntax 5.3.3
Contoh Penerapan Aturan Syntax 5.3.4
Aturan Penulisan Class dan Namespace 5.3.5
Contoh Penerapan Aturan Class dan Namespace 5.3.6
XXIII. Exception Handling 5.4
Apa itu Exception Handling 5.4.1
Hirarki Error pada PHP 5.4.2

Exception Handling pada PHP 5.4.3
XXIV. Anonymous Function dan Anonymous Class 5.5
Apa itu Anonymous Function 5.5.1
Contoh Penggunaan Anonymous Function 5.5.2
Apa itu Anonymous Class 5.5.3
Contoh Penggunaan Anonymous Class 5.5.4
XXV. Cara Membuat Variadic Function 5.6
XXVI. Instansiasi pada Konteks Statis 5.7
XXVII. Magic Method pada PHP 5.8
Apa itu Magic Method 5.8.1
__construct() dan __desctruct() 5.8.2
__set() dan __get() 5.8.3
__isset() dan __unset() 5.8.4
__sleep() dan __wakeup() 5.8.5
__call() dan __callStatic() 5.8.6
__toString() 5.8.7
XXVIII. Final Class dan Final Method 5.9
Apa itu Final Class 5.9.1
Contoh Penggunaan Final Class 5.9.2
Apa itu Final Method 5.9.3
Contoh Penggunaan Final Method 5.9.4
XXIX. Object sebagai Array 5.10
Apa itu Array Access 5.10.1
Contoh Penggunaan Array Access 5.10.2

Studi Kasus

XXX. Perhitungan Pajak PPH21 6.1

Cara Perhitungan Pajak PPH21 6.1.1

Persiapan Proyek 6.1.2

Pengelompokan Masalah 6.1.3

Penulisan Code 6.1.4

XXXI. Package Management dengan Composer 6.2

Apa itu Composer 6.2.1

Kenapa Menggunakan Composer 6.2.2

Instalasi Composer 6.2.3

Tentang composer.json 6.2.4

XXXII. Membuat Package Sendiri 6.3

Membuat composer.json 6.3.1

Autoload dengan Composer 6.3.2

Pendaftaran Package 6.3.3

Sinkronisasi Github dan Packagist 6.3.4

XXXIII. Design Pattern 6.4

Apa itu Design Pattern 6.4.1

Manfaat Penggunaan Design Pattern 6.4.2

Macam-Macam Design Pattern 6.4.3

XXXIV. Studi Kasus Membuat Framework Sederhana 6.5

Skop Proyek 6.5.1

Konsep Front Controller 6.5.2

HTTP Request dan HTTP Response 6.5.3

Mengarahkan Request dengan Router 6.5.4
Membuat Controller Class 6.5.5
Membuat Kernel Framework 6.5.6
Kesimpulan 6.5.7
XXXV. Studi Kasus Todo List Menggunakan OOP dan MVC
Tujuan dari Proyek Todo List 6.6.1 6.6
Pembuatan Database 6.6.2
Koneksi Database 6.6.3
Membuat Model Class 6.6.4
Membuat Todo Class 6.6.5
Membuat Controller Class 6.6.6
Menambahkan Template Engine pada Framework 6.6.7
Membuat Todo Controller Class 6.6.8
Kesimpulan 6.6.9

Penutup

Penutup 7.1

Kode: 16C4D834B

Pemrograman Berbasis Objek Modern

Menggunakan PHP

Menjadi Master Pemrograman Berbasis Objek (OOP) dengan PHP 7 - Next Generation

Muhamad Surya Iksanudin

Kata Pengantar

Segala puji bagi Allah SWT, yang senantiasa melimpahkan rahmat serta karunia-Nya kepada penulis sehingga dapat menyelesaikan buku dengan judul "Pemrograman Berbasis Objek Modern Menggunakan PHP" ini. Shalawat serta salam penulis haturkan kepada Rasulullah Sholallahu Alaihi Wassalam yang menjadi teladan terbaik bagi umat manusia. Rasul yang membawa kita dari jalan gelap menuju cahaya.

Terima kasih penulis sampaikan kepada seluruh pihak yang telah membantu terwujudnya buku ini terutama saudara-saudara seperjuangan di Komunitas Symfony Framework Indonesia yang tidak pernah lelah membantu penulis dalam penyusunan buku ini. Terima kasih juga penulis ucapkan khususnya kepada Bapak Hendro Wicaksono yang telah bersedia mengoreksi isi dari buku ini. Terima kasih penulis haturkan kepada Muhibbudin Suretno yang telah membuatkan sampul untuk buku ini.

Penulis menyadari dalam penulisan buku ini terdpat banyak kekurangan. Penulis mengharapkan kritik dan saran dari pembaca. Harapan penulis makalah ini bisa bermanfaat bagi pembaca dan bagi penelitian berikutnya.

Jakarta, Juli 2018

Muhamad Surya Iksanudin

Tentang Penulis



Muhamad Surya Iksanudin

Penulis lahir di Pemalang, 3 April 1988. Penulis tercatat sebagai salah satu member di Komunitas Symfony Framework Indonesia sebuah forum diskusi berbagai permasalahan yang ada di dunia pemrograman khususnya PHP dan lebih khusus lagi tentang Framework Symfony.

Penulis juga bekerja sebagai pekerja lepas sebagai *System Administrator* maupun *Web Developer*. Disela-sela waktu luangnya, penulis selalu meng-*update* informasi tentang teknologi secara umum maupun web teknologi secara khusus.

Diluar itu semua, penulis juga hobi *ngoding* dan telah membuat beberapa proyek yang dapat dilihat pada halaman Github penulis.

Penulis dapat dihubungi melalui kontak berikut:

Whatsapp: +62 878 000 939 15

Email: surya.iksanudin@gmail.com

Facebook: Muhamad Surya Iksanudin

LinkedIn: Muhamad Surya Iksanudin

Tentang Buku Ini

Buku ini saya dedikasikan untuk menjawab pertanyaan, masukkan dan kritik yang saya terima dari buku "Belajar Santai OOP PHP". Berbeda dengan buku "Belajar Santai OOP PHP" yang lebih *to the point*, buku ini mencoba menjelaskan lebih *detail* setiap pembahasan yang ada. Selain itu, buku ini juga membahas lebih banyak dengan contoh yang lebih bervariatif sehingga diharapkan pembaca dapat lebih mudah memahami konsep Pemrograman Berbasis Objek daripada buku sebelumnya.

Buku ini menyajikan pembahasan yang lebih mendalam dengan jumlah bab yang lebih banyak dan contoh yang lebih mendeketi *real case* penggunaan Pemrograman Berbasis Objek untuk menyelesaikan sebuah masalah tertentu pada pekerjaan.

Buku ini juga mengajarkan *best practice* serta *coding standard* yang berlaku global dalam komunitas dan ekosistem PHP seperti PSR (PHP *Standard Recomendations*) dan *Composer* sehingga setelah membaca buku ini, pembaca diharapkan dapat mengimplementasikannya dalam pekerjaan secara langsung.

Semoga buku ini dapat menjadi salah bacaan wajib *programmer* PHP yang ingin menguasai dan memahami konsep Pemrograman Berbasis Objek menggunakan bahasa pemrograman PHP.

Pemilik Buku

ID Buku : 16C4D834B

Nama : Google Play Books

Email : surya.kejawen@gmail.com

Lisensi

- Gitbook

Buku ini menggunakan Gitbook sebuah perangkat lunak *opensource* yang dikhususkan untuk menulis dokumen menggunakan format markdown. Gitbook memiliki lisensi Apache License 2.0.

- Visual Studio Code

Visual Studio Code digunakan sebagai *text editor* untuk menulis buku ini. Selain itu, Visual Studio Code juga penulis gunakan untuk menuliskan beberapa *listing code* yang ada dalam buku ini. Visual Studio Code adalah *text editor opensource* yang menggunakan lisensi MIT sehingga aman untuk digunakan baik untuk keperluan *commercial* maupun *non commercial*.

- Buku "Pemrograman Berbasis Objek Modern Menggunakan PHP"

Buku ini adalah hasil karya intelektual yang dilindungi oleh perundangan yang berlaku di Negara Kesatuan Republik Indonesia. Meski buku ini didistribusikan dalam format *e-book* namun lisensi buku ini bersifat *private*.

Anda tidak diperkenankan untuk meng-*copy* atau mendistribusikan ulang buku ini kepada orang lain. Bila Anda melakukan hal tersebut, maka Anda berarti telah melakukan pembajakan dan

Anda dapat dituntut secara hukum sesuai undang-undang yang berlaku di Negara Kesatuan Republik Indonesia.

Disetiap buku ini terdapat **kode unik** disudut kanan atas halaman judul yang menjadi identitas setiap *copy* dari buku ini. Kode unik tersebut menyimpan identitas pemilik buku yang sah sehingga bila kode tersebut digunakan oleh orang yang tidak berhak maka kami dapat **membatalkan segala *benefit*** yang melekat pada buku tersebut ketika pemesanan.

Dengan membeli buku ini, maka Anda berarti telah **mengetahui** dan **menyetujui** perihal lisensi yang ada pada buku ini.

Testimonial

- **Peter J. Kambey (Head of Executive PHP Indonesia Community)**

> Saya sangat gembira dengan kehadiran buku ini karena saya menyadari pengetahuan mengenai OOP (utamanya di PHP) terbukti menjadi hal yang amat penting dalam pengembangan aplikasi bisnis yang saya bangun.
>
> Untuk menularkan pengetahuan ini maka topik mengenai OOP menjadi salah satu *point* penting yang sering saya bagikan di setiap seminar dan *meetup* tehnikal dari komunitas PHP Indonesia dengan harapan agar mereka berusaha semaksimal mungkin untuk belajar dan tumbuh menjadi *developer-developer* handal di negeri ini.

- **Asep Badja Pradipta (CEO Tanibox)**

> Tidak banyak buku pemrograman PHP berbahasa Indonesia yang memiliki bahasan sangat mendalam. Buku *Pemrograman Berbasis Objek Moderen Dengan PHP* karya Surya ini berbeda, cocok dijadikan acuan bagi pemrogram pemula karena menggunakan bahasa yang sederhana, dan bisa juga menjadi bahan penyegaran bagi para pemrogram tingkat lanjut.

- **Hendro Wicaksono (Founder SLiMS Library Management System)**

> Kalau mencari buku membahas PHP yang membahas materi dari level pemula sampai mahir, lengkap, serius, tidak kacangan, dan ditulis oleh ahlinya di dunia nyata, maka membeli (dan membaca) buku ini adalah keputusan yang tepat. Dengan pengalaman tahunan membangun aplikasi skala enterprise berbasis OOP, M. Surya Ikhsanudin bisa menjelaskan dengan sangat detail disertai contoh yang mudah dicerna. *Very highly recommended book*!

- **Yayan Sopiyan (Senior Developer Zend Framework Indonesia)**

> Buku yang menarik untuk Anda gunakan sebagai panduan dalam membangun aplikasi berbasis PHP. Menggunakan teknik PHP modern, membahas berbagai contoh yang dimaksudkan dengan jelas dan terinci, agar diharapkan materi yang disampaikan dapat dipahami dengan mudah. Sebagai pengguna aktif dan aktivis PHP Zend Framework, saya merekomendasikan buku karangan Surya ini. *Good Job Bro!*

I. Pengenalan PHP

Pada bab ini kita akan membahas tentang sejarah PHP dari awal pertama kali dirilis hingga sekarang. Selain itu kita juga akan membahas tentang kelebihan dan kekurangan PHP sebagai bahasa pemrograman serta *market share* PHP di dunia *web development*.

Sejarah Bahasa Pemrograman PHP

Menurut wikipedia, Pada awalnya PHP merupakan kependekan dari Personal Home Page (Situs personal). PHP pertama kali dibuat oleh Rasmus Lerdorf pada tahun 1995. Pada waktu itu PHP masih bernama Form Interpreted (FI), yang wujudnya berupa sekumpulan skrip yang digunakan untuk mengolah data formulir dari web.

Selanjutnya Rasmus merilis kode sumber tersebut untuk umum dan menamakannya PHP/FI. Dengan perilisan kode sumber ini menjadi sumber terbuka, maka banyak pemrogram yang tertarik untuk ikut mengembangkan PHP.

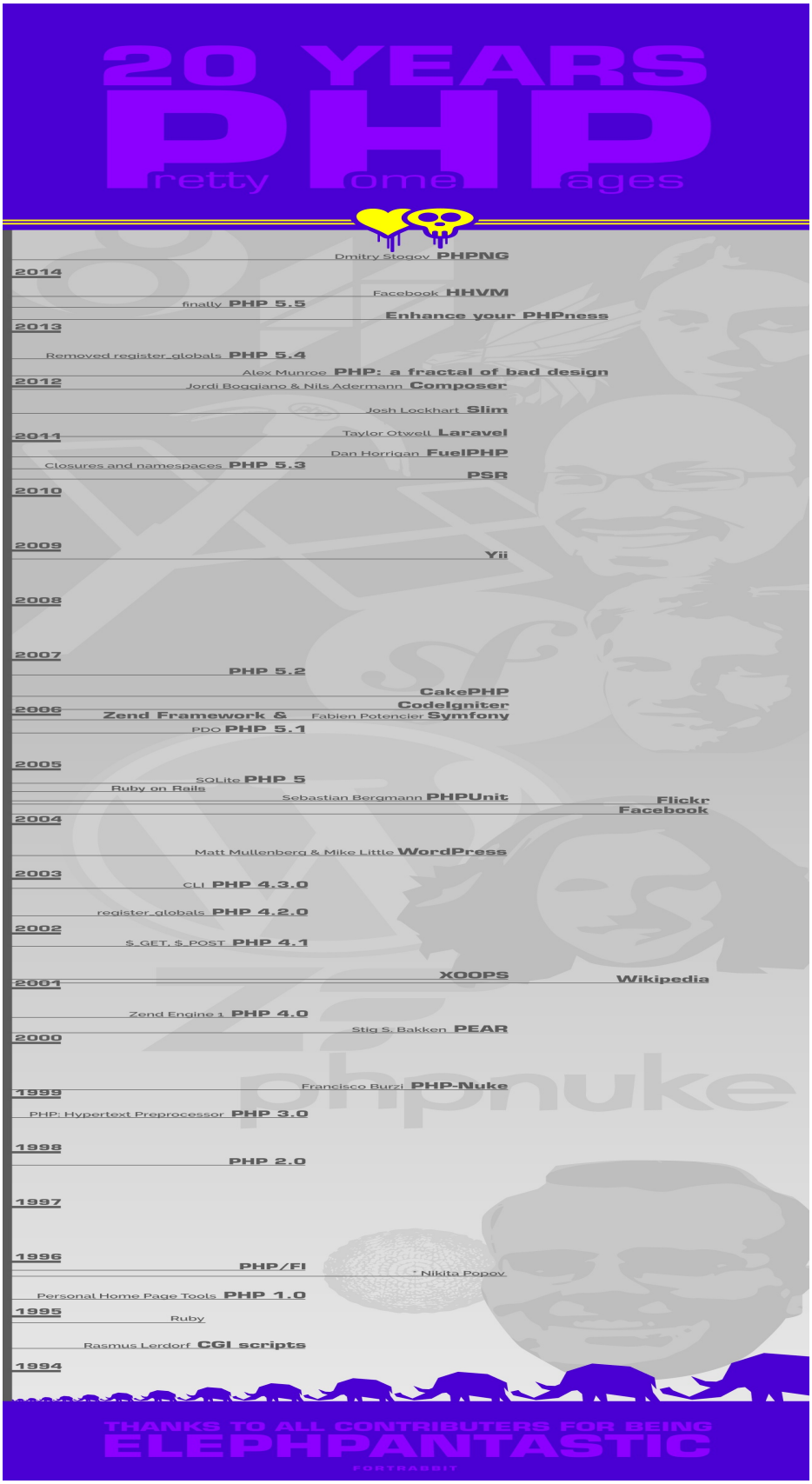
Pada November 1997, dirilis PHP/FI 2.0. Pada rilis ini, interpreter PHP sudah diimplementasikan dalam program C. Dalam rilis ini disertakan juga modul-modul ekstensi yang meningkatkan kemampuan PHP/FI secara signifikan.

Pada tahun 1997, sebuah perusahaan bernama Zend menulis ulang interpreter PHP menjadi lebih bersih, lebih baik, dan lebih cepat. Kemudian pada Juni 1998, perusahaan tersebut merilis interpreter baru untuk PHP dan meresmikan rilis tersebut sebagai PHP 3.0 dan singkatan PHP diubah menjadi akronim berulang PHP: Hypertext Preprocessing.

Pada pertengahan tahun 1999, Zend merilis interpreter PHP baru dan rilis tersebut dikenal dengan PHP 4.0. PHP 4.0 adalah versi PHP yang paling banyak dipakai pada awal abad ke-21. Versi ini banyak dipakai disebabkan kemampuannya untuk membangun aplikasi web kompleks tetapi tetap memiliki kecepatan dan stabilitas yang tinggi.

Pada Juni 2004, Zend merilis PHP 5.0. Dalam versi ini, inti dari interpreter PHP mengalami perubahan besar. Versi ini juga memasukkan model pemrograman berorientasi objek ke dalam PHP untuk menjawab perkembangan bahasa pemrograman ke arah paradigma berorientasi objek. Server web bawaan ditambahkan pada versi 5.4 untuk mempermudah pengembang menjalankan kode PHP tanpa meng-*install software server*.

Pada saat buku ini ditulis, PHP telah mencapai versi 7.2 dengan penambahan ekstensi dan perbaikan performa yang menjanjikan. Berikut adalah info grafis tentang sejarah dan perkembangan PHP serta ekosistemnya dari awal hingga tahun 2015 yaitu ketika PHP 7 atau yang juga dikenal dengan PHP Next Generation (PHPNG) dirilis publik.



Sumber: https://blog.fortrabbit.com

Kelebihan PHP

Sebagai bahasa pemrograman, PHP memiliki banyak kelebihan antara lain:

- Komunitas yang besar

Tidak dapat dipungkiri bahwa komunitas PHP sangat besar dan tersebar diseluruh dunia. Di Indonesia saja ada banyak komunitas yang berafiliasi dengan PHP baik itu pembahasan PHP secara umum maupun pembahasan secara khusus misalnya tentang *framework*. Di *facebook* ada *group* PHP Indonesia yang membahas PHP secara umum, dan ada pula Symfony Framework Indonesia yang membahas secara khusus tentang *framework* Symfony. Tidak hanya itu, di Telegram, WhatsApp pun banyak bertebaran *group* yang membahas tentang PHP.

- *Resources* yang melimpah

Dikarenakan komunitasnya yang besar, tentu saja akan berdampak pada kemudahan mencari *resources* yang berhubungan dengan PHP baik itu permasalahan yang sering terjadi, *library*, *software*, *CMS* hingga *framework* PHP banyak sekali bertebaran dan dengan mudah dapat ditemukan dengan *googling*.

- Mudah dipelajari

PHP adalah bahasa pemrograman sejuta umat. Hampir semua orang yang pernah bergelut dengan dunia *Web Development* pasti pernah menggunakannya atau setidaknya pernah sekedar

mencobanya. Tutorial untuk memulai belajar PHP pun dengan mudah ditemukan dengan mengetikkan kata kunci pada mesin pencari.

- Simpel

PHP itu simpel. *Syntax*-nya sangat sederhana dan mudah sekali dipelajari. *Saking* simpelnya, untuk memulai belajar PHP kita tidak perlu melakukan *setting* apapun, cukup *install* paket *software* seperti XAMPP atau WAMP maka Anda sudah dapat memulai belajar PHP.

- Mudah dan murah untuk *deployment*

Untuk men-*deploy* program PHP sangatlah mudah, kita cukup meng-*upload* ke *server hosting* yang harga juga sangat terjangkau bahkan ada yang gratis.

Dan masih banyak lagi kelebihan lainnya.

Kekurangan PHP

Banyakk orang yang bilang kekurangan utama PHP adalah bahwa PHP bahasa yang *weak type* dimana sebuah *variable* tidak memiliki tipe data sehingga menyulitkan ketika melakukan *debugging*. *Weak type* ini menyebabkan terjadinya *juggling* dimana sebuah *variable* yang tadinya berisi nilai `integer` misalnya dapat berubah menjadi berisi nilai `string` atau bahkan tipe data lainnya.

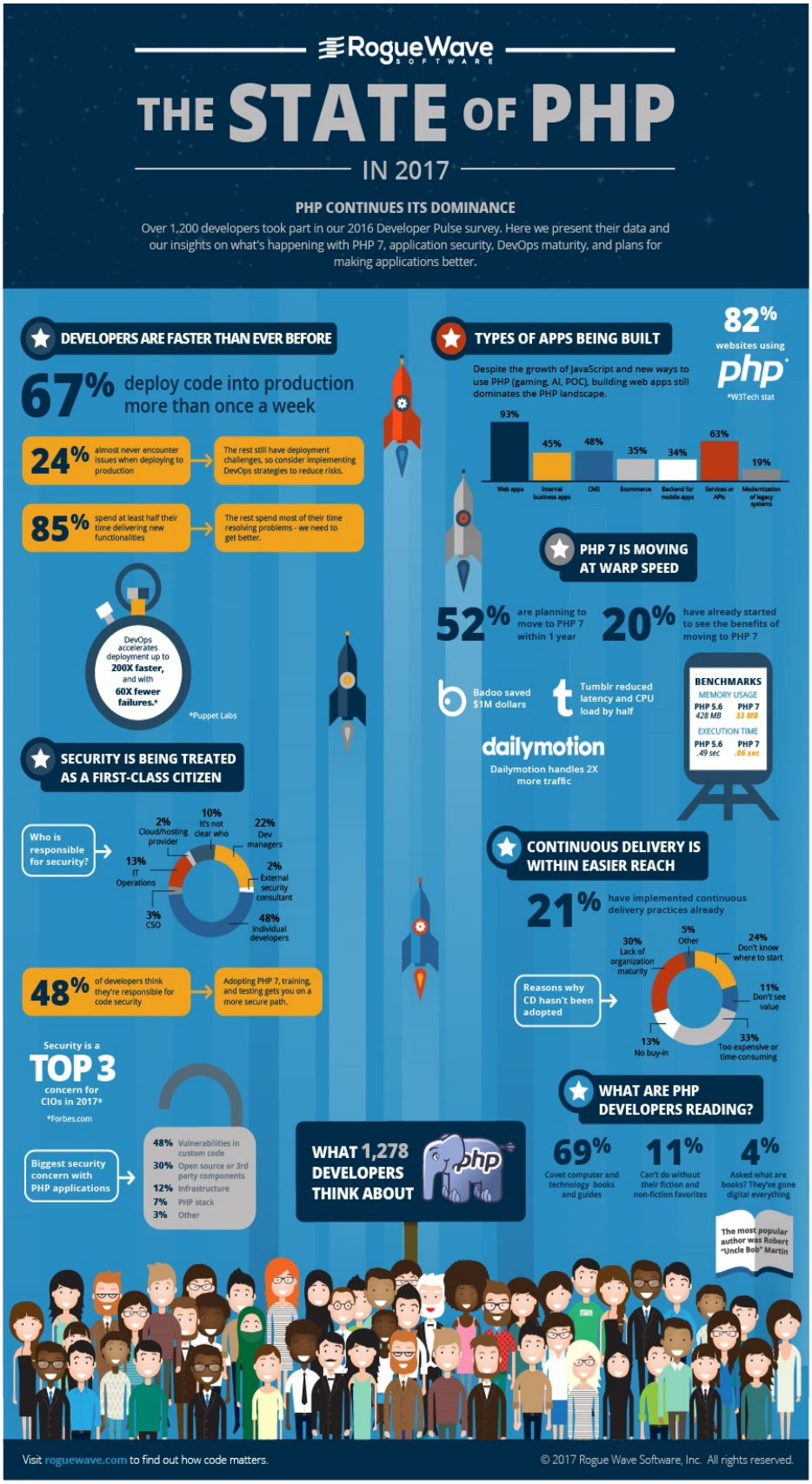
Selain *weak type*, PHP juga mempunya kekurangan lain yaitu inkonsistensi *API (Application Programming Interface)*. API disini bukan Web API yang mengembalikan `json` tapi API disini adalah fungsi bawaan dari PHP yang menjadi *interface* atau tatap muka antara kita sebagai *developer* dan bahasa pemrograman PHP itu sendiri. Contoh paling mudah dari ketidakkonsistenan PHP adalah dalam hal penamaan fungsi misalnya antara fungsi `substr` dan `str_replace`.

Kenapa `substr` tidak dinamai `str_sub`? Atau `str_replace` diganti jadi `replacestr`? Inilah yang membuat saya sebagai *developer* PHP malas untuk menghafalkan nama fungsi maupun urutan parameternya. Pada akhirnya saya lebih senang melihat dokumentasi resmi untuk melihat bagaimana sebuah fungsi bekerja.

Market Share PHP

Banyak yang menyangsikan PHP akan bertahan dan mampu bersaing di era Web 3.0 seperti sekarang ini. Namun komitmen PHP untuk terus menghadirkan inovasi dan menjawab kebutuhan user (Developer) tetap membawa PHP menjadi pemimpin sebagai bahasa yang paling digunakan untuk pembuatan website maupun aplikasi berbasis internet. Tidak tanggung-tanggung, PHP meninggalkan bahasa pemrograman lain dengan *market share* lebih dari 80% diikuti oleh ASP.Net dan Java.

Pencapaian ini membuktikan bahwa PHP masih tetap diminati, terlebih dengan hadirnya PHP generasi terbaru (PHP 7) yang menghadirkan banyak fitur baru dari segi kecepatan eksekusi, kestabilan serta keamanan. Berikut adalah info grafis tentang *market share* PHP.



Sumber: http://zend.com

PHP 7 - *The Next Generation*

Melihat kebelakang ketika PHP 5 pertama kali dirilis, versi ini adalah versi PHP pertama yang melahirkan banyak perubahan besar pada ekosistem PHP. Banyak sekali framework dan maupun tool pengembangan yang dirilis beberapa bulan setelah versi ini dirilis secara publik. *Framework* seperti Zend, Symfony, Prado, Cake, CodeIgniter, YII dan Laravel adalah sebagian dari banyak framework yang mengiringi perjalanan PHP 5 selama lebih dari satu dekade. *Tool* dan *library* lain seperti PHPUnit, PHP Code Sniffer, Doctrine, Propel pun ikut meramaikan ekosistem PHP.

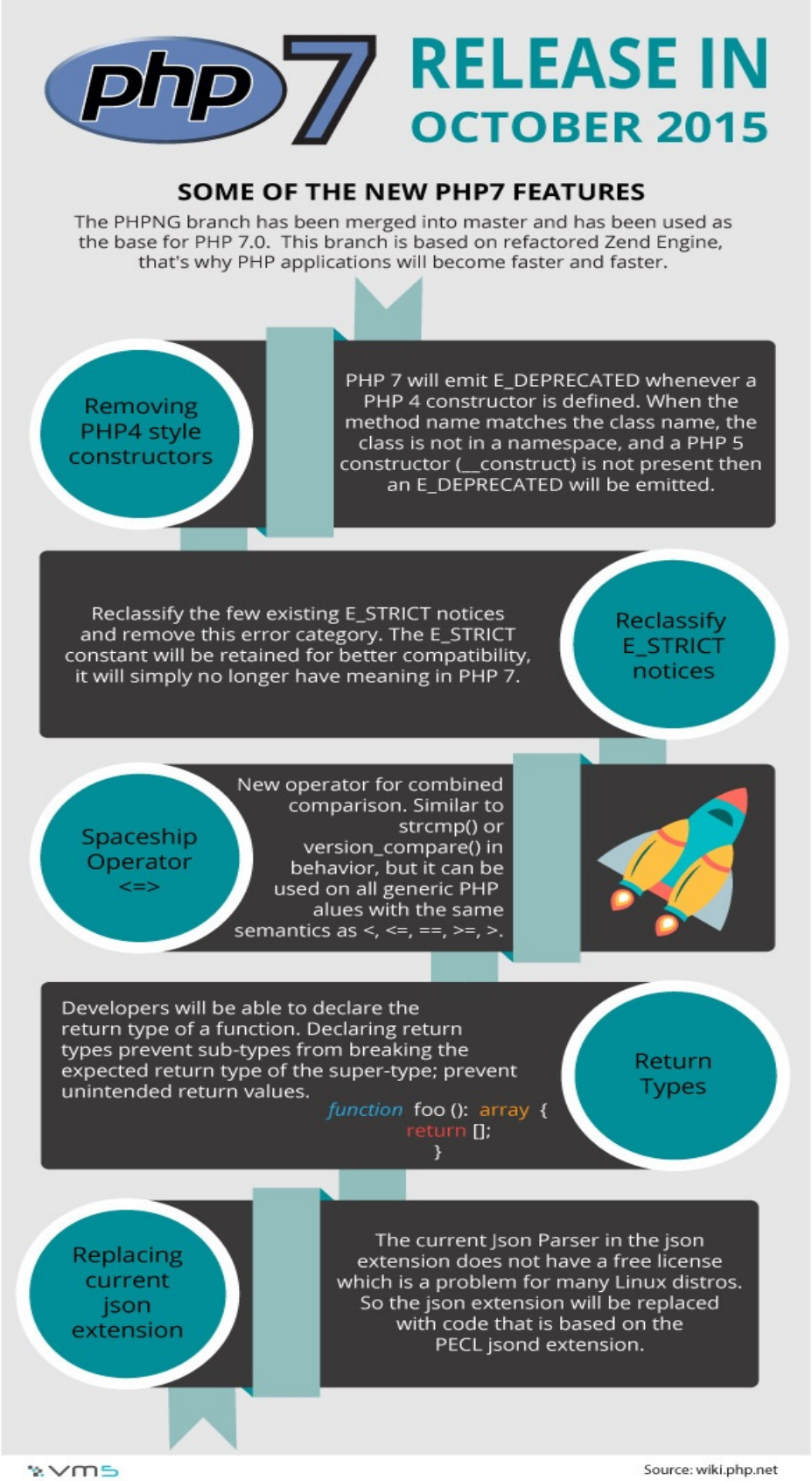
Ketika awal kemunculannya, PHP 5 yang mengusung konsep OOP, belum mampu menjawab beberapa kebutuhan seperti belum adanya `namespace`, `static type hinting`, `return type`, `closure` dan masih banyak lagi. Oleh karena ini, PHP terus berevolusi dan menambahkan fitur-fitur tersebut sedikit demi sedikit disamping juga tetap memberikan perbaikan keamanan dan stabilitas. Hingga PHP 5.6 yaitu versi terakhir dari PHP 5, PHP telah memiliki beberapa fitur seperti `namespace`, `array type hinting` dan `closure`.

Namun isu besar muncul pada PHP 5, yaitu masalah pada penggunaan memory yang boros serta waktu eksekusi yang lama. Masalah tersebut sebenarnya telah dijawab oleh Facebook dengan membuat *engine* yang diberinama HHVM (Hip Hop Virtual Machine) yaitu mesin yang mampu mengubah *code* PHP menjadi *bytecode* seperti pada Java, dan kemudian mengeksekusinya.

HHVM sendiri selain men-*support* PHP, juga men-*support* Hacklang, sebuah bahasa pemrograman yang *syntax*-nya sangat mirip dengan PHP yang dikembangkan oleh Facebook.

Kemunculan HHVM sendiri sebenarnya memberikan keresahan bagi Zend selaku pengembang PHP. Hal ini karena, selain HHVM, Facebook juga mempromosikan bahasa pemrograman baru yaitu Hacklang yang memang diciptakan untuk "menyaingi" PHP dan berjalan dengan optimal pada HHVM. Oleh karena itu, Zend kemudian me-*refactor Zend Engine* agar bisa lebih baik dari HHVM.

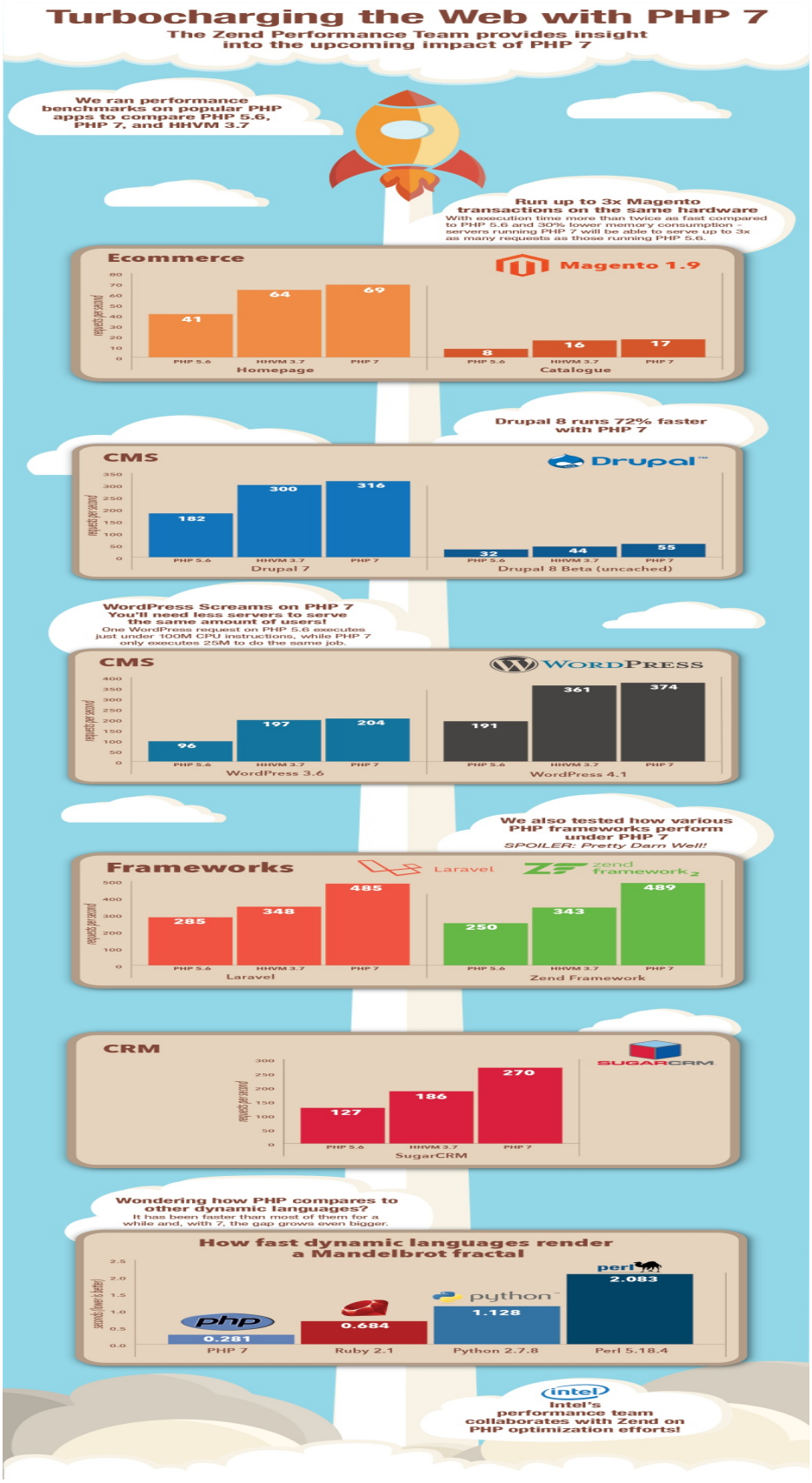
Setelah proses pengembangan yang cukup lama, pada Februari 2015 PHP 7 lahir sebagai *The Next Generation of PHP*. Banyak sekali perombakan yang dilakukan oleh Zend terutama pada performa dan penggunaan *memory* yang lebih kecil. Selain itu, beberapa fitur ditambahkan untuk melengkapi fitur OOP yang ada di PHP seperti `return type`, `static type hinting` selain juga perbaikan beberapa fitur. Dan berikut adalah info grafis tentang fitur yang ada pada PHP 7.



Sumber: http://vm5.eu

Perbandingan PHP 7 dengan HHVM dan PHP 5

Seperti yang sudah disebutkan diatas, PHP 7 memiliki banyak keunggulan terutama yang sangat menonjol adalah performa dan penggunaan *memory* yang lebih kecil. Perbaikan performa ini menjadikan PHP kembali memimpin dan mengungguli HHVM dari sisi waktu eksekusi dalam beberapa *benchmark*. Dan berikut adalah info grafis perbandingan antara PHP 7, HHVM maupun PHP 5 menggunakan beberapa *software* baik itu *framework*, CMS, maupun PHP 7 dibandingkan dengan bahasa pemrograman lain.



Sumber: http://www.zend.com

Dari info grafis diatas, terbukti bahwa PHP 7 mampu menjawab tantangan dan kebutuhan pasar. Dan ini menunjukkan komitmen dari Zend selaku pengembang dan perusahaan PHP untuk terus memberikan yang terbaik untuk pada developer PHP di seluruh dunia.

II. *Minimum Requirement Environment*

Bab ini berguna untuk menyamakan lingkungan pengembangan sehingga kedepannya tidak terjadi hal-hal yang tidak diharapkan.

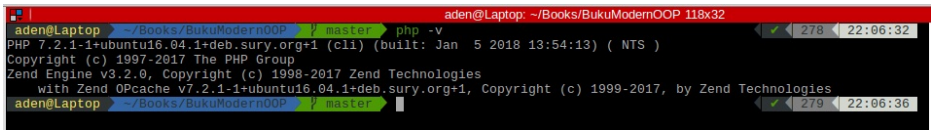
Versi PHP Minimum

Untuk Versi PHP yang akan kita gunakan sebagai acuan adalah PHP versi 7.2.1 dan karena kebetulan saya menggunakan Linux sebagai lingkungan pengembangan, saya tidak menggunakan paket *software* seperti XAMPP, WAMP atau sejenisnya. Bila Anda terbiasa dengan paket *software* maka saya menyarankan untuk menggunakan XAMPP dengan versi PHP yang sesuai.

Untuk men-*download* XAMPP, Anda dapat langsung mengunjungi halaman resminya *apache frineds* kemudian pilih sistem operasi sesuai yang Anda gunakan dan pastikan Anda men-*download* versi PHP yang sesuai. Langkah selanjutnya tinggal Anda *install* pada sistem operasi yang Anda gunakan.

Menjalankan PHP Melalui *Command Line*

Setelah melakukan instalasi PHP, tahap berikutnya adalah kita dapat menjalankan perintah PHP melalui terminal/*command line*. Untuk mengetes apakah kita sudah dapat menjalankan PHP melalui *command line*, kita dapat melakukannya dengan membuka terminal (untuk Linux dan MacOS) atau *command prompt* (untuk Windows) lalu mengetikkan perintah `php -v`. Bila kita telah dapat menjalankan perintah PHP maka akan muncul tampilan sebagai berikut:



Bila belum, maka Anda dapat mendaftarkan PHP kedalam *environment variable*. Untuk Windows dapat mengikuti tutorial pada *link* ini. Untuk Linux dan MacOS dapat mengikuti tutorial pada *link* ini. Anda hanya cukup mengganti lokasi (*path*) PHP sesuai dengan direktori atau *folder* tempat Anda meng-*install* XAMPP. Bila cara tersebut tidak dapat bekerja, Anda dapat mencoba me-*restart* PC Anda kemudian coba jalankan PHP melalui *command line* kembali.

III. Pengenalan Pemrograman Berbasis Objek

Pada bab ini kita akan membahas tentang apa itu pemrograman berbasis objek, kelebihan dan kekurangannya serta fitur apa saja yang harus ada dalam bahasa pemrograman yang mendukung pemrograman berbasis objek.

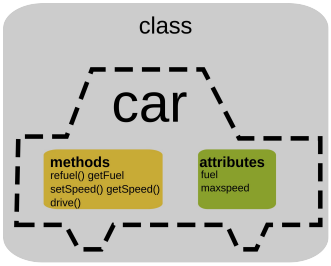
Apa itu Pemrograman Berbasis Objek

Menurut wikipedia, Pemrograman berorientasi objek merupakan paradigma pemrograman yang berorientasikan kepada objek. Semua data dan fungsi di dalam paradigma ini dibungkus dalam kelas-kelas atau objek-objek.

Dalam pemrograman berbasis objek, kita dituntut untuk memahami dan memetakan masalah kedalam *class* serta memecah masalah kedalam *class-class* yang lebih kecil dan simpel agar solusi yang dibuat lebih spesifik. Selanjutnya, *class-class* tersebut akan saling berkomunikasi dan berkolaborasi untuk memecahkan masalah yang kompleks. *Class-class* ini nantinya akan dirubah menjadi objek-objek pada saat *runtime*.

Setiap *class* dalam OOP mempunyai *method* atau fungsi serta *property* atau atribut. *Method* dalam *class* secara mudah diartikan sebagai segala kemampuan dari *class* atau apa saja yang dapat dilakukan oleh sebuah *class*. Sedangkan *property* adalah segala sesuatu yang dimiliki oleh *class*. Dalam OOP, *property* dan *method* dalam *class* saling bekerjasama membangun sebuah solusi dari suatu masalah.

Dalam beberapa referensi, *method* disebut juga sebagai *function* sedangkan *property* sering disebut juga sebagai *attribute*. Sehingga Anda tidak perlu bingung bila nantinya dibuku lain, Anda menemui istilah *function* dan *attribute* sebagai pengganti *method* dan *property*. Yang perlu Anda pahami bahwa *function* atau *method* adalah fitur atau kemampuan dari sebuah *class* sedangkan *property* atau *attribute* adalah segala sesuatu yang dimiliki oleh sebuah *class*.



Sumber: https://github.com

Kelebihan Pemrograman Berbasis Objek

Pemrograman berbasis objek atau biasa disebut OOP, memiliki banyak keunggulan dibandingkan paradigma pemrograman lainnya. Keunggulan pemrograman berbasis objek antara lain

sebagai berikut:

- Modularitas: pogram yang dibuat dapat dipecah menjadi modul-modul yang lebih kecil dan nantinya digabungkan menjadi solusi yang utuh.
- Fleksibilitas: karena setiap solusi dibuat dalam bentuk *class*, ketika terjadi perubahan maka hanya *class* tersebut saja yang perlu dirubah.
- Ekstensibilitas: penambahan *method* atau *property* dapat dilakukan dengan sangat mudah.
- *Reuse*: *class* dapat digunakan berkali-kali untuk proyek maupun modul yang lain.
- Mudah dimaintain: karena setiap *class* berdiri sendiri, maka untuk memaintain *class* tersebut jauh lebih mudah.
- Keamanan *code*: adanya visibilitas memberikan fitur keamanan dimana developer lain tidak bisa dengan bebas menggunakan fitur yang ada pada sebuah objek.
- Waktu development lebih cepat: karena *reusable* otomatis dapat mempersingkat waktu pengembangan program.

Dan masih banyak lagi keunggulan OOP yang lainnya.

Kekurangan Pemrograman Berbasis Objek

Selain kelebihan, pemrograman berbasis objek juga mempunyai kekurangan antara lain sebagai berikut:

- *Learning curve* yang lumayan tinggi.
- Ukuran program jauh lebih besar.
- Pemakaian *memory* lebih besar.

Kenapa Harus Belajar OOP

Setelah mengetahui kelebihan dan kekurangan OOP, sekarang kita sampai kepada kesimpulan kenapa kita harus belajar OOP. Tapi sebelum kita membahas kenapa kita harus belajar OOP, ada baiknya kita coba melihat realita yang ada. Di ekosistem PHP sekarang ini banyak sekali *framework* mulai dari yang kecil seperti Slim, Lumen, dan Zend Expressive hingga *framework* berskala besar seperti Zend Framework, dan Symfony. Bila kita cermati, hampir semua *framework* dibangun dengan menggunakan paradigma OOP sehingga dengan menguasai OOP, kita dapat memaksimalkan fitur yang ada dalam *framework* tersebut.

Dan berikut adalah beberapa alasan kenapa kita harus belajar OOP:

- OOP sangat cocok untuk pembuatan aplikasi berskala besar.
- OOP merupakan gerbang untuk menguasai *framework* dengan maksimal.
- OOP dapat digunakan diberbagai bahasa pemrograman yang mendukung OOP tidak hanya pada PHP.

- Dengan OOP kita dapat menerapkan *design pattern* dengan lebih mudah.

Dan masih banyak lagi alasan untuk belajar OOP.

Fitur Dasar Yang Ada pada OOP

Berikut adalah beberapa fitur dasar yang harus ada dalam bahasa pemrograman yang mendukung konsep OOP.

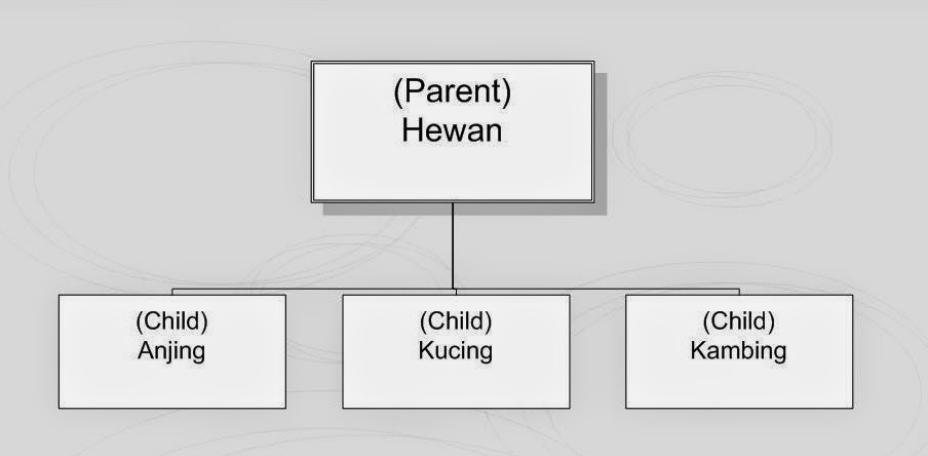
- Enkapsulasi

Enkapsulasi adalah sebuah fitur yang menggabungkan antara fungsionalitas dan data untuk menyembunyikan informasi. Enkapsulasi memungkinkan kita menggunakan fungsi dari sebuah objek tanpa perlu mengetahui detail dari apa yang terjadi dalam fungsi tersebut. Ekapsulasi mengatur bagaimana kita menggunakan objek, fungsi atau atribut apa saja yang dapat digunakan dan yang tidak dapat digunakan.

- Pewarisan

Pewarisan adalah sebuah fitur yang memungkinkan kita menggunakan fitur dari suatu *class* tanpa perlu mendefinisikan ulang semua *method* dan *property* yang ada pada *class* tersebut.

Pewarisan ini sangat bermanfaat jika kita ingin mempunyai sebuah *class* yang secara fitur mirip dengan *class* lain namun ada sebuah spesifikasi khusus yang spesifik dari *class* tersebut. Dalam OOP, *class* yang mewariskan sifat atau fitur disebut sebagai *parent class* sedangkan *class* yang diwarisi sifat atau fitur disebut *child class*. *Child class* memiliki semua fitur yang ada pada *parent class* ditambah dengan fitur spesifik miliknya sendiri.



Sumber: http://www.adityawiraha.com

- Polimorfisme

Polimorfisme adalah sebuah fitur yang erat kaitannya dengan fitur sebelumnya yaitu pewarisan. Fitur ini secara mudah adalah sebuah kemampuan dari objek untuk merespon atau mengolah secara berbeda terhadap input yang sama.

Saya agak kesulitan dalam menjelaskan bagaimana polimorfisme bekerja, namun nanti kita akan memahami hal tersebut ketika membahas tentang pewarisan, *abstract class*, *abtract method* dan *interface*. Untuk sekarang, Anda cukup pahami bahwa

polimorfisme adalah salah satu fitur yang harus ada dalam OOP dimana sebuah *method* yang sama pada *class* yang berbeda akan memberikan *output* yang berbeda pula jika digunakan.

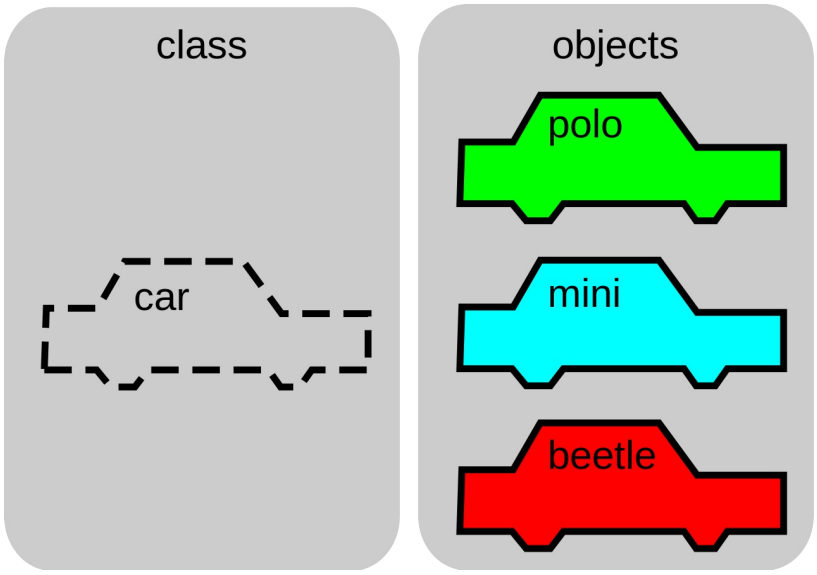
Itulah 3 konsep dasar yang harus ada pada bahasa pemrograman yang mendukung konsep OOP. Meski secara konsep sama, biasanya setiap bahasa pemrograman mempunyai cara implementasi yang berbeda dari fitur-fitur diatas.

IV. *Class* dan *Object*

Bab ini akan menjabarkan tentang apa itu *class*, bagaimana cara membuat *class*, apa itu *object*, proses pembuatan *object* serta memahami perbedaan antara *class* dan _object.

Apa itu Class

Secara mudah, *class* adalah cetakan atau *blueprint* dari objek. Didalam *class* terdapat *property* dan *method*. Dalam OOP, *class* merupakan kerangka dasar yang harus dibuat sebelum kita membuat *real object*. Untuk lebih jelas, perhatikan gambar berikut:



Sumber: cloudfront.net

Bila diperhatikan dari gambar diatas, dapat kita simpulkan bahwa sebuah *class* dapat memiliki banyak *object*. Dalam prakteknya, sebuah *class* memang dapat memiliki banyak *object* tergantung

dari berapa kali kita melakukan instansiasi. Instansiasi adalah proses atau mekanisme pembuatan *object* dalam OOP. Pada OOP, instansiasi ditandai dengan penggunaan *keyword* `new` diikuti nama dari *class* yang akan kita instansiasi.

Contoh *Class* pada PHP

Untuk membuat sebuah *class* pada PHP kita menggunakan *keyword* `class` diikuti nama dari *class* tersebut. Sebagai contoh kita akan membuat sebuah *class* `Mobil`, maka kita dapat membuatnya sebagai berikut:

```
<?php

class Mobil
{
}
```

Kita juga dapat menambahkan *method* atau *function* pada *class* sehingga *class* `Mobil` akan menjadi seperti berikut:

```
<?php

class Mobil
{
    public function jalan()
    {
        echo 'Mobil berjalan';
    }
}
```

Seperti yang sudah disebutkan pada pembahasan sebelumnya, bahwa selain memiliki *method*, *class* juga dapat memiliki *property*. Untuk membuat *property* pada *class* kita dapat menambah

Mudah sekali kan? kan langsung seperti berikut:

```
<?php

class Mobil
{
    public $roda;

    public function jalan()
    {
        echo 'Mobil berjalan';
    }
}
```

Anatomi *Class*

Untuk lebih memahami tentang struktur dari *class*, mari kita coba membedah kembali *class* `Mobil` diatas.

```
class Mobil
{
}
```

Keyword `class` adalah sebuah *keyword* yang digunakan PHP untuk menandai bahwa sebuah *class* didefinisikan. *Keyword* `class` diikuti oleh nama *class* dalam hal ini adalah `Mobil` dan

diikuti oleh kurung kurawal `{}`. Didalam kurung kurawal itulah segala *method* maupun *property* dari sebuah *class* didefinisikan.

```
public $roda;
```

Keyword `public` menandai bahwa sebuah *property* atau *method* memiliki visibilitas *public*. Di PHP terdapat 4 visibilitas yaitu *private, protected, public* dan *default* (tidak didefinisikan). Tentang visibilitas ini akan dibahas secara tersendiri pada pembahasan berikutnya. Setelah *keyword* `public` diikuti oleh *variable* `$roda` yang dalam konsep OOP disebut sebagai *property* yaitu sebuah *variable* yang dapat digunakan dalam lingkup *object*.

```
public function jalan()
{
    echo 'Mobil berjalan';
}
```

Pada contoh diatas, terdapat *keyword* `public` yang fungsinya telah dijelaskan sebelumnya. Setelah *keyword* `public` diikuti *keyword* `function` yang digunakan untuk menandai bahwa sebuah fungsi atau *method* didefinisikan. *Keyword* `function` diikuti oleh nama *method* yaitu `jalan` dan ikuti dengan tanda kurung `()` dilanjutkan dengan kurung kurawal `{}`.

Diantara kurung kurawal `{}` terdapat baris program yaitu `echo 'oleh _object_ Mobil berjalan';`, itulah yang disebut sebagai badan *method*. Didalam badan *method* kita dapat mendefinisikan *variable*, memanggil *method* lain atau bahwa memanggil *class* lain. Mudahnya, didalam *method* kita dapat mendefinisikan apa saja

yang kita dapat definisikan diluar *method*. Namun yang perlu diingat adalah bahwa segala *variable* yang didefinisikan dalam *method* bersifat *local* sehingga hanya dapat digunakan pada lingkup *method* tersebut. Jika Anda ingin membuat *variable* yang dapat diakses dalam lingkup *object* maka gunakanlah *property*.

Diantara tanda kurung `()` sebenarnya kita dapat mendefinisikan parameter yaitu *variable* yang akan dikirimkan ketika *method* tersebut dipanggil. Untuk contoh penggunaan parameter akan lebih jelas lagi pada pembahasan-pembahasan selanjutnya. Untuk saat ini kita cukup tahu dulu bahwa sebuah *method* dapat memiliki parameter.

Pembuatan *Object* (Instansiasi)

Seperti yang sudah dijelaskan sebelumnya bahwa sebuah *class* hanyalah *blueprint*, maka untuk membuatnya menjadi *real* kita perlu melakukan instansiasi. Proses instansiasi atau pembuatan *object* pada PHP ditandai dengan *keyword* `new` diikuti dengan nama *class*. Perhatikan contoh berikut:

```
<?php

class Mobil
{
    public $roda;

    public function jalan()
    {
        echo 'Mobil berjalan';
    }
```

```
}

$avanza = new Mobil();
```

Pada contoh diatas, kita menginstansiasi *class* `Mobil` dan memasukkannya kedalam *variable* `$avanza`. Selain cara diatas, kita juga dapat melakukan instansiasi tanpa menggunakan tanda kurung `()` setelah nama *class*.

```
<?php

class Mobil
{
    public $roda;

    public function jalan()
    {
        echo 'Mobil berjalan';
    }
}

$avanza = new Mobil;
```

Untuk memanggil *method* maupun *property* yang bersifat *public* dapat dilakukan sebagai berikut:

```
<?php

class Mobil
{
    public $roda;

    public function jalan()
    {
```

```
        echo 'Mobil berjalan';
    }
}

$avanza = new Mobil();
$avanza->roda = 4;

echo $avanza->jalan();
echo PHP_EOL;
echo $avanza->roda;
echo PHP_EOL;
```

Sehingga bila program diatas dijalankan maka akan menghasilkan *output* sebagai berikut:

```
php Mobil.php

Output:
Mobil berjalan
4
```

Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/RR865.

V. *Property* dan *Method*

Pembahasan yang akan dibahas pada bab ini adalah mengenai apa itu *property*, apa itu *method* serta memahami apa itu parameter dan bagaimana cara menyusun parameter yang baik.

Apa itu *Property*

Seperti yang sudah disinggung pada pembahasan sebelumnya, *property* adalah sebuah *variable* dapat digunakan dalam lingkup *class* atau *object*. *Property* sering disebut juga sebagai segala sesuatu yang dimiliki oleh *class*. *Property* memiliki visibilitas serta dapat memiliki nilai *default*. Perhatikan kembali contoh *class* `Mobil` berikut:

```
<?php

class Mobil
{
    public $roda;
}
```

Pada contoh diatas, *property* `$roda` memiliki visibilitas *public* dan tanpa nilai *default* atau nilai awal. Untuk memberikan nilai awal pada *property* `$roda`, kita dapat melakukannya sebagai berikut:

```
<?php
```

```
class Mobil
{
    public $roda = 4;
}
```

Pada contoh diatas, kita memberikan nilai awal pada *property* `$roda` dengan nilai `4`. Maka ketika kita melakukan instansiasi, secara otomasi *property* `$roda` akan langsung mempunyai nilai `4` sebagaimana yang telah kita definisikan.

```
<?php

class Mobil
{
    public $roda = 4;
}

$avanza = new Mobil();
echo $avanza->roda;
echo PHP_EOL;
```

Maka ketika program dieksekusi akan menghasilkan *output* sebagai berikut:

```
php Mobil.php

Output:
4
```

Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/mYNf4.

Apa itu *Method*

Jika *property* adalah segala sesuatu yang dimiliki oleh *class* atau *object*, maka *method* adalah segala sesuatu yang dapat dilakukan oleh *class* atau *object*. Sama seperti *property*, *method* juga memiliki visibilitas serta dapat memiliki parameter. Parameter dapat memiliki nilai awal atau *default value*. Bila parameter tidak memiliki *default value* maka parameter tersebut dianggap sebagai *mandatory parameter*. Perhatikan kembali *class* `Mobil` berikut:

```
<?php

class Mobil
{
    public function jalan()
    {
        echo 'Mobil berjalan';
    }
}
```

Pada contoh diatas, *method* `jalan()` memiliki visibilitas *public* dan tidak memiliki parameter. Untuk menambahkan parameter, kita dapat mendefinisikannya sebagai berikut:

```
<?php

class Mobil
{
    public function jalan($arah = 'depan')
    {
        echo 'Mobil berjalan ke arah '.$arah;
    }
}
```

Pada contoh diatas, kita menambahkan parameter `$arah` pada *method* `jalan()` serta menambahkan *default value* `depan` pada parameter tersebut. Kita dapat mengetes *code* diatas dengan cara sebagai berikut:

```
<?php

class Mobil
{
    public function jalan($arah = 'depan')
    {
        echo 'Mobil berjalan ke arah '.$arah;
    }
}

$avanza = new Mobil();

echo $avanza->jalan();
echo PHP_EOL;
echo $avanza->jalan('belakang');
echo PHP_EOL;
```

Ketika program diatas dijalankan, maka *output*-nya adalah sebagai berikut:

```
php Mobil.php

Output:
Mobil berjalan ke arah depan
Mobil berjalan ke arah belakang
```

Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/Klcha.

Bila kita perhatikan, *default value* akan ditimpa oleh nilai yang kita masukkan. Dalam contoh diatas, kita memasukkan nilai `belakang` sehingga parameter `$arah` yang tadinya berisi `depan` berganti menjadi `belakang`. Dalam contoh *real*, *default value* dapat digunakan untuk mengeset nilai awal `$port` MySQL dari koneksi *database* yang secara *default* adalah `3306` serta `$host` yang biasanya memiliki nilai awal `localhost`. Perhatikan contoh berikut:

```
<?php

class Koneksi
{
    public function connect($username, $password, $host = 'l
ocalhost', $port = 3306)
    {
        //Logic koneksi
    }
}
```

Pada contoh diatas, parameter `$username` dan `$password` bersifat *mandotory* sehingga bila kita tidak memasukkan nilai tersebut ketika memanggil fungsi `connect()` maka akan terjadi *error*. Perhatikan code dibawah ini:

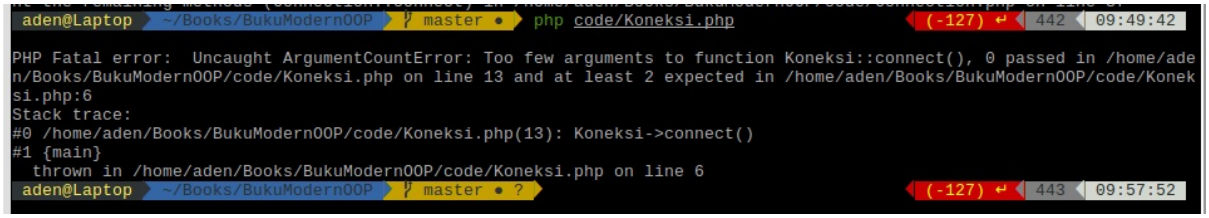
```
<?php

class Koneksi
{
    public function connect($username, $password, $host = 'l
ocalhost', $port = 3306)
```

```
    {
        //Logic koneksi
    }
}

$koneksi = new Koneksi();
$koneksi->connect();
```

Bila kita menjalankan program diatas maka akan terjadi *error* `PHP Fatal error: Uncaught ArgumentCountError: Too few arguments to function Koneksi::connect()` seperti gambar dibawah ini:



Ini terjadi karena kita tidak memasukkan nilai `$username` dan `$password` sedangkan parameter tersebut bersifat *mandatory*. Untuk memperbaikinya, kita cukup memasukkan nilai sebagai berikut:

```
<?php

class Koneksi
{
    public function connect($username, $password, $host = 'l
ocalhost', $port = 3306)
    {
        //Logic koneksi
    }
}

$koneksi = new Koneksi();
$koneksi->connect('root', '');
```

Bila kita mengeksekusi program diatas maka tidak akan terjadi *error* dan tidak mengeluarkan *output* apapun karena memang kita hanya membuat sebuah *method* kosong.

Urutan Parameter pada *Method*

Sebenarnya tidak ada aturan yang baku yang mengatur tentang urutan parameter pada sebuah *method*. Kita bisa saja mendefinisikan sesuka kita parameter-parameter dari sebuah *method*, namun tips berikut dapat dipertimbangkan untuk menghindari *error* dan memudahkan kita pada saat pemanggilan *method*.

- Parameter yang *mandatory* diletakkan didepan parameter yang memiliki *default value*.
- Penamaan parameter harus spesifik dan memiliki arti yang jelas serta menunjukkan kegunaan dari parameter tersebut.
- Gunakan *type hinting* (akan dibahas pada bab terpisah) pada parameter untuk menghindari kesalahan ketika memberikan nilai pada parameter.

Bila Anda perhatikan, apakah *class* `Koneksi` telah menerapkan tips diatas? Jawabannya adalah iya. Hanya saja, kita belum menerapkan *type hinting* karena memang belum dibahas.

VI. Visibilitas

Bab ini akan membahas tentang apa itu visibilitas dalam OOP, tingkatan visibilitas serta bagaimana mendefinisikan visibilitas dalam bahasa pemrograman PHP.

Apa itu Visibilitas

Visibilitas adalah salah satu fitur penting yang ada pada OOP. Fitur ini mengatur hak akses terhadap *property* maupun *method* dari sebuah *class*. Hak akses disini berbeda dengan hak akses pada aplikasi, hak akses disini adalah hak akses yang ada pada level bahasa pemrograman.

Visibilitas dalam OOP berperan penting dalam menjamin keamanan informasi yang terdapat pada *property* maupun *method*. Dengan fitur ini, *programmer* dapat membatasi dan mengatur *programmer* lainnya tentang bagaimana mengakses sebuah *property* atau *method* dari sebuah *class* atau fitur yang dibuatnya.

Tingkatan Visibilitas pada PHP

Dalam bahasa PHP, Visibilitas dibedakan menjadi 4 yaitu *private*, *protected*, *public* dan *default* atau tidak didefinisikan. Berikut adalah penjelasan masing-masing visibilitas.

- *Private*

Visibilitas *private* adalah visibilitas paling rendah pada OOP. Sebuah *property* atau *method* yang diberikan visibilitas *private* maka *property* atau *method* tersebut hanya dapat diakses dari lingkup *class* dimana *property* atau *method* tersebut didefinisikan. Untuk memberikan visibilitas *private* pada *property* atau *method* kita dapat menggunakan *keyword* `private` didepan *property* atau *method*. Perhatikan contoh berikut:

```
<?php

class Mobil
{
    private $roda = 4;

    private function jalan()
    {
        echo 'Mobil berjalan';
    }
}

$avanza = new Mobil();

echo $avanza->jalan();
echo PHP_EOL;
echo $avanza->roda;
echo PHP_EOL;
```

Bila program diatas dijalankan, maka akan muncul *error* `PHP Fatal error: Uncaught Error: Call to private method Mobil::jalan() from context`. Hal ini terjadi karena kita mencoba mengakses fungsi

`jalan()` yang memiliki visibilitas *private* diluar lingkup *class* yaitu dipanggil dari lingkup *object*. Hal yang sama juga akan terjadi pada *property* `$roda` dimana *property* tersebut juga memiliki visibilitas *private*.

Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/VQ18h.

Untuk mengakses *property* atau *method* dalam lingkup *class* digunakan *keyword* `$this` yang akan dibahas secara spesifik pada bab terpisah. Perhatikan contoh berikut:

```
<?php

class Mobil
{
    private $roda = 4;

    private function jalan()
    {
        echo 'Mobil berjalan';
    }

    public function jumlahRoda()
    {
        echo $this->roda;
    }
}

$avanza = new Mobil();

echo $avanza->jumlahRoda();
echo PHP_EOL;
```

Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/u9Z94. Bila program diatas dieksekusi, maka akan muncul *output* `4` sebagaimana nilai dari *property* `$roda` .

- *Protected*

Sebuah *property* atau *method* yang diberikan visibilitas *protected* maka *property* atau *method* tersebut dapat diakses dari lingkup *class* dimana *property* atau *method* tersebut didefinisikan serta turunan dari *class* tersebut. Untuk memberikan visibilitas *protected* pada *property* atau *method* kita dapat menggunakan *keyword* `protected` didepan *property* atau *method*. Perhatikan contoh berikut:

```
<?php

class Mobil
{
    private $roda = 4;

    protected function jalan()
    {
        echo 'Mobil berjalan';
    }

    public function jumlahRoda()
    {
        echo $this->roda;
    }
}

$avanza = new Mobil();
```

```
echo $avanza->jalan();
echo PHP_EOL;
```

Bila program diatas dijalankan, maka akan muncul *error* `PHP Fatal error: Uncaught Error: Call to protected method Mobil::jalan() from context` . Hal ini terjadi karena kita mencoba mengakses fungsi `jalan()` yang memiliki visibilitas *protected* diluar lingkup *class* yaitu dipanggil dari lingkup *object*.

Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/qPsFG.

- *Public*

Visibilitas *public* adalah visibilitas tertinggi pada OOP. Sebuah *property* atau *method* yang diberikan visibilitas *public* maka *property* atau *method* tersebut dapat diakses baik dari lingkup *class* maupun *object*. Untuk memberikan visibilitas *public* pada *property* atau *method* kita dapat menggunakan *keyword* `public` didepan *property* atau *method*. Mari kita melihat kembali contoh berikut:

```
<?php

class Mobil
{
    private $roda = 4;

    public function jumlahRoda()
    {
        echo $this->roda;
    }
}
```

```
$avanza = new Mobil();

echo $avanza->jumlahRoda();
echo PHP_EOL;
```

Method `jumlahRoda()` diberikan visibilitas *public* sehingga *method* tersebut dapat diakses dari luar *class* yaitu dalam lingkup *object*. Sehingga bila program diatas dieksekusi, maka akan muncul *output* `4` sebagaimana nilai dari *property* `$roda`. Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/umsp4.

- *Default* atau tidak didefinisikan

Bila kita tidak mendefinisikan visibilitas pada *property* atau *method* secara eksplisit, maka PHP secara *default* akan memberikan visibilitas pada *property* atau *method* tersebut dengan visibilitas *public*. Sehingga `public function jumlahRoda()` sama dengan `function jumlahRoda()` yaitu sama-sama memiliki visibilitas *public*.

Tips Visibilitas

Untuk menghindari kebingungan dan menjaga konsistensi *code* maka ada baiknya Anda menerapkan tips berikut:

- Definisikan visibilitas secara eksplisit.
- Gunakan visibilitas *private* atau *protected* pada *property*.
- Hindari penggunaan visibilitas *public* pada *property*, gunakan hanya jika memang benar-benar diperlukan.

- Batasi visibilitas pada *method* jika memungkinkan.

VII. Konsep Statis dan Konstanta

Pada bab ini kita akan membahas mengenai konsep *static* pada OOP, cara penerapannya hingga tingkatan visibilitasnya. Selain itu juga kita akan membahas tentang konstanta dan bagaimana mendefinisikan konstanta menggunakan bahasa pemrograman PHP.

Apa itu Konsep Statis

Dalam pemrograman berbasis objek segala sesuatu diibaratkan atau dimodelkan dalam *class* dan untuk merealisasikannya kita harus melakukan instansiasi. Sebagai contoh ketika kita ingin mengetahui jumlah roda pada *class* `Mobil` kita harus melakukan instansiasi *class* `Mobil` dan memasukkannya kedalam *variable* baru kemudian kita memanggilnya *method* `jumlahRoda()`. Itu adalah aturan dasar dari konsep OOP dimana segala sesuatu dimodelkan kedalam *class* atau *object*.

Konsep statis atau *static* adalah sebuah konsep yang keluar dari aturan dasar dari konsep OOP tersebut. Pada konsep *static* kita tidak perlu melakukan instansiasi untuk dapat memanggil sebuah *property* atau *method*. Karena tidak perlu melakukan instansiasi, maka kita dapat langsung memanggil *property* atau *method*

dengan menggunakan nama *class* diikuti `::` (*scope resolution operation*) diikuti *property* atau *method* (`NamaClass::$propertyStatic`).

Contoh Penerapan Konsep Statis

Untuk mendefinisikan sebuah *property* atau *method* menjadi statis, kita menggunakan *keyword* `static` seperti pada contoh berikut:

```
<?php

class Singa
{
    public static $KAKI = 4;

    public static function lari()
    {
        echo 'Singa berlari';
    }
}

echo Singa::$KAKI;
echo PHP_EOL;
echo Singa::lari();
echo PHP_EOL;
```

Bila program diatas dijalankan maka akan menghasilkan *output* sebagai berikut:

```
php Singa.php

Output:
4
```

```
Singa berlari
```

Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/NQ0Fv.

Untuk mengakses *property* atau *method static* pada lingkup *class* kita dapat menggunakan 3 cara yaitu dengan menggunakan nama *class* seperti cara diatas, menggunakan *keyword* `self` atau bisa juga menggunakan *keyword* `static`. Perhatikan contoh berikut:

```php
<?php

class Singa
{
    public static $KAKI = 4;

    public function kaki1()
    {
        echo Singa::$KAKI;
    }

    public function kaki2()
    {
        echo self::$KAKI;
    }

    public function kaki3()
    {
        echo static::$KAKI;
    }
}

$singa = new Singa();

echo $singa->kaki1();
echo PHP_EOL;
```

```
echo $singa->kaki2();
echo PHP_EOL;
echo $singa->kaki3();
echo PHP_EOL;
```

Bila program diatas dijalankan maka *output* dari *method* `kaki1()`, `kaki2` maupun `kaki3` akan sama yaitu `4`. Anda dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/dABh0.

Visibilitas pada Konsep Statis

Sama seperti *property* atau *method* pada umumnya yang memiliki visibilitas, *property* atau *method* yang statis pun tetap memiliki visibilitas. Tingkatan visibilitasnya pun sama seperti pada *property* atau *method* biasa yaitu *private*, *protected*, *public* dan *default*. Tidak ada perbedaan antara statis maupun non statis pada *property* dan *method* selain pada cara mengaksesnya saja.

Apa itu Konstanta

Bila kita kembali ke pelajaran jaman SMA, konstanta adalah sebuah nilai yang tidak berubah dan telah ditetapkan nilainya dari sejak awal. Contoh konstanta adalah `PI` yang nilainya `3.14` ada juga gaya gravitasi yang nilainya `9.8`. Tidak jauh berbeda dengan konsep diatas, pada OOP konstanta adalah sebuah nilai yang tidak dapat dirubah selama proses *runtime*.

Contoh Penerapan Konsep Konstanta

Untuk mendefinisikan sebuah konstanta kita menggunakan *keyword* `const` dan untuk mengakses konstanta didalam *class* kita menggunakan *keyword* `self` dan diluar *class* kita menggunakan nama *class*. Perhatikan contoh berikut:

```php
<?php

class Lingkaran
{
    public const PI = 3.14;

    public function luas($jari)
    {
        echo self::PI * $jari * $jari;
    }
}

$lingkaran = new Lingkaran();

echo Lingkaran::PI;
echo PHP_EOL;
$lingkaran->luas(7);
echo PHP_EOL;
```

Bila program diatas dijalankan maka *output*-nya adalah sebagai berikut:

```
Output:
3.14
153.86
```

Anda dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/IW70B.

Visibilitas pada Konsep Konstanta

Pada PHP 7.1, PHP menambahkan fitur baru pada konstanta yaitu visibilitas sehingga kita dapat memberikan visibilitas pada konstanta seperti halnya *property* dan *method*. Tingkatan visibilitasnya pun sama seperti pada *property* atau *method* yaitu *private*, *protected*, *public* dan *default*.

VIII. Keyword `$this` dan `self`

Bab ini akan membahas salah satu fitur yang wajib ada dalam bahasa pemrograman yang mendukung konsep OOP yaitu `$this` dan `self` serta perbedaan keduanya dalam bahasa pemrograman PHP.

Apa itu `$this`

Pada pembahasan sebelumnya kita sudah cukup banyak menggunakan *keyword* `$this` ketika memanggil *property* dalam lingkup *class* seperti pembahasan tentang visibilitas. Perhatikan kembali contoh dibawah ini:

```
<?php

class Mobil
{
    private $roda = 4;

    public function jumlahRoda()
    {
        echo $this->roda;
    }
}

$avanza = new Mobil();
```

```
echo $avanza->jumlahRoda();
echo PHP_EOL;
```

Pada pemrograman berbasis objek, konsep *keyword* `$this` pasti ada walaupun cara penulisan dan atau *syntax*-nya mungkin berbeda. Pada OOP, *keyword* `$this` adalah sebuah *variable* yang merujuk pada *object* yang diinstansiasi. *Variable* `$this` nantinya akan diganti dengan *variable* hasil instansiasi ketika sebuah *object* terbentuk. Pada contoh diatas, *variable* `$this` akan diganti oleh *variable* `$avanza`. Untuk lebih memahaminya, mari kita lihat contoh dibawah ini:

```
<?php

class Printer
{
    private $content;

    public function setContent($content)
    {
        $this->content = $content;
    }

    public function cetak()
    {
        echo $this->content;
    }
}

$printer1 = new Printer();
$printer1->setContent('Aku printer satu');
$printer1->cetak();
echo PHP_EOL;

$printer2 = new Printer();
```

```
$printer2->setContent('Aku printer dua');
echo $printer2->cetak();
echo PHP_EOL;
$printer1->cetak();
echo PHP_EOL;
```

Bila program diatas dieksekusi, maka akan menghasilkan *output* sebagai berikut:

```
php Printer.php

Aku printer satu
Aku printer dua
Aku printer satu
```

Anda juga dapat menjalankan program diatas secara *online* dengan membuka *link* berikut https://3v4l.org/vKJct.

Dari contoh diatas dapat kita pahami bahwa *keyword* `$this` itu merujuk pada spesifik *object* dan tidak bercampur antara satu *object* dengan *object* lainnya. Ini dibuktikan dengan hasil *output* antara `$printer1` dan `$printer2` yang berbeda walaupun keduanya sama-sama menginisiasi *class* `Printer`. Nilai `$content` pada `$printer1` tidak akan tertimpa oleh nilai `$content` pada `$printer2` karena keduanya merupakan *object* yang berbeda.

Secara mudah, *keyword* `$this` adalah sebuah *keyword* yang digunakan untuk merujuk pada *obejct* yang belum diketahui dan digunakan untuk mempermudah kita dalam menuliskan *code*. Keyword `$this` hanya dapat diakses dari internal *class* dan tidak

dapat diakses dari luar *class* atau pada lingkup *object*. Selain itu, *keyword* `$this` bersifat *read only* seperti halnya konstanta sehingga kita tidak dapat mengubah nilainya.

Apa itu `self`

Sama seperti *keyword* `$this`, *keyword* `self` juga memiliki pengertian dan fungsi yang sama. Yang membedakannya dengan *keyword* `$this` adalah bahwa *keyword* `self` hanya digunakan untuk memanggil *property* atau *method* yang bersifat statis dan juga untuk memanggil konstanta. Perhatikan kembali contoh berikut:

```
<?php

class Lingkaran
{
    const PI = 3.14;

    public function luas($jari)
    {
        echo self::PI * $jari * $jari;
    }
}

$lingkaran = new Lingkaran();

echo Lingkaran::PI;
echo PHP_EOL;
```

```
$lingkaran->luas(7);
echo PHP_EOL;
```

Sama seperti *keyword* `$this` , *keyword* `self` juga hanya dapat digunakan pada lingkup *class* dan tidak dapat digunakan pada lingkup *object*.

IX. *Return Value*

Setelah membaca bab ini diharapkan Anda akan memahami apa itu *return value* dan bagaimana cara mendefinisikan *return value* menggunakan bahasa pemrograman PHP.

Apa itu *Return Value*

Sebelum kita mengenal apa itu *return value*, ada baiknya saya sedikit bercerita sebagai penggandaian dari *return value*.

> Muhammad adalah seorang petani kedelai. Ia menanam benih kedelai kemudian merawatnya hingga benih itu tumbuh. Setelah tumbuh, ia memupuknya hingga pohon kedelai itu siap untuk dipanen.
>
> Muhammad kemudian memanen kedelai tersebut setelah dirasa kedelai tersebut siap untuk dipanen.

Dari pengandaian diatas, kita dapat mengibaratkan bahwa proses menanam hingga panen adalah sebuah proses yang terjadi didalam *method* atau fungsi. Sedangkan kedelai yang dihasilkan adalah *return value* dari proses tersebut.

Dalam OOP, *return value* adalah sebuah nilai yang dikembalikan oleh sebuah *method* ketika *method* tersebut dipanggil. Tipe data dari *return value* tidak harus sama dengan tipe data yang

dimasukkan melalui parameter. Bahkan ketika sebuah *method* tidak memiliki parameter sekalipun, *method* tersebut tetap dapat mengembalikan *return value*.

Contoh Penggunaan *Return Value*

Untuk membuat sebuah *method* dapat mengembalikan sebuah nilai kita menggunakan *keyword* `return`. Perhatikan kembali contoh *class* `Lingkaran` berikut:

```
<?php

class Lingkaran
{
    const PI = 3.14;

    public function luas($jari)
    {
        echo self::PI * $jari * $jari;
    }
}

$lingkaran = new Lingkaran();

$lingkaran->luas(7);
echo PHP_EOL;
```

Pada *class* `Lingkaran` diatas, *method* `luas()` langsung mengeluarkan *output* berubah `echo` hasil dari perkalian rumus luas lingkaran. Bila kita hendak mengubah agar *method* `luas()` tersebut mengembalikan *return value* maka kita harus mengganti `echo` dengan *keyword* `return` seperti contoh dibawah ini:

```php
<?php

class Lingkaran
{
    const PI = 3.14;

    public function luas($jari)
    {
        return self::PI * $jari * $jari;
    }
}

$lingkaran = new Lingkaran();

echo $lingkaran->luas(7);
echo PHP_EOL;
```

Kita juga dapat memasukkan *return value* kedalam sebuah *variable* sebagai berikut:

```php
<?php

class Lingkarancara-penggunaan
{
    const PI = 3.14;

    public function luas($jari)
    {
        return self::PI * $jari * $jari;
    }
}

$lingkaran = new Lingkaran();
$luas = $lingkaran->luas(7);

echo $luas;
```

```
  echo PHP_EOL;
```

Bagaimana dapat dipahami kan? Mudah bukan? Mari kita masuki dunia OOP PHP lebih dalam lagi.

X. *Constructor* dan *Destructor*

Pada bab ini akan dibahas apa itu *constructor*, apa itu *destructor* dan bagaimana cara penggunaannya pada bahasa pemrograman PHP.

Apa itu *Constructor*

Constructor adalah sebuah *method* khusus yang dieksekusi ketika sebuah *class* diinstansiasi. *Constructor* digunakan untuk mempersiapkan *object* ketika *keyword* `new` dipanggil. Dalam *constructor* kita dapat melakukan apapun yang kita dapat lakukan pada *method* biasa namun tidak bisa mengembalikan *return value*.

Muncul pertanyaan, kenapa *constructor* tidak dapat mengembalikan *return value*? Ya jelas lah tidak bisa mengembalikan *return value*, *kan keyword* `new` itu sudah mengembalikan berupa *object* dari *class* yang diinstansiasi. *Masa* kemudian *constructor* mengembalikan lagi nilai yang sesuai?

Misalnya, kita punya *class* `A` maka ketika menginisiasi *class* `A` tersebut dengan *keyword* `new` kedalam *variable* `$a` maka saat itu sebenarnya telah mengembalikan nilai berupa *object* `A` ke dalam *variable* `$a` tersebut.

Bagaimana jadinya jika didalam *constructor* kita dapat mengembalikan nilai dan kemudian membuat *constructor* dengan mengembalikan nilai *integer* `1` misalnya. Maka yang terjadi ketika

kita melakukan instansiasi *class* `A` dan fungsi *constructor* dipanggil, alih-alih kita mendapatkan *object* `A` yang ada kita justru mendapatkan *integer* `1`.

Bagaimana sudah jelas *kan* kenapa *contructor* hanya dapat berisi *logic* tapi tidak bisa mengembalikan *return value*?

Contoh Penggunaan *Constructor*

Untuk membuat *constructor* kita harus membuat *method* dengan nama `__construct()`. Perhatikan contoh berikut:

```
<?php

class Connection
{
    private $host;

    private $port;

    private $username;

    private $password;

    private $database;

    public function __construct($username, $password, $datab
ase, $host = 'localhost', $port = 3306)
    {
        $this->host = $host;
        $this->port = $port;
        $this->username = $username;
        $this->password = $password;
        $this->database = $database;
```

```
    }

    public function connect()
    {
        return new PDO(sprintf('mysql:host=%s;port=%d;dbname
=%s', $this->host, $this->port, $this->database), $this->use
rname, $this->password);
    }
}

$connection = new Connection('root', 'aden', 'quiz');
$pdo = $connection->connect();
```

Pada contoh diatas, kita membuat *class* `Connection` yang berfungsi untuk melakukan koneksi ke RDBMS MySQL dengan menggunakan *driver* `PDO` (*PHP Data Objects*). *Class* `Connection` mempunyai *constructor* yang memiliki 5 parameter dimana 3 diantaranya bersifat *mandatory*. Ketika kita melakukan instansiasi, maka perlu memasukkan parameter ke *class* `Connection` seperti pada contoh diatas.

Muncul sebuah pertanyaan, bagaimana bila kita tidak mendefinisikan *constructor* seperti pada contoh-contoh sebelumnya? Sebenarnya, secara *default* PHP akan membuat "*virtual*" *constructor* tanpa parameter dan tanpa *logic* jika kita tidak mendefinisikan *constructor* secara eksplisit. Perhatikan contoh dibawah ini:

```
<?php

class Connection
{
    public function __construct()
```

```
    {
    }
}
```

Kira-kira seperti itulah yang terjadi jika kita tidak mendefinisikan *constructor* secara eksplisit pada *class* yang kita buat. Namun seperti yang saya jelaskan diatas, *constructor* tersebut bersifat *virtual* sehingga tidak benar-benar ada.

Apa itu *Destructor*

Destructor adalah sebuah *method* khusus yang dieksekusi ketika sebuah *object* dihapus dari *memory*. Secara mudah, *destructor* adalah kebalikan dari *constructor*. Sama seperti pada *constructor*, PHP juga akan membuat *destructor* tanpa parameter dan tanpa *logic* jika kita tidak mendefinisikan *destructor* secara eksplisit. Berbeda dengan *constructor* yang dapat memiliki parameter, *destructor* tidak dapat memiliki parameter dan hanya dapat berisi *logic*.

Kenapa *desctrutor* tidak dapat memiliki parameter? Ya karena bagaimana caranya kita memasukkan sebuah nilai sesaat sebelum *object* dihancurkan?

Contoh Penggunaan *Desctructor*

Untuk membuat *desctrutor* kita harus membuat *method* dengan nama `__destruct()`. Perhatikan contoh berikut:

```php
<?php

class Connection
{
    public function __destruct()
    {
        echo 'Object dihapus dari memory';
    }
}

$connection = new Connection();
unset($connection);
```

Pada contoh diatas, ketika fungsi `unset()` dijalankan oleh PHP, maka akan mengeluarkan *output* `Object dihapus dari memory` sebagaimana yang didefinisikan pada *method* `__destruct()`.

Bila tidak ada sesuatu yang khusus yang harus dilakukan ketika *object* dihapus dari *memory*, sebaiknya kita tidak perlu membuat sebuah *destructor* secara eksplisit.

Jangan Bilang Siapa-Siapa

Saya tidak tahu apakah ini sebuah fitur ataukah sebuah *bug*, namun pada PHP kita dapat melakukan pemanggilan fungsi `__construct()` maupun `__destruct()` dari sebuah *object*. Perhatikan contoh berikut:

```php
<?php

class Connection
{
```

```
    public function __destruct()
    {
        echo 'Object dihapus dari memory';
    }
}

$connection = new Connection();
$connection->__destruct();
unset($connection);
```

Apakah Anda berfikir ketika kita memanggil fungsi `__desctruct()` secara langsung maka *object* akan dihapus dari *memory*? Kalau jawaban Anda adalah iya, maka Anda salah besar! Buktinya *variable* `$connection` tetap dapat di-`unset()` dan bukan `undefined variable`.

Pada PHP, kita dapat memanggil fungsi `__destruct()` maupun `__construct()` secara langsung sebagaimana fungsi biasa. Syarat agar fungsi tersebut dapat dipanggil adalah Anda harus mendefinisikannya secara eksplisit. Bagi saya ini sangat menyebalkan, kenapa? Perhatikan contoh berikut:

```
<?php

$a = new \DateTimeImmutable();

$b = $a->add(new \DateInterval('P1D'));

echo $b->format('Y-m-d');
echo PHP_EOL;
echo $a->format('Y-m-d');
echo PHP_EOL;
```

Menurut halaman dokumentasi resminya, *class* `DateTimeImmutable` adalah sebuah *class* yang seperti *class* `DateTime` namun nilai yang ada didalamnya tidak dapat berubah sesaat setelah kita melakukan instansiasi. Alih-alih mengubah nilai, `DateTimeImmutable` justru mengembalikan *object* `DateTimeImmutable` baru ketika kita mencoba untuk mengubah nilai yang ada didalamnya.

Pada contoh diatas misalnya, *variable* `$a` adalah *object* dari `DateTimeImmutable` lalu kemudian kita mencoba mengubahnya dengan menambahkan `1 hari` dari tanggal sekarang dengan memanggil *method* `add()`. Sesuai dengan dokumentasi resminya, maka ketika kita memanggil *method* `add()` yang terjadi bukannya nilai *object* `$a` yang berubah, tapi kita justru mendapatkan *object* baru yaitu `$b` dengan nilai baru.

```
php test_dateimmutable.php

Output:
2018-01-20
2018-01-19
```

Hal diatas adalah *behavior* yang benar dari `DateTimeImmutable`, namun karena kita tahu bahwa *constructor* dapat kita panggil walaupun *object* telah terbentuk, mari kita coba rubah nilai dari *object* `$a` yang seharusnya tidak dapat berubah itu.

```
<?php

$a = new \DateTimeImmutable();
```

```
$b = $a->add(new \DateInterval('P1D'));

echo $b->format('Y-m-d');
echo PHP_EOL;
echo $a->format('Y-m-d');
echo PHP_EOL;

$a->__construct($b->format('Y-m-d'));

echo $a->format('Y-m-d');
echo PHP_EOL;
```

Bila kita jalankan program diatas, maka hasilnya adalah sebagai berikut:

```
php test_dateimmutable.php

Output:
2018-01-20
2018-01-19
2018-01-20
```

Dan akhirnya, `DateTimeImmutable` yang seharusnya tidak dapat berubah nilainya pun dapat kita rubah nilainya.

Meski hal tersebut (memanggil fungsi `__construct` maupun `__destruct` secara langsung) tidak wajar dilakukan, namun hal tersebut dapat dilakukan dan justru dengan itulah kita dapat melakukan sesuatu yang seharusnya tidak dapat dilakukan.

XI. Enkapsulasi

Enkapsulasi adalah salah satu fitur yang ada pada bahasa pemrograman yang mendukung OOP yang memungkinkan kita mengatur bagaimana sebuah *object* dipresentasikan. Bab ini akan membahas tentang enkapsulasi secara lebih dalam dan mudah dipahami.

Apa itu Enkapsulasi

Enkapsulasi adalah sebuah mekanisme penyembunyian informasi pada *property* atau *method* dalam *class*. Enkapsulasi dalam konsep OOP diwujudkan dalam visibilitas yang telah kita bahas sebelumnya. Dengan enkapsulasi, kita bisa mengatur bagaimana sebuah *property* diset atau diakses nilainya. Bahkan kita juga bisa mengatur bagaimana sebuah *method* diakses atau dipanggil.

Penerapan Enkapsulasi

Enkapsulasi memungkinkan kita mengatur bagaimana program atau *object* lain berinteraksi dengan *object* yang kita buat. Sehingga dengan enkapsulasi kita bisa membuat pembatasan akses sekaligus mengatur cara mengakses *property* maupun *method*. Perhatikan kembali contoh berikut:

```
<?php
```

```
class Connection
{
    private $host;

    private $port;

    private $username;

    private $password;

    private $database;

    public function __construct($username, $password, $datab
ase, $host = 'localhost', $port = 3306)
    {
        $this->host = $host;
        $this->port = $port;
        $this->username = $username;
        $this->password = $password;
        $this->database = $database;
    }

    public function getConnection()
    {
        return new PDO(sprintf('mysql:host=%s;port=%d;dbname
=%s', $this->host, $this->port, $this->database), $this->use
rname, $this->password);
    }
}
```

Pada *class* `Connection` kita mengatur bahwa ketika *object* `Connection` dibuat, kita harus memasukkan parameter `$username` , `$password` dan seterusnya. Setelah itu, kita hanya

bisa mengakses *method* `getConnection()` sementara semua *property* yang ada tidak dapat kita akses secara langsung karena bersifat *private*.

```
<?php

class Mobil
{
    private $roda = 4;

    protected function roda()
    {
        return $this->roda + 2;
    }

    public function getRoda()
    {
        return $this->roda();
    }
}
```

Pada *class* `Mobil` diatas, kita memperbolehkan *object* untuk mengakses *method* `getRoda()` dimana *method* tersebut ternyata memanggil *method* `roda()` dan mengembalikan nilai dari *method* tersebut. *Object* `Mobil` tidak diberikan cara untuk bisa memanipulasi nilai dari *variable* `$roda` sehingga *object* tersebut cuma bisa menerima hasil pemrosesan dari *method* `getRoda()` tanpa bisa berbuat apa-apa.

Kesimpulan yang bisa kita peroleh adalah bahwa enkapsulasi memberikan kebebasan kepada kita untuk mengatur perilaku *object* dari *class* yang kita buat.

XII. Pewarisan

Pada bab ini akan dibahas apa itu pewarisan dan bagaimana menerapkan pewarisan pada bahasa pemrograman PHP.

Apa itu Pewarisan

Pewarisan adalah mekanisme pemberian sifat maupun ciri khusus dari induk (*parent*) kepada keturunannya (*child*). Bila kita hubungkan dengan konsep pada OOP maka pewarisan adalah mekanisme pemberikan *property* maupun *method* dari *parent class* kepada *child class*.

Property dan *method* yang diwariskan adalah *property*, *method* dan konstanta yang mempunyai visibilitas *protected* dan *public* sebagaimana yang telah dibawah pada bab sebelumnya.

Penerapan Pewarisan

Untuk membuat sebuah *class* menjadi turunan dari *class* lainnya, kita menggunakan *keyword* `extends` sebagai berikut:

```
<?php

class Hewan
{
    private $jenis;
```

```
    public function setJenis($jenis)
    {
        $this->jenis = $jenis;
    }

    public function getJenis()
    {
        return $this->jenis;
    }
}

class Kambing extends Hewan
{
}

class Harimau extends Hewan
{
}

class Singa extends Hewan
{
}

$kambing = new Kambing();
$kambing->setJenis('Herbivora');

$harimau = new Harimau();
$harimau->setJenis('Karnivora');

$singa = new singa();
$singa->setJenis('Karnivora');

echo $kambing->getJenis();
echo PHP_EOL;
echo $harimau->getJenis();
echo PHP_EOL;
echo $singa->getJenis();
```

```
echo PHP_EOL;
```

Pada contoh diatas, secara otomatis *class* `Kambing`, `Harimau` dan `Singa` memiliki semua *method* yang ada pada *class* `Hewan` sehingga bila program diatas dijalankan maka *output*-nya adalah sebagai berikut:

```
php Hewan.php

Output:
Herbivora
Karnivora
Karnivora
```

Pada bahasa pemrograman PHP tidak dikenal *multiple inheritance* sehingga kita setiap *child class* hanya boleh memiliki 1 *parent class*. Perhatikan contoh berikut:

```
<?php

class Hewan
{
    private $jenis;

    public function setJenis($jenis)
    {
        $this->jenis = $jenis;
    }

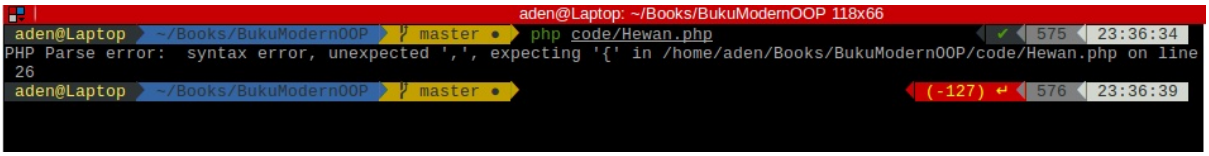
    public function getJenis()
    {
        return $this->jenis;
    }
}
```

```php
class Mamalia
{
    public function menyusui()
    {
        echo 'mik susu aaahhh';
    }
}

class Kambing extends Hewan, Mamalia
{
}

$kambing = new Kambing();
```

Bila program diatas dijalankan maka yang terjadi adalah *error syntax* karena setelah pada baris `, Mamalia` seharusnya adalah tanda kurung kurawan buka `{` seperti gambar berikut:



Namun kita dapat menggunakan *nested inheritance* untuk mengakali hal tersebut seperti tampak pada contoh dibawah ini:

```php
<?php

class Hewan
{
    private $jenis;

    public function setJenis($jenis)
    {
        $this->jenis = $jenis;
    }
```

```
    public function getJenis()
    {
        return $this->jenis;
    }
}

class Mamalia extends Hewan
{
    public function menyusui()
    {
        echo 'mik susu aaahhh';
    }
}

class Kambing extends Mamalia
{
}

$kambing = new Kambing();
$kambing->menyusui();
echo PHP_EOL;
```

XIII. *Overloading* dan *Overriding*

Bab ini akan menjabarkan apa itu *overloading*, apa itu *overriding* serta bagaimana menerapkannya pada bahasa pemrograman PHP. Selain itu, pada bab ini juga akan dibahas tentang apa itu *keyword* `parent` dan bagaimana cara penggunaannya.

Keyword `parent`

Pada pembahasan sebelumnya kita telah memahami tentang 2 *keyword* yang merujuk pada *object* yang diinstansiasi yaitu `$this` dan `self`. Pada pembahasan kali ini, saya akan menambahkan satu lagi *keyword* yang merujuk pada *object* yaitu `parent`.

Sama seperti `$this` dan `self` yang akan bersifat dinamis dan akan digantikan oleh *real object* ketika diinstansiasi, *keyword* `parent` pun demikian. Hanya saja bila *keyword* `$this` dan `self` itu merujuk pada *class* dimana ia didefinisikan, maka *keyword* `parent` itu merujuk kepada *parent class*.

Keyword `parent` hanya dapat digunakan pada *child class* dan hanya digunakan untuk memanggil *method* yang ada pada *parent class*. *Keyword* ini erat kaitannya dengan pembahasan berikutnya yaitu *overloading* dan *overriding*.

Apa itu *Overloading* dan *Overriding*

Berbeda dengan konsep OOP pada Java yang memungkinkan kita mendefinisikan ulang sebuah *method* dengan nama yang sama asalkan parameternya berbeda. Di PHP kita tidak diperbolehkan untuk melakukan hal tersebut sehingga konsep *overloading* dan *overriding* dalam OOP PHP tidak dapat dipisahkan.

Selain itu karena kita tidak dapat mendefinisikan ulang sebuah *method* dengan nama yang sama, maka konsep *overloading* dan *overriding* dalam OOP PHP menjadi bergantung dengan konsep pewarisan yang telah kita bahas sebelumnya.

Overriding secara mudah dipahami sebagai pendefinisian ulang sebuah *method* milik *parent class* oleh *child class*. Biasanya pendefinisian ulang tersebut diperlukan jika kita ingin menambahkan *logic* atau memodifikasi *logic* yang ada pada *method* tersebut. Penambahan atau modifikasi *logic* pada *method* di *child class* tersebut itulah yang disebut *overloading*.

Penerapan *Overloading* dan *Overriding*

Untuk lebih memahami konsep *overloading* dan *overriding* pada PHP, perhatikan contoh berikut:

```php
<?php

class PostRepository
{
    public function getLatestPost()
    {
        $posts = [
            [
                'id' => 1,
                'title' => 'Judul Pertama',
                'content' => 'Contoh Content Pertama',
            ],
            [
                'id' => 2,
                'title' => 'Judul Kedua',
                'content' => 'Contoh Content Kedua',
            ],
            [
                'id' => 3,
                'title' => 'Judul Ketiga',
                'content' => 'Contoh Content Ketiga',
            ],
        ];

        return $posts;
    }
}

class SuffledPostRepository extends PostRepository
{
    public function getLatestPost()
    {
        $posts = parent::getLatestPost();
```

```
            shuffle($posts);

            return $posts;
        }
    }

    $postRepository = new PostRepository();
    print_r($postRepository->getLatestPost());

    $suffledPostRepository = new SuffledPostRepository();
    print_r($suffledPostRepository->getLatestPost());
```

Pada contoh diatas kita mempunyai dua *class* yaitu `PostRepository` dan `SuffledPostRepository`, dimana *class* `SuffledPostRepository` adalah turunan dari *class* `PostRepository`. Pada *class* `SuffledPostRepository` kita meng-*override method* `getLatestPost()` dan menambahkan *logic* yaitu mengacak *index* dari `$posts`. Penambahan *logic* inilah yang disebut dengan *overloading*.

```
$posts = parent::getLatestPost();
```

Pada baris *code* diatas, kita memanggil *method* `getLatestPost()` milik *class* `PostRepository` sebagaimana telah saya jelaskan pada pembahasan penggunaan *keyword* `parent`. *Keyword* `parent` kita gunakan karena paham dan tahu dengan pasti bahwa *logic* pada *parent class* sudah sesuai dengan apa yang kita inginkan sehingga kita tidak perlu menulis ulang semua *code* tersebut. Kita hanya perlu sedikit perubahan untuk membuatnya sesuai dengan kebutuhan.

Untuk melihat *output* dari program diatas, Anda dapat membuka *link* berikut https://3v4l.org/R1UsN.

Agar semkain memahami tentang *override* dan *overload*, perhatikan contoh dibawah ini:

```
<?php

class Connection
{
    public function connect($database, $username, $password,
 $host = 'localhost')
    {
        throw new RuntimeException('Anda harus mengimplement
asikan method connect() sesuai dengan database driver yang A
nda gunakan.');
    }
}

class MySQLConnection extends Connection
{
    public function connect($database, $username, $password,
 $host = 'localhost')
    {
        /**
         * Ceritanya ini logic koneksi ke database MySQL
         *
         * Anda tidak bisa menggunakan _keyword_ `parent` un
tuk memanggil _method_ `connect()` milik _parent class_
         * karena akan mengakibatkan error.
         **/
    }
}

class PostgreSQLConnection extends Connection
{
    public function connect($database, $username, $password,
```

```
 $host = 'localhost')
    {
        /**
         * Ceritanya ini logic koneksi ke database PostgreSQ
L
         *
         * Anda tidak bisa menggunakan _keyword_ `parent` un
tuk memanggil _method_ `connect()` milik _parent class_
         * karena akan mengakibatkan error.
         **/
    }
}
```

Berbeda dengan contoh sebelumnya dimana kita dapat menggunakan *keyword* `parent` untuk mengambil *logic* pada *parent class*, pada contoh diatas, *parent class* memaksa kita untuk membuat *logic* sendiri sesuai dengan *database driver* yang kita gunakan. Ini terjadi karena saya men-*trigger exception* menggunakan *keyword* `throw` (pembahasan tentang *exception* dan *keyword* `throw` akan dibahas pada bab terpisah) sehingga akan memicu *error* ketika *method* `connect()` pada *class* `Connection` dipanggil.

Bagaimana sudah dapat dipahami bagaimana *override* dan *overload* bekerja? Untuk memaksa *child class* mengimplementasikan *logic*-nya sendiri seperti cara diatas sebenarnya tidak benar karena pada OOP ada namanya konsep *abstract* yang akan dibahas setelah pembahasan *override* dan *overload* ini.

XIV. *Abstract Class* dan *Abstract Method*

Pada bab ini akan dibahas secara lebih mendalam tentang apa itu *abstract class*, apa itu *abstract method* dan bagaimana cara penerapannya menggunakan bahasa pemrograman PHP.

Apa itu *Abstract Class*

Abstract class adalah sebuah *class* dalam OOP yang tidak dapat diinstansiasi atau dibuat *object*-nya. *Abstract class* biasanya berisi fitur-fitur dari sebuah *class* yang belum implementasikan. Seperti pada pembahasan sebelumnya tentang *class* `Connection` dimana kita harus membuat implementasi dari *class* tersebut dengan meng-*extends*-nya menjadi `MySQLConnection` dan `PostgreSQLConnection`. Karena *abstract class* harus diimplementasikan melalui proses pewarisan, maka dalam *abstract class* berlaku aturan-aturan yang ada pada konsep pewarisan yang telah kita bahas sebelumnya.

Didalam sebuah *abstract class* kita dapat membuat *property* dan *method* yang nantinya dapat digunakan oleh *child class*. Tentu saja *property* dan *method* yang dapat digunakan oleh *child class* adalah *property* dan *method* yang memiliki visibilitas *protected* dan *public*.

Apakah kita dapat mendefinisikan *property* dan *method* dengan visibilitas *private* pada *abstract class*? Tentu saja bisa, dan tidak ada yang melarangnya. Biasanya *property* atau *method* pada *abstract class* dibuat *private* karena kita tidak ingin *child class* dapat mengakses *property* atau *method* tersebut atau kita membuat cara lain untuk mengakses *private property*.

Apa itu *Abstract Method*

Sama seperti *abstract class*, *abstract method* adalah sebuah *method* yang harus diimplementasikan oleh *child class*. *Abstract method* hanya ada pada *abstract class* dan *interface* (akan dibahas secara terpisah).

Bila biasanya setiap *method* yang kita buat pasti mempunyai kurang kurawal `{}`, pada *abstract method* hal tersebut tidak dapat ditemui karena *abtract method* adalah sebuah *method* yang tidak memiliki *body* atau badan *method*.

Pada *child class*, *abstract method* harus didefinisikan ulang dan kita tidak dapat menggunakan *keyword* `parent` untuk memanggil *abstract method* pada *parent class*. Bila kita melakukan hal tersebut maka akan terjadi *error*.

Kegunaan *Abstract Class* dan *Abstract Method*

Saya banyak mendapat pertanyaan baik melalui *WhatsApp* maupun pesan di *Facebook* tentang apa *sih* kegunaan *abstract class* dan *abstract method*? Ada juga yang bertanya tentang kenapa sebuah *class* atau *method* harus dibuat menjadi *abstract*? Pembahasan kali ini saya dedikasikan untuk menjawab pertanyaan-pertanyaan tersebut, walaupun pastinya tidak mungkin dapat memuaskan semua orang yang butuh penjelasan tentang itu semua.

Secara mudah *abstract class* dan *abstract method* berguna untuk memastikan *child class* memiliki fitur-fitur yang telah ditentukan sebelumnya. *Abstract class* akan sangat berguna pada saat kita membahas tentang *type hinting* atau parameter *hinting*. Dengan *abstract class* dan *abstract method* kita bisa lebih percaya diri ketika memanggil sebuah *method* karena dapat dipastikan *method* tersebut dimiliki *child class*.

Contoh Penggunaan *Abstract Class* dan *Abstract Method*

Untuk mendefinisikan sebuah *class* atau *method* menjadi *abstract* kita menggunakan *keyword* `abstract` didepan *class* atau *method* tersebut. Untuk lebih jelasnya, mari kita ubah *class* `Connection` yang sebelumnya telah kita buat dengan cara yang tidak benar. Perhatikan contoh berikut:

```
<?php

abstract class Connection
```

```
{
    abstract public function connect($database, $username, $
password, $host = 'localhost');
}

class MySQLConnection extends Connection
{
    public function connect($database, $username, $password,
 $host = 'localhost')
    {
        /**
         * Ceritanya ini logic koneksi ke database MySQL
         *
         * Anda tidak bisa menggunakan _keyword_ `parent` un
tuk memanggil _method_ `connect()` milik _parent class_
         * karena akan mengakibatkan error.
         **/
    }
}

class PostgreSQLConnection extends Connection
{
    public function connect($database, $username, $password,
 $host = 'localhost')
    {
        /**
         * Ceritanya ini logic koneksi ke database PostgreSQ
L
         *
         * Anda tidak bisa menggunakan _keyword_ `parent` un
tuk memanggil _method_ `connect()` milik _parent class_
         * karena akan mengakibatkan error.
         **/
    }
}
```

Dengan cara demikian, maka *class* `MySQLConnection` dan `PostgreSQLConnection` dipaksa untuk mendefinisikan ulang *method* `connect()` sama seperti tujuan utama kita membuat *class* `Connection` pada bab sebelumnya. Bila kita tidak mengimplementasikan *method* `connect()` maka akan terjadi *error*. Perhatikan contoh berikut:

```
<?php

abstract class Connection
{
    abstract public function connect($database, $username, $
password, $host = 'localhost');
}

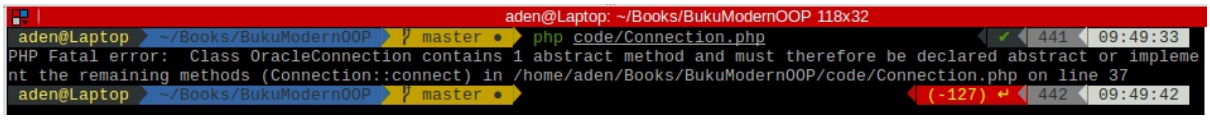
class MySQLConnection extends Connection
{
    public function connect($database, $username, $password,
 $host = 'localhost')
    {
        /**
         * Ceritanya ini logic koneksi ke database MySQL
         *
         * Anda tidak bisa menggunakan _keyword_ `parent` un
tuk memanggil _method_ `connect()` milik _parent class_
         * karena akan mengakibatkan error.
         **/
    }
}

class PostgreSQLConnection extends Connection
{
    public function connect($database, $username, $password,
 $host = 'localhost')
    {
        /**
```

```
            * Ceritanya ini logic koneksi ke database PostgreSQ
L
            *
            * Anda tidak bisa menggunakan _keyword_ `parent` un
tuk memanggil _method_ `connect()` milik _parent class_
            * karena akan mengakibatkan error.
            **/
        }
}

class OracleConnection extends Connection
{
}
```

Pada contoh diatas, kita membuat sebuah *child class* `OracleConnection` yang *extends* dari *class* `Connection` tanpa mengimplementasikan *method* `connect()`. Bila program diatas dijalankan maka akan terjadi *error* `PHP Fatal error: Class OracleConnection contains 1 abstract method and must therefore be declared abstract or implement the remaining methods (Connection::connect)` seperti gambar dibawah ini.



Perlu Anda ketahui bahwa didalam *abstract class* kita juga dapat membuat *non abstract method* sebagai berikut:

```
<?php

abstract class AbstractTaxCalculator
{
    private $previous;

    abstract public function maxPtkp();
```

```
    abstract public function minPtkp();

    abstract public function taxPercentage();

    public function getPrevious()
    {
        return $this->previous;
    }

    public function setPrevious($taxCalculator)
    {
        $this->previous = $taxCalculator;
    }

    public function calculate($ptkp)
    {
        $previousValue = 0;
        if ($previous = $this->getPrevious()) {
            $previousValue = $this->getPrevious()->calculate
($previous->maxPtkp());
            $ptkp -= $previous->maxPtkp();
        }

        return ($this->taxPercentage() * $ptkp) + $previousV
alue;
    }

    public function isSupportPtkp($ptkp)
    {
        if ($ptkp < $this->maxPtkp() && $ptkp >= $this->minP
tkp()) {

            return true;
        }

        return false;
    }
```

```
}
```

Bagaimana apakah sudah lebih bisa memahami apa itu *abstract class* dan *abtract method*? Contoh diatas hanyalah sebuah ilustrasi dari sebuah *abstract class* yang memiliki *abstract method* dan *non abstract method*.

Pada pembahasan-pembahasan selanjutnya, kita akan semakin memahami kegunaan *abstract class* dan *abtract method* secara lebih mendalam dengan melihat langsung penggunaan *abstract class* dan *abstract method* dalam *real project*.

XV. *Interface*

Bab ini akan menerangkan tentang apa itu *interface* dalam konsep OOP serta bagaimana cara penggunaannya. Selain itu juga pada bab ini akan membahas tentang *trait*, sebuah fitur yang dapat membuat *code* yang kita buat semakin *reusable*, serta bagaimana *best practice* dalam penggunaan *interface* dan *trait*.

Apa itu *Interface*

Apa yang Anda bayangkan jika mendengar kata *interface*? Mungkin yang terbayang adalah sebuah halaman web yang lengkap dengan biasa Anda sebut dengan istilah *interface* atau *user interface*. Tapi dalam pemrograman berbasis objek, konsep *interface* yang dimaksud sangat berbeda jauh dengan *interface* yang Anda bayangkan tersebut.

Dalam pemrograman berbasis objek, *interface* adalah sebuah *class* yang semua *method*-nya adalah *abstract method*. Karena semua *method*-nya adalah *abstract method* maka *interface* pun harus diimplementasikan oleh *child class* seperti halnya pada *abstract class*. Hanya saja bila kita sebelumnya menggunakan *keyword* `extends` untuk mengimplementasikan sebuah *abstract class*, maka pada *interface* kita menggunakan *keyword* `implements` untuk mengimplementasikan sebuah *interface*.

Contoh Penggunaan *Interface*

Untuk lebih memahami bagaimana sebuah *interface* bekerja, coba perhatikan contoh berikut:

```php
<?php

interface HewanInterface
{
    public function getJenis();
}

class Kambing implements HewanInterface
{
    public function getJenis()
    {
        return 'Herbivora';
    }
}

class Harimau implements HewanInterface
{
    public function getJenis()
    {
        return 'Karnivora';
    }
}

class Singa implements HewanInterface
{
    public function getJenis()
    {
        return 'Karnivora';
    }
}
```

Pada contoh diatas, kita memiliki *interface* `HewanInterface` dan mempunyai 1 *abstract method* yaitu `getJenis()` serta memiliki 3 implementasi yaitu `Kambing` , `Harimau` dan `Singa` . Setiap *class* yang mengimplementasikan *interface* `HewanInterface` harus membuat implementasi dari *method* `getJenis()` seperti tampak pada contoh diatas.

Selain itu juga kita dapat menggabungkan antara *interface* dan *abstract class* sebagaimana tampak pada contoh dibawah ini:

```
<?php

interface HewanInterface
{
    public function getJenis();
}

abstract class Hewan
{
    private $jenis;

    public function setJenis($jenis)
    {
        $this->jenis = $jenis;
    }

    public function getJenis()
    {
        return $this->jenis;
    }
}

class Kambing extends Hewan implements HewanInterface
{
}
```

```
class Harimau extends Hewan implements HewanInterface
{
}

class Singa extends Hewan implements HewanInterface
{
}

$kambing = new Kambing();
$kambing->setJenis('Herbivora');

$harimau = new Harimau();
$harimau->setJenis('Karnivora');

$singa = new singa();
$singa->setJenis('Karnivora');

echo $kambing->getJenis();
echo PHP_EOL;
echo $harimau->getJenis();
echo PHP_EOL;
echo $singa->getJenis();
echo PHP_EOL;
```

Dengan membuat *code* seperti diatas, *child class* kita tidak perlu mengimplementasikan *abstract method* milik *interface* `HewanInterface` karena sudah diimplementasikan oleh *abstract class* `Hewan`. Dengan cara tersebut, maka *code* kita semakin rapi, mudah dibaca, dan *reusable*. Bila *code* diatas dijalankan, maka *output*-nya adalah sebagai berikut:

```
php Hewan.php

Output:
Herbivora
```

```
Karnivora
Karnivora
```

Mungkin sampai disini Anda belum terlalu paham kenapa kita harus membuat *interface* sementara kita bisa menggunakan *abstract class* karena keduanya terlihat sama saja. Bahkan disatu sisi, *abstract class* jauh lebih menjanjikan karena kita masih bisa membuat *logic*, sementara pada *interface* karena semuanya adalah *abstract method* secara otomatis kita tidak diperkenankan membuat *logic* pada *interface*.

Pada prateknya, *interface* dan *abstract class* memiliki fungsi yang berbeda. *Interface* berfungsi untuk memastikan *child class* memiliki fitur-fitur yang telah ditetapkan dalam *interface*, sementara *abstract class* berguna untuk memberikan fitur-fitur dasar pada *child class* dimana fitur-fitur tersebut dapat digunakan secara bersama-sama oleh *child class*.

Seperti contoh diatas, dimana *inteface* `HewanInterface` memiliki fitur `getJenis()`, sementara *abstract class* `Hewan` memberikan fitur `setJenis()` dan `getJenis()` untuk dapat dipakai secara bersama-sama oleh *child class* `Singa`, `Harimau`, dan `Kambing`.

Tidak seperti pada pewarisan, pada *interface* kita dapat menggunakan *multiple interface* sekaligus seperti pada contoh berikut:

```php
<?php

interface HewanInterface
{
    public function getJenis();
```

```
}

interface MamaliaInterface
{
    public function menyusui();
}

abstract class Hewan
{
    private $jenis;

    public function setJenis($jenis)
    {
        $this->jenis = $jenis;
    }

    public function getJenis()
    {
        return $this->jenis;
    }
}

class Kambing extends Hewan implements HewanInterface, Mamal
iaInterface
{
    public function menyusui()
    {
        echo 'nyusu kambing';
    }
}

class Harimau extends Hewan implements HewanInterface, Mamal
iaInterface
{
    public function menyusui()
    {
        echo 'nyusu maung';
    }
```

```
}

class Singa extends Hewan implements HewanInterface, Mamalia
Interface
{
    public function menyusui()
    {
        echo 'nyusu singa';
    }
}
```

Membuka Wawasan

Di era milenial seperti sekarang ini penggunaan *interface* sangat masif. Banyak *framework* dan *library* yang kalau kita mau membaca *source code*-nya maka akan mudah sekali bagi kita untuk menemukan *interface*.

Penggunaan *interface* tidak lain karena fitur yang dimiliki *interface* itu sendiri yaitu sebagai hirarki tertinggi pada *parameter casting* (akan dibahas pada bab tersendiri) dimana setiap *object* yang mengimplementasikan sebuah *interface* akan valid jika dimasukkan kedalam *method* yang menggunakan *interface* tersebut sebagai *type hinting* atau *parameter casting*.

Seperti pada *framework* Laravel, dimana *interface* akan sangat mudah ditemukan pada *folder* `Contracts` seperti nampak pada Github *repository* Laravel berikut.

Pada paradigma pemrograman modern, ada istilah "*interface as contract*" yang maksudnya adalah *interface* digunakan pada *parameter casting* sebagai pengikat bahwa *object* yang akan

dimasukkan kedalam *method* pasti memiliki fitur-fitur atau *method-method* yang didefinisikan pada *interface* tersebut.

Sehingga dengan menggunakan *interface* tersebut sebagai *paramter casting* pada *method* maka didalam *method* tersebut kita bisa dengan percaya diri untuk menggunakan *method-method* yang ada pada *interface* tanpa takut terjadi *error* `undefined method`.

XVI. *Method Chaining*

Tak terasa sudah 15 bab kita bahas dari awal buku ini hingga sekarang. Pada bab kali ini kita akan sedikit *refreshing* dan membahas sebuah konsep ringan dari OOP yang banyak diterapkan pada *framework* yaitu *method chaining*.

Apa itu *Method Chaining*

Seperti yang sudah dijelaskan diatas, *Method Chaining* adalah salah satu konsep yang ada pada pemrograman berbasis objek. *Method Chaining* memungkinkan kita memanggil *method* secara berantai dalam satu baris sekaligus. Fungsi *method chaining* adalah mempersingkat pemanggilan *method* yang tadinya kita perlu memanggil *method* dalam beberapa baris, dengan *method chaining* kita dapat melakukannya dengan hanya satu baris *code*. Perhatikan ilustrasi berikut:

```
$mutator = new StringMutator('Muhamad Surya Iksanudin');
$mutator->bold();
$mutator->underline();
$mutator->italic();
$mutator->strike();
```

Dengan *method chaining* kita dapat membuatnya menjadi lebih singkat seperti pada ilustrasi berikut:

```
$mutator = new StringMutator('Muhamad Surya Iksanudin');
$mutator->bold()->underline()->italic()->strike();
```

Bagaimana terlihat lebih simpel bukan? *Method chaining* ini juga umum digunakan pada *framework* salah satunya adalah CodeIgniter yang menggunakan metode *method chaining* pada *library query builder* miliknya.

```
$this->db->select('*')->from('user')->where('id = 1')->get()
;
```

Familiar kan dengan *syntax* diatas? Ingin tau bagaimana cara membuatnya? Pembahasan selanjutnya akan menjelaskan secara men-*detail* bagaimana cara membuat *method chaining* dan bagaimana sebuah *method* dapat dirangkai seperti itu.

Cara Pembuatan *Method Chaining*

Untuk membuat sebuah *method* dapat dipanggil secara berantai seperti pada contoh diatas, kita perlu memahami kembali konsep *return value* dan penggunaan dari *keyword* `$this`. Keduanya diperlukan karena untuk membuat *method chaining* kita perlu mengembalikan *object* itu sendiri kepada pemanggil. Perhatikan contoh berikut:

```
<?php

class StringMutator
{
```

```
    private $word;

    public function __construct($word)
    {
        $this->word = $word;
    }

    public function bold()
    {
        $this->word = sprintf('<b>%s</b>', $this->word);

        return $this;
    }

    public function underline()
    {
        $this->word = sprintf('<u>%s</u>', $this->word);

        return $this;
    }

    public function italic()
    {
        $this->word = sprintf('<i>%s</i>', $this->word);

        return $this;
    }

    public function strike()
    {
        $this->word = sprintf('<strike>%s</strike>', $this->
word);

        return $this;
    }

    public function __toString()
    {
```

```
        return $this->word;
    }
}
```

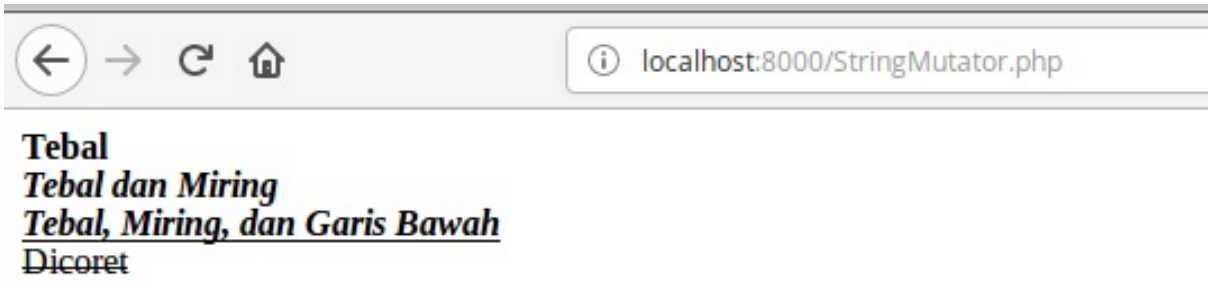
Pada contoh diatas, setiap *method* dalam *class* mengembalikan *object* itu sendiri yang ditandai dengan `return $this`. Dengan mengembalikan *object* itu sendiri maka kita bisa langsung memanggil *method* yang ada pada *object* tersebut secara langsung dari *method* yang kita panggil sebelumnya. Kita bisa memanggil *method* `bold()` setelah memanggil *method* `italic()` secara berurutan seperti berikut:

```
$mutator->italic()->bold();
```

Pada *class* `StringMutator` diatas saya menggunakan *magic method* `__toString()` yaitu sebuah *method* yang akan dieksekusi secara otomatis ketika kita mencoba mencetak *object* dengan *keyword* `echo` misalnya sehingga *code* kita menjadi semakin singkat lagi seperti pada contoh dibawah ini:

```
echo (new StringMutator('Tebal'))->bold();
echo '<br/>';
echo (new StringMutator('Tebal dan Miring'))->bold()->italic
();
echo '<br/>';
echo (new StringMutator('Tebal, Miring, dan Garis Bawah'))->
bold()->italic()->underline();
echo '<br/>';
echo (new StringMutator('Dicoret'))->strike();
echo '<br/>';
```

Bila program diatas dijalankan dan dibuka melalui *browser* maka *output*-nya akan nampak sebagai berikut:



XVII. Pengelompokan Berkas

Bab ini akan membahas tentang bagaimana cara mengelompokkan berkas pada OOP PHP agar aplikasi yang kita bangun lebih terorganisir dan lebih mudah dimaintain.

Pengelompokan Berkas pada OOP PHP

Ketika membangun sebuah aplikasi apalagi aplikasi tersebut tergolong besar, mengelompokkan aplikasi berdasarkan modul adalah salah satu cara mengelompokkan berkas. Pengelompokkan tersebut biasanya berdasarkan fungsionalitas dari sebuah modul dalam sebuah aplikasi.

Dalam paradigma pemrograman berbasis objek, pengelompokkan berkas berdasarkan fungsionalitasnya dimanakan *packaging* atau pemaketan. *Packinging* biasanya dilakukan dengan mengelompokkan beberapa *class* yang masih satu fungsionalitas menjadi satu *folder*.

Namun akan menjadi masalah jika ternyata didalam *folder* yang berlainan terdapat *class* yang memiliki nama yang sama. Misalnya kita punya dua *folder* yaitu `Http` dan `Api` dimana pada masing-masing *folder* tersebut terdapat *class* `Request` yang mengimplementasikan *interface* `RequestInterface` yang berfungsi untuk menangani *request* pada masing-masing fungsionalitas.

Susunan *folder* tersebut nampak sebagai berikut:



Dan berikut adalah isi dari masing-masing file:

```php
<?php
// filename: RequestInterface.php

interface RequestInterface
{
    public function handle();
}
```

```php
<?php
// filename: Http/Request.php

class Request implements RequestInterface
{
    public function handle()
    {
        echo 'Handle Http Request';
    }
}
```

```php
<?php
// filename: Api/Request.php

class Request implements RequestInterface
```

```
{
    public function handle()
    {
        echo 'Handle Api Request';
    }
}
```

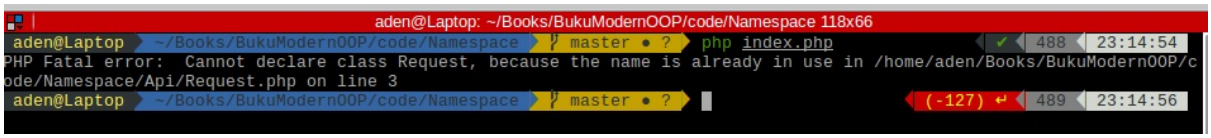
Bila kita membuat *file* pemanggil semisal `index.php` dengan isi sebagai berikut:

```
<?php

require __DIR__.'/RequestInterface.php';
require __DIR__.'/Http/Request.php';
require __DIR__.'/Api/Request.php';


$request = new Request();
$request->handle();
echo PHP_EOL;
```

Maka ketika kita menjalankan *file* `index.php` tersebut akan terjadi *error* karena PHP menganggap bahwa *class* `Request` telah didefinisikan sebelumnya. Pesan *error* tersebut adalah `PHP Fatal error: Cannot declare class Request, because the name is already in use` seperti pada gambar berikut:



Untuk mengatasi hal tersebut, PHP mempunyai sebuah fitur yang disebut *namespacing*. Fitur ini akan dibahas lebih lanjut pada bahasan berikutnya.

Keyword `namespace` dan `use`

Konsep *namespacing* sebenarnya sangat erat kaitannya dengan kehidupan kita. Konsep ini biasa dipakai oleh pengembang perumahan untuk mengelompokkan rumah berdasarkan lokasinya. Pengelompokan tersebut biasanya didasarkan pada lorong atau jalan yang ada diperumahan tersebut yaitu kemudian disebut blok.

Blok inilah yang disebut *namespacing* dimana setiap blok memiliki beberapa rumah yang letaknya sama yaitu masih dalam satu lorong. Rumah-rumah itu kemudian diberikan nomer yang sama dengan nomer rumah yang ada pada blok lain semisal pada blok A terdapat nomer rumah 1, kemudian pada blok B juga ada nomer rumah 1. Walaupun memiliki nomer rumah yang sama, namun tidak akan terjadi salah alamat, karena walaupun nomer rumahnya sama tapi lokasi bloknya berbeda.

Penggelompokkan tersebut diimplementasikan oleh PHP dalam bentuk *namespace* dimana dalam *namespace* yang berbeda kita dapat mendefinisikan nama *class* yang sama. Untuk mendefinisikan suatu *namespace* kita menggunakan *keyword* `namespace` dan untuk menggunakan suatu *namespace* kita menggunakan *keyword* `use`.

Contoh Penggunaan *Keyword* `namespace` dan `use`

Seperti yang sudah dijelaskan diatas, *keyword* `namespace` dan `use` digunakan untuk mengemlompokkan *class*. Untuk memahami bagaimana penggunaan *keyword* tersebut, mari kita kembali ke contoh sebelumnya dan memberikan *namespace* pada *class* `Request` sebagai berikut:

```
<?php
// filename: Http/Request.php

namespace Http;

use RequestInterface;

class Request implements RequestInterface
{
    public function handle()
    {
        echo 'Handle Http Request';
    }
}
```

```
<?php
// filename: Api/Request.php

namespace Api;

use RequestInterface;

class Request implements RequestInterface
{
    public function handle()
    {
        echo 'Handle Api Request';
    }
}
```

Kita perlu menggunakan `use` ketika memanggil *interface* `RequestInterface` karena jika tidak maka *interface* `RequestInterface` akan dianggap bagian dari *namespace* tersebut. Hal ini akan mengakibatkan *error* karena *interface* `RequestInterface` tidak memiliki atau bukan bagian dari *namespace* tersebut.

Kemudian kita juga harus mengubah *file* `index.php` untuk menggunakan *class* `Request` sesuai dengan *namespace* yang telah kita buat.

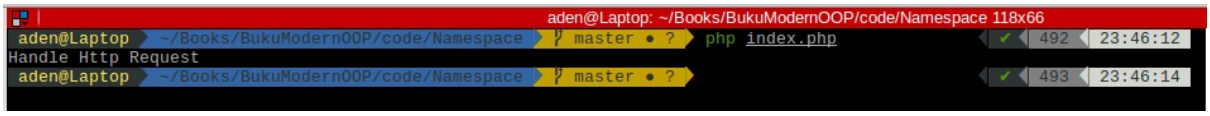
```
<?php

require __DIR__.'/RequestInterface.php';
require __DIR__.'/Http/Request.php';
require __DIR__.'/Api/Request.php';

use Http\Request;

$request = new Request();
$request->handle();
echo PHP_EOL;
```

Jika program diatas dieksekusi maka *output*-nya adalah `Handle Http Request` seperti pada gambar berikut:



Selanjutnya, mari kita mencoba memanggil juga *class* `Api\Request` (penyebutan nama *class* bersamaan dengan nama *namespace*-nya disebut juga dengan FQCN atau *Fully Qualified*

Class Name) pada `index.php` sehingga program kita akan berubah menjadi sebagai berikut:

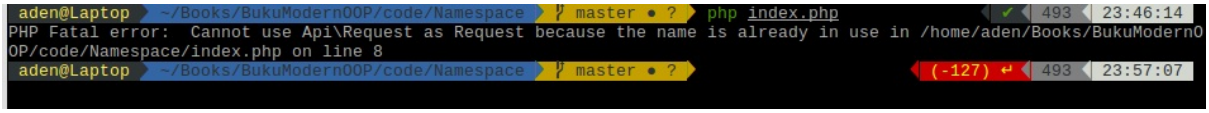
```
<?php

require __DIR__.'/RequestInterface.php';
require __DIR__.'/Http/Request.php';
require __DIR__.'/Api/Request.php';

use Http\Request;
use Api\Request;

$request = new Request();
$request->handle();
echo PHP_EOL;
$apiRequest = new Request();
$apiRequest->handle();
echo PHP_EOL;
```

Bila kita menjalakan program diatas ternyata justru terjadi *error* `PHP Fatal error: Cannot use Api\Request as Request because the name is already in use` seperti gambar dibawah:



Untuk mengatasi *error* tersebut, kita dapat memanggil *class* menggunakan FQCN tanpa menggunakan *keyword* `use` sehingga *file* `index.php` akan jadi seperti berikut:

```
<?php

require __DIR__.'/RequestInterface.php';
require __DIR__.'/Http/Request.php';
require __DIR__.'/Api/Request.php';
```

```
$request = new Http\Request();
$request->handle();
echo PHP_EOL;
$apiRequest = new Api\Request();
$apiRequest->handle();
echo PHP_EOL;
```

Dan bila kita jalankan kembali *file* `index.php` maka *output*-nya adalah sebagai berikut:

```
php index.php


Output:
Handle Http Request
Handle Api Request
```

Pada banyak kasus kita perlu menggunakan *class* lain yang masih satu *namespace*, untuk menggunakan *class* lain yang masih satu *namespace* kita tidak perlu menggunakan *keyword* `use` dan dapat langsung menggunakannya seperti pada contoh berikut:

```
<?php
// filename: Http/OtherClass.php

namespace Http;

class OtherClass
{
    private $request;

    public function __construct()
    {
```

```
            $this->request = new Request();
        }
    }
```

Pada contoh diatas, *class* `Request` otomatis akan merujuk pada *class* `Http\Request` sebab *class* `Http\Request` memiliki *namespace* yang sama dengan *class* `OtherCass` yaitu `Http`.

Memberikan alias dengan *keyword* `as`

Meski kita dapat memanggil *class* menggunakan FQCN untuk mengatasi *error* ketika menggunakan `use` dengan nama akhir *class* yang sama seperti pada contoh diatas. Namun cara tersebut kurang elegan bahkan akan membuat *code* yang kita buat tampak jelek bila ternyata *class* yang akan kita gunakan memiliki *namespace* yang panjang.

Cara yang terbaik untuk mengatasi *error* tersebut diatas adalah dengan memberikan alias pada *class* yang kita gunakan menggunakan *keyword* `as`. Sehingga *code* pada *file* `index.php` akan menjadi seperti berikut:

```
<?php

require __DIR__.'/RequestInterface.php';
require __DIR__.'/Http/Request.php';
require __DIR__.'/Api/Request.php';

use Http\Request as HttpRequest;
use Api\Request as ApiRequest;
```

```
$request = new HttpRequest();
$request->handle();
echo PHP_EOL;
$apiRequest = new ApiRequest();
$apiRequest->handle();
echo PHP_EOL;
```

Bagaimana tampak lebih simpel dan lebih nyaman dibaca bukan?

XVIII. Parameter *Casting* dan *Return Type Declaration*

Setelah membaca bab ini diharapkan Anda dapat memahami apa itu parameter *casting* dan apa itu *return type declaration* serta bagaimana cara menggunakannya pada kasus nyata.

Apa itu Parameter *Casting*

Parameter *casting* atau yang biasa disebut juga dengan *type hinting* adalah sebuah cara untuk memastikan bahwa parameter yang dimasukkan kedalam sebuah *method* sesuai dengan tipe data yang telah kita tentukan. Dengan kata lain, *type hinting* adalah cara kita memaksa pemanggil *method* untuk memasukkan parameter yang tepat dan sesuai.

Dari sisi *developer*, *type hinting* memberikan kepercayaan diri karena kita dapat memastikan dan menentukan secara spesifik tipe data yang bisa masuk kedalam *method*. Hal ini akan mengurangi kemungkinan *error* atau *bug* yang terjadi yang disebabkan oleh kesalahan tipe data baik yang terjadi karena kesalahan pengolahan maupun kesalahan karena data yang salah yang berasal dari *database*.

Hingga PHP versi 5.6, PHP hanya mengenal *type hinting* untuk tipe data `array`, `callable`, dan `object` (termasuk didalamnya `interface`). Baru pada PHP versi 7, PHP menambahkan lebih

banyak *type hinting*, yaitu dengan menambahkan dukungan untuk tipe data `string`, `int`, `float` dan `bool`. Penambahan tersebut semakin menegaskan komitmen PHP untuk menjadi bahasa pemrograman yang *type safety*.

Scalar Type Hinting

Scalar type atau yang juga disebut sebagai *primitive type* adalah tipe data yang hanya memuat satu data dalam dirinya. Contoh dari tipe data *scalar* adalah `string`, `int`, `float`, dan `bool`. Namun dalam contoh nanti saya juga akan memasukkan `array` karena menurut saya `array` dan tipe data *scalar* pada *type hinting* tidak jauh berbeda sehingga akan lebih mudah untuk membahasnya sekaligus.

Untuk mendefinisikan *type casting* atau *type hinting* pada sebuah *method* kita menambahkan tipe data atau *cast type* sebelum pendefinisian *variable* parameter. Perhatikan contoh berikut:

```
<?php

class ParameterCasting
{
    public function __construct(array $arrayType)
    {
    }

    public function stringCast(string $stringType)
    {
    }

    public function intCast(int $intType)
```

```
        {
        }

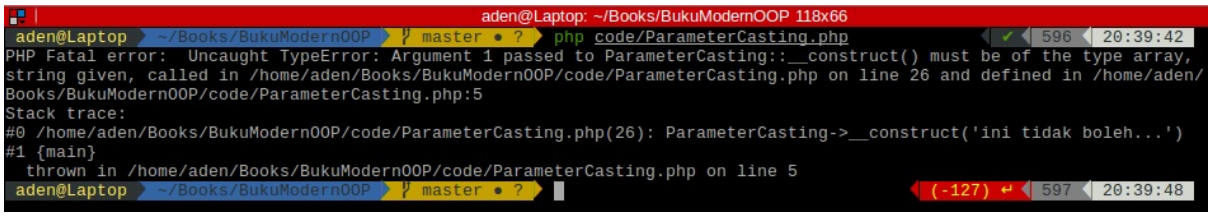
        public function floatCast(float $floatType)
        {
        }

        public function booleanCast(bool $booleanType)
        {
        }
    }
```

Pada contoh diatas, ketika kita membuat *object* dari *class* `ParameterCasting` maka kita wajib memasukkan parameter berupa `array` bila kita tidak memasukkan parameter pasti akan terjadi *error* seperti yang sudah kita sama-sama pahami. Namun jika kita memasukkan parameter selain tipe data `array`, karena kita sudah melakukan *casting* bahwa yang boleh dimasukkan hanya tipe data `array`, maka juga akan mengakibatkan *error*. Perhatikan contoh dibawah ini:

```
$object = new ParameterCasting('ini tidak boleh masuk');
```

Bila program diatas dijalankan, maka akan terjadi *error* `PHP Fatal error: Uncaught TypeError: Argument 1 passed to ParameterCasting::__construct() must be of the type array, string given` seperti pada gambar dibawah:



Namun bila kita menggantinya dengan tipe data `array` seperti pada contoh dibawah ini:

```
$object = new ParameterCasting(array());
```

Maka tidak akan terjadi *error* karena kita memasukkan parameter sesuai dengan spesifikasi yang sudah kita buat sebelumnya. Begitu juga dengan *method* yang lain, jika kita salah memasukkan parameter maka akan terjadi *error* seperti sebelumnya.

Khusus untuk tipe data `float` secara *default* kita bisa memasukkan tipe data `int` ataupun `float` (*numeric*) dan ini berlaku juga sebaliknya untuk `int` karena PHP menganggap keduanya adalah sama-sama *numeric*. Kita dapat membuat PHP membedakan tipe data `float` dan `int` secara spesifik dengan menggunakan *strict mode* yaitu dengan menambahkan baris *code* `declare(strict_types=1)` diatas sebuah *file* yang inginkan. Perhatikan contoh berikut:

```
<?php

class ParameterCasting
{
    public function __construct(array $arrayType)
    {
    }

    public function stringCast(string $stringType)
    {
    }

    public function intCast(int $intType)
    {
```

```
    }

    public function floatCast(float $floatType)
    {
    }

    public function booleanCast(bool $booleanType)
    {
    }
}

$object = new ParameterCasting(array());
$object->floatCast(1);
$object->intCast(1.0);
```

Pada contoh diatas tidak akan terjadi *error* seperti yang sudah saya jelaskan sebelumnya walaupun kita memasukkan `int` kedalam *method* `floatCast()` yang di-*casting* hanya menerima `float`, begitu juga pada *method* `intCast()` yang di-*casting* hanya menerima `int` tetapi kita tetap bisa memasukkan tipe data `float` kedalamnya.

Untuk membuat keduanya secara spesifik hanya menerima tipe data sesuai dengan *casting*-nya, maka kita tambahkan `declare(strict_types=1)` dibagian paling atas sebagai berikut:

```
<?php

declare(strict_types=1);

class ParameterCasting
{
    public function __construct(array $arrayType)
    {
    }
```

```
    public function stringCast(string $stringType)
    {
    }

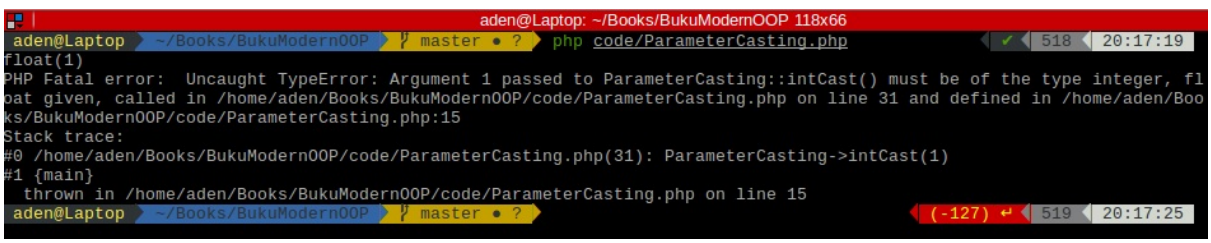
    public function intCast(int $intType)
    {
    }

    public function floatCast(float $floatType)
    {
        var_dump($floatType);
    }

    public function booleanCast(bool $booleanType)
    {
    }
}

$object = new ParameterCasting(array());
$object->floatCast(1);
$object->intCast(1.0);
```

Dan bila program diatas dijalankan kembali, maka akan terjadi *error* `PHP Fatal error: Uncaught TypeError: Argument 1 passed to ParameterCasting::intCast() must be of the type integer, float given` karena kita mencoba memasukkan tipe data `float` kedalam method `intCast()`, sedangkan `int 1` yang kita masukkan kedalam *method* `floatCast()` secara otomatis diubah oleh PHP menjadi tipe *floating point* seperti pada gambar dibawah:



Object Type Hinting

Sama seperti pada *scalar type hinting*, *object type hinting* mempunyai fungsi untuk membatasi nilai parameter yang dimasukkan kedalam sebuah *method*. Bedanya adalah yang sekarang kita *casting* parameter menggunakan nama *class* atau *interface*.

Pada *object type hinting* saya tidak akan menunjukkan bagaimana jika yang di-*passing* kedalam *method* tipe datanya tidak sesuai karena sudah dijelaskan pada pembahasan sebelumnya. Pada pembahasan kali ini kita akan fokus kepada hirarki pada *object type hinting*. Perhatikan contoh berikut:

```
<?php

class Post
{
    private $title;

    private $content;

    public function __construct(string $title, string $conte
nt)
    {
        $this->title = $title;
        $this->content = $content;
    }

    public function getTitle()
    {
        return $this->title;
    }
```

```
    public function getContent()
    {
        return $this->content;
    }
}
```

```
<?php

namespace Mutator;

class StringMutator
{
    public static function bold(string $word)
    {
        return sprintf('<b>%s</b>', $word);
    }

    public static function underline(string $word)
    {
        return sprintf('<u>%s</u>', $word);
    }

    public static function italic(string $word)
    {
        return sprintf('<i>%s</i>', $word);
    }

    public static function strike(string $word)
    {
        return sprintf('<strike>%s</strike>', $word);
    }
}
```

```
<?php
```

```
namespace Mutator;

use Post;

class PostMutator
{
    private $post;

    public function __construct(Post $post)
    {
        $this->post = $post;
    }

    public function getBoldTitle()
    {
        return StringMutator::bold($this->post->getTitle());
    }

    public function getItalicContent()
    {
        return StringMutator::italic($this->post->getContent
());
    }
}
```

Pada contoh diatas terdapat 3 *class* yaitu `Post`, `Mutator\PostMutator`, dan `Mutator\StringMutator`. Dimana *class* `Post` menjadi *type hinting* pada *constructor class* `Mutator\PostMutator` sedangkan *class* `Mutator\StringMutator` adalah *class* utilitas yang dibutuhkan oleh *class* `Mutator\PostMutator` untuk memanipulasi isi dari *class* `Post`. Untuk lebih jelas bagaimana cara menggunakannya, perhatikan contoh *file* pemanggil dibawah ini:

```
<?php
```

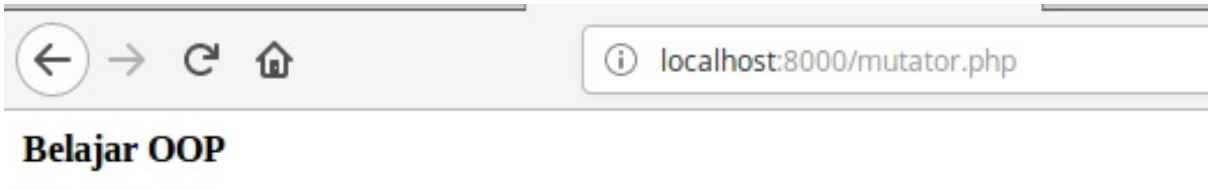
```
require __DIR__.'/Post.php';
require __DIR__.'/Mutator/PostMutator.php';
require __DIR__.'/Mutator/StringMutator.php';

use Mutator\PostMutator;

$post = new Post('Belajar OOP', 'Saya sedang belajar OOP PHP'
);
$mutator = new PostMutator($post);

echo $mutator->getBoldTitle();
```

Bila program diatas dijalankan di *browser* maka *output*-nya tampak sebagai berikut:



Perlu Anda pahami bahwa pada *object type hinting* berlaku hirarki dimana semakin tinggi maka suatu hirarki maka *object* yang dapat dimasukkan kedalam sebuah *method* akan semakin banyak dan begitu sebaliknya, jika semakin kebawah maka semakin spesifik juga *object* yang dapat dimasukkan kedalam sebuah *method*.

Susunan hirarki tersebut dari atas kebawah adalah *interface*, *abstract class*, *parent class*, dan *child class* sesuai dengan hirarki pada pewarisan. Sehingga bila kita menggunakan *interface* sebagai *type hinting* pada sebuah *method* maka semua *class* yang mengimplementasikan *interface* tersebut akan valid jika

dimasukkan atau di-*passing* didalam *method* tersebut. Hal ini berlaku juga pada *abstract class* dan seterusnya. Perhatikan contoh berikut:

```
<?php

interface PostInterface
{
    public function getTitle();

    public function getContent();
}
```

```
<?php

class Post implements PostInterface
{
    private $title;

    private $content;

    public function __construct(string $title, string $conte
nt)
    {
        $this->title = $title;
        $this->content = $content;
    }

    public function getTitle()
    {
        return $this->title;
    }

    public function getContent()
    {
        return $this->content;
```

```
    }
}
```

```
<?php

namespace Mutator;

use PostInterface;

class PostMutator
{
    private $post;

    public function __construct(PostInterface $post)
    {
        $this->post = $post;
    }

    public function getBoldTitle()
    {
        return StringMutator::bold($this->post->getTitle());
    }

    public function getItalicContent()
    {
        return StringMutator::italic($this->post->getContent
());
    }
}
```

Pada contoh diatas kita menambahkan *interface* `PostInterface` dan membuat *class* `Post` mengimplementasikan *interface* tersebut. Kemudian kita juga mengubah *type hinting* pada

constructor class `Mutator\PostMutator` menjadi menggunakan *interface* `PostInterface`. Setelah itu kita ubah *file* pemanggil menjadi sebagai berikut:

```
<?php

require __DIR__.'/PostInterface.php';
require __DIR__.'/Post.php';
require __DIR__.'/Mutator/PostMutator.php';
require __DIR__.'/Mutator/StringMutator.php';

use Mutator\PostMutator;

$post = new Post('Belajar OOP', 'Saya sedang belajar OOP PHP'
);
$mutator = new PostMutator($post);

echo $mutator->getBoldTitle();
```

Bila kita buka program diatas menggunakan *browser* maka hasilnya akan sama dengan program sebelumnya, bukan? Ini menunjukkan bahwa dengan menggunakan *interface* sebagai *type hinting* maka *class* yang mengimplementasikan *interface* dianggap sebagai parameter yang valid.

Mari kita coba lagi dengan membuat *class* `PostItalicTitle` yaitu sebuah *class* yang mengimplementasikan *interface* `PostInterface` juga untuk membuktikan bahwa hirarki parameter yang telah saya jelaskan diatas benar-benar bekerja dengan baik.

```
<?php

class PostItalicTitle implements PostInterface
```

```
{
    private $title;

    private $content;

    public function __construct(string $title, string $conte
nt)
    {
        $this->title = $title;
        $this->content = $content;
    }

    public function getTitle()
    {
        return sprintf('<i>%s</i>', $this->title);
    }

    public function getContent()
    {
        return $this->content;
    }
}
```

Kemudian kita ubah *file* pemanggil menjadi seperti berikut:

```
<?php

require __DIR__.'/PostInterface.php';
require __DIR__.'/Post.php';
require __DIR__.'/PostItalicTitle.php';
require __DIR__.'/Mutator/PostMutator.php';
require __DIR__.'/Mutator/StringMutator.php';

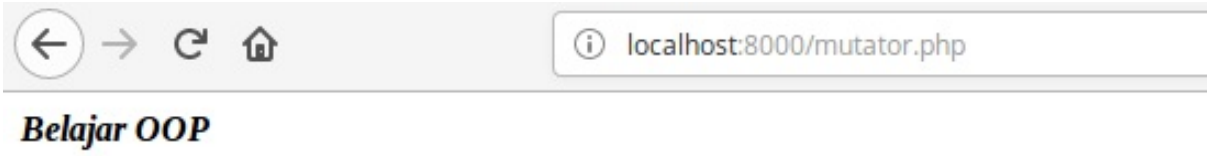
use Mutator\PostMutator;

$post = new PostItalicTitle('Belajar OOP', 'Saya sedang bela
jar OOP PHP');
```

```
$mutator = new PostMutator($post);

echo $mutator->getBoldTitle();
```

Bila kita buka program diatas menggunakan *browser* maka *output*-nya akan tampak seperti berikut:



Kita juga bisa mengubah *class* `PostItalicTitle` menjadi *child class* dari *class* `Post` seperti berikut:

```
<?php

class PostItalicTitle extends Post
{
    public function getTitle()
    {
        return sprintf('<i>%s</i>', parent::getTItle());
    }
}
```

Kemudian kita ubah *file* pemanggil menjadi seperti berikut:

```
<?php

require __DIR__.'/PostInterface.php';
require __DIR__.'/Post.php';
require __DIR__.'/PostItalicTitle.php';
require __DIR__.'/Mutator/PostMutator.php';
require __DIR__.'/Mutator/StringMutator.php';
```

```
use Mutator\PostMutator;

$post = new PostItalicTitle('Belajar OOP', 'Saya sedang bela
jar OOP PHP');
$mutator = new PostMutator($post);

echo $mutator->getBoldTitle();
```

Bila kita buka program diatas menggunakan *browser* maka hasilnya akan sama dengan program sebelumnya.

Nullable Type Hinting

Nullable type hinting adalah fitur baru yang ditambahkan pada PHP versi 7.1. *Nullable type hinting* memungkinkan kita memasukkan parameter `null` pada *parameter casting*. Untuk menggunakan fitur ini kita hanya cukup menambahkan tanda tanya `?` didepan *type hinting*. Perhatikan contoh berikut:

```
<?php

namespace Mutator;

class StringMutator
{
    public static function bold(?string $word)
    {
        return sprintf('<b>%s</b>', $word);
    }
}
```

Pada contoh ditas, kita dapat memasukkan nilai `string` atau `null` kedalam *method* `bold()`. Hal ini juga berlaku pada semua tipe data termasuk tipe data *object* (*interface*, *abstract class*, *parent class*, dan *child class*). Dengan *nullable hinting* maka kita dapat memanggil *method* `bold()` dengan `StringMutator::bold(null)` tanpa takut terjadi *error*.

Namun bila kita memanggil *method* tersebut tanpa memasukkan parameter `StringMutator::bold()` tetap akan terjadi *error* karena *method* `bold()` memerlukan parameter walaupun itu `null`.

Apa itu *Return Type Declaration*

Sama seperti *parameter casting*, *return type declaration* atau biasa disebut juga *return hinting* berfungsi untuk memastikan nilai yang dikembalikan oleh sebuah *method* telah sesuai dengan tipe data yang diinginkan. Pada *return hinting* berlaku semua aturan yang ada pada *type hinting* sehingga saya tidak perlu menjelaskan ulang permasalahan tersebut. Pada *return hinting* juga berlaku *nullable hinting* seperti pada *type hinting*.

Cara Penggunaan *Return Type Declaration*

Untuk mendeklarasikan *return type* kita menggunakan tanda titik dua `:` setelah tanda kurung `()` dan sebelum tanda kurung kurawal `{}`. Untuk lebih jelasnya, mari kita modifikasi contoh yang sudah ada diatas seperti dibawah ini:

```php
<?php

interface PostInterface
{
    public function getTitle(): string;

    public function getContent(): string;
}
```

```php
<?php

class Post implements PostInterface
{
    private $title;

    private $content;

    public function __construct(string $title, string $conte
nt)
    {
        $this->title = $title;
        $this->content = $content;
    }

    public function getTitle(): string
    {
        return $this->title;
    }

    public function getContent(): string
    {
        return $this->content;
    }
}
```

```php
<?php

namespace Mutator;

class StringMutator
{
    public static function bold(string $word): string
    {
        return sprintf('<b>%s</b>', $word);
    }

    public static function underline(string $word): string
    {
        return sprintf('<u>%s</u>', $word);
    }

    public static function italic(string $word): string
    {
        return sprintf('<i>%s</i>', $word);
    }

    public static function strike(string $word): string
    {
        return sprintf('<strike>%s</strike>', $word);
    }
}
```

```php
<?php

namespace Mutator;

use PostInterface;

class PostMutator
{
    private $post;
```

```
    public function __construct(PostInterface $post)
    {
        $this->post = $post;
    }

    public function getBoldTitle(): string
    {
        return StringMutator::bold($this->post->getTitle());
    }

    public function getItalicContent(): string
    {
        return StringMutator::italic($this->post->getContent
());
    }

    public function getOriginalPost(): PostInterface
    {
        return $this->post;
    }
}
```

Pada contoh diatas kita memberikan *return hinting* pada semua *class* yang sebelumnya kita gunakan. Kemudian saya menambahkan *method* baru pada *class* `PostMutator` yaitu *method* `getOriginalPost()` yang mengembalikan *object* dari *interface* `PostInterface` untuk memberikan contoh bagaimana menggunakan *object* sebagai *return hinting*.

Untuk mendefinisikan *nullable return hinting* kita menggunakan tanda tanya `?` dibelakang tanda titik dua `:` dan sebelum tipe data. Perhatikan contoh berikut:

```
<?php
```

```php
namespace Mutator;

use PostInterface;

class PostMutator
{
    private $post;

    public function __construct(?PostInterface $post)
    {
        $this->post = $post;
    }

    public function getOriginalPost(): ? PostInterface
    {
        return $this->post;
    }
}
```

Selain itu, kita juga dapat memastikan sebuah *method* tidak mengembalikan apapun atau biasa disebut *void* dengan menggunakan *keyword* `void` pada *return type hinting* seperti pada contoh dibawah ini:

```php
<?php

namespace Mutator;

use PostInterface;

class PostMutator
{
    private $post;

    public function __construct(?PostInterface $post)
```

```
    {
        $this->post = $post;
    }

    public function getOriginalPost(): ? PostInterface
    {
        return $this->post;
    }

    public function noReturnValue(): void
    {
    }
}
```

Bagaimana apakah penjelasan diatas dapat dipahami? Kalau belum silahkan langsung dicoba agar lebih paham bagaimana semua itu bekerja.

XIX. *Recursive Function*

Bagaimana apakah kepala terasa hampir mendidih setelah membaca penjelasan tentang *parameter casting* dan *return type hinting* pada bab sebelumnya? Kalau iya, mari kita *refreshing* sejenak dengan membahas sesuatu yang ringan yaitu tentang *recursive function*.

Apa itu *Recursive Function*

Recursive function adalah sebuah *function* yang memanggil dirinya sendiri dalam badan *function*-nya. *Recursive function* biasanya dipakai untuk menyelesaikan permasalahan yang mempunyai pola dasar yang berulang seperti perhitungan faktorial.

Keuntungan menggunakan *recursive function* adalah mempersingkat *code* yang kita tulis. Namun yang perlu diperhatikan adalah bahwa kita harus benar-benar paham bagaimana *function* tersebut bekerja. Jika kita tidak paham bagaimana *nested call* yang terjadi didalam *recursive function* bisa saja bukan solusi singkat yang didapat tapi justru permasalah yang justru kita sama sekali tidak mengetahui bagaimana cara mengatasinya.

Recursive function sangat perlu dipelajari dan dipahami oleh *programmer* karena dalam banyak kasus *recursive function* terbukti mampu menyelesaikan permasalahan yang kompleks dan dinamis

dengan lebih mudah.

Contoh Penggunaan *Recursive Function*

Sebelum membuat *recursive function* mari kita terlebih dahulu memahami tentang faktorial bilangan. Faktorial secara mudah dipahami sebagai perkalian beruntun dari `n` nilai hingga `n = 1` dimana `n` adalah bilangan positif. Secara matematika, faktorial dilambangkan dengan `n!` .

Dan berikut adalah contoh faktorial bilangan dari 1 hingga 7:

```
1! = 1
2! = 2 X 1
3! = 3 X 2 X 1
4! = 4 X 3 X 2 X 1
5! = 5 X 4 X 3 X 2 X 1
6! = 6 X 5 X 4 X 3 X 2 X 1
7! = 7 X 6 X 5 X 4 X 3 X 2 X 1
```

Bagaimana membuat program untuk menghitung faktorial tersebut dengan dan tanpa *recursive function* mari kita coba buat. Pertama kali kita buat program menghitung faktorial tanpa *recursive function* terlebih dahulu. Perhatikan contoh berikut:

```php
<?php

class Faktorial
{
    public static function nonRecursive(int $number)
```

```
    {
        $result = 1;
        for ($i = 1; $i <= $number; $i++) {
            $result *= $i;
        }

        return $result;
    }
}

echo Faktorial::nonRecursive(4);
echo PHP_EOL;
```

Bila kita jalankan program diatas, maka hasilnya adalah `24` karena `4 X 3 X 2 X 1 = 24` seperti berikut:

```
php Faktorial.php

Output:
24
```

Cara diatas adalah cara menghintung faktorial tanpa menggunakan *recursive function*, lalu bagaimana cara menghitung faktorial bilangan menggunakan *recursive function*? Perhatikan contoh dibawah ini:

```
<?php

class Faktorial
{
    public static function nonRecursive(int $number)
    {
        $result = 1;
        for ($i = 1; $i <= $number; $i++) {
```

```
            $result *= $i;
        }

        return $result;
    }

    public static function recursive(int $number)
    {
        if (2 > $number) {
            return $number;
        }

        return $number * self::recursive($number - 1);
    }
}

echo Faktorial::nonRecursive(5);
echo PHP_EOL;
echo Faktorial::recursive(5);
echo PHP_EOL;
```

Pada baris `$number * self::recursive($number - 1)` inilah yang disebut *nested call* pada *recursive function*. Jika program diatas dijalankan maka *output*-nya adalah sebagai berikut:

```
php Faktorial.php

Output:
120
120
```

Selain untuk faktorial, kita juga dapat menggunakan *recursive function* untuk menyelesaikan deret *fibonacci* yang urutannya adalah sebagai berikut:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
 987, ...
```

Bila ingin mengetahui bagaimana rumus bilangan *fibonacci* maka silahkan klik langsung halaman wikipedia ini.

Untuk mengetahui nilai deret *fibonacci* pada `n` *index* maka kita dapat membuatnya program sebagai berikut:

```
<?php

class Fibonacci
{
    public static function calculate(int $limit)
    {
        if (0 === $limit || 1 === $limit) {
            return $limit;
        }

        if (2 === $limit) {
            return 1;
        }

        return self::calculate($limit - 1) + self::calculate
($limit - 2);
    }
}

// index dimulai dari 0
// 0, 1, 1, 2, 3, 5, 8
echo Fibonacci::calculate(6);
echo PHP_EOL;
```

Jika program diatas dijalankan maka *output*-nya adalah sebagai berikut:

```
php Fibonacci.php


Output:
8
```

Sepanjang pengalaman saya melamar pekerjaan, pengetahuan tentang *recursive function* sering kali ditanyakan dan dijadikan bahan *test*. Semoga dengan adanya bab ini dapat menjadikan Anda mampu memahami dan mengaplikasikan *recursive function* dalam kasus nyata.

Yang terpenting ketika membuat *recursive function* adalah Anda harus memastikan bahwa *function* tersebut akan berhenti. Jika Anda tidak dapat memastikan hal tersebut, maka yang terjadi program Anda akan terus berputar sampai *memory* habis.

XX. *Late Static Bindings*

Pada bab ini kita akan membahas tentang apa itu *late static binding* dan cara penggunaannya pada bahasa pemrograman PHP.

Apa itu *Late Static Bindings*

Late static bindings adalah salah satu fitur PHP yang bekerja hanya pada *static inheritance* dimana *child class* meng-*override* *static function* milik *parent class*. Secara bahasa yang sederhana *late static bindings* adalah memasukkan *method static* ke dalam *method static* yang lainnya dimana salah satu dari *method static*-nya di-*override* melalui konsep pewarisan.

Late static bindings bekerja dengan menyimpan nama *class* pada akhir *"non-forwading call"*. Hal ini terjadi karena penggunaan *keyword* `self` hanya mereferensi terhadap *class* dimana *keyword* tersebut dideklarasikan. Artinya jika kita memanggil *method* pada *parent class* melalui *child class* maka *keyword* `self` tidak bekerja sebagaimana yang kita harapkan.

Contoh *Late Static Bindings*

Untuk lebih memahami bagaimana *late static bindings* bekerja, perhatikan contoh dibawah ini:

```php
<?php

class StringMutator
{
    public static function bold(string $word): string
    {
        return sprintf('<b>%s</b>', $word);
    }

    public static function italic(string $word): string
    {
        return sprintf('<i>%s</i>', $word);
    }

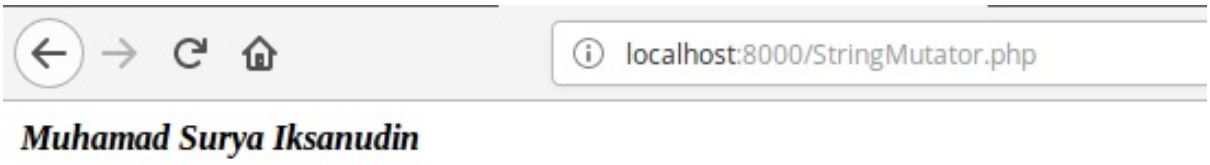
    public static function boldItalic(string $word): string
    {
        return self::italic(self::bold($word));
    }
}

class ChildStringMutator extends StringMutator
{
    public static function bold(string $word): string
    {
        return '<b>STATIC LATE BINDINGS</b>';
    }
}

echo ChildStringMutator::boldItalic('Muhamad Surya Iksanudin'
);
```

Bila melihat contoh diatas, seharusnya ketika kita memanggil `ChildStringMutator::boldItalic('Muhamad Surya Iksanudin')` maka *output*-nya adalah `<i><b>STATIC LATE BINDINGS</b></i>` karena

kita telah meng-*override* *method* `bold()` pada *class* `ChildStringMutator`. Namun bila kita menjalankan program diatas dan membuka *browser* maka yang muncul adalah sebagai berikut:



Atau yang kalau kita tulis dalam *plain text* adalah `<i><b>Muhamad Surya Iksanudin</b></i>` tidak sesuai dengan harapan kita. Ini terjadi karena *method* `italic` dan `bold()` ketika dipanggil pada *method* `boldItalic()` menggunakan referensi `self` sehingga *method* `italic()` dan terutama `bold()` karena *method* tersebut yang kita *override*, hanya merujuk pada *class* `StringMutator`.

Untuk mengatasi hal tersebut, PHP menyediakan *keyword* `static` yang memiliki fitur *late static bindings* sebagaimana yang sedang kita bahas sekarang. Perhatikan contoh berikut:

```
<?php

class StringMutator
{
    public static function bold(string $word): string
    {
        return sprintf('<b>%s</b>', $word);
    }

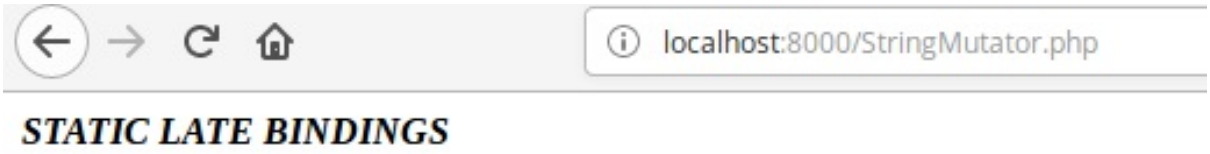
    public static function italic(string $word): string
    {
        return sprintf('<i>%s</i>', $word);
    }
```

```
    public static function boldItalic(string $word): string
    {
        return static::italic(static::bold($word));
    }
}

class ChildStringMutator extends StringMutator
{
    public static function bold(string $word): string
    {
        return '<b>STATIC LATE BINDINGS</b>';
    }
}

echo ChildStringMutator::boldItalic('Muhamad Surya Iksanudin'
);
```

Dan bila kita buka program diatas menggunakan *browser* maka hasilnya adalah sebagai berikut:



Bagaimana sudah sesuai dengan espektasi kita *kan*? Setelah mengetahui bagaimana *late static bindings* bekerja, apakah Anda akan tetap menggunakan *keyword* `self` untuk memanggil *static method* ataukah akan beralih menggunakan *keyword* `static`? Pilihan ada ditangan Anda.

XXI. *Trait*

Pada pembahasan kali ini kita akan mempelajari tentang fitur yang ada pada OOP PHP yaitu *trait*. Selain mempelajari apa itu *trait* dan cara penggunaannya, dalam bab ini juga akan dibahas tentang *best practice* ketika menggunakan fitur *trait* tersebut.

Apa itu *Trait*

Seperti yang sudah dibahas sebelumnya bahwa dalam OOP PHP tidak mengenal *multiple inheritance* sehingga sebuah *class* dipaksa hanya boleh 1 *parent class*. Meski kita dapat menggunakan fitur *nested inheritance*, namun dalam beberapa kasus fitur tersebut belum dapat mengakomodir kebutuhan kita.

Untuk mengatasi hal tersebut, PHP memperkenalkan fitur baru yang disebut *trait*. *Trait* adalah sebuah mekanisme dalam PHP yang memungkinkan kita menggunakan (*reuse*) *code* pada sebuah *trait* seperti halnya pada pewarisan. Namun tidak seperti pada pewarisan yang hanya mewariskan *protected* dan *public* saja, pada *trait* semua *code* dapat diwariskan.

Dalam bahasa yang sangat gampang, *trait* adalah *copy paste code* yang dilalukan melalui bahasa pemrograman. Sehingga bila sebuah *class* menggunakan *trait* maka *interpreter* PHP akan mem-*paste*-kan *code* pada *trait* tersebut ke *class* yang menggunakannya.

Trait juga dapat diibaratkan seperti potongan *puzzle* yang dapat dipasangkan ke *class* apapun. Anda dapat menggunakannya pada *abstract class* maupun *class* biasa (bukan *abstract class*).

Trait adalah kebalikan dari *interface* dimana setiap *method* dalam sebuah *trait* harus berupa *non abstract method*.

Penggunaan *Trait*

Untuk membuat sebuah *trait* kita menggunakan *keyword* `trait` diikuti nama *trait* tersebut. Perhatikan contoh berikut:

```
<?php

trait StringMutator
{
    public function bold(string $word): string
    {
        return sprintf('<b>%s</b>', $word);
    }

    public function italic(string $word): string
    {
        return sprintf('<i>%s</i>', $word);
    }

    public function boldItalic(string $word): string
    {
        return $this->italic($this->bold($word));
    }
}
```

Dan untuk menggunakan *trait* kita memakai *keyword* `use` seperti halnya pada pemanggilan *class* pada *namespace*. Bedanya, kalau pemanggilan *class* pada *namespace* posisi *keyword* `use` berada diluar dari *block class*, maka pada *trait* posisi *keyword* `use` berada di dalam *block class*. Perhatikan contoh berikut:

```
<?php

class Post
{
    private $title;

    private $content;

    public function __construct(string $title, string $conte
nt)
    {
        $this->title = $title;
        $this->content = $content;
    }

    public function getTitle(): string
    {
        return $this->title;
    }

    public function getContent(): string
    {
        return $this->content;
    }
}

trait StringMutator
{
    public function bold(string $word): string
    {
```

```
        return sprintf('<b>%s</b>', $word);
    }

    public function italic(string $word): string
    {
        return sprintf('<i>%s</i>', $word);
    }

    public function boldItalic(string $word): string
    {
        return $this->italic($this->bold($word));
    }
}

class PostMutator
{
    use StringMutator;//Pemanggilan _trait_

    private $post;

    public function __construct(Post $post)
    {
        $this->post = $post;
    }

    public function boldTitle(): string
    {
        return $this->bold($this->post->getTitle());
    }
}

$postMutator = new PostMutator(new Post('Judul', 'Ini contoh
 content dari berita.'));
echo $postMutator->boldTitle();
```

Pada contoh diatas, *class* `PostMutator` secara otomatis akan mewarisi semua fitur atau *method* yang ada pada *trait* `StringMutator` sehingga pada *class* `PostMutator` kita dapat memanggil *method* `bold()` milik `StringMutator`. Untuk melihat hasilnya, silahkan Anda buka sendiri program diatas melalui *browser* karena saya sedang malas mendemokan buat Anda, *hehehe*.

Dalam satu *class* kita dapat menggunakan lebih dari satu *trait* seperti pada contoh berikut:

```
<?php

class Post
{
    private $title;

    private $content;

    public function __construct(string $title, string $conte
nt)
    {
        $this->title = $title;
        $this->content = $content;
    }

    public function getTitle(): string
    {
        return $this->title;
    }

    public function getContent(): string
    {
        return $this->content;
    }
```

```
}

trait BoldText
{
    public function bold(string $word): string
    {
        return sprintf('<b>%s</b>', $word);
    }
}

trait ItalicText
{
    public function italic(string $word): string
    {
        return sprintf('<i>%s</i>', $word);
    }
}

class PostMutator
{
    use BoldText;
    use ItalicText;

    private $post;

    public function __construct(Post $post)
    {
        $this->post = $post;
    }

    public function boldItalicTitle(): string
    {
        return $this->italic($this->bold($this->post->getTit
le()));
    }
}

$postMutator = new PostMutator(new Post('Judul', 'Ini contoh
```

```
 content dari berita.'));
echo $postMutator->boldItalicTitle();
```

Meski dalam halaman dokumentasi resminya, *trait* memiliki banyak sekali fitur, namun pada prakteknya jarang sekali *programmer* yang menggunakan fitur-fitur tersebut selain daripada fitur-fitur yang umum. Hal ini untuk mengurangi kompleksitas program dan agar program tersebut lebih mudah dibaca dan di-*trace*.

Best Practice pada *Trait*

Seperti yang sudah dijelaskan diatas, *trait* adalah sebuah fitur yang sangat *powerful* dalam konsep OOP PHP. *Saking powerful*-nya, fitur ini dapat membuat Anda dalam masalah besar (kesulitan melakukan *tracing code*) jika Anda tidak bijak dalam menggunakannya. Penggunaan *trait* secara berlebihan akan mengakibatkan kompleksitas program menjadi meningkat dan membuat kita semakin sulit dalam melakukan *tracing*.

Untuk menghindari hal tersebut, ada baiknya Anda memperhatikan hal-hal dibawah ini:

- Jangan membuat *nested trait*.

Pembuatan *nested trait*, dimana sebuah *trait* menggunakan *trait* lainnya, hanya akan membuat Anda semakin sulit dalam melakukan *tracing code*.

- Gunakan *trait* seminimal mungkin.

Meski sangat *powerful* tapi *trait* itu seperti micin. Kalau terlalu banyak akan bikin *bego* kalau kata anak milenial.

- Gunakan *trait* sebagai aktifator sebuah *interface*.

Bila diperlukan, buatlah *trait* yang berisi semua *method* yang ada pada *interface* khususnya pada *interface* yang sering digunakan. Untuk lebih jelas perhatikan contoh *real* berikut:

```
<?php
/*
 * This file is part of the Symfony package.
 *
 * (c) Fabien Potencier <fabien@symfony.com>
 *
 * For the full copyright and license information, please vi
ew the LICENSE
 * file that was distributed with this source code.
 */
namespace Symfony\Component\DependencyInjection;
/**
 * ContainerAwareInterface should be implemented by classes
that depends on a Container.
 *
 * @author Fabien Potencier <fabien@symfony.com>
 */
interface ContainerAwareInterface
{
    public function setContainer(ContainerInterface $contain
er = null);
}
```

```
<?php
/*
 * This file is part of the Symfony package.
```

```
 *
 * (c) Fabien Potencier <fabien@symfony.com>
 *
 * For the full copyright and license information, please vi
ew the LICENSE
 * file that was distributed with this source code.
 */
namespace Symfony\Component\DependencyInjection;
/**
 * ContainerAware trait.
 *
 * @author Fabien Potencier <fabien@symfony.com>
 */
trait ContainerAwareTrait
{
    /**
     * @var ContainerInterface
     */
    protected $container;
    public function setContainer(ContainerInterface $contain
er = null)
    {
        $this->container = $container;
    }
}
```

Sehingga bila kita membuat *class* `A` yang mengimplementasikan *interface* `Symfony\Component\DependencyInjection\ContainerAwareInterface` maka *class* `A` tersebut cukup menggunakan *trait* `Symfony\Component\DependencyInjection\ContainerAwareTrait` tanpa perlu menulis ulang *method* `setContainer()` pada *class* `A` seperti berikut:

```
<?php
```

```
use Symfony\Component\DependencyInjection\ContainerAwareInte
rface;
use Symfony\Component\DependencyInjection\ContainerAwareTrait
;

class A implements ContainerAwareInterface
{
    use ContainerAwareTrait;
}
```

Setidaknya 3 poin simpel diatas cukup untuk membuat program Anda sedikit berkurang kompleksitasnya ketika menggunakan *trait*.

XXII. *Coding Standard*

Pada pembahasan kali ini kita akan mempelajari tentang *coding standard* yang berlaku pada ekosistem PHP, apa itu PSR serta kenapa kita harus menulis *code* sesuai dengan PSR.

Apa itu FIG dan PSR

PHP *Standard Recomendations* atau yang lebih dikenal dengan PSR adalah kumpulan aturan atau *standard* yang diakui dan berlaku pada ekosistem PHP. Penetapan standarisasi tersebut bertujuan untuk memudahkan kolaborasi antar *developer* PHP diseluruh dunia.

PSR merupakan bentuk kontribusi nyata komunitas PHP untuk kemajuan PHP itu sendiri. Dengan adanya PSR, *developer* PHP diseluruh dunia bisa mempunyai *standard* dan *coding style* yang seragam dan ketika mereka membuat *project open source*, *developer* PHP lainnya dapat dengan mudah untuk ikut berkontribusi.

Salah satu manfaat dari PSR adalah lahirnya *composer* sebuah *dependency manager* yang dibuat untuk memudahkan *developer* PHP untuk melakukan instalasi, meng-*update* maupun menghapus *package* atau *library* yang dibutuhkan dalam sebuah proyek dengan mudah. Pembahasan tentang *composer* akan dibahas secara lebih spesifik pada bab tersendiri.

Penerapan PSR saat ini sudah sangat lazim dilakukan oleh *developer* terutama yang menggunakan *framework* seperti Symfony, Laravel, Slim, dan Zend. Sehingga bisa dikatakan sebagai seorang *developer* kita wajib mengetahui dan menerapkan PSR.

Kenapa harus Menerapkan PSR

Dengan mengikuti sebuah *standard* tertentu secara langsung maupun tidak langsung akan memberikan manfaat kepada kita, begitu pula ketika kita mengikuti *standard* yang ada pada ekosistem PHP yaitu PSR. Dengan mengikuti PSR kita akan mendapatkan manfaat antara lain:

> - *Code* program menjadi lebih rapi dan terstruktur
> - *Code* yang kita buat menjadi semakin *reusable* karena dapat digunakan oleh *developer* lain yang menerapkan PSR dengan mudah.
> - Mudah mengganti suatu fungsional atau *library* dengan fungsional atau *library* lainnya karena mempunyai *standard* yang sama.
> - Memudahkan kita berkolaborasi dengan *developer* lainnya.

Aturan Penulisan *Syntax*

Aturan penulisan *syntax* pada PSR diatur dalam PSR-1 dan kemudian disempurnakan dengan PSR-2. Dalam rekomendasi PSR, aturan menulis *syntax* adalah sebagai berikut:

> - Hanya boleh menggunakan *tag* `<?php >` dan `<?= >` dan tidak boleh menggunakan variasi *tag* yang lain.
> - Penulisan *namespace*, dan *class* harus mengikuti aturan *autoload* pada PSR-0 dan PSR-4.
> - Penulisan nama *class*, *interface* dan *trait* harus menggunakan `StudlyCaps` yaitu mirip seperti `camelCase` hanya saja huruf awal harus besar.
> - *Constanta* harus ditulis dalam huruf kapital dan menggunakan *underscore* (`_`) sebagai pemisah kata.
> - Penulisan nama *method* harus menggunakan `camelCase` .
> - Penulisan nama *property* tidak atur secara ketat, hanya saja harus konsisten.
> - *Indent* pada harus menggunakan 4 spasi dan bukan *tab*.

Dan masih banyak lagi, silahkan baca sendiri atau gunakan *Google Translate* bila Anda kesulitan menerjemahkannya.

Contoh Penerapan Aturan *Syntax*

Sebenarnya dari awal bila Anda memperhatikan, contoh-contoh yang kita tulis sebenarnya telah menerapkan aturan-aturan yagn ada pada PSR-1 maupun PSR-2 diatas seperti pada contoh

berikut:

```
<?php
// filename: Api/Request.php

namespace Api;

use RequestInterface;

class Request implements RequestInterface
{
    public function handle()
    {
        echo 'Handle Api Request';
    }
}
```

Menggunakan *PHP CS Fixer*

Banyaknya aturan dalam PSR-1 dan PSR-2 membuat kita kadang tanpa sadar melanggar atau tidak menerapkannya. Dan bila kita seorang *programmer* yang ketat dalam menerapkan aturan, hal itu tentu perlu ditangani. Namun untuk mengetahui apakah *code* yang kita tulis telah sesuai dengan *standard* atau belum adalah sesuatu yang sulit jika proyek yang kita bangun cukup besar dan *code* yang telah kita tulis sudah cukup banyak.

Beruntunglah di komunitas PHP ada orang yang *aware* dengan hal tersebut dan mau membuatkan *tool* untuk mengatasi hal tersebut. Adalah Fabien Potencier dan Dariusz Ruminski, dua *developer* ini mengembangkan sebuah *tool* yang diberinama PHP *CS Fixer* atau

PHP *Coding Standard Fixer* sebuah yang dapat membetulkan *code* yang Anda tulis sesuai dengan aturan *coding standard* yang berlaku.

Untuk menggunakan *tool* tersebut, Anda dapat men-*download file phar*-nya melalui website Sensiolab dan menyimpannya pada *root* proyek yang Anda kerjakan.

Untuk menggunakannya, Anda cukup mengetikan dari *terminal* atau *command prompt* perintah berikut:

```
php php-cs-fixer.phar fix <folder> --rules=@PSR2
```

Dengan menjalankan perintah singkat diatas, maka PHP *CS Fixer* akan membetulkan *syntax code* Anda secara otomatis mengikuti aturan PSR-2 yang otomatis juga mengikut aturan PSR-1.

Untuk lebih *detail* tentang PHP *CS Fixer* serta opsi perintahnya, Anda dapat membacanya langsung pada website Sensiolab yang *link*-nya telah ditautkan diatas.

Aturan Penulisan *Class* dan *Namespace*

Pada PSR, aturan penamaan *class* dan *namespace* diatur melalui PSR-0 yang kemudian disempurnakan melalui PSR-4. Dalam rekomendasi PSR, aturan penulisan nama *class* dan *namespace* adalah sebagai berikut:

> - Sebuah *class* secara FQCN harus ditulis dengan

> `<VendorName>\(<Namespace>\)*<ClassName>`
>
> - `<VendorName>` dapat berisi nama *package* atau *library* atau nama perusahaan atau organisasi pemilik *library* tersebut.
> - Diperbolehkan pada `<Namespace>` menggunakan *nested namespace* seperti `<Namspace1>\<Namspace2>` dan seterusnya.
> - Setiap *namespace* harus dikonversi kedalam *folder* kerja.
> - Tanda *underscore* (`_`) pada `<Namespace>` tidak memiliki arti apapun.
> - Bila pada `<ClassName>` terdapat *underscore* (`_`) maka harus dikonversi kedalam *folder*.
> - `<ClassName>` harus diawali dengan huruf besar.
> - Aturan ini berlaku juga untuk *interface* dan *trait*.
> - Satu *file* hanya boleh berisi satu *class* saja.

Contoh Penerapan Aturan *Class* dan *Namespace*

Contoh penerapan PSR-0 dan PSR-4 dapat dilihat pada *class* `PostMutator` pada bab sebelumnya. Isi dari *class* `PostMutator` adalah sebagai berikut:

```
<?php
```

```
namespace Mutator;

use PostInterface;

class PostMutator
{
    private $post;

    public function __construct(PostInterface $post)
    {
        $this->post = $post;
    }

    public function getBoldTitle(): string
    {
        return StringMutator::bold($this->post->getTitle());
    }

    public function getItalicContent(): string
    {
        return StringMutator::italic($this->post->getContent
());
    }

    public function getOriginalPost(): PostInterface
    {
        return $this->post;
    }
}
```

Dengan menerapkan aturan-aturan diatas, maka secara otomatis *code* yang kita tulis akan menjadi lebih mudah dibaca dan mudah dimaintain.

XXIII. *Exception Handling*

Exception handling adalah salah satu fitur penting yang harus ada pada setiap program yang kita buat agar program yang dibuat menjadi lebih aman. Pada bab ini akan dibawah tentang apa itu *exception handling* dan penerapannya dalam bahasa pemrograman PHP.

Apa itu *Exception Handling*

Exception adalah sebuah kondisi dimana program berhenti dan tidak bisa melanjutkan eksekusi *code* atau biasa yang kita sebut dengan *error*. Kecuali karena kesalahan *syntax*, pada PHP *error* tersebut dapat kita tangkap dan kita tangani agar program yang seharusnya berhenti dapat tetap melanjutkan eksekusi atau mengirimkan pesan *error* yang *human readable* atau dapat dipahami oleh pengguna awam.

Pada PHP, ketika program terjadi *error* baik karena kesalahan memasukkan parameter, kesalahan aritmatika seperti pembagian dengan nol (`0`) atau yang lainnya, PHP akan menangkap *error* tersebut dan menganggapnya sebagai *object*. Karena dianggap sebagai *object* maka ketika *error* tersebut akan ditampilkan pada layar, maka secara otomatis akan memanggil *method* `__toString()`.

Kita dapat menangkap *object error* tersebut dan kemudian menanganinya sesuai dengan keinginan kita atau biasa disebut *exception handling*. Jadi *exception handling* adalah mekanisme untuk menangkap *error* yang terjadi pada program dan kemudian menanganinya.

Hirarki *Error* pada PHP

PHP merombak secara besar-besaran mekanisme *error reporting* pada PHP 7 dan menambahkan banyak *class* baru untuk menangani *error* sesuai dengan konteksnya. Bila pada PHP 5 kita hanya mengenal *class* `Exception` dan turunannya untuk menangani *error* pada program, maka pada PHP 7 kita harus membedakan antara *exception* dan *error* karena pada PHP 7, PHP memperkenalkan *class* baru yaitu *class* `Error`.

Secara umum, kesalahan yang dilakukan *programmer* seperti salah memasukkan parameter, pembagian dengan nol (`0`), kurang memasukkan parameter dan kesalahan-kesalahan lain yang dilakukan secara tidak sengaja oleh *programmer* masuk dalam kategori *error* sehingga akan dianggap sebagai *object* dari *class* `Error`.

Sedangkan kesalahan program yang terjadi karena disengaja oleh sang *programmer* biasanya masuk dalam kategori *exception* meskipun Anda dapat membuatnya masuk kategori *error* namun sangat tidak dianjurkan. Contoh kesalahan program yang sengaja adalah sebagai berikut:

```
<?php

class Connection
{
    public function connect()
    {
        throw new RuntimeException('Anda harus mengimplement
asikan method connect() sesuai dengan database driver yang A
nda gunakan.');
    }
}
```

Ketika kita membuat *object* `Connection` dan memanggil *method* `connect()` maka kita akan memicu *exception* yaitu `RuntimeException`.

Class `Error` dan `Exception` adalah *class* yang mengimplementasikan *interface* `Throwable` dan memiliki beberapa *child class*. Adapun hirarki *class* `Error` dan `Exception` adalah sebagai berikut:

> - Throwable
>   - Error
>   - ArithmeticError
>   - DivisionByZeroError
>   - AssertionError
>   - ParseError
>   - TypeError
>   - ArgumentCountError

> - Exception
>   - RuntimeException
>   - InvalidArgumentException
>   - ...

Karena baik berupa *class* maka kita pun dapat membuat *child class* dari *class* `Error` maupun `Exception` sesuai dengan kebutuhan kita.

Exception Handling pada PHP

Sebelum membahas tentang *exception handling* pada PHP, ada baiknya kita pahami dulu alur penanganan *error* yang dilakukan oleh PHP. Perhatikan contoh sederhana berikut:

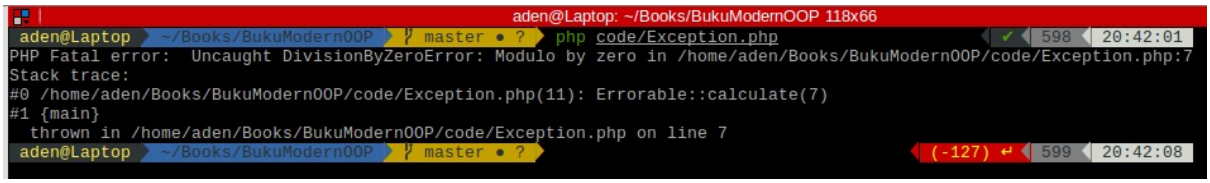
```
<?php

class Errorable
{
    public static function calculate(int $number)
    {
        return ($number % 0); //Bila menggunakan $number / 0
 maka hanya akan memunculkan _warning_ bukan _error_
    }
}

Errorable::calculate(7);

echo 'Ini tidak dieksekusi';
echo PHP_EOL;
```

Pada contoh diatas ketika kita memanggil *method* `calculate()` akan terjadi *error* disebabkan oleh pembagian dengan nol (`0`). PHP secara otomatis akan membuat *object* `DivisionByZeroError` dan kemudian mencari *block* `catch` untuk kemudian memberikan *error* tersebut agar dapat ditangani. Bila *block* `catch` tidak ditemukan, maka PHP akan mencoba memanggil *function* didefinisikan pada fungsi `set_exception_handler()` . Bila fungsi `set_exception_handler()` tidak didefinisikan, maka PHP akan mengubahnya menjadi menjadi *fatal error*, yaitu mencetak *error* pada layar.



Mari kita fokus pada kalimat, **kemudian mencari *block* `catch` untuk kemudian memberikan *error* tersebut agar dapat ditangani**. Inti dari kalimat tersebutlah yang akan coba kita lakukan, yaitu menangani *error* yang terjadi.

Untuk menangani *error* tersebut, kita harus membuat baris program yang mungkin dapat memicu *error* berada didalam *block* `try` dan kemudian bila terjadi *error* kita akan menangkapnya menggunakan *block* `catch` . Perhatikan contoh diatas:

```
<?php

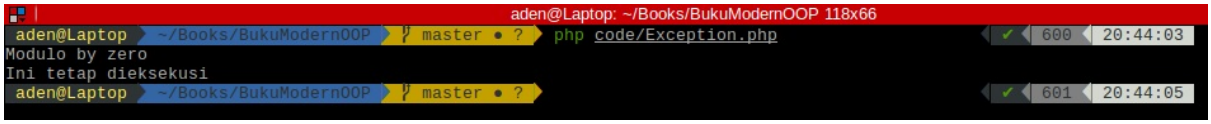
class Errorable
{
    public static function calculate(int $number)
    {
        return ($number % 0);
    }
```

```
}

try {
    Errorable::calculate(7);
} catch (DivisionByZeroError $e) {
    echo $e->getMessage();
    echo PHP_EOL;
}

echo 'Ini tetap dieksekusi';
echo PHP_EOL;
```

Ketika program diatas dijalakan maka akan mengeluarkan *output* sebagai berikut:



Dengan menggunakan *block* `try {} catch () {}` kita dapat membuat program kita tetap berjalan walaupun dalam kondisi *error* atau menampilkan pesan yang lebih manusiawi agar pada dipahami pengguna yang awam. Kita juga dapat menangkap *error* yang kita sengaja buat sama seperti yang kita lakukan diatas. Perhatikan contoh berikut:

```
<?php

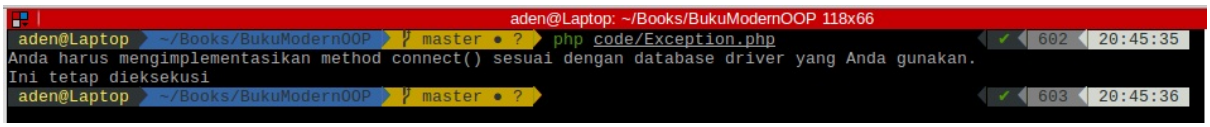
class Connection
{
    public function connect()
    {
        throw new RuntimeException('Anda harus mengimplement
asikan method connect() sesuai dengan database driver yang A
nda gunakan.');
    }
```

```
}

try {
    $connection = new Connection();
    $connection->connect();
} catch (RuntimeException $e) {
    echo $e->getMessage();
    echo PHP_EOL;
}

echo 'Ini tetap dieksekusi';
echo PHP_EOL;
```

Bila dijalakan program diatas akan mengeluarkan *output* sebagai berikut:



Selain itu kita juga dapat menggunakan *interface* `Throwable` untuk menangani `Error` dan `Exception` maupun *child class* dari keduanya sekaligus karena `Throwable` adalah *interface* dari kedua *class* tersebut.

Kita juga dapat menggunakan *multiple* `catch` untuk menangani sebuah *error* seperti pada contoh berikut:

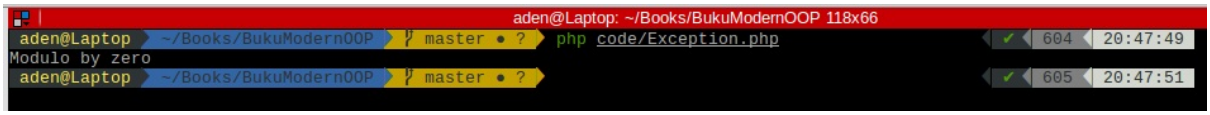
```
<?php

class Errorable
{
    public static function calculate(int $number)
    {
        return ($number % 0);
    }
```

```
}

try {
    Errorable::calculate(7);
} catch (Exception $e) {
    echo 'Tidak masuk kesini';
    echo PHP_EOL;
} catch (DivisionByZeroError $e) {
    echo $e->getMessage();
    echo PHP_EOL;
}
```

Ketika program diatas dijalakan maka akan mengeluarkan *output* sebagai berikut:



Block `catch` yang diatas tidak dieksekusi karena *block* tersebut hanya menangani *object* `Exception` sedangkan *error* yang dihasilkan oleh program adalah *object* `DivisionByZeroError` sehingga akan ditangkap oleh *block* `catch` berikutnya.

Kita juga dapat membuat sebuah program dieksekusi walaupun terjadi *error* dan tak dapat ditangani oleh *block* `catch` menggunakan *block* `finally` seperti contoh dibawah ini:

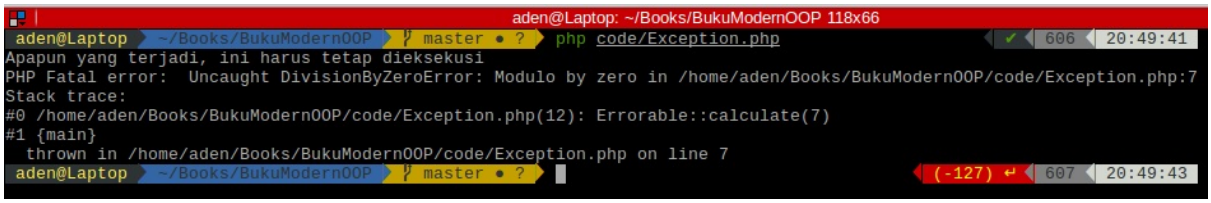
```
<?php

class Errorable
{
    public static function calculate(int $number)
    {
        return ($number % 0);
    }
```

```
}

try {
    Errorable::calculate(7);
} catch (Exception $e) {
    echo 'Tidak masuk kesini';
    echo PHP_EOL;
} finally {
    echo 'Apapun yang terjadi, ini harus tetap dieksekusi';
    echo PHP_EOL;
}
```

Bila dijalakan program diatas akan mengeluarkan *output* sebagai berikut:



Bila melihat *output* diatas, terlihat *block* `finally` lebih dahulu dieksekusi daripada *error* yang muncul. Seperti yang telah dijelaskan diatas, hal ini terjadi karena ketika *error* terjadi, PHP akan mencari *block* `catch` yang dapat menangani *error* tersebut. Bila itu tidak ditemukan maka PHP akan mengecek apakah ada *error handler* yang di-*set* pada fungsi `set_exception_handler()`. Bila tidak ada, maka PHP akan mengubahnya menjadi menjadi *fatal error*, yaitu mencetak *error* pada layar.

Karena ketika mencari *block* `catch` untuk menangani *error* tersebut tidak ada sedangkan *block* `finally` masih dalam satu *block* dengan *block* `catch`, maka PHP mengeksekusi *block* `finally` terlebih dahulu sebelum mencari *error handler*.

XXIV. *Anonymous Function* dan *Anonymous Class*

Pada pembahasan kali ini kita akan memahami tentang apa itu *anonymous class*, *anonymous function* dan cara penggunaaan keduanya.

Apa itu *Anonymous Function*

Pada pemrograman *modern*, *anonymous function* sangat umum digunakan terutama pada bahasa pemrograman Javascript yang menurut saya paling banyak. *Anonymous function* adalah *function* tanpa nama dan hanya memiliki *body function* saja atau yang lazim juga disebut *lambda function* atau *closure function*. Perbedaan antara *lambda* dan *closure* adalah bahwa dalam *closure* kita dapat menggunakan *variable* dari luar konteks *function* tersebut.

Anonymous function berguna ketika kita ingin membuat solusi yang spesifik tanpa dan hanya digunakan sekali saja sehingga akan lebih efektif kita membuat *anonymous function* daripada membuat sebuah *class* secara utuh atau membuat *function* secara terpisah.

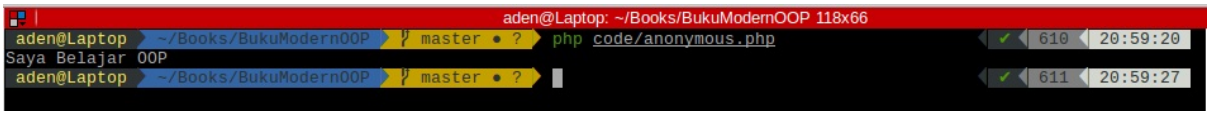
Contoh Penggunaan *Anonymous Function*

Untuk membuat *anonymous function* pada PHP sama seperti pada bahasa pemrograman lainnya. Kita dapat membuat *anonymous function* sebagai *callback* dari suatu *function* seperti yang lazim kita gunakan pada Javascript. Perhatikan contoh berikut:

```
<?php

echo preg_replace_callback('/[a-z]+/i', function (array $mat
ch) {
    return 'PHP' === $match[0] ? 'OOP' : $match[0];
}, 'Saya Belajar PHP');
echo PHP_EOL;
```

Bila program diatas dijalankan maka *output*-nya adalah `Saya Belajar OOP` seperti tampak pada gambar dibawah ini:



Kita juga bisa memasukkan *anonymous function* kedalam sebuah *variable* seperti pada contoh berikut:

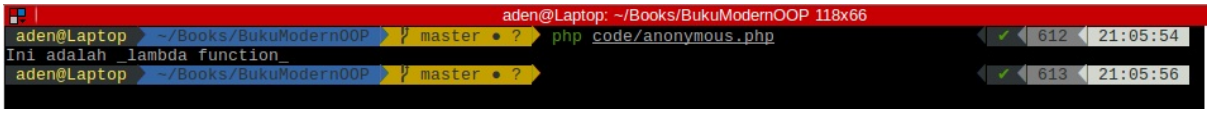
```
<?php

$print = function ($word) {
    echo $word;
};


$print('Ini adalah _lambda function_');
```

```
echo PHP_EOL;
```

Bila program diatas dieksekusi maka *output*-nya adalah sebagai berikut:



Selain kedua cara diatas, kita juga dapat menggunakan *variable* dari luar untuk dimasukkan kedalam *anonymous function* atau yang biasa disebut *closure* seperti pada contoh dibawah ini:

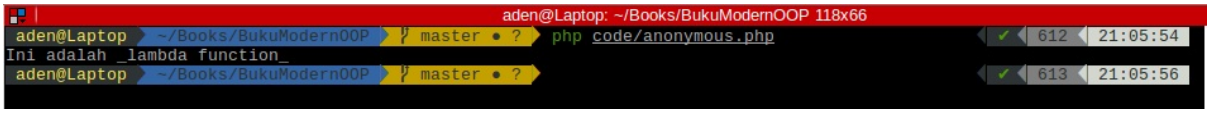
```
<?php

$word = 'Ini adalah _closure_';

$print = function () use ($word) {
    echo $word;
};


$print();
echo PHP_EOL;
```

Dan bila program diatas dijalankan maka *output*-nya adalah sebagai berikut:



Demikianlah beberapa contoh penggunaan *anonymous function* pada PHP. Untuk contoh penggunaan yang lebih banyak, Anda dapat membuka dokumentasi resminya pada halaman ini.

Apa itu *Anonymous Class*

Sama seperti konsep *anonymous function*, *anonymous class* adalah sebuah *class* tanpa nama yang dibuat sesaat sebelum proses instansiasi atau setelah *keyword* `new` dideklarasikan. *Anonymous class* sangat membantu jika kita memerlukan *object* yang simpel dan hanya sekali digunakan. Namun pada prakteknya, *anonymous class* jarang digunakan.

Contoh Penggunaan *Anonymous Class*

Untuk membuat *anonymous class* kita cukup mendeklarasikan *keyword* `new` diikuti *keyword* `class` seperti pada contoh dibawah ini:

```
<?php

$foo = new class{
    public function foo()
    {
        echo 'foo';
    }
};

$foo->foo();
```

```
echo PHP_EOL;
```

Anonymous class kita juga meng-*extends class* lainnya, mengimplementasikan *interface* serta dapat menggunakan *trait*. Perhatikan contoh berikut:

```
<?php

class SebuahClass {}

interface SebuahInterface {}

trait SebuahTrait {}

var_dump(new class(10) extends SebuahClass implements Sebuah
Interface {
    private $num;

    public function __construct($num)
    {
        $this->num = $num;
    }

    use SebuahTrait;
});
```

Hanya sebatas itu saja yang dapat saya jelaskan tentang *anonymous class* karena saya sendiri seingat saya tidak pernah menggunakannya.

XXV. Cara Membuat *Variadic Function*

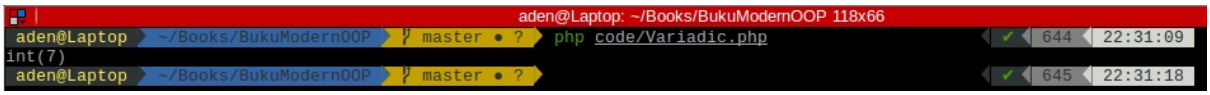
Variadic function adalah sebuah fungsi yang dapat menerima *parameter* secara dinamis. Secara *default* sebenarnya semua *function* dalam PHP adalah *variadic function* karena dapat menerima *parameter* berapapun yang dimasukkan kedalamnya. Perhatikan contoh berikut:

```
<?php

class Variadic
{
    public static function foo(int $number)
    {
        var_dump($number);
    }
}

Variadic::foo(7, 'ini', 'tetap', 'masuk');
```

Jika program diatas dijalankan maka tidak akan terjadi *error* dan mengeluarkan *ouput* sebagai berikut:



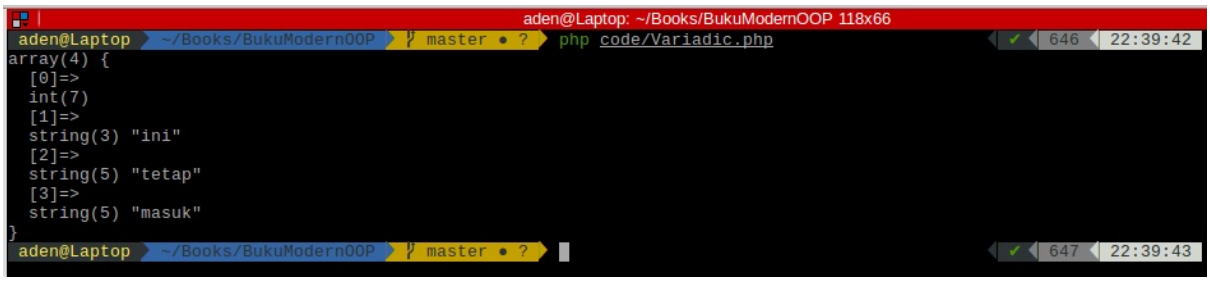
Kita dapat menangkap *parameter-parameter* dari *variadic function* dengan menggunakan `func_get_args()` seperti pada contoh berikut:

```php
<?php

class Variadic
{
    public static function foo(int $number)
    {
        var_dump(func_get_args());
    }
}

Variadic::foo(7, 'ini', 'tetap', 'masuk');
```

Jika program diatas dieksekusi maka *ouput*-nya sebagai berikut:



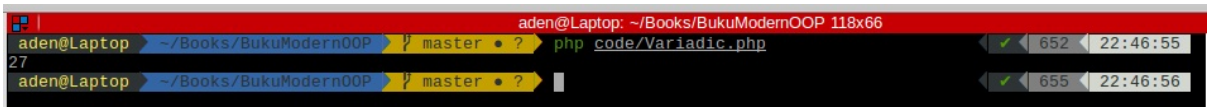
Kita juga dapat menggunakan titik tiga (`...`) untuk mendefiniskan *variadic function* dan cara ini adalah cara yang direkomendasikan. Perhatikan contoh berikut:

```php
<?php

class Variadic
{
    public static function add(int ...$numbers)
    {
        return array_sum($numbers);
    }
}

echo Variadic::add(7, 9, 5, 6);
echo PHP_EOL;
```

Bila program diatas dijalankan maka *ouput*-nya adalah sebagai berikut:



Pada contoh diatas terlihat bahwa *variable* `$numbers` yang merupakan *variadic paramter* ditangkap sebagai *array integer*. Dengan *syntax* diatas kita dapat menambahkan *type hinting* pada *variadic parameter* sehingga dapat menjamin *parameter* yang masuk sesuai dengan spesifikasi yang telah kita tentukan. Selain kita juga memasukkan parameter secara *variadic*, perhatikan contoh berikut:

```
<?php

class Variadic
{
    public static function add(int ...$numbers)
    {
        return array_sum($numbers);
    }
}

$numbers = [7, 9, 5, 6];

echo Variadic::add(...$numbers);
echo PHP_EOL;
```

Bagaimana apakah penjelasan diatas sudah cukup membuka pemahaman tentang *variadic function*? Anda dapat mengembangkan *syntax* diatas sesuai dengan kebutuhan Anda.

XXIX. Instansiasi pada Konteks Statis

Mari kita *refreshing* sejenak agar apa yang telah kita pelajari dapat lebih mengena dan tersimpan dalam hati kita. Pembahasan kali ini akan ringan saja yaitu tentang cara instansiasi *object* pada konteks *static*.

Seperti yang sudah kita pahami bahwa konsep *static* adalah sebuah konsep yang keluar dari aturan dasar pada pemrograman berbasis objek. Dalam konteks *static* kita tidak perlu membuat *object* terlebih dahulu untuk menggunakannya seperti pada contoh berikut:

```
<?php

class A
{
    public static function foo()
    {
        echo 'foo';
    }
}

A::foo();
```

Lalu bagaimana jika kita ingin menginstansiasi *class* pada konteks *static*. Perhatikan contoh berikut:

```php
<?php

class A
{
    public function bar()
    {
        echo 'bar';
    }

    public static function foo()
    {
        echo 'foo';
        //saya ingin memanggil _method_ `bar()` disini
    }
}

A::foo();
```

Bagaimana agar saya dapat melakukan hal seperti yang saya tulis pada komentar program diatas?

Untuk dapat memanggil *method* `bar()` dari *static method* `foo()` kita dapat melakukannya menggunakan cara *instansiasi* seperti biasanya yaitu menggunakan *keyword* `new` diikuti dengan nama *class* tersebut. Perhatikan contoh berikut:

```php
<?php

class A
{
    public function bar()
    {
        echo 'bar';
    }
```

```
    public static function foo()
    {
        echo 'foo';
        (new A())->bar();
    }
}

A::foo();
```

Selain itu kita juga dapat menggunakan *keyword* `self` untuk menginstansiasi *class* dimana *keyword* tersebut dideklarasikan. Dalam hal ini, kita dapat menginstansiasi *class* `A` menggunakan *keyword* `self` seperti pada contoh dibawah ini:

```
<?php

class A
{
    public function bar()
    {
        echo 'bar';
    }

    public static function foo()
    {
        echo 'foo';
        (new self())->bar();
    }
}

A::foo();
```

Kita juga dapat menggunakan *keyword* `static` untuk menginstansiasi *class* pada *static method* yang merujuk pada *class* dimana *keyword* tersebut dideklarasikan. Selain itu juga, *keyword* `static` memiliki fitur *late static bindings* sehingga lebih direkomendasikan. Perhatikan contoh berikut:

```
<?php

class A
{
    public function bar()
    {
        echo 'bar';
    }

    public static function foo()
    {
        echo 'foo';
        (new static())->bar();
    }
}

A::foo();
```

Bila dijalankan, ketiga program diatas akan menghasilkan *output* yang sama yaitu kata `foobar`.

Selain cara diatas, kita juga dapat menggunakan *scope resolution operator* (`::`) untuk memanggil *non static method* dari *static method* dalam lingkup *class*. Perhatikan contoh berikut:

```
<?php

class A
```

```
{
    public function bar()
    {
        echo 'bar';
    }

    public static function foo()
    {
        echo 'foo';
        static::bar();
    }
}

A::foo();
```

Cara diatas juga berlaku pada *keyword* `self` sehingga kita dapat mengganti *keyword* `static` dengan *keyword* `self`.

Magic Method pada PHP

Pada bab ini kita akan membahas tentang *magic method*, apa fungsinya dan ada berapa *magic method* dalam PHP.

Apa itu *Magic Method*

Magic method adalah sekumpulan *method* yang secara *default* didaftarkan oleh PHP pada *object* ketika *object* tersebut dibuat. *Magic method* akan dipanggil secara otomatis pada kondisi tertentu sesuai dengan skop dari *magic method* tersebut.

Magic method pada PHP ditandai dengan *double underscore* (`__`) sebagai awalan dari nama *method*. Contoh *magic method* yang sudah kita bahas maupun sudah kita gunakan adalah *method* `__construct()` , `__destruct()` , dan `__toString()` .

Karena *magic method* menggunakan awalan *double underscore* (`__`), maka ada baiknya kita tidak memberi nama *method* dengan memakai *double underscore* sebagai awalan. Hal ini dimaksudkan agar kita terhindar dari kebingungan.

`__construct()` dan `__desctruct()`

Pada pembahasan sebelumnya kita telah membahas secara mendalam tentang *magic method* `__construct()` dan `__destruct()` sehingga kali ini saya hanya cukup mengulang bahwa *magic method* `__construct()` dan `__destruct()` adalah *magic method* yang berhubungan langsung dengan *object creation* dimana *magic method* `__construct()` dipanggil pada saat pembuatan *object* atau instansiasi, sementara *magic method* `__destruct()` dipanggil ketika *object* dihapus dari *memory*.

`__set()` dan `__get()`

Magic method `__set()` dan `__get()` adalah *magic method* yang dipanggil pada proses pemberian nilai dan pengaksesan nilai pada suatu *property*. *Magic method* ini banyak digunakan pada *modern framework* terutama *framework* yang berhubungan dengan *database* atau yang biasa dikenal dengan sebutan ORM.

Magic method `__set()` akan dipanggil secara otomatis ketika kita hendak memberikan nilai pada suatu *property* yang tidak dapat diakses. Misalnya kita akan memasukkan nilai pada *property* yang memiliki visibilitas *private* ataupun *protected* ataupun ketika kita ingin memasukkan nilai ke *property* yang belum didefinisikan. Perhatikan contoh berikut:

```
<?php

class MagicMethod
{
    private $name;
}
```

```
$magic = new MagicMethod();
$magic->name = 'Muhamad Surya Iksanudin';
```

Bila kita jalankan program diatas tentu akan terjadi *error* karena kita mencoba mengakses dan memberi nilai *property* `$name` yang memiliki visibilitas *private*. Untuk mengatasi masalah tersebut, kita dapat menggunakan *magic method* `__set()` sebagai berikut:

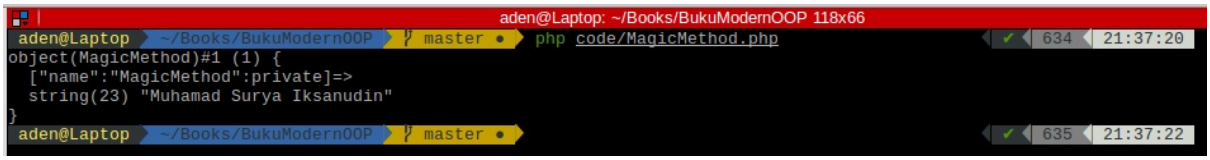
```
<?php

class MagicMethod
{
    private $name;

    public function __set($property, $value)
    {
        if ('name' === $property) {
            $this->name = $value;
        } else {
            throw new ParseError(sprintf('Undefined property
 %s in class %s', $property, __CLASS__));
        }
    }
}

$magic = new MagicMethod();
$magic->name = 'Muhamad Surya Iksanudin';
```

Bila kita menjalankan program diatas maka tidak akan terjadi *error* dan bila `var_dump()` *variable* `$magic` maka hasilnya tampak seperti gambar berikut:



Terlihat bahwa *property* `$name` telah memiliki nilai sebagaimana yang kita set diatas. Perlu diketahui bahwa *magic method* `__set()` di PHP secara *default* memiliki *logic* sebagai berikut:

```
<?php

class MagicMethod
{
    public function __set($property, $value)
    {
        $this->{$property} = $value;
    }
}
```

Sehingga kita dapat memasukkan *property* apapun kedalam sebuah *class* tanpa takut *error* seperti berikut:

```
<?php

class MagicMethod
{
    public function __set($property, $value)
    {
        $this->{$property} = $value;
    }
}

$magic = new MagicMethod();
$magic->name = 'Muhamad Surya Iksanudin';

var_dump($magic);
```

Sehingga kita dapat menuliskan *code* diatas secara singkat sebagai berikut:

```
<?php

class MagicMethod
{
}

$magic = new MagicMethod();
$magic->name = 'Muhamad Surya Iksanudin';

var_dump($magic);
```

Property yang dibuat secara *on fly* tersebut otomatis memiliki visibilitas *public* karena seperti yang telah dibahas pada pembahasan visibilitas bahwa visibilitas *default* pada PHP adalah *public*.

Kebalikan dari *magic method* `__set()` adalah *magic method* `__get()`. *Magic method* ini akan dipanggil ketika kita mencoba mengakses *property* yang tidak dapat diakses. Perhatikan contoh berikut:

```
<?php

class MagicMethod
{
    private $name;

    public function __construct($name)
    {
        $this->name = $name;
    }
```

```
}

$magic = new MagicMethod('Muhamad Surya Iksanudin');
echo $magic->name;
echo PHP_EOL;
```

Bila kita jalankan program diatas tentu akan terjadi *error* karena kita mencoba mengakses *property* `$name` yang memiliki visibilitas *private*. Untuk mengatasi masalah tersebut, kita dapat menggunakan *magic method* `__get()` sebagai berikut:

```
<?php

class MagicMethod
{
    private $name;

    public function __construct($name)
    {
        $this->name = $name;
    }

    public function __get($property)
    {
        if ('name' === $property) {
            return $this->name;
        } else {
            throw new ParseError(sprintf('Undefined property
 %s in class %s', $property, __CLASS__));
        }
    }
}

$magic = new MagicMethod('Muhamad Surya Iksanudin');
echo $magic->name;
echo PHP_EOL;
```

Untuk melihat bagaimana *magic method* `__set()` dan `__get()` diterapkan pada kasus nyata, Anda dapat melihat *code* dari Eloquent yaitu ORM milik *framework* Laravel berikut:

```
// source: https://github.com/illuminate/database/blob/maste
r/Eloquent/Model.php#L1404

public function __get($key)
{
    return $this->getAttribute($key);
}

/**
 * Dynamically set attributes on the model.
 *
 * @param  string  $key
 * @param  mixed  $value
 * @return void
 */
public function __set($key, $value)
{
    $this->setAttribute($key, $value);
}
```

`__isset()` dan `__unset()`

Magic method berikutnya adalah `__isset()` dan `__unset()`, yaitu *magic method* yang akan dipanggil otomatis ketika kita mencoba mengecek eksistensi atau ada tidaknya suatu *property* pada sebuah *class*.

Magic method `__isset()` akan dipanggil ketika kita memanggil fungsi `isset()` atau `empty()` untuk mengecek ada tidaknya *property* yang tidak dapat diakses dari luar. Perhatikan contoh berikut:

```
<?php

class MagicMethod
{
    private $name;
}

$magic = new MagicMethod();

var_dump(isset($magic->name));
```

Tanpa kita menjalankan program diatas, kita dapat tahu bahwa *output* program diatas adalah `false` karena *property* `$name` tidak dapat diakses sehingga dianggap tidak ada. Dengan menggunakan *magic method* `__isset()` kita dapat memanipulasi *output* dari pengecekan `isset()` diatas. Perhatikan contoh berikut:

```
<?php

class MagicMethod
{
    private $name;

    public function __isset($property)
    {
        if ('name' === $property) {
            return true;
        }
```

```
    }
}

$magic = new MagicMethod();

var_dump(isset($magic->name));
```

Sekarang bila kita jalankan kembali program diatas maka *ouput*-nya adalah `true`. Untuk mengecek eksistensi sebuah *property* secara presisi, saya sarankan Anda menggunakan fungsi `property_exists()` karena fungsi tersebut jauh lebih presisi daripada fungsi `isset()`.

Kebalikan dari *magic method* `__isset()` adalah *magic method* `__unset()`. *Magic method* ini akan dipanggil ketika kita mencoba meng-*unset* sebuah *property* yang tidak dapat diakses. *Magic method* `__unset()` biasanya digunakan untuk memanipulasi *array key* pada sebuah *class* dimana *array key* tersebut dianggap sebagai *property* jika diakses dari luar atau biasa disebut *virtual property*. Perhatikan contoh berikut:

```
<?php

class MagicMethod
{
    private $data = [
        'name' => 'Muhamad Surya Iksanudin',
        'address' => 'In your hearth',
    ];

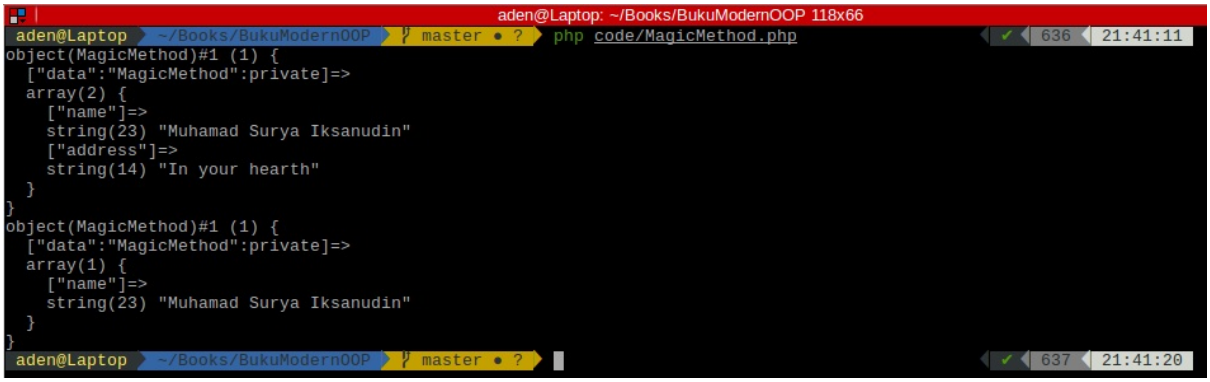
    public function __unset($property)
    {
        if (isset($this->data[$property])) {
            unset($this->data[$property]);
```

```
        }
    }
}

$magic = new MagicMethod();

var_dump($magic);
unset($magic->address);
var_dump($magic);
```

Bila program diatas dijalankan maka *output*-nya adalah sebagai berikut:



Pada gambar diatas terlihat bahwa sebelum fungsi `unset()` dipanggil, *property* `$data` memiliki dua indeks yaitu `name` dan `address`, namun setelah fungsi `unset()` dipanggil, *property* `$data` hanya memiliki satu indeks yaitu `name` saja.

Untuk melihat bagaimana *magic method* `__isset()` dan `__unset()` diterapkan pada kasus nyata, Anda dapat melihat *code* dari Eloquent yaitu ORM milik *framework* Laravel berikut:

```
// source: https://github.com/illuminate/database/blob/maste
r/Eloquent/Model.php#L1472

public function __isset($key)
{
```

```
        return $this->offsetExists($key);
    }

    /**
     * Unset an attribute on the model.
     *
     * @param  string  $key
     * @return void
     */
    public function __unset($key)
    {
        $this->offsetUnset($key);
    }
```

`__sleep()` dan `__wakeup()`

Magic method selanjutnya yang akan kita bahas adalah *magic method* `__sleep()` dan `__wakeup()`. *Magic method* ini akan dipanggil secara otomatis pada proses *serialization* dan *unserialization*. *Serialization* yaitu sebuah proses untuk mengubah *object* menjadi data yang dapat disimpan baik dalam *persistent storage* seperti RDBMS atau ditransmisikan lagi ke program lain melalui *network*, sedangkan *unserialization* adalah kebalikan dari *serialization*.

Magic method `__sleep()` akan dipanggil ketika kita memanggil fungsi `serialize()` untuk melakukan serialisasi pada *object*. Pada *magic method* `__sleep()` kita hanya perlu mengembalikan *array* berupa nama *property* yang akan kita *serialize*. Perhatikan contoh berikut:

```php
<?php

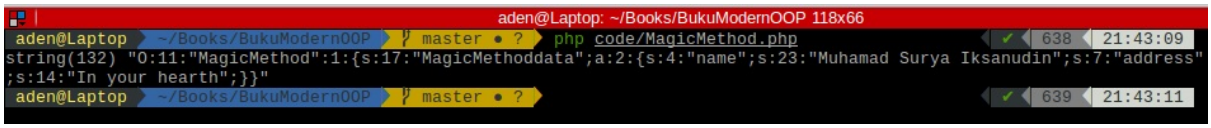
class MagicMethod
{
    private $data = [
        'name' => 'Muhamad Surya Iksanudin',
        'address' => 'In your hearth',
    ];

    public function __sleep()
    {
        return ['data'];
    }
}

$magic = new MagicMethod();

var_dump(serialize($magic));
```

Bila program diatas dijalankan maka *output*-nya adalah sebagai berikut:



Perlu diketahui bahwa *magic method* `__sleep()` hanya akan memproses *property* yang secara visibilitas masuk dalam jangkauan *class* dimana *magic method* `__sleep()` tersebut dipanggil. Misalkan pada kasus pewarisan, *magic method* `__sleep()` didefinisikan pada *parent class* dimana kita mengembalikan *private property* yang ada pada *parent class*, kemudian kita melakukan *serialization* pada *child class* sehingga secara otomatis *magic method* `__sleep()` pada *parent class*

dipanggil, maka PHP tidak akan mampu memproses *private class* tersebut dan akan mengeluarkan *notice* atau peringatan. Perhatikan contoh berikut:

```php
<?php

class ParentClass
{
    private $myOwnProperty;

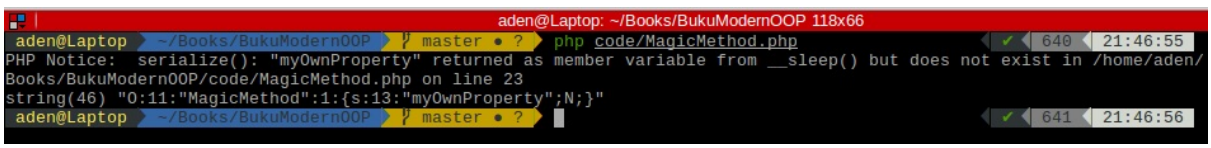
    public function __sleep()
    {
        return ['myOwnProperty'];
    }
}

class MagicMethod extends ParentClass
{
    private $data = [
        'name' => 'Muhamad Surya Iksanudin',
        'address' => 'In your hearth',
    ];
}

$magic = new MagicMethod();

var_dump(serialize($magic));
```

Bila program diatas dieksekusi maka *output*-nya adalah sebagai berikut:



Untuk melihat bagaimana *magic method* `__sleep()` diterapkan pada kasus nyata, Anda dapat melihat *code* dari Doctrine yaitu ORM berikut:

```
// source: https://github.com/doctrine/doctrine2/blob/master
/lib/Doctrine/ORM/Mapping/ClassMetadata.php#L305

public function __sleep()
{
    $serialized = [];

    // This metadata is always serialized/cached.
    $serialized = array_merge($serialized, [
        'declaredProperties',
        'fieldNames',
        //'embeddedClasses',
        'identifier',
        'className',
        'parent',
        'table',
        'valueGenerationPlan',
    ]);

    // The rest of the metadata is only serialized if necess
ary.
    if ($this->changeTrackingPolicy !== ChangeTrackingPolicy
::DEFERRED_IMPLICIT) {
        $serialized[] = 'changeTrackingPolicy';
    }

    if ($this->customRepositoryClassName) {
        $serialized[] = 'customRepositoryClassName';
    }

    if ($this->inheritanceType !== InheritanceType::NONE) {
        $serialized[] = 'inheritanceType';
        $serialized[] = 'discriminatorColumn';
```

```
            $serialized[] = 'discriminatorValue';
            $serialized[] = 'discriminatorMap';
            $serialized[] = 'subClasses';
        }

        if ($this->isMappedSuperclass) {
            $serialized[] = 'isMappedSuperclass';
        }

        if ($this->isEmbeddedClass) {
            $serialized[] = 'isEmbeddedClass';
        }

        if ($this->isVersioned()) {
            $serialized[] = 'versionProperty';
        }

        if ($this->lifecycleCallbacks) {
            $serialized[] = 'lifecycleCallbacks';
        }

        if ($this->entityListeners) {
            $serialized[] = 'entityListeners';
        }

        if ($this->namedQueries) {
            $serialized[] = 'namedQueries';
        }

        if ($this->namedNativeQueries) {
            $serialized[] = 'namedNativeQueries';
        }

        if ($this->sqlResultSetMappings) {
            $serialized[] = 'sqlResultSetMappings';
        }

        if ($this->cache) {
```

```
            $serialized[] = 'cache';
        }

        if ($this->readOnly) {
            $serialized[] = 'readOnly';
        }

        return $serialized;
    }
```

Lawan dari *magic method* `__slepp()` adalah *magic method* `__wakeup()`, yaitu *magic method* yang akan dipanggil ketika kita memanggil fungsi `unserialize()`. Perhatikan contoh berikut:

```
<?php

class Connection
{
    protected $link;

    private $database;

    private $username;

    private $password;

    private $host;

    private $port;

    public function __construct($database, $username, $passw
ord, $host = 'localhost', $port = 3306)
    {
        $this->host = $host;
        $this->port = $port;
        $this->username = $username;
```

```
        $this->password = $password;
        $this->database = $database;

        $this->connect();
    }

    public function __sleep()
    {
        return array('host', 'port', 'username', 'password',
'database');
    }

    public function __wakeup()
    {
        $this->connect();
    }

    private function connect()
    {
        $this->link = new PDO(sprintf('mysql:host=%s;port=%d
;dbname=%s', $this->host, $this->port, $this->database), $th
is->username, $this->password);
    }
}
```

Pada contoh diatas, ketika kita memanggil fungsi `unserialize()` maka secara otomatis koneksi akan di-*reset* yaitu dengan memanggil *method* `connect()`.

Untuk contoh nyata penggunaan *method* `__wakeup()`, Anda dapat melihatnya pada potongan *code* dari *framework* Laravel berikut:

```
// source: https://github.com/laravel/framework/blob/5.5/src
/Illuminate/Database/Eloquent/Model.php#L1500
```

```
/**
 * When a model is being unserialized, check if it needs to
be booted.
 *
 * @return void
 */
public function __wakeup()
{
    $this->bootIfNotBooted();
}
```

`__call()` dan `__callStatic()`

Magic method `__call()` dan `__callStatic()` adalah *magic method* yang akan dipanggil secara otomatis ketika kita memanggil *method* maupun *static method* yang tidak dapat diakses.

Magic method `call()` akan dipanggil secara otomatis ketika kita memanggil sebuah *method* yang tidak dapat diakses pada konteks *object*. Perhatikan contoh berikut:

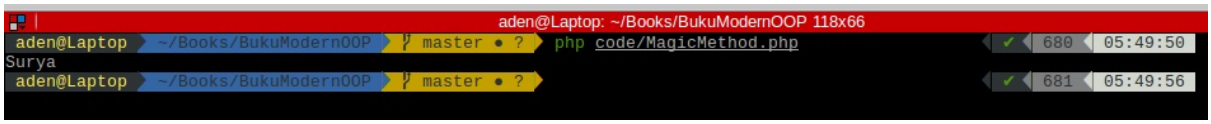
```
<?php

class MagicMethod
{
    private function foo(string $name)
    {
        echo $name;
    }
```

```
    public function __call($name, $arguments)
    {
        if ('foo' === $name) {
            $this->foo($arguments[0]);
        } else {
            throw new Error(sprintf('Undefined method %s on
class %s', $name, __CLASS__));
        }
    }
}


$magic = new MagicMethod();
$magic->foo('Surya');
echo PHP_EOL;
```

Pada contoh diatas kita memanggil *method* `foo()` yang memiliki visibilitas *private* yang seharusnya tidak dapat diakses, tapi dengan menggunakan *magic method* `__call()`, kita tetap dapat memanggilnya dan tidak terjadi *error*. Bila program diatas dijalankan maka hasilnya adalah sebagai berikut:



Perlu diingat bahwa parameter yang dimasukkan kedalam *method*, misal pada contoh diatas adalah *string* `Surya`, ketika masuk ke *magic method* `__call()` akan dikonversi menjadi *array*.

Magic method selanjutnya adalah *magic method* `__callStatic()`, yaitu *magic method* yang akan dipanggil ketika kita memanggil *method* yang tidak dapat diakses pada konteks *static*. Perhatikan contoh berikut:

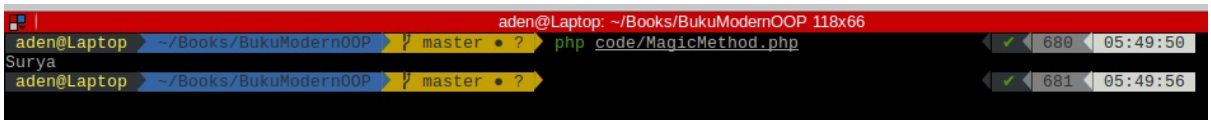
```php
<?php

class MagicMethod
{
    private function foo(string $name)
    {
        echo $name;
    }

    public static function __callStatic($name, $arguments)
    {
        if ('foo' === $name) {
            self::foo($arguments[0]);
        } else {
            throw new Error(sprintf('Undefined method %s on
class %s', $name, __CLASS__));
        }
    }
}


MagicMethod::foo('Surya');
echo PHP_EOL;
```

Bila contoh program diatas dieksekusi maka akan menghasilkan *output* sebagai berikut:



Magic method `__call()` dan `__callStatic()` cukup sering digunakan pada *framework* Laravel seperti pada potongan *code* berikut:

```
// source: https://github.com/laravel/framework/blob/5.5/src
```

```
/Illuminate/Database/Eloquent/Model.php#L1464

public function __call($method, $parameters)
{
    if (in_array($method, ['increment', 'decrement'])) {
        return $this->$method(...$parameters);
    }

    return $this->newQuery()->$method(...$parameters);
}

/**
 * Handle dynamic static method calls into the method.
 *
 * @param  string  $method
 * @param  array  $parameters
 * @return mixed
 */
public static function __callStatic($method, $parameters)
{
    return (new static)->$method(...$parameters);
}
```

`__toString()`

Magic method terakhir yang akan kita bahas adalah `__toString()`. Meski masih banyak *magic method* dalam PHP, namun tidak mungkin kita membahasnya satu per satu sehingga saya putuskan untuk membahasnya hingga sampai pada *magic method* `__toString()` saja.

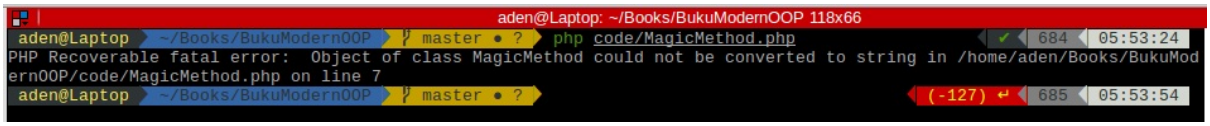
Magic method `__toString()` akan dipanggil secara otomatis ketika kita berusaha mencetak sebuah *object* seperti ketika kita menggunakan *keyword* `echo`. Perhatikan contoh berikut:

```php
<?php

class MagicMethod
{
}

echo new MagicMethod();
echo PHP_EOL;
```

Pada contoh diatas, bila kita jalankan maka akan terjadi *error* seperti gambar dibawah ini:



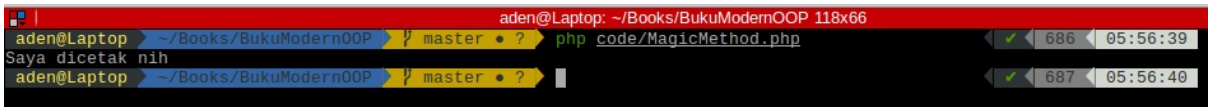
Hal ini terjadi karena kita berusaha mencetak sebuah *object* ke layar menggunakan *keyword* `echo` dan hal tersebut tidak diperbolehkan. Untuk mengatasi hal tersebut, kita harus mengkonversi *object* kedalam *string* yaitu dengan menggunakan *magic method* `__toString()` seperti pada contoh berikut:

```php
<?php

class MagicMethod
{
    public function __toString()
    {
        return 'Saya dicetak nih';
    }
}

echo new MagicMethod();
echo PHP_EOL;
```

Sehingga bila dijalankan maka *ouput*-nya adalah sebagai berikut:



Magic method `__toString()` sangat mudah ditemukan pada *framework* dan cukup sering digunakan pada proyek. Pada *framework*, penggunaan *magic method* `__toString()` dapat dilihat pada potongan *code* dari *framework* Symfony berikut:

```
// source: https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Security/Csrf/CsrfToken.php#L55

/**
 * Returns the value of the CSRF token.
 *
 * @return string The token value
 */
public function __toString()
{
    return $this->value;
}
```

Final *Class* dan Final *Method*

Pada bab ini kita akan membahas tentang apa itu final *class*, apa itu final *method*, kegunaan keduanya serta bagaimana mengaplikasikannya dalam *code* yang kita buat.

Apa itu Final *Class*

Pada beberapa kasus di dunia pemrograman berbasis objek, kita kadang perlu memastikan bahwa sebuah *class* yang kita buat tidak dapat diturunkan menggunakan *keyword* `extends`. Ketika kasus tersebut terjadi, kita dapat menggunakan *keyword* `final` sebelum *keyword* `class` untuk memastikan bahwa *class* tersebut tidak dapat diturunkan.

Contoh Penggunaan Final *Class*

Untuk lebih jelasnya tentang bagaimana final *class* bekerja, perhatikan contoh berikut:

```
<?php

class FinalClass
{
}

class ChildClass extends FinalClass
```

```
{
}
```

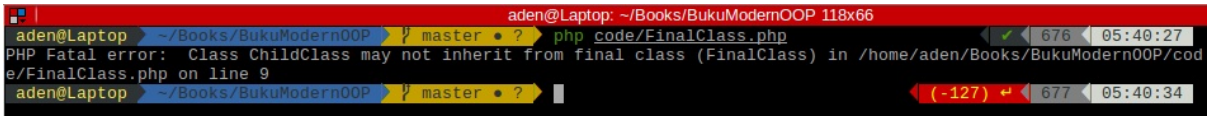
Pada contoh diatas, kita ingin memastikan bahwa *class* `FinalClass` tidak dapat diwariskan, maka kita perlu menambahkan *keyword* `final` pada *class* `FinalClass` tersebut menjadi seperti berikut:

```
<?php

final class FinalClass
{
}

class ChildClass extends FinalClass
{
}
```

Sehingga bila *code* diatas dijalankan akan menghasilkan *error* sebagai berikut:



Pada kasus nyata, final *class* digunakan pada *framework* Api Platform, sebuah *framework* yang khusus dibuat untuk membangun aplikasi RESTful *web* API, seperti pada potongan *code* berikut:

```
//source: https://github.com/api-platform/core/blob/master/src/DataProvider/ChainCollectionDataProvider.php

<?php
```

```
/*
 * This file is part of the API Platform project.
 *
 * (c) Kévin Dunglas <dunglas@gmail.com>
 *
 * For the full copyright and license information, please vi
ew the LICENSE
 * file that was distributed with this source code.
 */

declare(strict_types=1);

namespace ApiPlatform\Core\DataProvider;

use ApiPlatform\Core\Exception\ResourceClassNotSupportedExce
ption;

/**
 * Tries each configured data provider and returns the resul
t of the first able to handle the resource class.
 *
 * @author Kévin Dunglas <dunglas@gmail.com>
 */
final class ChainCollectionDataProvider implements ContextAw
areCollectionDataProviderInterface
{
    private $dataProviders;

    /**
     * @param ContextAwareCollectionDataProvclassiderInterfa
ce[]|CollectionDataProviderInterface[] $dataProviders
     */
    public function __construct(array $dataProviders)
    {
        $this->dataProviders = $dataProviders;
    }
```

```
    /**
     * {@inheritdoc}
     */
    public function getCollection(string $resourceClass, str
ing $operationName = null, array $context = [])
    {
        foreach ($this->dataProviders as $dataProvider) {
            try {
                if ($dataProvider instanceof RestrictedDataP
roviderInterface
                    && !$dataProvider->supports($resourceCla
ss, $operationName, $context)) {
                    continue;
                }

                return $dataProvider->getCollection($resourc
eClass, $operationName, $context);
            } catch (ResourceClassNotSupportedException $e)
{
                @trigger_error(sprintf('Throwing a "%s" in a
 data provider is deprecated in favor of implementing "%s"',
 ResourceClassNotSupportedException::class, RestrictedDataPr
oviderInterface::class), E_USER_DEPRECATED);
                continue;
            }
        }

        return [];
    }
}
```

Apa itu Final *Method*

Sama seperti pada pada final *class*, final *method* adalah sebuah *method* yang tidak dapat di-*override* ketika sebuah *class* diturunkan. Bila kita menggunakan final *class* maka secara otomatis semua *method* yang ada akan menjadi final *method*, maka dengan final *method* kita dapat memilih *method* mana saja yang ingin kita buat menjadi final dan tidak dapat di-*override* sementara *class*-nya tetap dapat diturunkan.

Meski saya jarang menemukan kasus nyata penggunaan final *method* namun konsep final *method* ini cukup penting dipahami karena merupakan salah satu fitur yang ada pada OOP terutama PHP.

Contoh Penggunaan Final *Method*

Sama seperti pada final *class*, untuk membuat sebuah *method* menjadi final, kita dapat mengunakan *keyword* `final` sebelum pendefinisian visibilitas pada *method*. Perhatikan contoh berikut:

```
<?php

class FinalClass
{
    final public function finalMethod()
    {
    }
}
```

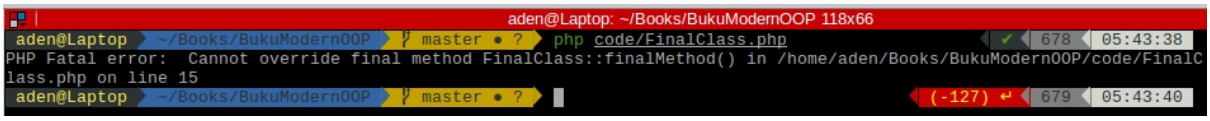
Sehingga bila kita mencoba untuk meng-*override* *method* `finalMethod()` sebagaimana contoh dibawah ini:

```php
<?php

class FinalClass
{
    final public function finalMethod()
    {
    }
}

class ChildClass extends FinalClass
{
    public function finalMethod()
    {
    }
}
```

Maka akan terjadi *error* sebagaimana tampak pada gambar dibawah ini:



Penggunaan final *method* sangat jarang karena biasanya *programmer* langsung menggunakan final *class* untuk membatasi penurunan. Hal ini karena final *class* akan membuat semua *method* menjadi final *method* secara otomatis.

XXIX. *Object* sebagai *Array*

Anda pengguna *framework* Laravel? Apakah Anda familiar dengan *syntax* `$app['events']` atau sejenisnya? Pada pembahasan kali ini kita akan membahas cara membuat *syntax* demikian.

Apa itu *Array Access*

Pada beberapa *framework modern* seperti pada Laravel dan Silex penggunaan *syntax* `$app['events']` adalah hal yang biasa dilakukan. Padahal kalau kita telusuri bahwa *variable* `$app` adalah sebuah *object* dan bukan *array*.

Array access adalah personifikasi sebuah *object* yang seakan-akan dia adalah *array*. Karena *object* tersebut berperan sebagai *array* maka kita dapat menggunakan *syntax array* seperti `$object['sesuatu']` ataupun `$object['sesuatu'] = 'nilai'`.

Contoh Penggunaan *Array Access*

Pada PHP, untuk membuat sebuah *object* dapat berperan sebagai *array* maka *class* dari *object* tersebut harus mengimplementasikan *interface* `ArrayAccess`. *Interface* `ArrayAccess` memiliki 4 *method* yaitu `offsetSet()`, `offsetExists()`, `offsetUnset()`, dan `offsetGet()`. Untuk lebih jelasnya, perhatikan contoh dibawah ini:

```php
<?php

class ArrayAccessClass implements ArrayAccess
{
    private $container;

    public function offsetSet($offset, $value)
    {
        if (is_null($offset)) {
            $this->container[] = $value;
        } else {
            $this->container[$offset] = $value;
        }
    }

    public function offsetExists($offset)
    {
        return isset($this->container[$offset]);
    }

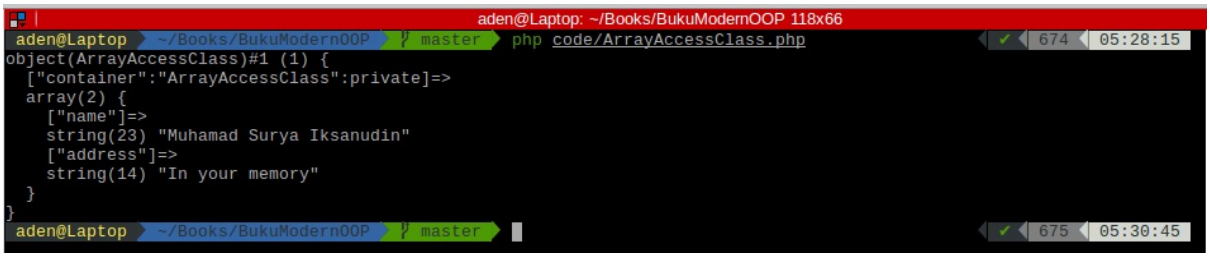
    public function offsetUnset($offset)
    {
        unset($this->container[$offset]);
    }

    public function offsetGet($offset)
    {
        return isset($this->container[$offset]) ? $this->con
tainer[$offset] : null;
    }
}

$object = new ArrayAccessClass();
$object['name'] = 'Muhamad Surya Iksanudin';
$object['address'] = 'In your memory';

var_dump($object);
```

Jika program diatas dieksekusi maka akan menghasilkan *output* sebagai berikut:



Dari contoh diatas terlihat semua *key* yang kita definisikan akan dikonversi jadi *key* pada *property* `$container` sebagaimana yang kita definisikan pada *method* `offsetSet()` .

Perlu diketahui bahwa *object* yang mengimplementasikan *interface* `ArrayAccess` hanya dapat digunakan untuk akses *array* seperti diatas namun tidak dapat digunakan kedalam *interator* seperti `for () {}` atau `foreach ()` .

XXX. Perhitungan Pajak PPH21

Pajak Penghasilan Pasal 21 atau biasa disingkat PPH21 adalah salah satu komponen wajib pada perhitungan penggajian suatu perusahaan. Pada bab ini kita akan mencoba membuat sebuah modul untuk menghitung besaran PPH21 yang harus dibayarkan berdasarkan Penghasilan Kena Pajak atau disingkat PKP. Tentang apa itu PPH21 dapat Anda baca pada halaman ini.

Cara Perhitungan PPH21

Dalam perhitungan PPH21, sebenarnya terdapat banyak parameter yang harus dimasukkan, baik sebagai penambah maupun pengurangnya. Namun pada kesempatan kali ini, kita akan fokus pada hasil akhir parameter-parameter tersebut yaitu PKP (Penghasilan Kena Pajak). Karena menggunakan PKP, maka kita cukup menghitung besaran pajak berdasarkan golongan PKP-nya saja.

Dalam kasus ini, kita menganggap bahwa semua yang dihitung pajaknya melalui modul ini adalah orang yang memiliki NPWP sehingga aturan perhitungan pajaknya adalah sebagai berikut:

> - Wajib Pajak dengan penghasilan tahunan sampai dengan Rp50.000.000,- adalah 5%

> - Wajib Pajak dengan penghasilan tahunan di atas Rp50.000.000,- sampai dengan Rp250.000.000,- adalah 15%
> - Wajib Pajak dengan penghasilan tahunan di atas Rp250.000.000,- sampai dengan Rp500.000.000,- adalah 25%
> - Wajib Pajak dengan penghasilan tahunan di atas Rp500.000.000,- adalah 30%

Berdasarkan ketentuan diatas, misalkan Surya mempunyai PKP sebesar Rp60.000.000,- setahun maka perhitungannya adalah sebagai berikut:

```
PKP = 60.000.000

5% X 50.000.000 = 2.500.000
15% X 10.000.000 = 1.500.000

PPH21 = 2.500.000 + 1.500.000 = 4.000.000
```

Bagaimana sudah mulai mengerti cara menghitungnya? Jadi dari PKP tidak langsung dikalikan dengan persentase tapi digunakan persentase berjenjang. Untuk lebih memahami, perhatikan lagi contoh berikut:

```
PKP = 300.000.000

5% X 50.000.000 = 2.500.000
15% X 200.000.000 = 30.000.000
25% X 50.000.000 = 12.500.000
```

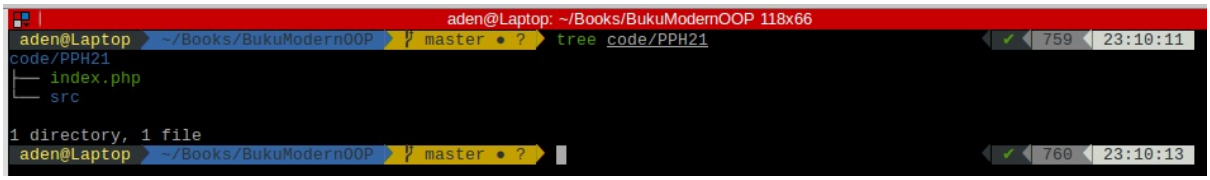
```
PPH21 = 2.500.000 + 30.000.000 + 12.500.000 = 45.000.000
```

Dari kedua contoh diatas, kita dapat memahami bahwa perhitungan PPH21 itu menggunakan perhitungan berjenjang dimana persentase yang lebih kecil akan tetap dihitung dan jadi pengurang persentase selanjutnya.

Persiapan Proyek

Pada tahap ini saya ingin kita menyamakan persepsi bahwa apa yang kita bangun adalah sebuah modul perhitungan PPH21 yang menerapkan semua pembahasan yang telah kita pelajar dari awal hingga akhir. Adapun permasalah yang timbul seperti *code* yang tidak optimal dan lain sebagainya adalah sebuah pembahasan lain yang tidak akan kita permasalahkan disini.

Untuk menyamakan persepsi maka saya akan membuat *folder* kerja `PPH21` yang didalamnya terdapat *folder* `src` dan *file* `index.php`. *Folder* `src` adalah tempat kita menyusun solusi dalam bentuk *code* sementara *file* `index.php` adalah *file* yang akan kita gunakan untuk mengetes solusi yang kita buat. Sehingga susunan *folder* kita akan tampak seperti berikut:



Pengelompokan Masalah dan Solusi

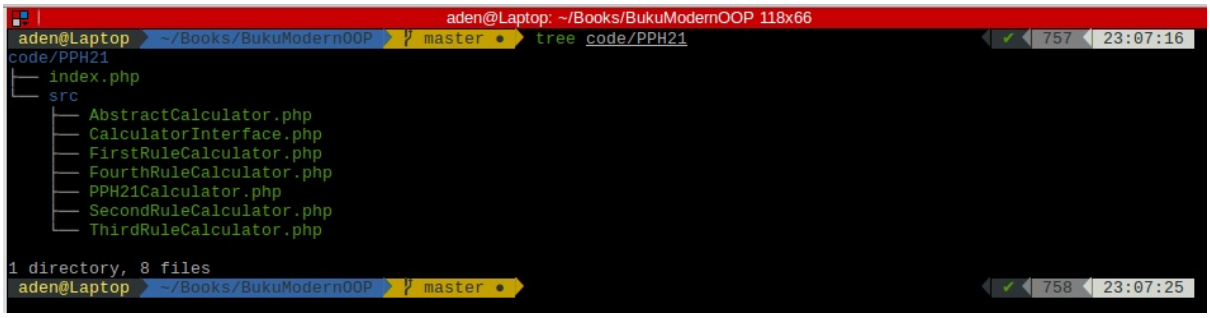
Bila kita mengacu pada aturan PPH21 diatas, maka kita dapat mengelompokkan permasalahan berdasarkan aturan tersebut yaitu:

> - Wajib Pajak dengan penghasilan tahunan sampai dengan Rp50.000.000,- adalah 5%
> - Wajib Pajak dengan penghasilan tahunan di atas Rp50.000.000,- sampai dengan Rp250.000.000,- adalah 15%
> - Wajib Pajak dengan penghasilan tahunan di atas Rp250.000.000,- sampai dengan Rp500.000.000,- adalah 25%
> - Wajib Pajak dengan penghasilan tahunan di atas Rp500.000.000,- adalah 30%

Setiap dari aturan diatas adalah sebuah masalah tersendiri yang dapat kita representasikan sebagai *class*. Sehingga kita sedikitnya akan membuat 5 *class* untuk menyelesaikan masalah PPH21 tersebut dimana 4 *class* adalah sebagai *class* solusi dan 1 *class* sebagai *class wrapper* atau penggabung yang akan menyatukan semua solusi tersebut sehingga lebih mudah untuk digunakan. *Class wrapper* inilah yang nantinya akan dipanggil dari *file* `index.php`.

Untuk menyamakan solusi, maka kita perlu membuat *interface* untuk masing-masing *class* solusi agar ketika kita panggil pada *class wrapper*, kita dapat memastikan bahwa setiap *class* solusi memiliki *method* atau fitur yang sama.

Dan kita akan menambahkan *abstract class* untuk digunakan bersama-sama antar *class* solusi agar tidak tidak mengulang-ngulang *code* yang dapat digunakan bersama-sama. Sehingga sekarang susunan *folder* kita akan nampak seperti berikut:



Mungkin sampai disini Anda bingung kenapa harus banyak *file* dan *class* yang terlibat, padahal permasalahnya cukup simpel. Ini tidak lain karena kita akan menerapkan semua yang telah kita pelajari sehingga Anda akan lebih paham bagaimana OOP menyelesaikan masalah dan bagaimana satu *class* dengan *class* lainnya saling berinteraksi membangun solusi.

Penulisan Code

Code pertama yang kita tulis adalah *file* `CalculatorInterface.php` yang berisi *interface* dari semua kalkulator kita. Dan berikut adalah *code*-nya:

```
<?php
```

```
//file: CalculatorInterface.php

namespace ModernOOP\StudiKasus\PPH21;

interface CalculatorInterface
{
    public function calculate(float $pkp): float;

    public function maxPkp(): float;

    public function minPkp(): float;

    public function taxPercentage(): float;
}
```

Pada *code* diatas terlihat bahwa *interface* `CalculatorInterface` memiliki 4 *methods* yang harus diimplementasikan oleh *class* yang mengimplementasikan *interface* tersebut. Dengan *interface* kita dapat memastikan bahwa seluruh *class* memiliki fitur yang sama yang telah didefinisikan pada *interface* tersebut.

Dari *interface* diatas, kita dapat membuat *abstract class* karena pada perhitungan PPH21 kita dapat menggunakan *recursive function*. Berikut adalah *code* untuk *abstract class* `AbstractCalculator` pada *file* `AbstractCalculator.php`.

```
<?php
//file: AbstractCalculator.php

namespace ModernOOP\StudiKasus\PPH21;

abstract class AbstractCalculator implements CalculatorInter
face
{
    private $chain;
```

```
    public function __construct(?CalculatorInterface $chain
= null)
    {
        $this->chain = $chain;
    }

    public function calculate(float $pkp): float
    {
        $previousValue = 0;
        if ($previous = $this->chain) {
            $previousValue = $this->chain->calculate($previo
us->maxPkp());
            $pkp -= $previous->maxPkp();
        }

        return ($this->taxPercentage() * $pkp) + $previousVa
lue;
    }
}
```

Sedikit saya jelaskan tentang *code* diatas. Pada baris berikut:

```
private $chain;

public function __construct(?CalculatorInterface $chain = nu
ll)
{
    $this->chain = $chain;
}
```

Kita mengaitkan antara *rule* yang satu dengan *rule* sebelumnya sebagaimana perhitungan PPH21 yang telah kita simulasikan diatas. Misalnya *rule* kedua (`15%`) terkait dengan *rule* pertama

(`5%`) dan seterusnya sehingga kita memerlukan baris *code* tersebut.

```
$previousValue = 0;
if ($previous = $this->chain) {
    $previousValue = $this->chain->calculate($previous->maxP
kp());
    $pkp -= $previous->maxPkp();
}
```

Pada baris diatas, kita mengecek apakah dalam *rule* tersebut terkait dengan *rule* sebelumnya dan kalau ada kaitan dengan *rule* sebelumnya maka akan dikalkulasi terlebih dahulu. Hal tersebut dilakukan secara *recursive* dengan memanggil *method* `calculate()` hingga tidak ada *rule* yang dikaitkan dengan *rule* tersebut dan kemudian hasil perhitungannya dimasukkan kedalam *variable* `$previousValue` .

```
return ($this->taxPercentage() * $pkp) + $previousValue;
```

Nilai dari *variable* `$previousValue` kemudian dijadikan faktor penambah pada perhitungan akhir dari PPH21 tersebut. *Code* pada *method* `calculate()` dari *abstract class* `AbstractCalculator` diatas adalah inti dari modul perhitungan PPH21 yang kita buat karena nantinya semua *child class* akan menggunakan *method* tersebut.

Code selanjutnya adalah *code* dari *file* `FirstRuleCalculator.php` dan berikut adalah isinya:

```
<?php
```

```
//file: FirstRuleCalculator.php

namespace ModernOOP\StudiKasus\PPH21;

class FirstRuleCalculator extends AbstractCalculator
{
    public function maxPkp(): float
    {
        return 50000000;
    }

    public function minPkp(): float
    {
        return 0;
    }

    public function taxPercentage(): float
    {
        return 0.05;
    }
}
```

Terlihat simpel sekali bukan? Itu karena kita sudah membuat *abstract class* `AbstractCalculator` yang berisi *logic* inti dari modul kita sehingga *child class* hanya mengimplementasikan *method-method* yang belum diimplementasikan saja.

Untuk *class-class* selanjutnya pun *code*-nya akan mirip-mirip dengan *class* `FirstRuleCalculator` karena memang hanya tinggal mengubah isi dari *method* `minPkp()`, `maxPkp()` dan `taxPercentage()` saja. Dan berikut adalah *code*-nya.

```
<?php
//file: SecondRuleCalculator.php
```

```php
namespace ModernOOP\StudiKasus\PPH21;

class SecondRuleCalculator extends AbstractCalculator
{
    public function maxPkp(): float
    {
        return 250000000;
    }

    public function minPkp(): float
    {
        return 50000000;
    }

    public function taxPercentage(): float
    {
        return 0.15;
    }
}
```

```php
<?php
//file: ThirdRuleCalculator.php

namespace ModernOOP\StudiKasus\PPH21;

class ThirdRuleCalculator extends AbstractCalculator
{
    public function maxPkp(): float
    {
        return 500000000;
    }

    public function minPkp(): float
    {
        return 250000000;
    }
```

```
    public function taxPercentage(): float
    {
        return 0.25;
    }
}
```

```
<?php
//file: FourthRuleCalculator.php

namespace ModernOOP\StudiKasus\PPH21;

class FourthRuleCalculator extends AbstractCalculator
{
    public function maxPkp(): float
    {
        return 10000000000000000;//bikin sebesar mungkin sam
pai impossible ada orang setahun dapat segitu
    }

    public function minPkp(): float
    {
        return 500000000;
    }

    public function taxPercentage(): float
    {
        return 0.3;
    }
}
```

Bagaimana, jadi terlihat lebih simpel *kan* dengan adanya *abstract class* `AbstractCalculator` ? Selanjutnya kita akan membuat *class wrapper* untuk memudahkan dalam pemanggilan pada *file* `index.php` . *Class wrapper* `PPH21Calculator` tersebut berisi sebagai berikut:

```php
<?php
//file: PPH21Calculator.php

namespace ModernOOP\StudiKasus\PPH21;

final class PPH21Calculator
{
    private $calculators;

    public function __construct(CalculatorInterface ...$calc
ulators)
    {
        $this->calculators = $calculators;
    }

    public function calculate(float $pkp): float
    {
        foreach ($this->calculators as $calculator) {
            if ($pkp < $calculator->maxPkp() && $pkp >= $cal
culator->minPkp()) {
                return $calculator->calculate($pkp);
            }
        }
    }
}
```

Setelah semua *class* telah dibuat, maka selanjutnya kita tinggal buat *file* pemanggilnya yaitu `index.php`. *File* tersebut berisi sebagai berikut:

```php
<?php

require __DIR__.'/src/CalculatorInterface.php';
require __DIR__.'/src/AbstractCalculator.php';
require __DIR__.'/src/FirstRuleCalculator.php';
require __DIR__.'/src/SecondRuleCalculator.php';
```

```php
require __DIR__.'/src/ThirdRuleCalculator.php';
require __DIR__.'/src/FourthRuleCalculator.php';
require __DIR__.'/src/PPH21Calculator.php';

use ModernOOP\StudiKasus\PPH21\FirstRuleCalculator;
use ModernOOP\StudiKasus\PPH21\SecondRuleCalculator;
use ModernOOP\StudiKasus\PPH21\ThirdRuleCalculator;
use ModernOOP\StudiKasus\PPH21\FourthRuleCalculator;
use ModernOOP\StudiKasus\PPH21\PPH21Calculator;

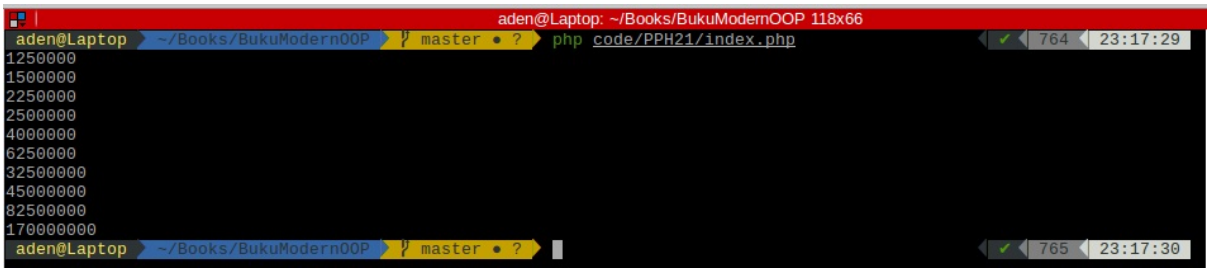

$first = new FirstRuleCalculator();
$second = new SecondRuleCalculator($first);
$third = new ThirdRuleCalculator($second);
$fourth = new FourthRuleCalculator($third);

$calculator = new PPH21Calculator($first, $second, $third, $
fourth);

//1.250.000
echo $calculator->calculate(25000000);
echo PHP_EOL;
//1.500.000
echo $calculator->calculate(30000000);
echo PHP_EOL;
//2.250.000
echo $calculator->calculate(45000000);
echo PHP_EOL;
//2.500.000
echo $calculator->calculate(50000000);
echo PHP_EOL;
//4.000.000
echo $calculator->calculate(60000000);
echo PHP_EOL;
//6.250.000
echo $calculator->calculate(75000000);
echo PHP_EOL;
//32.500.000
```

```
echo $calculator->calculate(250000000);
echo PHP_EOL;
//45.000.000
echo $calculator->calculate(300000000);
echo PHP_EOL;
//82.500.000
echo $calculator->calculate(450000000);
echo PHP_EOL;
//170.000.000
echo $calculator->calculate(750000000);
echo PHP_EOL;
```

Komentar pada *file* `index.php` diatas adalah espektasi yang seharusnya dari perhitungan PPH21 yang benar. Jika *file* `index.php` tersebut dieksekusi maka *output*-nya sebagai berikut:



Dari gambar diatas, terlihat antara espektasi dan *output* program dihasilkan telah sama. Sampai disini, berarti program perhitungan PPH21 yang kita buat telah selesai dan hasilnya sesuai dengan aturan yang berlaku.

Dengan studi kasus diatas, kita telah menerapkan semua yang kita pelajari tentang OOP mulai dari *class* hingga *final class*. Selain itu kita juga dapat mengambil kesimpulan bahwa dengan OOP kita dapat menyelesaikan permasalahan dengan lebih mudah karena kita dapat memecah masalah menjadi bagian yang lebih kecil seperti pada *class-class* `FirstRuleCalculator`,

`SecondRuleCalculator`, dan seterusnya sehingga solusi yang dibuat pun menjadi lebih simpel dan fokus pada satu masalah kecil saja.

Sebenarnya untuk menyelesaikan masalah PPH21 tersebut, kita dapat membuatnya menjadi lebih simpel hanya satu *class* saja sebagai berikut:

```
<?php

class PPH21Calculator
{
    private function firstRule(float $pkp): float
    {
        if (0 < $pkp && 50000000 >= $pkp) {//0 - 50jt
            return $pkp * 0.05;
        }

        return 0;
    }

    private function secondRule(float $pkp): float
    {
        if (50000000 < $pkp && 250000000 >= $pkp) {//50jt -
250jt
            $pkp -= 50000000;

            $prev = $this->firstRule(50000000);

            return ($pkp * 0.15) + $prev;
        }

        return 0;
    }
```

```
    private function thirdRule(float $pkp): float
    {
        if (250000000 < $pkp && 500000000 >= $pkp) {//250jt
- 500jt
            $pkp -= 250000000;

            $prev = $this->secondRule(250000000);

            return ($pkp * 0.25) + $prev;
        }

        return 0;
    }

    private function fourthRule(float $pkp): float
    {
        if (500000000 < $pkp && 10000000000000000 >= $pkp) {
//> 500jt
            $pkp -= 500000000;

            $prev = $this->thirdRule(500000000);

            return ($pkp * 0.3) + $prev;
        }

        return 0;
    }

    public function calculate(float $pkp): float
    {
        return $this->firstRule($pkp) ?: $this->secondRule($
pkp) ?: $this->thirdRule($pkp) ?: $this->fourthRule($pkp);
    }
}

$pph21 = new PPH21Calculator();
```

```
//1250000
echo $pph21->calculate(25000000);
echo PHP_EOL;
//1500000
echo $pph21->calculate(30000000);
echo PHP_EOL;
//2250000
echo $pph21->calculate(45000000);
echo PHP_EOL;
//2500000
echo $pph21->calculate(50000000);
echo PHP_EOL;
//4000000
echo $pph21->calculate(60000000);
echo PHP_EOL;
//6250000
echo $pph21->calculate(75000000);
echo PHP_EOL;
//32500000
echo $pph21->calculate(250000000);
echo PHP_EOL;
//45000000
echo $pph21->calculate(300000000);
echo PHP_EOL;
//82500000
echo $pph21->calculate(450000000);
echo PHP_EOL;
//170000000
echo $pph21->calculate(750000000);
echo PHP_EOL;
```

Bagaimana, apakah Anda merasa *zonk*? Tidak perlu merasa demikian karena contoh diatas hanya sebagai perbandingan saja dan apa yang telah Anda buat jauh lebih mudah dimaintain seandainya terdapat perubahan peraturan maupun penambahan *rule* baru dikemudian hari.

XXXI. *Package Management* dengan *Composer*

Salah satu hal tersulit dalam mengerjakan sebuah proyek *software development* adalah me-*manage dependency library* yang dibutuhkan dalam proyek tersebut. Hal ini akan dipermudah dengan sebuah *tool* yaitu *composer* yang akan kita bahas pada bab ini.

Apa itu *Composer*

Composer adalah sebuah *tool* yang bertujuan untuk memudahkan *developer* dalam me-*manage dependency* pada proyek berbasis PHP. *Composer* seperti npm pada NodeJs, yum pada Redhat, bundler pada Ruby, atau apt pada Ubuntu.

Composer dapat digunakan untuk meng-*install*, meng-*update* dan menghapus *library* yang kita gunakan dalam proyek kita. *Composer* pertama kali diperkenalkan oleh Jordi Boggiano dan Nils Adermann tidak lama setelah PSR-0 disetujui sebagai *standard* pada ekosistem PHP.

Composer pertama kali digunakan sebagai *tool* instalasi utama oleh *framework* Symfony karena Jordi adalah salah satu *core developer framework* tersebut. Sehingga penamaan *folder* yang digunakan oleh *composer* pun menggunakan konsensus yang berlaku pada *framework* Symfony yaitu *folder* `vendor`.

Kenapa Menggunakan *Composer*

Selain sebagai *dependency manager*, *composer* juga memiliki fitur utama yaitu *autoload* dimana kita tidak perlu lagi meng-*include file* PHP satu per satu seperti yang terjadi pada pembahasan sebelumnya. Pada pembahasan sebelumnya, ketika kita hendak menggunakan *file* atau *class* kita harus meng-*include* sebagai berikut:

```
require __DIR__.'/src/CalculatorInterface.php';
require __DIR__.'/src/AbstractCalculator.php';
require __DIR__.'/src/FirstRuleCalculator.php';
require __DIR__.'/src/SecondRuleCalculator.php';
require __DIR__.'/src/ThirdRuleCalculator.php';
require __DIR__.'/src/FourthRuleCalculator.php';
require __DIR__.'/src/PPH21Calculator.php';
```

Bila kebutuhan *file* atau *class* kita sedikit, mungkin hal tersebut tidak terlalu masalah, namun jika kebutuhan *file* atau *class* yang kita perlukan banyak, maka hal tersebut akan sangat menyulitkan dan cukup memakan banyak *line code*. Dengan *composer* hal tersebut tak perlu terjadi karena *composer* dapat digunakan sebagai *autoloader* yaitu dengan memanggil *file* `autoload.php` yang ada di *folder* `vendor` sebagai berikut:

```
require __DIR__ . '/vendor/autoload.php';
```

Dengan satu baris diatas maka kita tidak perlu lagi menggunakan *block* `require` lagi sehingga kita bisa menghapus *block* `require` yang panjang diatas sehingga `index.php` akan menjadi sebagai

berikut:

```
<?php

require __DIR__ . '/vendor/autoload.php';

use ModernOOP\StudiKasus\PPH21\FirstRuleCalculator;
use ModernOOP\StudiKasus\PPH21\SecondRuleCalculator;
use ModernOOP\StudiKasus\PPH21\ThirdRuleCalculator;
use ModernOOP\StudiKasus\PPH21\FourthRuleCalculator;
use ModernOOP\StudiKasus\PPH21\PPH21Calculator;


$first = new FirstRuleCalculator();
$second = new SecondRuleCalculator($first);
$third = new ThirdRuleCalculator($second);
$fourth = new FourthRuleCalculator($third);

$calculator = new PPH21Calculator($first, $second, $third, $
fourth);

//1250000
echo $calculator->calculate(25000000);
echo PHP_EOL;
//1500000
echo $calculator->calculate(30000000);
echo PHP_EOL;
//2250000
echo $calculator->calculate(45000000);
echo PHP_EOL;
//2500000
echo $calculator->calculate(50000000);
echo PHP_EOL;
//4000000
echo $calculator->calculate(60000000);
echo PHP_EOL;
//6250000
echo $calculator->calculate(75000000);
```

```
echo PHP_EOL;
//32500000
echo $calculator->calculate(250000000);
echo PHP_EOL;
//45000000
echo $calculator->calculate(300000000);
echo PHP_EOL;
//82500000
echo $calculator->calculate(450000000);
echo PHP_EOL;
//170000000
echo $calculator->calculate(750000000);
echo PHP_EOL;
```

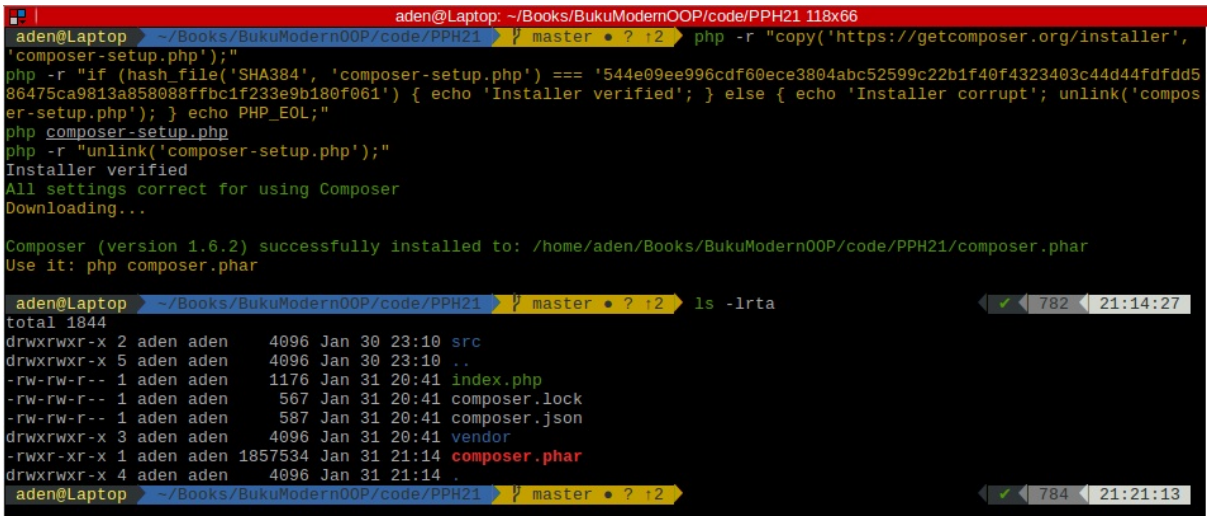
Sehingga dapat disimpulkan, selain sebagai *dependency manager*, *composer* juga digunakan sebagai *autoloader* untuk meminimalkan penggunaan `require` pada *code* kita.

Instalasi *Composer*

Untuk menginstall *composer* sangatlah mudah, Anda cukup meng-*copy script* berikut:

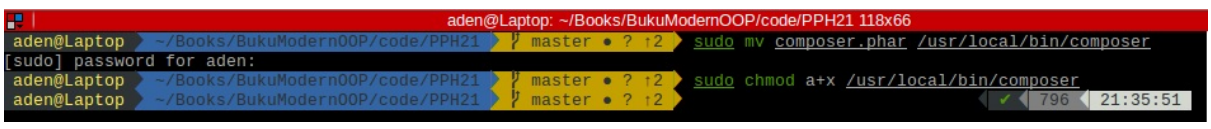
```
php -r "copy('https://getcomposer.org/installer', 'composer-
setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php') === '5
44e09ee996cdf60ece3804abc52599c22b1f40f4323403c44d44fdfdd586
475ca9813a858088ffbc1f233e9b180f061') { echo 'Installer veri
fied'; } else { echo 'Installer corrupt'; unlink('composer-s
etup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Pada *script* diatas, kita melakukan 4 hal yaitu *download installer*, verifikasi `SHA384`, menjalankan *setup* dan kemudian menghapus *file installer*. Bila semua tahap diatas berhasil, maka pada *folder* tempat kita menjalankan *script* diatas akan terdapat *file* `composer.phar` seperti pada gambar berikut:

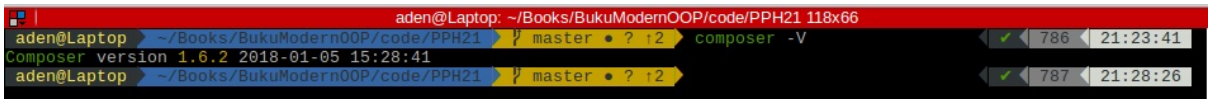


Bila Anda menggunakan sistem operasi Windows, Anda dapat menggunakan *installer* yang lebih mudah melalui link berikut sehingga nantinya Anda dapat membuka *command prompt* dan menjalankan perintah `composer -V`.

Sementara untuk pengguna sistem operasi Linux atau Unix-*Like* kita dapat memindahkan *file* `composer.phar` dan mengganti nama menjadi `/usr/local/bin/composer` kemudian menjalakan perintah `sudo chmod a+x /usr/local/bin/composer` sehingga akan tampak sebagai berikut:



Baik menggunakan Windows, Linux, maupun Unix-*Like*, pada akhirnya Anda dapat menjalankan perintah `composer -V` seperti pada gambar berikut:



Tentang `composer.json`

Untuk menggunakan *composer*, Anda harus membuat *file* `composer.json` pada *root project* dan kemudian mendeskripsikan kebutuhan kita pada *block* `require` di *file* `composer.json` sebagai berikut:

```
{
    "require": {
        "monolog/monolog": "1.23.*"
    }
}
```

Pada *code* diatas, kita mendeskripsikan bahwa proyek yang kita bangun membutuhkan *package* `monolog/monolog` dengan versi `1.23.*` yang artinya adalah versi `1.23.n` dimana `n` adalah versi terakhir dari versi dasar `1.23`. Penulisan versi sendiri mengikuti *standard* Semantic Versioning dan merujuk pada *tag* di *repository* Github.

Pada contoh diatas, ketika kita menjalankan `composer update` maka secara otomatis *composer* akan men-*download package* `monolog/monolog` dari Packagist dan menyimpannya di *folder* `vendor` sejajar dengan *file* `composer.json`.

Bagian penting lainnya dari `composer.json` adalah *block* `autoload` dimana kita mendeskripsikan *namespace* dari proyek kita serta *root source*-nya. Perhatikan contoh berikut:

```
"autoload": {
    "psr-4": {
        "ModernOOP\\StudiKasus\\PPH21\\": "src/"
    }
}
```

Pada contoh diatas, kita memberitahu *composer* bahwa proyek kita memiliki namespace `ModernOOP\StudiKasus\PPH21` dan merujuk pada *folder* `src`. Ini berarti bahwa semua *class* yang ada di *folder* `src` harus dimulai dengan *namespace* `ModernOOP\StudiKasus\PPH21` sebagaimana aturan pada PSR-4.

Untuk keterangan lebih lengkap tentang *file* `composer.json`, Anda dapat membacanya pada halaman ini.

XXXII. Membuat *Package* Sendiri

Pada bab ini kita akan membahas bagaimana membuat *package composer* sendiri dan kemudian mendaftarkannya ke Packagist. *Package* yang akan kita daftarkan adalah modul kalkulator PPH21 yang telah kita buat sebelumnya.

Membuat `composer.json`

Seperti yang sudah dibahas pada bab sebelumnya tentang *composer*, kali ini kita akan membuat *file* `composer.json` untuk modul kalkulator PPH21 kita. Isi dari *file* `composer.json` tersebut adalah sebagai berikut:

```
{
    "type": "library",
    "license": "proprietary",
    "name": "modernoop/pph21",
    "authors": [
        {
            "name": "Muhamad Surya Iksanudin",
            "email": "surya.iksanudin@gmail.com",
            "homepage": "https://github.com/ad3n"
        }
    ],
    "require": {
        "php": "^7.2"
    },
```

```
    "config": {
        "preferred-install": {
            "*": "dist"
        },
        "sort-packages": true,
        "classmap-authoritative": true,
        "optimize-autoloader": true
    },
    "autoload": {
        "psr-4": {
            "ModernOOP\\StudiKasus\\PPH21\\": "src/"
        }
    }
}
```

File tersebut diletakkan sejajar *folder* PPH21 atau pada *root project*. Pada *file* `composer.json` diatas, *autoload* PSR-4 kita arahkan ke *folder* `src` karena semua *class* yang kita buat berada disana. Kemudian jalankan perintah `composer update` hingga selesai maka pada *folder* PPH21 akan muncul *folder* baru yaitu *folder* `vendor` yang berisi *folder* `composer` dan *file* `autoload.php`.

Tidak ada *folder* lain karena kita tidak menggunakan *library* apapun sebagai *dependency* dari modul kalkulator PPH21 tersebut.

Autoload dengan *Composer*

Sebelum mendaftarkan *package* kita ke Packagist, terlebih dahulu kita tes hasil instalasi kita dengan mengubah *file* `index.php` yang berisi banyak `require` seperti dibawah ini:

```
require __DIR__.'/src/CalculatorInterface.php';
```

```
require __DIR__.'/src/AbstractCalculator.php';
require __DIR__.'/src/FirstRuleCalculator.php';
require __DIR__.'/src/SecondRuleCalculator.php';
require __DIR__.'/src/ThirdRuleCalculator.php';
require __DIR__.'/src/FourthRuleCalculator.php';
require __DIR__.'/src/PPH21Calculator.php';
```

Menjadi *autoload* dengan hanya sebuah baris `require` sebagai berikut:

```
require __DIR__ . '/vendor/autoload.php';
```

Sehingga isi dari *file* `index.php` berubah menjadi sebagai berikut:

```
<?php

require __DIR__ . '/vendor/autoload.php';

use ModernOOP\StudiKasus\PPH21\FirstRuleCalculator;
use ModernOOP\StudiKasus\PPH21\SecondRuleCalculator;
use ModernOOP\StudiKasus\PPH21\ThirdRuleCalculator;
use ModernOOP\StudiKasus\PPH21\FourthRuleCalculator;
use ModernOOP\StudiKasus\PPH21\PPH21Calculator;


$first = new FirstRuleCalculator();
$second = new SecondRuleCalculator($first);
$third = new ThirdRuleCalculator($second);
$fourth = new FourthRuleCalculator($third);

$calculator = new PPH21Calculator($first, $second, $third, $
fourth);

//1250000
echo $calculator->calculate(25000000);
```

```
echo PHP_EOL;
//1500000
echo $calculator->calculate(30000000);
echo PHP_EOL;
//2250000
echo $calculator->calculate(45000000);
echo PHP_EOL;
//2500000
echo $calculator->calculate(50000000);
echo PHP_EOL;
//4000000
echo $calculator->calculate(60000000);
echo PHP_EOL;
//6250000
echo $calculator->calculate(75000000);
echo PHP_EOL;
//32500000
echo $calculator->calculate(250000000);
echo PHP_EOL;
//45000000
echo $calculator->calculate(300000000);
echo PHP_EOL;
//82500000
echo $calculator->calculate(450000000);
echo PHP_EOL;
//170000000
echo $calculator->calculate(750000000);
echo PHP_EOL;
```

Kita kemudian dapat mengetesnya dengan memanggil *file* `index.php` menggunakan perintah `php index.php` dari *root project*. Bila tidak ada *error* maka berarti instalasi *composer* kita telah berhasil dan kita sudah menggunakan fitur *autoload* dari *composer*.

Pendaftaran *Package*

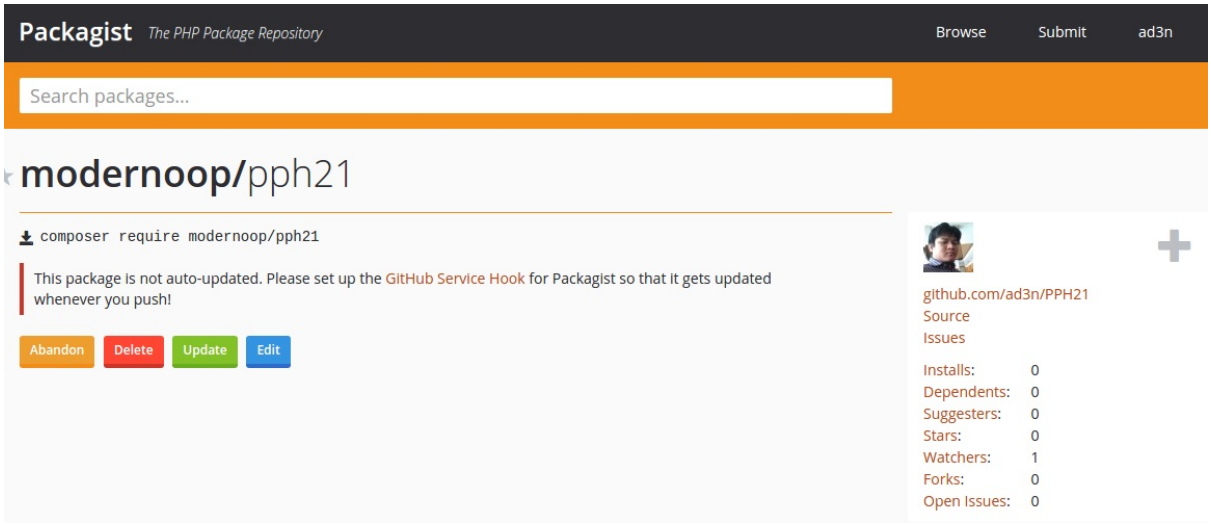
Sebelum mendaftarkan *package* ke Packagist, terlebih dahulu kita harus meng-*upload*-nya ke Github. Untuk tutorial bagaimana meng-*upload* proyek atau *package* kita ke Github, Anda dapat membaca tutorial lengkapnya pada *link* ini.

Anggap saja Anda sudah meng-*upload* proyek kita ke Github dengan *URL* `https://github.com/ad3n/PPH21`, maka selanjutnya kita buka website Packagist yaitu `https://packagist.org` kemudian kita pilih menu *sign in* dan pilih *Use Github*. Ikuti prosesnya hingga kemudian kita otomatis akan *login* dengan tampil halaman sebagai berikut:



Kemudian kita pilih menu *submit* dan kemudian masukkan *URL repository* kita yaitu `https://github.com/ad3n/PPH21` dan kemudian pilih tombol *check*. Bila nama yang kita daftarkan telah ada, maka Anda dapat mengganti baris `"name": "modernoop/pph21"` dengan nama yang sesuai dengan keinginan Anda. Setelah itu kemudian klik tombol *Submit*.

Bila kita berhasil mendaftarkan *package* kita, maka akan muncul tampilan sebagai berikut:

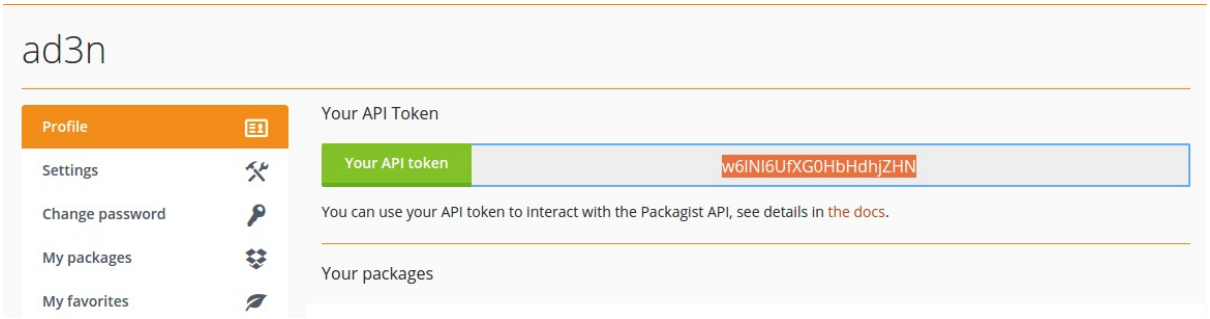


Bila sudah maka kita dapat meng-*install* *package* kita menggunakan *composer* dengan menjalankan perintah `composer require modernoop/pph21`. Secara otomatis didalam *folder* `vendor` akan muncul *folder* `modernoop/pph21` yang berisi modul kalkulator PPH21.

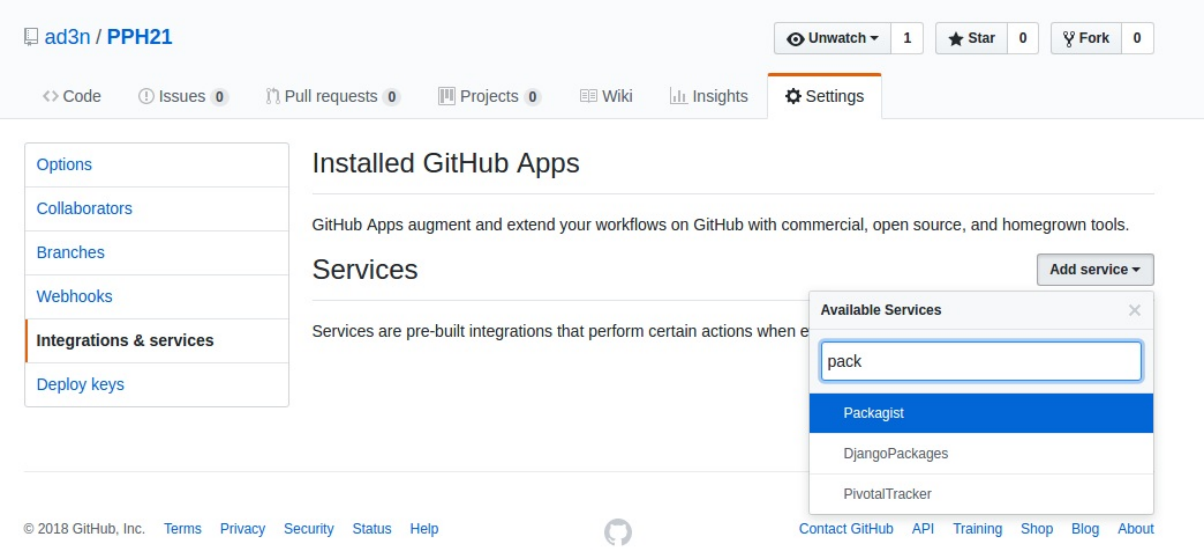
Sinkronisasi Github dan Packagist

Secara *default*, Packagist tidak melakukan sinkronisasi antara *repository* kita yang di Github dengan *package* yang kita daftarkan pada Packagist. Untuk itu kita perlu menggunakan Github *Hook Services* untuk melakukan hal tersebut.

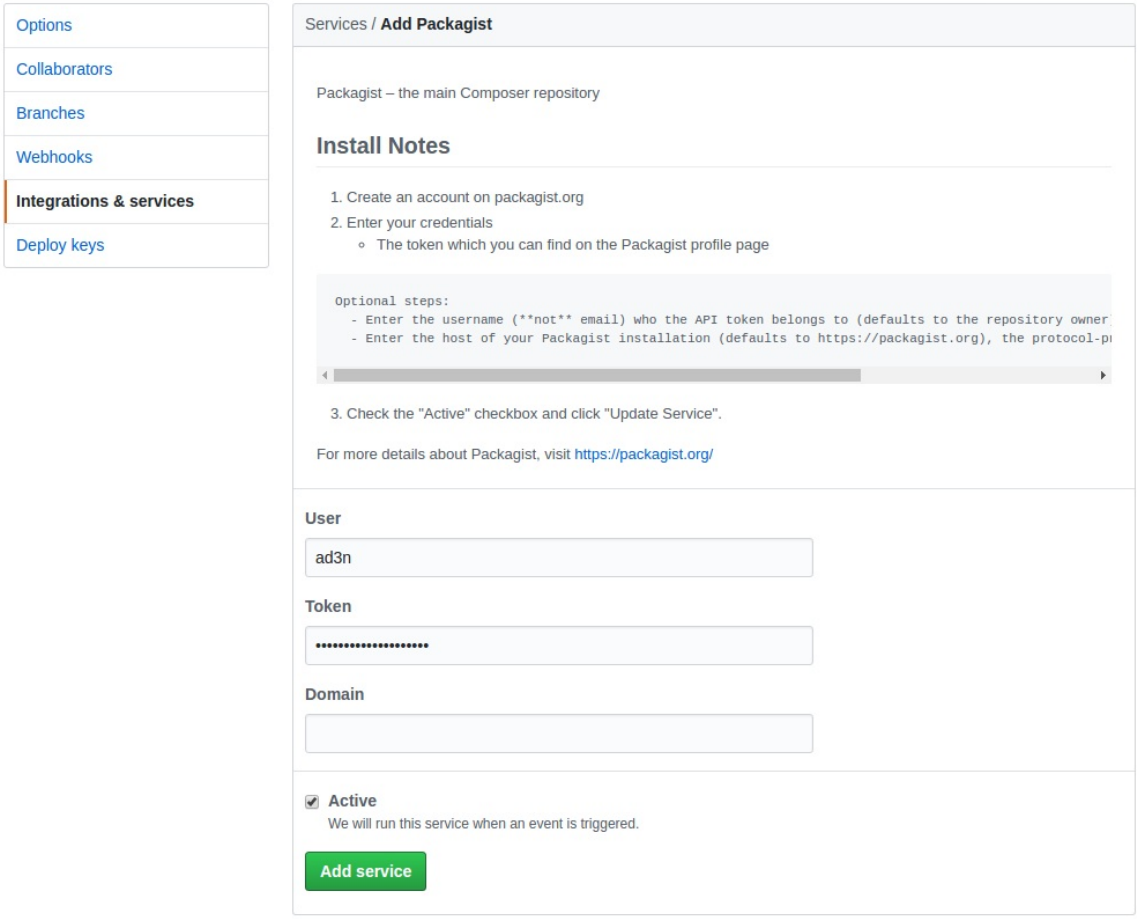
Untuk melakukan hal tersebut, pertama kali kita harus meng-*copy* API *Token* yang ada pada halaman *profile* Packagist terlebih dahulu. Halaman API *Token* tampak seperti gambar berikut:



Kemudian kita kembali ke halaman *repository* kita di Github dan klik menu *setting* dan pilih *integrations and services* kemudian klik tombol *add service* lalu ketikan Packagist seperti tampak pada gambar berikut:



Lalu isikan *username* dengan *username* Github, kemudian *token* dengan API *Token* dari Packagist. Pada isian URL, Anda dapat mengosonginya. Kemudian klik tombol *Add Service* seperti pada gambar berikut:



Dengan cara seperti itu, maka jika kita meng-*update code* kita di Github *repository* maka secara otomatis pada Packagist juga akan ter-*update*.

XXXIII. *Design Pattern*

Perkembangan teknologi dan metodelogi pengembangan *software* memunculkan *tool-tool* yang dapat digunakan untuk membantu menyelesaikan masalah dan mempercepat pengambangan *software* itu sendiri. Salah satu *tool* tersebut adalah *design pattern* yang akan kita bahas pada bab ini.

Apa itu *Design Pattern*

Design pattern adalah solusi umum yang berupa *best practice* yang digunakan untuk menyelesaikan permasalah-permasalahan yang muncul pada pengembangan *software*. *Design pattern* semakin berkembang dengan adanya konsep OOP karena hampir semua *design pattern* diimplementasikan menggunakan konsep tersebut.

Design pattern banyak diimplementasikan dalam pembuatan sebuah *framework*. Dalam sebuah *framework* terdapat banyak *design pattern* yang digunakan dan saling mendukung satu dengan lainnya untuk menyelesaikan permasalahan yang ada pada *framework* tersebut.

Manfaat Penggunaan *Design Pattern*

Pemahaman tentang *design pattern* sangat bermanfaat dalam mempercepat perancangan solusi serta memberikan kemudahan bagi *developer* lain untuk memahami *code* yang kita tulis. Selain itu, *code* yang ditulis menggunakan *design pattern* akan lebih mudah untuk di-*refactoring* dikemudian hari.

Manfaat lain penggunaan *design pattern* adalah efisiensi *code*, dimana *code* yang kita tulis menjadi lebih singkat dan lebih mudah fokus dalam menyelesaikan permasalahan tertentu sehingga ketika masalah tersebut berkembang, maka kita dapat mengembangkan solusi yang baru dengan lebih mudah.

Macam-Macam *Design Pattern*

Secara garis besar, *design pattern* dibagi menjadi 3 yaitu:

- Kreasional (*Creational*)

Pattern ini fokus pada bagaimana sebuah *object* diinstansiasi. Contoh *pattern* ini adalah *abstract factory*, *builder*, *pool*, *singleton*, dan lain sebagainya.

- Struktural

Pattern ini fokus pada bagaimana mempermudah *class* atau *object* yang memiliki fungsionalitas baru berkolaborasi. Contoh *pattern* ini adalah *decorator*, *adapter*, *composite*, dan lain sebagainya.

- Behavioral

Pattern ini mengatur bagaimana komunikasi antar *class* dalam menyelesaikan sebuah masalah. Contoh *pattern* ini antara lain *chain of responsibilities*, *strategy*, *observer* dan masih banyak lagi.

Pada modul PPH21 yang telah kita buat sebelumnya, kita menggunakan *chain of responsibilities pattern* untuk dimana setiap kalkulator dikaitkan dengan kalkulator lainnya. Pembahasan tentang *design pattern* akan dibuat pada buku yang terpisah, tunggu selalu buku-buku dari saya ya (promosi tidak terselubung).

XXXIV. Studi Kasus Membuat Framework Sederhana

Pada pembahasan kali ini kita akan mencoba membuat *framework* sendiri menggunakan *package* yang telah ada di Packagist.

Skop Proyek

Agar pembahasan pada bab ini tidak melebar jauh, maka saya akan membatasi pembahasan kali ini yaitu tentang bagaimana menampilkan pesan `Hello World` dan pesan `Selamat Datang, {nama}` menggunakan mekanisme *framework*. *Framework* yang kita bangun pada bab ini adalah sebuah *framework* sederhana yang hanya dapat menangani *request* yang sederhana tanpa ada interaksi dengan *database* dan *form*.

Konsep *Front Controller*

Sebagai awal pembuatan *framework* kita saya telah membuat *folder* `ModernFramework` sebagai *root project* kita dan berisi 2 *file* yaitu `hello.php` dan `greeting.php` dimana isi dari masing-masing *file* adalah sebagai berikut:

```
<?php
//filename: hello.php
```

```
echo 'Hello World';
```

```
<?php
//filename: greeting.php

echo sprintf('Selamat Datang, %s', $_GET['nama']);
```

Secara normal, untuk memanggil *file* tersebut maka kita harus membuat *browser* dan mengetikkan `http://localhost:8000/hello.php` untuk *file* `hello.php` dan `http://localhost:8000/greeting.php?nama=Surya` untuk *file* `greeting.php`.

Anda tidak perlu bingung kenapa ada angka `8000` setelah `localhost`, itu hanyalah nomer *port* yang dihasilkan ketika saya menggunakan *built-in web server* di PHP menggunakan perintah:

```
php -S localhost:8000
```

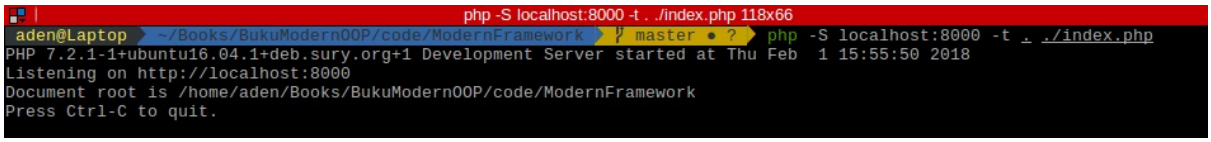
Karena pada *framework* biasanya menggunakan 1 *file* `index.php` sebagai *front controller* maka saya membuat *file* `index.php` sejajar dengan *file* `hello.php` dan `greeting.php` dengan isi sebagai berikut:

```
<?php
//filename: index.php

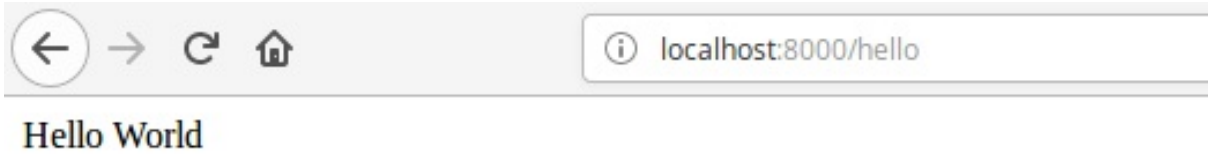
if ('/hello' === $_SERVER['REQUEST_URI']) {
    require 'hello.php';
}
```

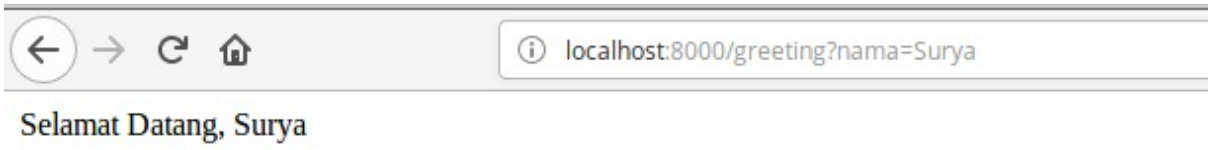
```
if (false !== strpos($_SERVER['REQUEST_URI'], '/greeting'))
{
    require 'greeting.php';
}
```

Ketika kita menjalankan *built-in web server* menggunakan perintah dibawah ini:



Kemudian membuat *browser* maka hasilnya adalah sebagai berikut:





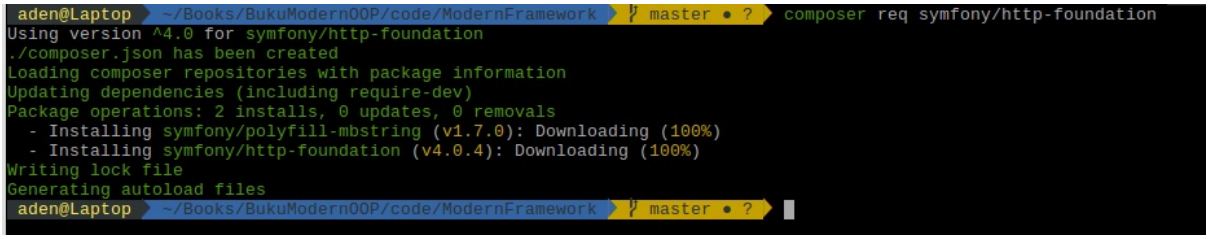
Sampai tahap ini kita telah mengimplementasikan konsep *front controller* yaitu hanya menggunakan 1 *file* saja sebagai pengatur *request* yang masuk yaitu *file* `index.php` .

HTTP Request dan *HTTP Response*

Pada dasarnya ketika klien membuat sebuah halaman *web* maka sebenarnya dia telah melakukan *request* ke *server* yang disebut sebagai HTTP *Request* dan kemudian ketika *server* memberikan *response*, setelah memproses *request* tersebut sebelumnya, itulah yang disebut sebagai HTTP *Response*.

Pada pembahasan kali ini kita akan menggunakan OOP untuk memproses *request* dan *response* pada *framework* yang kita buat. Untuk itu kita perlu meng-*install package* menggunakan *composer*. *Package* yang akan kita gunakan adalah `symfony/http-foundation`, *package* yang sama yang digunakan juga oleh *framework* Laravel untuk menangani *request* dan *response*.

Untuk meng-*install* kita cukup menjalankan perintah `composer req symfony/http-foundation` dari *root project* dan secara otomatis *composer* akan membuat *file* `composer.json` dan meng-*install*-kan *package* tersebut untuk kita sebagaimana gambar berikut:



Setelah berhasil meng-*install*, selanjutnya kita perlu mengubah *file* `index.php` dan mendaftarkan *composer autoloader* sebagai berikut:

```
<?php
//filename: index.php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
```

```
$request = Request::createFromGlobals();
$path = $request->getPathInfo();

$route = ['/hello' => 'hello.php', '/greeting' => 'greeting.
php'];

if (isset($route[$path])) {
    include $route[$path];
}
```

Dengan *code* diatas, kita telah mengimplementasikan Symfony *request* (`Symfony\Component\HttpFoundation\Request`) untuk mengarahkan permintaan klien. Sedikit penjelasan tentang *code* diatas, baris *code* `$request = Request::createFromGlobals()` adalah proses instansiasi *object* `Request` berdasarkan PHP *super global variable* (`$_GET` , `$_POST` , dan seterusnya).

Kita perlu mengubah juga *file* `hello.php` dan `greeting.php` untuk mengimplementasikan Symfony *response* agar *framework* yang kita buat lebih *powerful*.

```
<?php
//filename: hello.php

use Symfony\Component\HttpFoundation\Response;

$response = new Response();
$response->setContent('Hello World');

$response->send();
```

```
<?php
```

```
//filename: greeting.php

use Symfony\Component\HttpFoundation\Response;

$response = new Response();
$response->setContent(sprintf('Selamat Datang, %s', $request
->get('nama')));

$response->send();
```

Pada baris `$request->get('nama')` ini mengambil parameter dari *request* yaitu `nama`. Sehingga bila kita membuka halaman *web* misal `http://localhost:8000/greeting?nama=Surya`, maka `$request->get('nama')` akan mengembalikan nilai `Surya`.

Karena pada *file* `hello.php` dan `greeting.php` sama-sama menggunakan *variable* `$response`, maka kita dapat memindahkannya ke `index.php` agar lebih simpel dan mudah dibaca. Sehingga *file-file* tersebut akan menjadi sebagai berikut:

```
<?php
//filename: index.php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();
$path = $request->getPathInfo();

$response = new Response();

$route = ['/hello' => 'hello.php', '/greeting' => 'greeting.
php'];
```

```
if (isset($route[$path])) {
    include $route[$path];
}

$response->send();
```

```
<?php
//filename: hello.php

$response->setContent('Hello World');
```

```
<?php
//filename: greeting.php

$response->setContent(sprintf('Selamat Datang, %s', $request
->get('nama')));
```

Bagaimana menjadi lebih simpel bukan? Kita juga dapat menambahkan *response* jika `404` jika ternyata halaman yang di-*request* tidak ditemukan.

```
<?php
//filename: index.php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();
$path = $request->getPathInfo();
```

```
$response = new Response();

$route = ['/hello' => 'hello.php', '/greeting' => 'greeting.
php'];

if (isset($route[$path])) {
    include $route[$path];
} else {
    $response->setContent('Halaman tidak ditemukan');
    $response->setStatusCode(Response::HTTP_NOT_FOUND);
}

$response->send();
```

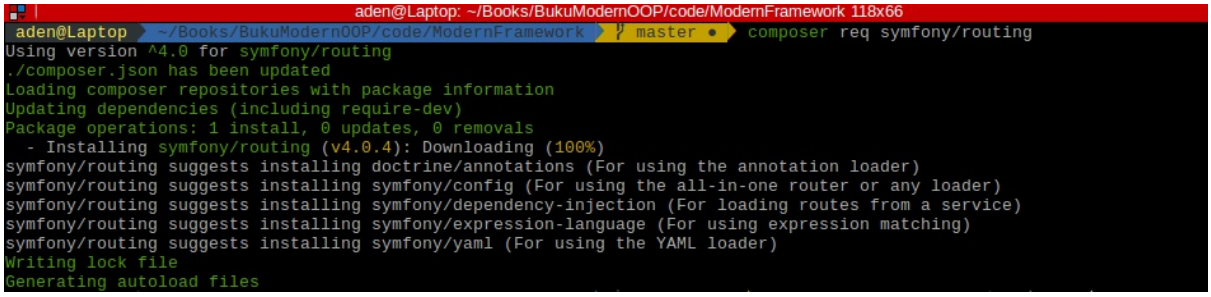
Sampai disini pembahasan kita tentang *request* dan *response* pada *framework* kita sebagai implementasi dari HTTP *Request* dan HTTP *Response*. Untuk pemahaman lebih mendalam tentang komponen Symfony HTTP *Foundation*, Anda dapat membacanya melalui *link* ini.

Mengarahkan *Request* dengan *Router*

Pada *framework modern* seperti Symfony, Zend, dan Laravel, *routing system* adalah salah satu komponen penting. Sama halnya dengan *framework-framework* tersebut, *framework* yang kita

bangun pun akan menerapkan *routing system* untuk mengarahkan *request* menuju halaman yang akan memproses *request* tersebut hingga menghasilkan *response*.

Untuk menerapkan *routing system* kita perlu meng-*install package* `symfony/routing` dengan menjalankan perintah `composer req symfony/routing` dari *root project* sebagaimana gambar berikut:



Sebelum kita menggunakan *routing system* pada *framework* yang kita buat, ada baiknya kita pahami terlebih dahulu bagaimana Symfony *routing* bekerja. Perhatikan contoh berikut:

```
<?php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\Routing\Matcher\UrlMatcher;

$request = Request::createFromGlobals();
$routes = new RouteCollection();

$routes->add('hello', new Route('/hello'));
$routes->add('greeting', new Route('/greeting/{nama}', ['nam
a' => 'Surya']));
```

```
$context = new RequestContext();
$context->fromRequest($request);

$matcher = new UrlMatcher($routes, $context);

print_r($matcher->match('/hello'));
/**
 * Array
 * (
 *     [_route] => hello
 * )
 */

print_r($matcher->match('/greeting'));
/**
 * Array
 * (
 *     [nama] => Surya
 *     [_route] => greeting
 * )
 */

print_r($matcher->match('/greeting/Ihsan'));
/**
 * Array
 * (
 *     [nama] => Ihsan
 *     [_route] => greeting
 * )
 */
```

Dari contoh diatas terlihat bahwa ketika kita melakukan *matching* terhadap *path request* maka Symfony *routing* akan mengembalikan sebuah *array* dengan indeks `_route` untuk nama dari *route* yang sesuai dan *route param* jika *route* tersebut memiliki parameter.

Setelah kita memahami bagaimana Symfony *routing* bekerja, selanjutnya kita ubah *file* `index.php` untuk mengimplementasikan *routing system* sebagai berikut:

```
<?php
//filename: index.php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\Exception\ResourceNotFoundExce
ption;

$request = Request::createFromGlobals();
$routes = new RouteCollection();

$routes->add('hello', new Route('/hello'));
$routes->add('greeting', new Route('/greeting/{nama}', ['nam
a' => 'Surya']));

$context = new RequestContext();
$context->fromRequest($request);

$matcher = new UrlMatcher($routes, $context);

try {
    $response = new Response();

    extract($matcher->match($request->getPathInfo()));

    include sprintf('%s.php', $_route);
} catch (ResourceNotFoundException $e) {
```

```
    $response = new Response('Halaman tidak ditemukan', Resp
onse::HTTP_NOT_FOUND);
}

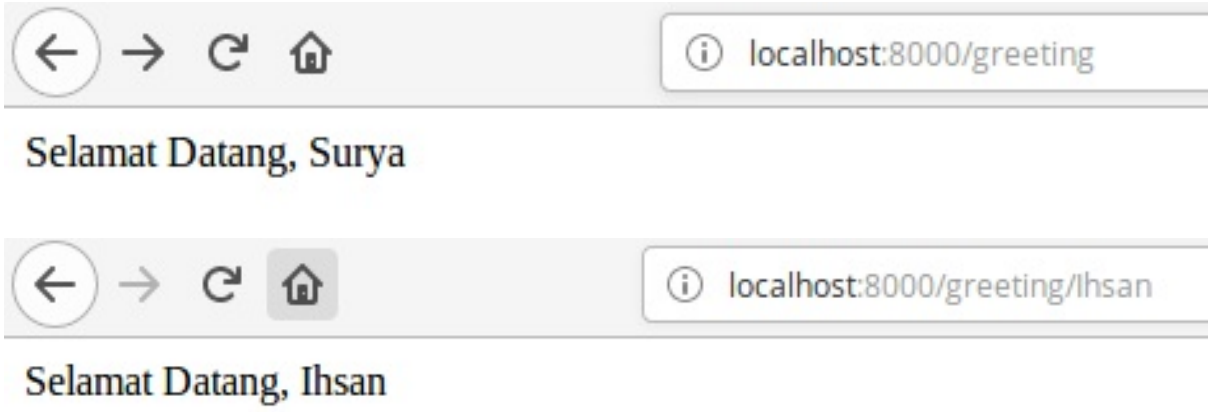
$response->send();
```

Kita juga perlu mengubah *file* `greeting.php` menjadi seperti berikut:

```
<?php
//filename: greeting.php

$response->setContent(sprintf('Selamat Datang, %s', $nama));
```

Dengan *code* diatas, ketika kita membuka URL `http://localhost:8000/greeting` maka `$nama` akan berisi `Surya` sebagai *default route param*. Namun jika kita membuka URL `http://localhost:8000/greeting/Ihsan` maka `$nama` akan berisi `Ihsan` seperti terlihat pada gambar berikut:



Pada *code* diatas, jika *route* semakin banyak maka *file* `index.php` akan semakin panjang sehingga kita perlu memindahkan *routing* dan membuat *file* sendiri untuk mendefinisikan *routing* tersebut.

Saya membuat *file* `config/routes.php` sejajar `index.php` dan memindahkan *routing* ke dalam *file* tersebut.

```
<?php
//filename: config/routes.php

use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$routes = new RouteCollection();

$routes->add('hello', new Route('/hello'));
$routes->add('greeting', new Route('/greeting/{nama}', ['nam
a' => 'Surya']));
```

Untuk memanggil *file* tersebut, saya mengubah *file* `index.php` sebagai berikut:

```
<?php
//filename: index.php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\Exception\ResourceNotFoundExce
ption;

$request = Request::createFromGlobals();

include __DIR__.'/config/routes.php';
```

```
$context = new RequestContext();
$context->fromRequest($request);

$matcher = new UrlMatcher($routes, $context);

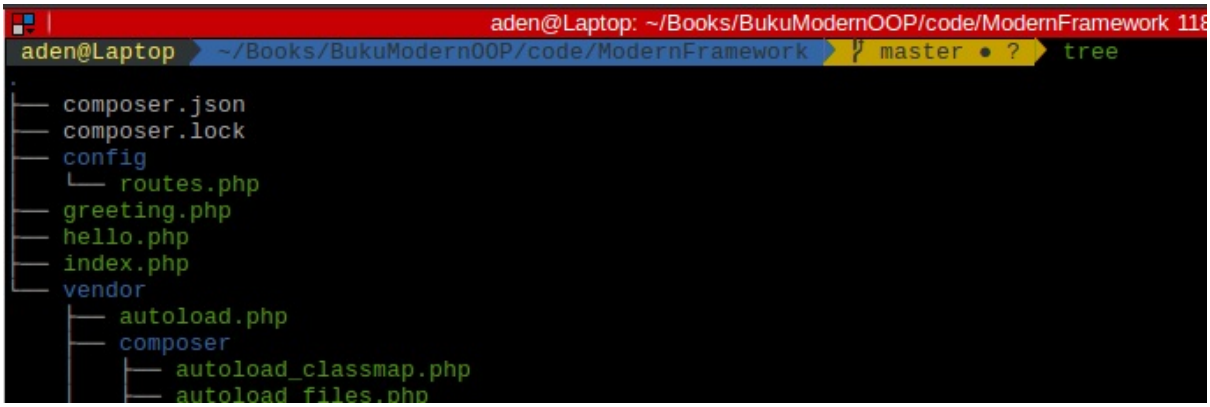
try {
    $response = new Response();

    extract($matcher->match($request->getPathInfo()));

    include sprintf('%s.php', $_route);
} catch (ResourceNotFoundException $e) {
    $response = new Response('Halaman tidak ditemukan', Resp
onse::HTTP_NOT_FOUND);
}

$response->send();
```

Sehingga susunan *folder* kita sekarang menjadi seperti berikut:



Untuk mengetahui secara lebih mendalam tentang Symfony *routing*, Anda dapat membaca dokumentasi resminya pada *link* ini.

Membuat *Kernel Framework*

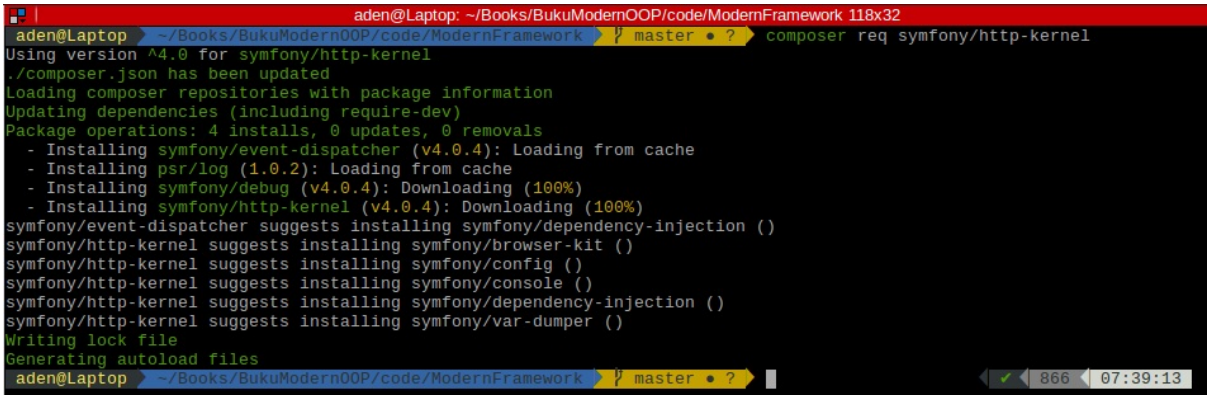
Sejatinya *file* `index.php` hanya dijadikan sebagai *front controller* tanpa ada *logic* disana. Untuk itu kita perlu memindahkan *logic* yang ada pada *file* `index.php` ke dalam *file* tersendiri. *File* inilah yang nantinya akan mengarahkan *request*, mendelegasikan *request* ke *controller* hingga menerima dan mengembalikan *response* ke klien.

Oleh karena itu saya membuat *folder* `src` dan membuat *namespace* `ModernFramework` sebagai *root namespace* serta mendaftarkan *namespace* tersebut ke `composer.json` agar dapat diregistrasi ke *composer autoloader*.

```
{
    "require": {
        "symfony/http-foundation": "^4.0",
        "symfony/routing": "^4.0"
    },
    "autoload": {
        "psr-4": {
            "ModernFramework\\": "src/"
        }
    }
}
```

Untuk meng-*update composer autoloader* kita perlu menjalankan perintah `composer dump-autoload` terlebih dahulu.

Untuk membuat *kernel* kita membutuhkan *package* `symfony/http-kernel` dan untuk meng-*install*-nya kita cukup menjalankan perintah `composer req symfony/http-kernel` dari *root project* seperti gambar dibawah ini:



Setelah itu kita membuat *file* `Application.php` di dalam *folder* `src` dengan isi sebagai berikut:

```
<?php
//filename: src/Application.php

namespace ModernFramework;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\Routing\Exception\ResourceNotFoundExce
ption;
use Symfony\Component\Routing\Matcher\UrlMatcher;

class Application implements HttpKernelInterface
{
    public function handle(Request $request, $type = self::M
ASTER_REQUEST, $catch = true)
    {
        include __DIR__.'/../config/routes.php';

        $context = new RequestContext();
        $context->fromRequest($request);

        $matcher = new UrlMatcher($routes, $context);

        try {
```

```
            $response = new Response();

            extract($matcher->match($request->getPathInfo())
);

            include sprintf('%s/../%s.php', __DIR__, $_route
);

            return $response;
        } catch (ResourceNotFoundException $e) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }
    }
}
```

Dan kemudian kita perlu mengubah *file* `index.php` untuk menyesuaikan dengan perubahan tersebut.

```
<?php
//filename: index.php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use ModernFramework\Application;

$request = Request::createFromGlobals();

$kernel = new Application();

$response = $kernel->handle($request);
$response->send();
```

Bagaimana jadi tampak lebih simpel *kan file* `index.php` yang kita miliki? Selanjutnya kita akan membuat *controller* untuk menggantikan *file* `greeting.php` dan `hello.php`.

Membuat *Controller Class*

Sejauh ini *framework* yang kita buat telah menerapkan *kernel* sehingga *file* `index.php` kita menjadi lebih simpel. Namun semua itu masih sangat sederhana. Kita perlu memisahkan *logic* dan membuat *controller* untuk memproses *request* dan memberikan *response*.

Di dalam *folder* `src` kita membuat *folder* baru yaitu `Controller` dan membuat *file controller* `HelloController.php` sebagai berikut:

```
<?php
//filename: src/Controller/HelloController.php

namespace ModernFramework\Controller;

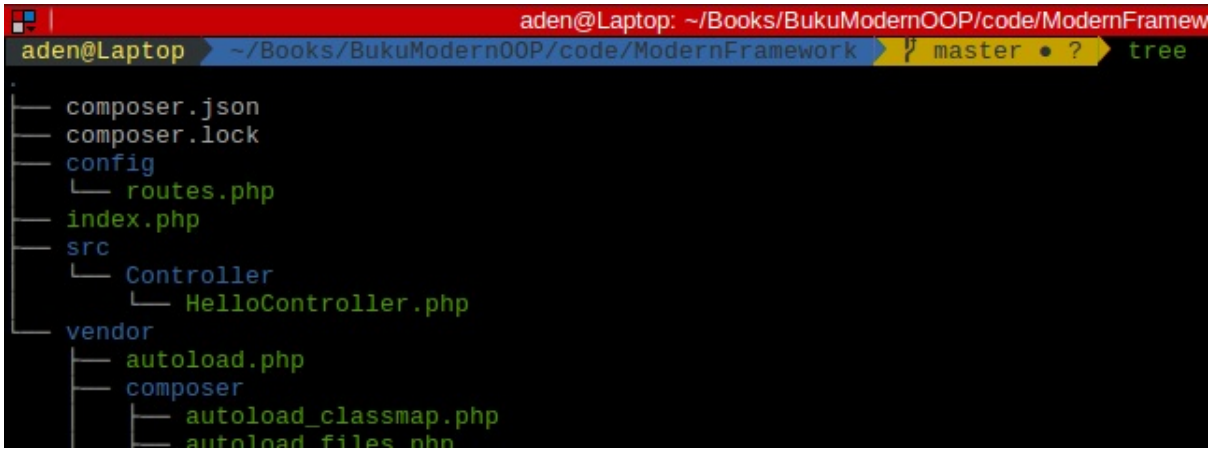
use Symfony\Component\HttpFoundation\Response;

class HelloController
{
    public function hello()
    {
        return new Response('Hello World');
    }

    public function greet($nama)
    {
        return new Response(sprintf('Selamat Datang, %s', $n
ama));
```

```
    }
}
```

Sampai disini *controller* kita belum dapat digunakan, namun kita sudah bisa menghapus *file* `hello.php` dan `greeting.php` yang berada pada *root project* sehingga sususan *folder* kita menjadi sebagai berikut:



Setiap *action* atau *method* pada *controller* harus mengembalikan *object* `Response` bila tidak maka akan terjadi *error*. Dan agar kita dapat mengarahkan *request* ke *controller* maka kita perlu mengubah *routing* yang telah kita buat dengan menambahkan spesial indeks `_controller` sebagai berikut:

```
<?php
//filename: config/routes.php

use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$routes = new RouteCollection();

$routes->add('hello', new Route('/hello', [
    '_controller' => 'ModernFramework\Controller\HelloContro
ller::hello',
```

```
]));
$routes->add('greeting', new Route('/greeting/{nama}', [
    'nama' => 'Surya',
    '_controller' => 'ModernFramework\Controller\HelloContro
ller::greet',
]));
```

Dengan penambahan *code* diatas, maka kita dapat menggunakan *controller resolver* dan *argument resolver* untuk mendapatkan *object controller* serta parameter dari *controller* tersebut. Kita menambahkan *controller resolver* dan *argument resolver* pada *file* `Application.php` sebagai berikut:

```
<?php
//filename: src/Application.php

namespace ModernFramework;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Controller\ControllerResolv
er;
use Symfony\Component\HttpKernel\Controller\ArgumentResolver
;
use Symfony\Component\Routing\Exception\ResourceNotFoundExce
ption;
use Symfony\Component\Routing\Matcher\UrlMatcher;

class Application implements HttpKernelInterface
{
    public function handle(Request $request, $type = self::M
ASTER_REQUEST, $catch = true)
    {
        include __DIR__.'/../config/routes.php';
```

```
        $context = new RequestContext();
        $context->fromRequest($request);

        $matcher = new UrlMatcher($routes, $context);

        $controllerResolver = ControllerResolver();
        $argumentResolver = ArgumentResolver();

        try {
            $request->attributes->add($matcher->match($reque
st->getPathInfo()));

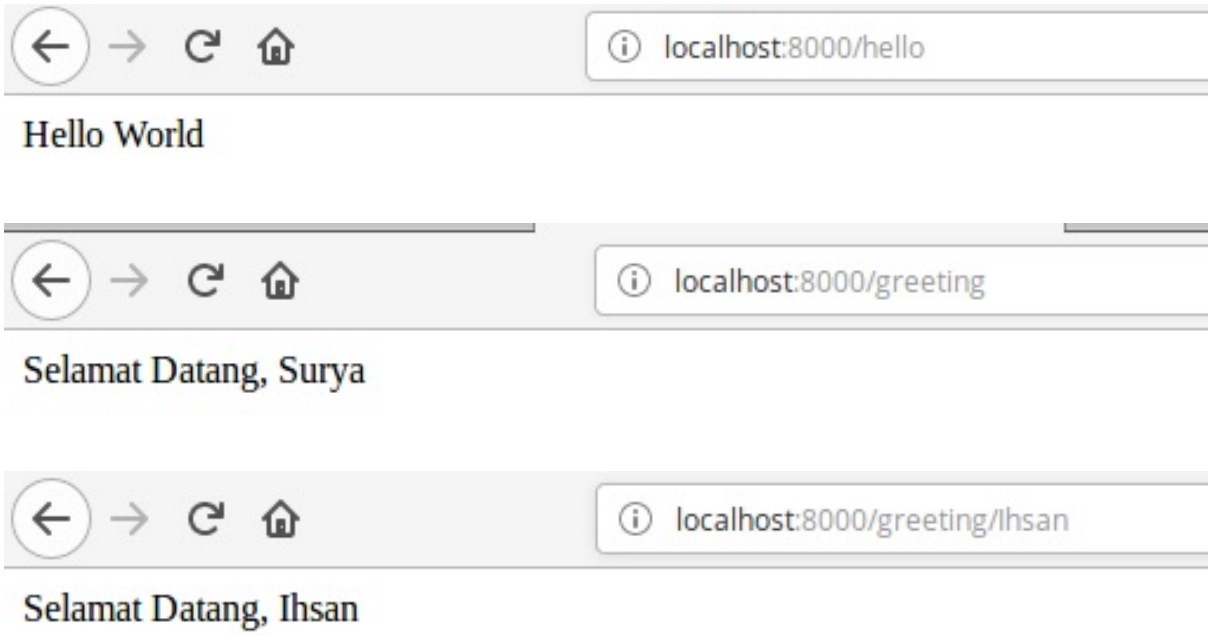
            $controller = $controllerResolver->getController
($request);
            $arguments = $argumentResolver->getArguments($re
quest, $controller);

            return call_user_func_array($controller, $argume
nts);
        } catch (ResourceNotFoundException $e) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }
    }
}
```

Sedikit penjelasan dari perubahan *code* diatas, pada baris *code* `$request->attributes->add($matcher->match($request->getPathInfo()))` kita memasukkan hasil dari `$matcher->match($request->getPathInfo())` menjadi *attribute* dari *request* agar dapat dipanggil menggunakan `$request->get()` maupun `$request->attributes->get()`. Hal tersebut diperlukan karena kita

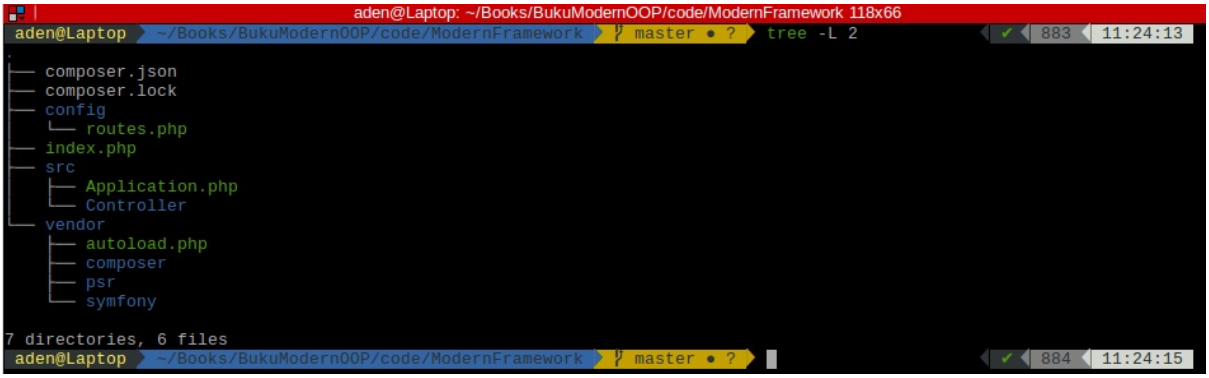
menggunakan *controller resolver* dan *argument resolver* untuk mendapatkan *controller* dan parameter dari *method* atau *action* pada *controller*.

Dengan perubahan *file* `Application.php` seperti diatas, maka setiap *request* akan diarahkan ke *controller* sehingga ketika kita membuka *browser*, maka hasilnya adalah sebagai berikut:



Kesimpulan

Sampai disini berarti pembahasan kita tentang cara membuat *framework* sederhana telah usai. Untuk membuat sebuah *framework* yang sederhana ternyata tidak terlalu sulit, kita hanya memerlukan 3 *package* dari komponen Symfony, dengan susunan *folder* terakhir adalah sebagai berikut:



Meski sangat sederhana, namun kita dapat belajar tentang bagaimana sebuah *request* masuk, diarahkan ke *controller* untuk kemudian diproses oleh *controller* hingga mengembalikan *response*. Pada pembahasan selanjutnya, kita akan mengembangkan *framework* yang telah kita buat untuk dapat berinteraksi dengan *database*.

XXXV. Studi Kasus *Todo List* Menggunakan OOP dan MVC

Pembahasan kali ini adalah pengembangan dari pembahasan sebelumnya. Pada pembahasan kali ini, kita akan mengembangkan *framework* yang telah kita buat pada pembahasan sebelumnya dengan menambahkan fitur koneksi *database* dan *template engine*.

Saya meng-*copy* seluruh *code* pada pembahasan sebelumnya dan mengganti nama *folder*-nya menjadi `Todo` agar membedakan antara *code* sebelum dan setelah penambahan fitur *database* dan *template engine*.

Tujuan dari Proyek *Todo List*

Tujuan utama dari pembuatan aplikasi *todo list* ini adalah memahami konsep MVC (*Model* - *View* - *Controller*). Selain tujuan utama tersebut, pada pembahasan kali ini kita juga akan belajar bagaimana menambahkan fitur koneksi *database* serta *template engine* pada *framework* yang kita buat.

Pembuatan *Database*

Pada aplikasi *todo list* ini kita hanya akan menggunakan 1 tabel *database* yaitu `todo` dengan kolom `id`, `activity`, dan `is_done`. Kita akan menggunakan *database engine* SQLite namun Anda dapat menggunakan MySQL atau MariaDB atau RDBMS apapun yang didukung oleh PDO (PHP Data Objects).

Saya menggunakan SQLite hanya karena bagi saya SQLite sangat ringan dan *portable* sehingga cocok untuk proses pengembangan aplikasi. Sementara untuk kebutuhan produksi, penggunaan SQLite sangat tidak dianjurkan.

Untuk SQL dari tabel `todo` adalah sebagai berikut:

```
CREATE TABLE todo
(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    activity VARCHAR(255) NOT NULL,
    is_done INTEGER DEFAULT 0
);
```

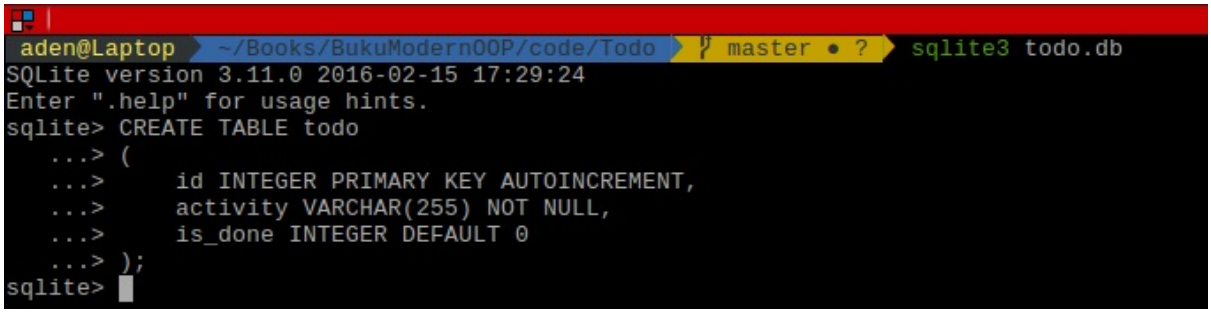
Untuk membuat *database* dengan SQLite, kita cukup membuat *file* `todo.db` pada *root project* dan kemudian menjalankan perintah melalui *command line* atau *command prompt* sebagai berikut:

```
sqlite3 todo.db
```

Maka akan muncul *shell* SQLite sebagai berikut:

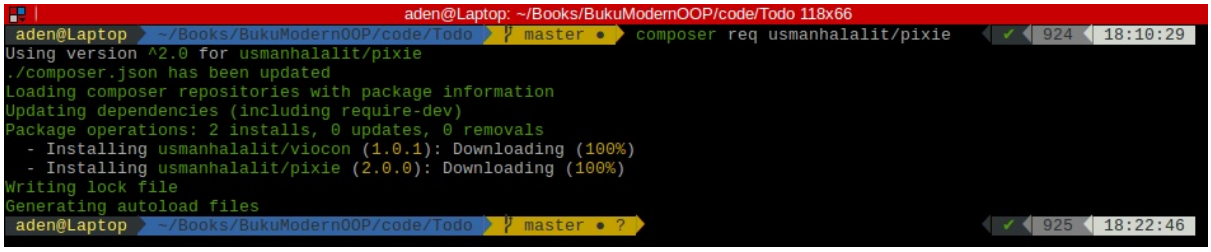


Kita kemudian mengetikkan perintah SQL diatas dan kemudian tekan *enter* untuk menjalankan seperti pada gambar berikut:



Koneksi *Database*

Setelah membuat *database*, tahap selanjutnya adalah membuat koneksi antara *database* dan aplikasi. Untuk membuat koneksi saya menggunakan *package* `pixie` dan untuk meng-*install*-nya kita cukup mengetikkan `composer req usmanhalalit/pixie` dari *root project* seperti gambar berikut:



Dan kemudian kita membuat *file* `Database.php` didalam *folder* `src/Util` dengan isi sebagai berikut:

```
<?php
//filename: src/Util/Database.php

namespace ModernFramework\Util;

use Pixie\Connection;
use Pixie\QueryBuilder\QueryBuilderHandler;
```

```
class Database
{
    private static $connection;

    private static $pdo;

    private function __construct($host, $database, $driver =
'mysql', $username = 'root', $password = null, $port = 3306,
 $charset = 'utf8')
    {
        $config = [
            'driver' => $driver,
            'host' => $host,
            'database' => $database,
            'username' => $username,
            'password' => $password,
            'charset' => $charset,
            'port' => $port,
        ];

        static::$connection = new Connection($driver, $confi
g);
        static::$pdo = static::$connection->getPdoInstance()
;
    }

    public static function connect($host, $database, $driver
 = 'mysql', $username = 'root', $password = null, $port = 33
06, $charset = 'utf8')
    {
        return new static($host, $database, $driver, $userna
me, $password, $port, $charset);
    }

    public function execute(string $query, array $parameters)

    {
```

```
        $statement = static::$pdo->prepare($query);

        foreach ($parameters as $parameter => $value) {
            $statement->bindValue(sprintf(':%s', $parameter)
, $value);
        }

        $statement->execute();
    }

    public function createQueryBuilder()
    {
        return new QueryBuilderHandler(static::$connection);
    }
}
```

Setelah itu kita harus membuat konfigurasi *database* pada *file* `config/database.php` dengan isi sebagai berikut:

```
<?php
//filename: config/database.php

use ModernFramework\Util\Database;

$database = Database::connect('localhost', 'todo.db', 'sqlit
e');
```

Tahap terakhir kita ubah `Application.php` agar menge-*load file* `config/database.php` sebagai berikut:

```
<?php
//filename: src/Application.php

namespace ModernFramework;
```

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Controller\ControllerResolv
er;
use Symfony\Component\HttpKernel\Controller\ArgumentResolver
;
use Symfony\Component\Routing\Exception\ResourceNotFoundExce
ption;
use Symfony\Component\Routing\Matcher\UrlMatcher;

class Application implements HttpKernelInterface
{
    public function handle(Request $request, $type = self::M
ASTER_REQUEST, $catch = true)
    {
        include __DIR__.'/../config/routes.php';
        include __DIR__.'/../config/database.php';

        $context = new RequestContext();
        $context->fromRequest($request);

        $matcher = new UrlMatcher($routes, $context);

        $controllerResolver = new ControllerResolver();
        $argumentResolver = new ArgumentResolver();

        try {
            $request->attributes->add($matcher->match($reque
st->getPathInfo()));

            $controller = $controllerResolver->getController
($request);
            $arguments = $argumentResolver->getArguments($re
quest, $controller);
```

```
            return call_user_func_array($controller, $argume
nts);
        } catch (ResourceNotFoundException $e) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }
    }
}
```

Sampai disini kita telah selesai membuat koneksi antara *database* dan aplikasi. Tahap selanjutnya adalah membuat model agar dapat berinteraksi dengan *database*.

Membuat *Model Class*

Setelah selesai membuat koneksi *database*, maka tahap selanjutnya adalah membuat *class* `Model` sebagai dasar dari semua model yang akan kita buat. *Class* `Model` ini adalah sebuah *abstract class* sehingga wajib di-*extends* oleh *child class*.

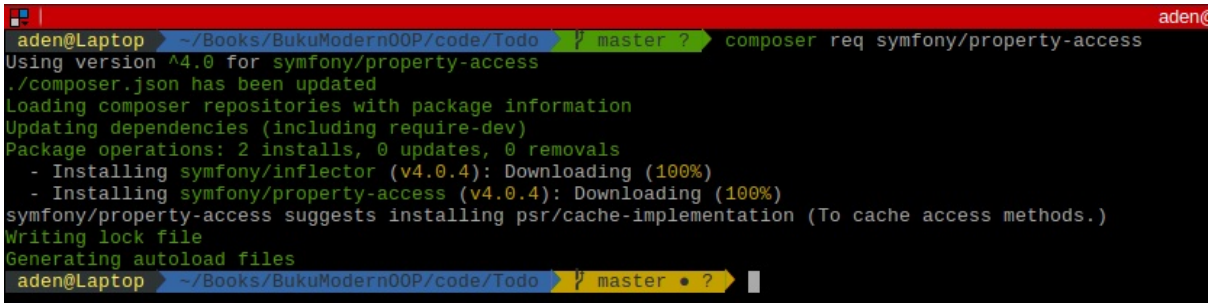
Pada *class* `Model` kita akan menerapkan *active record pattern* seperti yang digunakan juga oleh *framework* Laravel. Bila Anda pengguna *framework* Laravel, mungkin Anda tidak asing dengan *syntax* berikut:

```
$object->save();
```

Pada model, kita akan juga akan menerapkan *syntax* tersebut. Hal ini bertujuan agar Anda juga memahami bagaimana *framework* seperti Laravel dan lainnya bekerja untuk menangani operasi

database. Namun tentu saja apa yang kita buat ini masih sangat sederhana dan perlu pengembangan lebih lanjut.

Untuk membuat *class* `Model` kita perlu menambahkan terlebih dahulu *package* `symfony/property-access` sebagai *package* utilitas. Untuk meng-*install* *package* tersebut, kita cukup mengetikkan perintah `composer req symfony/property-access` seperti gambar berikut:



Setelah instalasi berhasil, selanjutnya kita membuat *class* `Model` sebagai berikut:

```
<?php
//filename: src/Model/Todo.php

namespace ModernFramework\Model;

use ModernFramework\Util\Database;
use Symfony\Component\PropertyAccess\PropertyAccess;

abstract class Model
{
    private $connection;

    abstract public function getId(): ? int;

    public function __construct(Database $connection)
    {
        $this->connection = $connection;
```

```
    }

    public function save(): void
    {
        $class = new \ReflectionClass(get_class($this));

        $table = strtolower($class->getShortName());
        $properties = array_filter($class->getProperties(\Re
flectionProperty::IS_PRIVATE|\ReflectionProperty::IS_PROTECT
ED), function ($property) {
            return 'id' === $property ? false : true;
        });

        $columns = array_map(function($property) {
           return $this->toUnderScore($property->getName());

        }, $properties);

        $accessor = PropertyAccess::createPropertyAccessor()
;
        $parameters = [];
        foreach ($columns as $key => $property) {
            $parameters[$property] = $accessor->getValue($th
is, $property);
        }

        if ($this->getId()) {
            $this->update($table, $columns, $parameters);
        } else {
            $this->insert($table, $columns, $parameters);
        }
    }

    public function delete()
    {
        if ($id = $this->getId()) {
            $class = new \ReflectionClass(get_class($this));
            $table = strtolower($class->getShortName());
```

```
            $this->connection->execute(sprintf('DELETE FROM
%s WHERE id = :id', $table), ['id' => $id]);
        }
    }

    public function find(int $id): ? self
    {
        $class = new \ReflectionClass(get_class($this));
        $table = strtolower($class->getShortName());

        $result = $this->connection->createQueryBuilder()->f
rom($table)->find($id);
        if (!$result) {
            return $result;
        }

        return $this->normalize($result);
    }

    public function findAll(): array
    {
        $class = new \ReflectionClass(get_class($this));
        $table = strtolower($class->getShortName());

        $results = $this->connection->createQueryBuilder()->
from($table)->get();
        foreach ($results as $key => $result) {
            $results[$key] = $this->normalize($result);
        }

        return $results;
    }

    private function insert(string $table, array $columns, a
rray $values): void
    {
        $parameters = array_map(function($column) {
```

```
            return sprintf(':%s', $column);
        }, $columns);

        $this->connection->execute(sprintf('INSERT INTO %s(%
s) VALUES (%s)', $table, implode(',', $columns), implode(','
, $parameters)), $values);
    }

    private function update(string $table, array $columns, a
rray $values): void
    {
        $parameters = array_map(function($column) {
            return sprintf('%s = :%s', $column, $column);
        }, $columns);

        $this->connection->execute(sprintf('UPDATE %s SET %s
 WHERE id = :id', $table, implode(',', $parameters)), $value
s);
    }

    private function toUnderScore(string $column): string
    {
        return strtolower(preg_replace('/([a-z])([A-Z])/', '
$1_$2', str_replace(' ', '_', $column)));
    }

    private function normalize(\stdClass $data): Model
    {
        $clone = clone $this;
        $accessor = PropertyAccess::createPropertyAccessor()
;
        foreach (json_decode(json_encode($data), true) as $p
roperty => $value) {
            $accessor->setValue($clone, $property, $value);
        }

        return $clone;
    }
```

```
}
```

Dengan meng-*extends class* `Model` diatas, maka *child class* akan dapat menggunakan *syntax* berikut:

```
//Untuk operasi insert dan update
$object->save();

//Untuk operasi delete
$object->delete();

//Untuk mendapatkan data berdasarkan id
$object->find($id);

//Untuk mendapatkan semua data
$object->findAll();
```

Membuat *Todo Class*

Setelah membuat *class* `Model`, selanjutnya kita perlu membuat *class* `Todo` yang meng-*extends class* `Model`. *Class* inilah yang nantinya akan berinteraksi dengan *database* secara langsung.

Class `Todo` merepresentasikan tabel `todo` di *database* sehingga nama *property*-nya adalah nama kolom pada tabel *database*. Dan berikut adalah *class* `Todo` tersebut:

```
<?php
//filename: src/Model/Todo.php

namespace ModernFramework\Model;
```

```
use ModernFramework\Util\Database;

class Todo extends Model
{
    const DONE = 1;

    const TODO = 0;

    private $id;

    private $activity;

    private $isDone;

    public function __construct(Database $connection)
    {
        parent::__construct($connection);
        $this->isDone = self::TODO;
    }

    public function getId(): ? int
    {
        return $this->id;
    }

    public function setId(string $id): void
    {
        $this->id = (int) $id;
    }

    public function getActivity(): string
    {
        return $this->activity;
    }

    public function setActivity(string $activity): void
    {
        $this->activity = $activity;
```

```
    }

    public function isDone(): bool
    {
        return (bool) $this->isDone;
    }

    public function done(): void
    {
        $this->isDone = self::DONE;
    }

    public function setIsDone(string $done): void
    {
        $this->isDone = (int) $done === self::DONE ? self::D
ONE : self::TODO;
    }
}
```

Karena ketika diambil dari *database* semua tipe data menjadi `string` maka baik `$id` maupun `$done` harus kita konversi dahulu menjadi `integer` agar sesuai dengan kebutuhan kita.

Sampai disini kita telah selesai membuat *class* model untuk tabel `todo`. Tahap selanjutnya adalah pembuatan *controller* dan dilanjutkan dengan pembautan *template*.

Membuat *Controller Class*

Berbeda dengan *controller* pada pembahasan sebelumnya, pada pembahasan kali ini, *controller* yang akan kita buat akan memiliki *parent class* seperti halnya pada model. Hal ini dilakukan agar

kompleksitas dapat dipecah dan *parent class* dapat digunakan kembali pada *controller-controller* yang lainnya.

Class controller yang akan kita buat bersifat *abstract* sama seperti halnya *class* `Model` dan nantinya pada *class controller* inilah kita memasukkan model agar dapat digunakan oleh *controller*. Dan berikut adalah *class controller* tersebut:

```
<?php
//filename: src/Controller/Controller.php

namespace ModernFramework\Controller;

use ModernFramework\Model\Model;
use ModernFramework\Util\Database;
use Symfony\Component\HttpFoundation\Response;

abstract class Controller
{
    private $connection;

    public function setConnection(Database $connection): void

    {
        $this->connection = $connection;
    }

    protected function getConnection(): Database
    {
        return $this->connection;
    }

    protected function getModel(string $model): Model
    {
        return new $model($this->connection);
    }
```

```
    protected function render(string $view): Response
    {
        return new Response($view);
    }
}
```

Untuk dapat menyesuaikan dengan *class controller* diatas, kita perlu mengubah *file* `Application.php` dengan menambahkan *code* `$controller->setConnection($database)` sebelum pemanggilan *controller* untuk memasukkan koneksi ke *controller*. Sehingga *code* pada *file* `Application.php` menjadi seperti dibawah ini:

```
<?php
//filename: src/Application.php

namespace ModernFramework;

use ModernFramework\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Controller\ControllerResolv
er;
use Symfony\Component\HttpKernel\Controller\ArgumentResolver
;
use Symfony\Component\Routing\Exception\ResourceNotFoundExce
ption;
use Symfony\Component\Routing\Matcher\UrlMatcher;

class Application implements HttpKernelInterface
{
    const BASE_PATH = __DIR__.'/../';
```

```
    public function handle(Request $request, $type = self::M
ASTER_REQUEST, $catch = true)
    {
        include self::BASE_PATH.'config/routes.php';
        include self::BASE_PATH.'config/database.php';

        $context = new RequestContext();
        $context->fromRequest($request);

        $matcher = new UrlMatcher($routes, $context);

        $controllerResolver = new ControllerResolver();
        $argumentResolver = new ArgumentResolver();

        try {
            $request->attributes->add($matcher->match($reque
st->getPathInfo()));

            $controller = $controllerResolver->getController
($request);
            $arguments = $argumentResolver->getArguments($re
quest, $controller);

            if ($controller instanceof Controller) {
                $controller->setConnection($database);
            }

            return call_user_func_array($controller, $argume
nts);
        } catch (ResourceNotFoundException $e) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }
    }
}
```

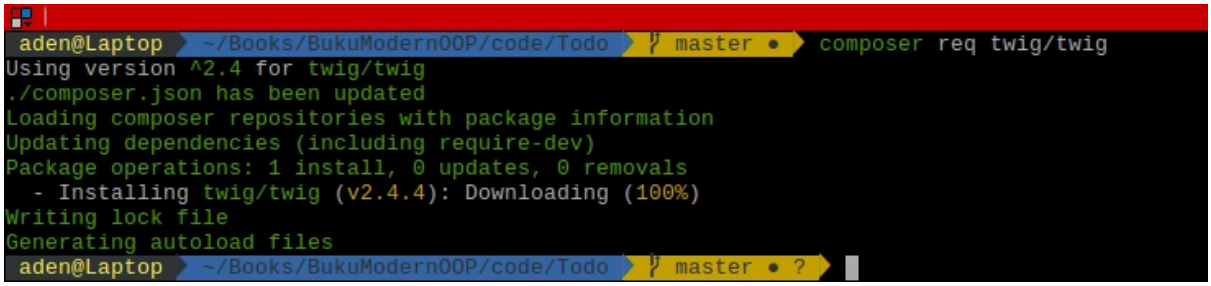
Dengan cara diatas, maka setiap *controller* yang kita buat secara otomatis memiliki koneksi *database*. Hal ini akan memudahkan kita ketika berinteraksi dengan model.

Menambahkan *Template Engine* pada *Framework*

Agar *code* yang kita tulis menjadi lebih bersih dan lebih mudah dibaca serta tidak bercampur antara *file controller* dan model dengan *file view* maka kita perlu menambahkan *template engine* pada *framework* kita. Selain berfungsi untuk memisahkan antara presentasi dan *logic*, *template engine* juga berguna untuk memudahkan kolaborasi dengan *frontend developer*.

Pada *framework* yang kita buat, untuk *template engine* kita akan menggunakan Twig. Saya memilih Twig karena Twig di-*compile* menjadi *plain* PHP *code* sehingga lebih cepat. Selain itu Twig aman karena menggunakan *auto escaping* serta mudah digunakan.

Untuk meng-*install* Twig kita cukup mengetikkan perintah `composer req twig/twig` seperti gambar berikut:



Setelah selesai melakukan instalasi, selanjutnya kita buat *class* `View` yang akan menggunakan Twig sebagai *template engine*. Isi dari *class* `View` tersebut adalah sebagai berikut:

```
<?php
//filename: src/View/View.php

namespace ModernFramework\View;

use Symfony\Component\HttpFoundation\Response;

class View
{
    private $twig;

    public function __construct(string $templatePath, string
 $cachePath = null)
    {
        $this->twig = new \Twig_Environment(
            new \Twig_Loader_Filesystem($templatePath),
            ['cache' => null !== $cachePath ? $cachePath : f
alse]
        );
    }

    public function render(string $template, array $variable
s = [])
    {
        return new Response($this->twig->render($template, $
variables));
    }
}
```

Kita membutuhkan *template path* sebagai *base directory* dari *template*. Selain itu kita juga membutuhkan *cache path* sebagai tempat menyimpan *file* hasil *compile* dari Twig. Fitur *cache* ini sangat disarankan pada *environment production*, namun bila Anda tidak ingin menggunakannya, Anda dapat mengabaikannya.

Penggunaan *class* `View` ini nantinya sama seperti `$this->load->view()` pada CodeIgniter atau `$this->render()` pada Symfony ketika dipanggil dari *controller*. Untuk itu kita perlu mengubah *abstract class* `Controller` kita sebagai berikut:

```
<?php
//filename: src/Controller/Controller.php

namespace ModernFramework\Controller;

use ModernFramework\Application;
use ModernFramework\Model\Model;
use ModernFramework\View\View;
use ModernFramework\Util\Database;
use Symfony\Component\HttpFoundation\Response;

abstract class Controller
{
    private $connection;

    private $template;

    public function __construct()
    {
        $this->template = new View(Application::BASE_PATH.'t
emplate', Application::BASE_PATH.'cache');
    }

    public function setConnection(Database $connection): void
```

```
    {
        $this->connection = $connection;
    }

    protected function getConnection(): Database
    {
        return $this->connection;
    }

    protected function getModel(string $model): Model
    {
        return new $model($this->connection);
    }

    protected function render(string $template, array $varia
bles = []): Response
    {
        return $this->template->render($template, $variables
);
    }
}
```

Pada *root project* kita buat 2 *folder* yaitu *folder* `template` untuk tempat menyimpan *template* kita, serta `cache` sebagai tempat menyimpan *file* hasil *compile* dari Twig.

Ketika kita mengubah *template* maka kita harus menghapus semua isi pada *folder* `cache` agar kita dapat melihat perubahan yang telah kita buat.

Sampai tahap ini kita telah mengintegrasikan antara *controller* dengan model serta *controller* dengan *view*. Tahap selanjutnya adalah *todo controller* dan membuat operasi *CRUD* untuk tabel

`todo`. Bagaimana semakin menarik bukan?

Membuat *Todo Controller Class*

Tahap terakhir dari pembahasan kita adalah pembuatan *controller* `TodoController` yang akan menangani *request* dan mengembalikan *response* kepada klien. Pada *class* `TodoController` nantinya *model* dan *view* akan digunakan.

Pada tahap awal, kita akan menampilkan data seluruh *todo* dari *database* sebagai berikut:

```
<?php
//filename: src/Controller/TodoController.php

namespace ModernFramework\Controller;

use Symfony\Component\HttpFoundation\Response;

class TodoController extends Controller
{
    public function index()
    {
        return $this->render('index.html.twig', ['todos' =>
[]]);
    }
}
```

Action method `index()` diatas belum dapat menampilkan data yang ada di *database*, namun nantinya kit akan menampilkan semua data dari tabel `todo` pada *action method* `index()`.

Setelah membuat *action method*, kita harus membuat *template* `index.html.twig` pada *folder* `template` dengan isi sebagai berikut:

```
<!-- filename: template/index.html.twig -->
<table>
    <thead>
        <tr>
            <td>No</td>
            <td>Tugas</td>
            <td>Selesai?</td>
            <td>Pilihan</td>
        <tr>
    </thead>
    <tbody>
    {% for i, todo in todos %}
        <tr>
            <td>{{ (i + 1) }}</td>
            <td>{{ todo.activity }}</td>
            <td>{{ todo.isDone ? 'Selesai' : 'Belum' }}</td>
            <td>{% if false == todo.isDone %}<a href="/todo/
' ~ todo.id ~ '/done">Selesai</a> | {% endif %}<a href="/tod
o/{{ todo.id }}/edit">Edit</a> | <a href="/todo/{{ todo.id }
}/delete">Hapus</a></td>
        </tr>
    {% endfor %}
    </tbody>
</table>
<p><a href="/todo/new">Tambah</a></p>
```

Setelah itu kita harus mendaftarkan *action method* `index()` ke *route* agar dapat kita panggil menggunakan *browser*. Untuk mendaftarkan *route*, kita cukup menambahkan baris *code* berikut:

```
$routes->add('todo_index', new Route('/todo', [
```

```
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::index',
]));
```

Secara lengkap, *file* `routes.php` kita adalah sebagai berikut:

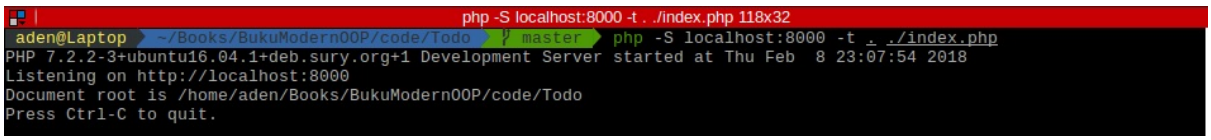
```
<?php
//filename: config/routes.php

use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

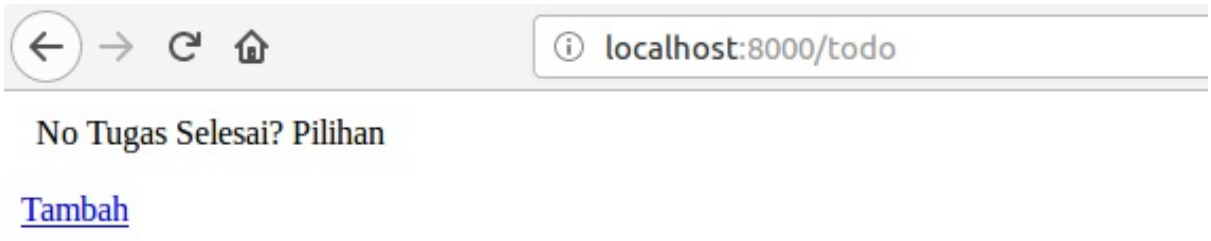
$routes = new RouteCollection();

$routes->add('todo_index', new Route('/todo', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::index',
]));
```

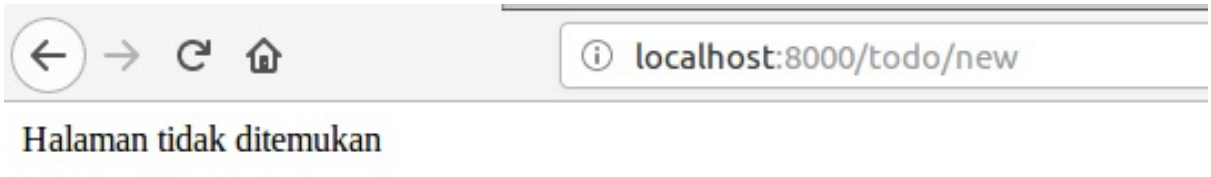
Selanjutnya kita tinggal menjalankan *built-in web server* dengan mengetikkan perintah `php -S localhost:8000 -t . ./index.php` dari *root project* seperti pada gambar berikut:



Dan kemudian kita dapat membuka *browser* dan mengetikkan alamat `http://localhost:8000/todo`, maka akan muncul halaman sebagai berikut:



Dari gambar terlihat tidak ada data yang ditampilkan, yang ada hanya *link* `Tambah` yang aktif. Kita dapat membuka *link* tersebut dengan mengekliknya maka akan muncul pesan `Halaman tidak ditemukan` seperti berikut:



Hal tersebut wajar karena kita belum membuat *action method* untuk *route* `/todo/new`. Agar halaman tersebut tidak muncul pesan tersebut, kita harus membuat *action method* `new()` sebagai berikut:

```
<?php
//filename: src/Controller/TodoController.php

namespace ModernFramework\Controller;

use ModernFramework\Model\Todo;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

class TodoController extends Controller
{
    public function index()
    {
        return $this->render('index.html.twig', ['todos' =>
```

```
[]]);
    }

    public function new(Request $request)
    {
        $activity = $request->get('activity');
        if ($activity) {
            $todo = $this->getModel(Todo::class);
            $todo->setActivity($activity);
            $todo->save();
        }

        return $this->render('new.html.twig');
    }
}
```

Kemudian kita membuat *file* `new.html.twig` dengan isi sebagai berikut:

```
<!-- filename: template/new.html.twig -->
<form action="/todo/new" method="POST">
    <input type="text" name="activity" />
    <button type="submit">Simpan</button>
</form>
```

Dan selanjutnya kita perlu mendaftarkan *action method* `new()` kedalam *route* sebagai berikut:

```
<?php
//filename: config/routes.php

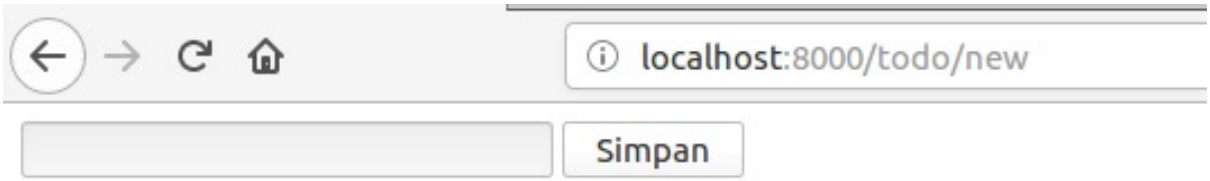
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$routes = new RouteCollection();
```

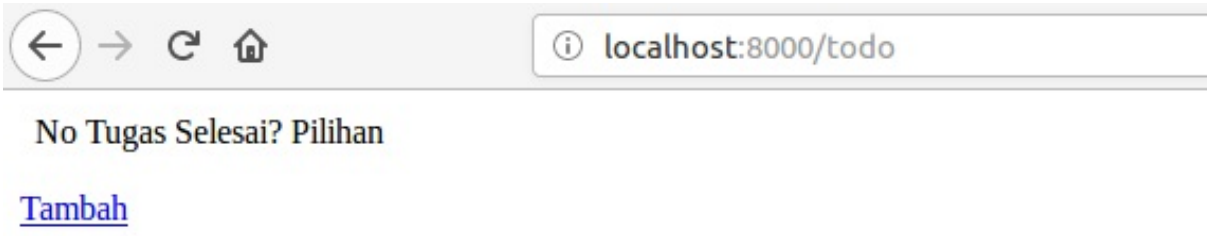
```
$routes->add('todo_index', new Route('/todo', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::index',
]));

$routes->add('todo_new', new Route('/todo/new', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::new',
]));
```

Kita dapat me-*refresh* kembali *browser* kita maka akan muncul halaman sebagai berikut:



Dan ketika kita mengisi *form* tersebut dan mengeklik tombol `Simpan` maka kan kembali ke halaman *list* seperti pada gambar berikut:



Tidak perlu khawatir karena memang pada *action method* `index()` kita belum memanggil data dari *database*. Agar data yang telah kita masukkan muncul, maka kita perlu mengubah *action method* `index()` menjadi sebagai berikut:

```
    public function index()
    {
        return $this->render('index.html.twig', ['todos' =>
$this->getModel(Todo::class)->findAll()]);
    }
```

Setelah kita *refresh* kembali halaman `/todo` maka data yang kita masukkan akan muncul seperti gambar berikut:



Terlihat *list todo* kita sudah muncul, selanjutnya kita klik *link* `Selesai` maka akan muncul `Halaman tidak ditemukan` sehingga kita perlu membuat *action method*-nya terlebih dahulu. *Action method* untuk menangani *link* `Selesai` tersebut adalah *action method* `done()` dengan isi sebagai berikut:

```
    public function done($id)
    {
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }

        $todo->done();

        return new RedirectResponse('/todo');
```

```
    }
```

Secara lengkap, *class* `TodoController` akan menjadi sebagai berikut:

```
<?php
//filename: src/Controller/TodoController.php

namespace ModernFramework\Controller;

use ModernFramework\Model\Todo;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\RedirectResponse;

class TodoController extends Controller
{
    public function index()
    {
        return $this->render('index.html.twig', ['todos' =>
$this->getModel(Todo::class)->findAll()]);
    }

    public function new(Request $request)
    {
        $activity = $request->get('activity');
        if ($activity) {
            $todo = $this->getModel(Todo::class);
            $todo->setActivity($activity);
            $todo->save();

            return new RedirectResponse('/todo');
        }

        return $this->render('new.html.twig');
    }
```

```
    public function done($id)
    {
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }

        $todo->done();
        $todo->save();

        return new RedirectResponse('/todo');
    }
}
```

Dan kemudian kita daftarkan *action method* tersebut kedalam *route* sebagai berikut:

```
<?php
//filename: config/routes.php

use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$routes = new RouteCollection();

$routes->add('todo_index', new Route('/todo', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::index',
]));

$routes->add('todo_new', new Route('/todo/new', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::new',
]));

$routes->add('todo_done', new Route('/todo/{id}/done', [
```

```
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::done',
]));
```

Setelah kita *refresh* kembali halaman `/todo` maka data yang kita masukkan akan muncul seperti gambar berikut:



Sampai tahap ini kita sudah dapat menampilkan *list*, tambah dan meng-*update* `todo` menjadi `Selesai`. Tahap selanjutnya kita akan membuat halaman *edit* sebagai berikut:

```
<!-- filename: template/edit.html.twig -->
<form action="/todo/{{ todo.id }}/edit" method="POST">
    <input type="text" name="activity" value="{{ todo.activi
ty }}" />
    <button type="submit">Simpan</button>
</form>
```

Kemudian kita membuat *action method* `edit()` dengan isi sebagai berikut:

```
    public function edit(Request $request, $id)
    {
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
```

```
esponse::HTTP_NOT_FOUND);
        }

        if (Request::METHOD_POST === $request->getMethod())
{
            $todo->setActivity($request->get('activity'));
            $todo->save();

            return new RedirectResponse('/todo');
        }

        return $this->render('edit.html.twig', ['todo' => $t
odo]);
    }
```

Secara lengkap *class* `TodoController` menjadi sebagai berikut:

```
<?php
//filename: src/Controller/TodoController.php

namespace ModernFramework\Controller;

use ModernFramework\Model\Todo;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\RedirectResponse;

class TodoController extends Controller
{
    public function index()
    {
        return $this->render('index.html.twig', ['todos' =>
$this->getModel(Todo::class)->findAll()]);
    }

    public function new(Request $request)
    {
```

```
        $activity = $request->get('activity');
        if ($activity) {
            $todo = $this->getModel(Todo::class);
            $todo->setActivity($activity);
            $todo->save();

            return new RedirectResponse('/todo');
        }

        return $this->render('new.html.twig');
    }

    public function done($id)
    {
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }

        $todo->done();
        $todo->save();

        return new RedirectResponse('/todo');
    }

    public function edit(Request $request, $id)
    {
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }

        if (Request::METHOD_POST === $request->getMethod())
{
            $todo->setActivity($request->get('activity'));
            $todo->save();
```

```
            return new RedirectResponse('/todo');
        }

        return $this->render('edit.html.twig', ['todo' => $t
odo]);
    }
}
```

Dan kemudian kita daftarkan *action method* tersebut ke dalam *route* kita sebagai berikut:

```
<?php
//filename: config/routes.php

use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$routes = new RouteCollection();

$routes->add('todo_index', new Route('/todo', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::index',
]));

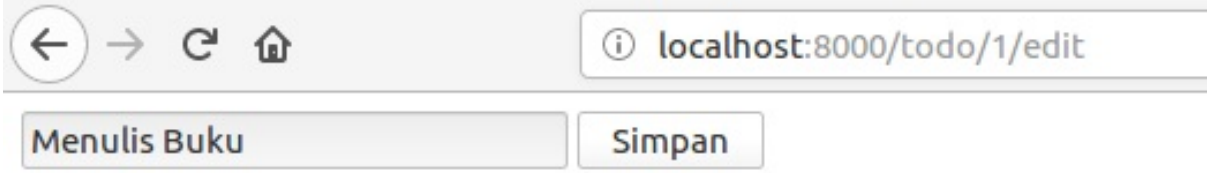
$routes->add('todo_new', new Route('/todo/new', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::new',
]));

$routes->add('todo_done', new Route('/todo/{id}/done', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::done',
]));

$routes->add('todo_edit', new Route('/todo/{id}/edit', [
    '_controller' => 'ModernFramework\Controller\TodoControl
```

```
ler::edit',
]));
```

Setelah itu kita klik *link* `edit` akan muncul halaman *edit* seperti gambar berikut:



Dan kemudian kita coba *edit* isian kemudian klik tombol simpan maka akan kembali ke halaman *list* namun dengan data yang sudah diperbarui sebagai berikut:



Action terakhir yang harus kita buat adalah *action method* `delete()` dengan isi sebagai berikut:

```
    public function delete($id)
    {
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }
```

```
        $todo->delete();

        return new RedirectResponse('/todo');
    }
```

Sehingga *class* `TodoController` akan menjadi sebagai berikut:

```
<?php
//filename: src/Controller/TodoController.php

namespace ModernFramework\Controller;

use ModernFramework\Model\Todo;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\RedirectResponse;

class TodoController extends Controller
{
    public function index()
    {
        return $this->render('index.html.twig', ['todos' =>
$this->getModel(Todo::class)->findAll()]);
    }

    public function new(Request $request)
    {
        $activity = $request->get('activity');
        if ($activity) {
            $todo = $this->getModel(Todo::class);
            $todo->setActivity($activity);
            $todo->save();

            return new RedirectResponse('/todo');
        }

        return $this->render('new.html.twig');
```

```
    }

    public function done($id)
    {
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }

        $todo->done();
        $todo->save();

        return new RedirectResponse('/todo');
    }

    public function edit(Request $request, $id)
    {
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }

        if (Request::METHOD_POST === $request->getMethod())
{
            $todo->setActivity($request->get('activity'));
            $todo->save();

            return new RedirectResponse('/todo');
        }

        return $this->render('edit.html.twig', ['todo' => $t
odo]);
    }

    public function delete($id)
    {
```

```
        $todo = $this->getModel(Todo::class)->find($id);
        if (!$todo) {
            return new Response('Halaman tidak ditemukan', R
esponse::HTTP_NOT_FOUND);
        }

        $todo->delete();

        return new RedirectResponse('/todo');
    }
}
```

Dan kemudian kita daftarkan *action method* `delete()` tersebut kedalam *route* sebagai berikut:

```
<?php
//filename: config/routes.php

use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$routes = new RouteCollection();

$routes->add('todo_index', new Route('/todo', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::index',
]));

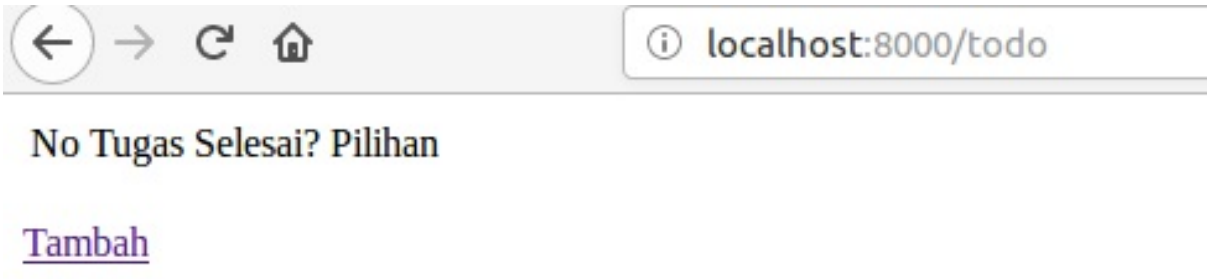
$routes->add('todo_new', new Route('/todo/new', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::new',
]));

$routes->add('todo_done', new Route('/todo/{id}/done', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::done',
]));
```

```
$routes->add('todo_edit', new Route('/todo/{id}/edit', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::edit',
]));

$routes->add('todo_delete', new Route('/todo/{id}/delete', [
    '_controller' => 'ModernFramework\Controller\TodoControl
ler::delete',
]));
```

Setelah kita klik *link* `Hapus` maka akan kembali halaman `/todo` dan data telah dihapus seperti pada gambar berikut:



Kesimpulan

Sampai disini berarti kita telah membuat operasi *CRUD* untuk tabel `todo`. Dengan selesainya operasi *CRUD* tersebut maka selesai juga pembahasan kita tentang *todo list* menggunakan MVC. Kesimpulan yang dapat kita ambil adalah bahwa membuat *framework MVC* tidaklah sulit asalkan kita memahami bagaimana alur *request* dan *response* dengan baik.

Memang *framework* yang kita buat belum memiliki fitur *_validasi*, keamanan, form dan lain sebagainya, namun sudah dapat digunakan untuk operasi *CRUD* sederhana.

Penutup

Terima kasih saya sampaikan kepada Anda karena Anda telah meluangkan waktu untuk membaca tulisan dari saya ini. Tak lupa saya memohon maaf atas segala kesalahan penulisan, *code* maupun gambar yang saya sajikan dalam tulisan ini.

Saya berharap Anda tidak membagikan buku ini kepada siapapun seraya mengajak mereka yang menginginkan buku ini untuk membeli langsung buku agar saya semakin termotivasi untuk menulis.

Jangan lupa untuk menantikan seri buku dari saya lainnya dan terus *upgrade skill* Anda dan jangan pernah puas dengan apa yang Anda raih saat ini. Akhir kata dari saya *"Raihlah kemuliaan di Dunia dengan ilmu, gapailah kemuliaan di Akhirat dengan ilmu, dan capailah kemuliaan keduanya dengan ilmu"*.

Wassalamu'alaikum Warohmatullah Wabarokatuhu.