

11. Case Study: Database Query Builder

Di awal buku saya pernah menulis bahwa proses pembuatan kode program dengan OOP butuh sebuah perencanaan. Alasannya karena dalam OOP kita dituntut untuk merancang kode program yang bisa di-“scaling”, yakni tetap mudah dikelola seiring meningkatnya kompleksitas aplikasi. Akan sangat repot jika harus merombak ulang seluruh kode program hanya untuk mengakomodasi beberapa fitur baru.

Misalnya saat ini kita diminta membuat aplikasi Sistem Informasi Sekolah untuk sebuah yayasan pendidikan. Yayasan ini memiliki 3 jenjang sekolah, yakni SD, SMP dan SMA. Setelah aplikasi jadi, ternyata ada permintaan untuk menambah jenjang PAUD (Pendidikan Anak Usia Dini).

Jika di awal kita sudah mempersiapkan semuanya dengan baik, maka tidak ada masalah. Penambahan ini bisa diatasi dengan membuat beberapa tabel baru di database. Sebaliknya, jika kode program dibuat tanpa perencanaan, sangat mungkin harus merombak total seluruh kode program.

Tiga prinsip dasar pemrograman object: *inheritance*, *polymorphism* dan *encapsulation* bisa dipakai untuk mengatasi masalah ini.

Salah satunya adalah dengan *encapsulation*, dimana kita memisahkan kode program menjadi modul-modul mandiri, terpisah dengan kode program utama. Misalnya kode program yang berhubungan dengan database menjadi satu modul, kode program untuk memproses form menjadi satu modul, kode program yang menangani tampilan menjadi 1 modul, dst.

Dengan perencanaan yang matang, setiap modul nantinya bisa dipakai ulang untuk aplikasi lain, tidak hanya untuk 1 aplikasi saja. Proses validasi form misalnya, akan selalu dibutuhkan untuk setiap halaman yang memiliki form. Daripada membuat proses validasi berulang kali, akan lebih baik kita buat sebuah modul terpisah.

Di dalam programming, modul-modul ini sering juga disebut sebagai **library**, **plugin**, atau **add-on** yang sama-sama berarti sebuah “kode tambahan”. Namun proses pembuatan library memang butuh waktu yang kadang kala bisa lebih rumit daripada langsung membuat kode program tanpa library. Kita harus memikirkan bagaimana caranya agar library tersebut bersifat universal, mudah digunakan dan juga bisa dikembangkan lebih jauh lagi.

Terdapat 2 pilihan, apakah membuat library sendiri atau menggunakan library yang sudah jadi. PHP merupakan bahasa pemrograman yang sudah matang sehingga tersedia banyak pilihan

library yang ditulis oleh berbagai programmer professional. Untungnya lagi, mayoritas library ini bisa di dapat dengan gratis.

Kembali, karena konsep OOP yang memang mendorong pemisahan kode program, maka hampir semua library dibuat menggunakan pemrograman object.

Satu aplikasi biasanya butuh beberapa library, mulai dari mengakses database, memproses form, menangani cookie, hingga mengirim email. Beberapa library ada yang di-bundle menjadi 1 paket lengkap, dan itulah yang menjadi **framework**. Secara sederhana, framework terdiri dari kumpulan library siap pakai untuk memudahkan pembuatan aplikasi.

Sebagai contoh, jika menggunakan framework **Code Igniter**, proses mengambil tabel barang dari database cukup dengan 1 perintah berikut:

```
$query = $this->db->get('barang');
```

Jika kita menggunakan **mysqli** atau **PDO**, kode programnya tentu akan lebih panjang.

Dalam bab ini (dan juga bab berikutnya) kita tidak akan belajar cara menggunakan library yang sudah jadi, tapi membuat library itu sendiri. Studi kasus ini menjadi praktek yang bagus dalam penerapan konsep OOP.

11.1. MySQL Query Builder

Apa yang akan kita rancang adalah sebuah library untuk memudahkan pengaksesan database.

Dalam framework PHP seperti *Code Igniter* atau *Laravel*, teknik ini dikenal sebagai **Query Builder**.

Nantinya, kita hanya perlu memanggil beberapa method untuk mengakses database, kemudian method itulah yang akan melakukan "kerja" membuat dan menjalankan perintah query MySQL. Sebagai contoh, untuk menampilkan seluruh tabel barang, kita bisa menggunakan kode berikut:

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->get('barang');
```

Di baris 2 saya menggunakan perintah `require` bawaan PHP untuk meng-import file `DB.php`. Di dalam file `DB.php` ini terdapat pendefinisian **class** `DB` yang berisi berbagai method. Inilah class yang akan kita rancang sesaat lagi.

Method `DB::getInstance()` di baris 3 dipakai untuk membuat object dari class `DB`. Perintah instansiasi ini akan memproses semua "urusan" dengan MySQL, yang salah satunya membuka koneksi ke database MySQL.

Di baris 5 saya menampung hasil dari perintah `$DB->get('barang')` ke dalam variabel `$tabelBarang`. Method `get()` adalah salah satu method yang tersedia di dalam class `DB`. Method ini dipakai untuk mengambil data tabel MySQL. Hasilnya, variabel `$tabelBarang` akan berisi seluruh data tabel `barang` dalam bentuk array.

Anda bisa perhatikan bahwa kita tidak menulis satu pun perintah query MySQL. Ini semua akan di proses oleh method `$DB->get()`.

Ketika method `$DB->get('barang')` dipanggil, class `DB` akan memproses argument `'barang'` menjadi query `"SELECT * FROM barang"`. Inilah yang dimaksud bahwa class `DB` yang akan kita rancang merupakan sebuah **query builder**, dimana library tersebut bisa meng-generate dan menjalankan query MySQL secara otomatis.

Contoh lain, proses input ke tabel `barang` bisa dilakukan dengan perintah berikut:

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $DB->insert('barang',[  
6     'nama_barang' => 'Philips Blender HR 2157',  
7     'jumlah_barang' => 11,  
8     'harga_barang' => 629000  
9 ]);
```

Perintah di baris 2 dan 3 masih sama seperti sebelumnya.

Di baris 5 saya mengakses method `$DB->insert()` yang dipakai untuk proses `INSERT` ke database. Argument pertama method berupa nama tabel yang akan diinput, yakni `'barang'`. Kemudian argument kedua berisi *associative array* yang terdiri dari pasangan nama kolom dan nilai.

Apa yang akan dilakukan oleh class `DB` adalah mengkonversi perintah di baris 5 – 9, menjadi query berikut:

```
INSERT INTO barang ('nama_barang', 'jumlah_barang', 'harga_barang') VALUES  
( 'Philips Blender HR 2157', 11, 629000 )
```

Contoh terakhir, untuk menghapus data tabel bisa dilakukan dengan kode berikut:

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $DB->delete('barang',[ 'id_barang' '=', 4 ]);
```

Perintah di baris 5 artinya saya ingin menghapus data di tabel `barang` dengan kondisi kolom `'id_barang' = 4`. Secara internal nantinya pemanggilan method ini akan dikonversi menjadi perintah query berikut:

```
DELETE FROM barang WHERE id_barang = 4
```

Itulah beberapa contoh method yang akan kita rancang. Total, class `DB` memiliki sekitar 15 method untuk berbagai keperluan, termasuk update data, pencarian dengan query `LIKE`, menangani kondisi `WHERE`, memeriksa nilai nilai unik, dll. Semuanya akan kita bahas secara bertahap.

Dapat juga dilihat bahwa class `DB` sangat memudahkan pembuatan aplikasi. Kode program di file utama menjadi lebih singkat karena "kerja keras" dalam mengakses database sudah ditangani oleh class `DB`.

Dalam perancangan class `DB`, saya akan memakai PDO *prepared statement*. Class `DB` di sini berperan sebagai sebuah "layer" atau "interface" di atas PDO. Dengan mengakses class `DB`, kita tidak perlu menulis langsung method-method PDO. Si pengguna library (bisa jadi itu adalah programmer lain), juga tidak perlu memahami prinsip kerja di dalam class `DB`, tapi cukup cara pakainya saja.

11.2. Class DB

Library database yang akan kita rancang adalah sebuah **class** yang saya beri nama `DB`. Class `DB` ini disimpan dalam file `DB.php`. Anda bebas jika ingin menggunakan nama class serta nama file yang berbeda. Proses pengaksesan class `DB` dilakukan dari file terpisah, yang dalam contoh ini saya akses dari `index.php`.

Dengan demikian, kita butuh 2 buah file: `DB.php` dan `index.php`. Sepanjang pembahasan nanti, kedua file ini akan diakses secara bergantian. Silahkan buka teks editor dan ketik kode program berikut ke dalam file `DB.php`:

01.DB_class/DB.php

```
1 <?php
2 class DB{
3
4 }
```

Yup, hanya berupa class kosong yang akan kita isi sesaat lagi. Berikutnya, buat file `index.php` yang berisi kode program berikut:

01.DB_class/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = new DB();
4
5 echo "<pre>";
6 var_dump($DB);
7 echo "</pre>";
```

Di baris 2 saya menggunakan perintah `require` bawaan PHP untuk meng-import file `DB.php`. Karena langsung ditulis dengan nama file, artinya file `index.php` harus berada di dalam folder yang sama dengan file `DB.php`. Jika file `DB.php` berada di folder lain, silahkan tambah alamat path ke dalam perintah `require`.

Selanjutnya di baris 3 saya membuat variabel `$DB` sebagai object hasil instansiasi dari class `DB`. Object `$DB` ini kemudian ditampilkan dengan fungsi `var_dump()` di baris 5 – 7. Fungsi `var_dump()` di sini semata-mata untuk melihat apa isi dari variabel `$DB`.

Berikut hasil tampilan dari file `index.php`:

```
object(DB)#1 (0) { }
```

Hasilnya, perintah `var_dump($DB)` memperlihatkan bahwa variabel `$DB` sudah berisi object dari class `DB`.

11.3. Class DB: Constructor

Langkah pertama dalam pembuatan sebuah library database adalah membuka koneksi ke database server, yang dalam contoh kita menggunakan database MySQL (atau tepatnya MariaDB bawaan XAMPP).

Karena proses koneksi ini selalu dibutuhkan, akan sangat pas jika ditulis ke dalam constructor. Informasi seputar *host*, *nama database*, *user name* dan *password* MySQL akan saya simpan ke dalam *private property* seperti kode berikut:

02.DB_constructor/DB.php

```
1 <?php
2 class DB{
3
4     // Property untuk koneksi ke database mysql
5     private $_host = '127.0.0.1';
6     private $_dbname = 'ilkoom';
7     private $_username = 'root';
8     private $_password = '';
9
10    // Property internal dari class DB
11    private $_pdo;
12 }
```

Dalam kode program ini saya membuat 5 buah property. Kelima property menggunakan hak akses `private` sehingga hanya bisa diakses dari dalam class `DB` saja.

Setiap property diawali dengan tanda garis bawah "`_`" (*underscore*) semata-mata agar mudah dibedakan dengan property yang memiliki hak akses lain. Maksudnya, jika sebuah property diawali dengan *underscore*, maka saya bisa pastikan itu adalah `private` property. Teknik ini

hanya kebiasaan beberapa programmer dan bukan sebuah kewajiban. Anda boleh mengikuti cara ini atau menggunakan nama variabel biasa.

Sesuai dengan namanya, property `$_host` dipakai untuk menyimpan alamat komputer host atau IP address dari komputer tempat MySQL server berada. Karena MySQL server ada di komputer yang sama, maka saya bisa memakai alamat 'localhost' atau IP address '127.0.0.1'.

Berikutnya property `$_dbname` dipakai untuk menampung nama database. Dalam contoh ini saya menggunakan database 'ilkoom'. Artinya, di MySQL server harus sudah tersedia database bernama `ilkoom`.

Property `$_username` dan `$_password` dipakai untuk menampung nama username dan password untuk mengakses ke MySQL Server. Dalam contoh ini saya akan login sebagai user `root` dengan password berupa string kosong.

Property terakhir yakni `$_pdo` di siapkan untuk menampung koneksi PDO dari PHP ke database MySQL. Isi untuk property ini akan kita tulis di dalam constructor sebagai berikut:

02.DB_constructor/DB.php

```

1 <?php
2 class DB{
3
4     // Property untuk koneksi ke database mysql
5     private $_host = '127.0.0.1';
6     private $_dbname = 'ilkoom';
7     private $_username = 'root';
8     private $_password = '';
9
10    // Property internal dari class DB
11    private $_pdo;
12
13    // Constructor untuk pembuatan PDO Object
14    public function __construct(){
15        try {
16            $this->_pdo = new PDO('mysql:host='.$this->_host.';dbname='.$this->_dbname,
17                                  $this->_username, $this->_password);
18            $this->_pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
19        } catch (PDOException $e){
20            die("Koneksi / Query bermasalah: ".$e->getMessage().
21                " (".$e->getCode()."')");
22        }
23    }
24
25 }
```

Kode program ini adalah sambungan dari sebelumnya.

Di dalam method constructor (baris 14 - 23), saya langsung memanggil class PDO untuk membuat koneksi ke database. Hasil koneksi ini ditampung ke dalam property `$this->_pdo` yang sudah kita siapkan sebelumnya. Sekedar pengingat, `$this` adalah variabel khusus yang

merujuk ke "object saat ini".

Sebagai argument dari pembuatan PDO object, saya merangkai isi dari empat private property: `$_host`, `$_dbname`, `$_username` dan `$_password`. Kode programnya tampak rumit karena penambahan variabel `$this` untuk mengakses setiap private property. Pada dasarnya perintah di baris 16 – 17 sama seperti berikut:

```
$this->_pdo = new PDO("mysql:host=127.0.0.1;dbname=ilkoom", "root", "");
```

Selanjutnya baris 18 berisi pengaturan agar pesan error di tampilkan sebagai sebuah exception. Exception ini akan di tangkap oleh kode program di baris 19 – 22. Tidak ada hal yang baru di sini karena sama persis seperti yang kita pakai pada bab tentang PDO.

Saya memakai fungsi `die()` untuk menampilkan pesan error. Artinya jika class `DB` gagal mengakses database, fungsi `die()` akan langsung menghentikan kode program.

Sebagai uji coba, silahkan akses kembali file `index.php`. Jika tidak ada masalah akan tampil hasil berikut:

```
object(DB)#1 (5) {  
    ["_host":"DB":private]=>  
    string(9) "127.0.0.1"  
    ["_dbname":"DB":private]=>  
    string(6) "ilkoom"  
    ["_username":"DB":private]=>  
    string(4) "root"  
    ["_password":"DB":private]=>  
    string(0) ""  
    ["_pdo":"DB":private]=>  
    object(PDO)#2 (0) {  
    }  
}
```

Hasil ini berasal dari perintah `var_dump($DB)` di dalam file `index.php`. Isinya memperlihatkan seluruh property yang ada di dalam class `DB`. Perhatikan bahwa property `_pdo` di bagian terakhir sudah berisi `object(PDO)`, artinya proses koneksi dengan database MySQL sukses dilakukan.

Untuk menguji jika terjadi kesalahan, saya akan tukar isi property `$_password` di dalam class `DB` dengan sebuah karakter 'x':

```
private $_password = 'x';
```

Berikut hasil dari `index.php`:

```
Koneksi / Query bermasalah: SQLSTATE[HY000] [1045] Access denied for user  
'root'@'localhost' (using password: YES) (1045)
```

Tampil pesan error karena seharusnya user `root` tidak memiliki password. Namun pesan error

ini juga menunjukkan bahwa block try – catch yang kita tulis sudah berhasil menangkap exception. Silahkan ubah kembali isi property `$_password` menjadi string kosong.

Sepanjang bab ini saya akan mengakses tabel `barang` yang ada di database `ilkoom`. Tabel `barang` ini sudah kita buat sebelumnya, atau anda bisa jalankan file `generate_tabel_barang.php` untuk membuat ulang tabel barang beserta isinya.

11.4. Class DB: Singleton Pattern

Constructor class `DB` yang baru saja kita tulis sudah berjalan dengan baik, namun juga masih bisa disempurnakan lebih lanjut.

Untuk aplikasi yang kompleks, dalam satu halaman kadang perlu mengakses database MySQL beberapa kali. Jika halaman tersebut dibuat secara berurutan dari atas ke bawah, proses instansiasi class `DB` hanya perlu dilakukan sekali di awal (biasanya di baris paling atas).

Namun jika halaman tersebut terdiri dari beberapa modul atau library yang saling memanggil satu sama lain, ada kemungkinan class `DB` di instansiasi lebih dari 1 kali. Pada kode program kita saat ini, setiap kali class `DB` di instansiasi, koneksi ke database MySQL akan selalu dibuat ulang.

Sebagai praktik, silahkan buka file `DB.php` lalu tambah 1 perintah `echo` ke dalam constructor class `DB`:

03.DB_constructor_problem/DB.php

```
1 <?php
2 class DB{
3 ...
4 ...
5     public function __construct(){
6         try {
7             ...
8                 $this->_pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
9                 echo "Koneksi dibuat <br>";
10            } catch (PDOException $e){
11                ...
12            }
13        }
14    }
```

Saya menyisipkan perintah `echo "Koneksi dibuat
"` di baris 9 agar kita bisa melihat hasil tampilan setiap kali constructor class `DB` diakses.

Selanjutnya, buka file `index.php` dan isi dengan kode berikut:

03.DB_constructor_problem/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = new DB();
4 $DB = new DB();
5 $DB = new DB();
```

Dalam file `index.php` ini saya melakukan 3 kali instansiasi class `DB`. Bagaimana hasilnya?

```
Koneksi dibuat
Koneksi dibuat
Koneksi dibuat
```

Seperti yang terlihat, constructor class `DB` akan dipanggil sebanyak 3 kali, yakni satu untuk setiap proses instansiasi. Artinya, koneksi ke MySQL juga akan dibuat sebanyak 3 kali.

Dalam kondisi ideal, koneksi ke MySQL cukup dilakukan 1 kali saja meskipun class `DB` dibuat berulang kali. Salah satu solusi untuk menerapkan konsep seperti ini adalah menggunakan teknik *singleton pattern*, yang merupakan bagian dari konsep pemrograman object yang disebut sebagai **Design Pattern**.

Sekilas Tentang Design Pattern

Dalam ilmu programming (terutama pemrograman berorientasi objek) terdapat istilah yang dikenal sebagai **Design Pattern**. Secara sederhana, *design pattern* adalah teknik atau cara penulisan tertentu untuk membuat kode program "berperilaku khusus".

Perilaku khusus yang saya maksud adalah kode program tersebut bisa dibentuk supaya mengikuti aturan-aturan tertentu, misalnya hanya bisa di instansiasi 1 kali saja, atau sebuah object hanya bisa dibuat dengan memanggil method dari class induk.

Istilah **Design Pattern** mulai populer sejak tahun 1995, yang berasal dari buku "*Design Patterns: Elements of Reusable Object-Oriented Software*". Buku ini ditulis oleh 4 ilmuwan komputer ternama yang dikenal sebagai **The "Gang of Four"**, yakni Erich Gamma, Richard Helm, Ralph Johnson dan John Vlissides ([wikipedia](#)).

Buku tersebut berisi 23 pola atau programming pattern, diantaranya *builder pattern*, *factory method pattern*, *adapter pattern*, dan termasuk juga *singleton pattern*. Sampai saat ini buku tersebut masih menjadi patokan dasar bagi yang ingin belajar *design pattern*.

Apakah saya juga harus belajar design pattern?

Idealnya **iya**, karena teknik *design pattern* ini dibuat berdasarkan pengalaman programmer-programmer sebelumnya yang sudah lebih dulu menemui masalah yang sama.

Namun design pattern termasuk materi advanced yang cukup rumit. Agar bisa memahami design pattern, seorang programmer harus paham konsep pemrograman

object secara mendalam serta pernah beberapa kali membuat project dengan konsep OOP.

Selain itu ada juga pendapat bahwa *design pattern* ini terlalu rumit dan relatif jarang dipakai dalam pembuatan aplikasi "normal". Kebanyakan pola / *pattern* baru diperlukan jika anda membuat library atau framework sendiri. Jika hanya sebagai pengguna framework (seperti yang akan banyak kita lakukan), cukup ikuti alur yang sudah ditetapkan oleh framework tersebut.

Saran saya, jika anda memang berencana menjadi programmer professional, *design pattern* ini menjadi salah satu materi yang diperlukan. Seseorang yang sudah menguasai *design pattern*, pasti mahir dalam pemrograman berorientasi objek. Sebaliknya, bagi yang paham dengan OOP belum tentu menguasai *design pattern*.

Tantangan lain (setidaknya untuk saat ini) anda harus mencari referensi dalam buku bahasa inggris karena materi *design pattern* yang berbahasa indonesia masih cukup langka.

Kembali ke class DB, kita akan rancang kode program untuk membuat **singleton pattern**. Tujuannya agar constructor class DB hanya bisa dipanggil 1 kali saja.

Pertama, buat sebuah `static` property yang nanti akan dipakai untuk menampung object dari class DB. Property ini saya beri nama `$_instance` dan diberi nilai awal `null`. :

```
private static $_instance = null;
```

Berikutnya, ubah hak akses constructor class DB menjadi `private`:

```
private function __construct(){
    ...
}
```

Loh, kalau constructor dibuat private, bukannya kita tidak akan bisa meng-instansiasi class DB?

Betul, karena constructor menjadi private, kita tidak akan bisa membuat object dari class DB **di luar class**. Namun proses pembuatan object masih bisa dilakukan dari dalam class DB itu sendiri, yakni dari static method `getInstance()` berikut:

```
1 public static function getInstance(){
2     if(!isset(self::$_instance)) {
3         self::$_instance = new DB();
4     }
5     return self::$_instance;
6 }
```

Inilah method yang digunakan untuk membuat **singleton pattern**. Kondisi `if` di baris 2 akan memeriksa apakah property `$_instance` berisi sesuatu atau tidak, yakni `if(!isset(self::$_instance))`.

Jika property `$_instance` tidak berisi sesuatu (atau berisi `null`), maka perintah di baris 3 akan dijalankan. Perintah ini dipakai untuk mengisi property `$_instance` dengan object dari class `DB`. Perhatikan cara penulisannya, yakni `self::$_instance = new DB()`. Sebagai pengingat, `self` adalah keyword khusus yang merujuk ke class saat ini. Kita tidak menggunakan `$this` karena `$_instance` adalah sebuah static property.

Jika ternyata property `$_instance` berisi sesuatu (`not null`), maka kondisi `if` akan dilewati dan method ini mengembalikan isi dari property `$_instance` tersebut.

Berikut kode program lengkap dari file `DB.php`:

04.DB_constructor_singleton/DB.php

```

1  <?php
2  class DB{
3
4      // Property untuk koneksi ke database mysql
5      private $_host = '127.0.0.1';
6      private $_dbname = 'ilkoom';
7      private $_username = 'root';
8      private $_password = '';
9
10     // Property internal dari class DB
11     private static $_instance = null;
12     private $_pdo;
13
14     // Constructor untuk pembuatan PDO Object
15     private function __construct(){
16         try {
17             $this->_pdo = new PDO('mysql:host='.$this->_host.';dbname='.$this->_dbname,
18                                 $this->_username, $this->_password);
19             $this->_pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
20             echo "Koneksi dibuat <br>";
21         } catch (PDOException $e){
22             die("Koneksi / Query bermasalah: ".$e->getMessage().
23                 " (".$e->getCode().")");
24         }
25     }
26
27     // Singleton pattern untuk membuat class DB
28     public static function getInstance(){
29         if(!isset(self::$_instance)) {
30             self::$_instance = new DB();
31         }
32         return self::$_instance;
33     }
34 }
```

Sekarang proses pembuatan object dari class DB juga sudah berubah. Kita tidak bisa lagi menggunakan perintah `$DB = new DB()` karena constructor class DB sudah "di kunci" dengan hak akses `private`. Proses pembuatan object dari class DB sekarang menggunakan method `DB::getInstance()`.

Berikut isi file `index.php` yang dipakai untuk membuat object class DB:

04.DB_constructor_singleton/index.php

```
1 <?php
2 require 'DB.php';
3
4 $DB = DB::getInstance();
5
6 echo "<pre>";
7 var_dump($DB);
8 echo "</pre>";
```

Hasil kode program:

Koneksi dibuat

```
object(DB)#1 (5) {
    ["_host":"DB":private]=>
    string(9) "127.0.0.1"
    ["_dbname":"DB":private]=>
    string(6) "ilkoom"
    ["_username":"DB":private]=>
    string(4) "root"
    ["_password":"DB":private]=>
    string(0) ""
    ["_pdo":"DB":private]=>
    object(PDO)#2 (0) {
    }
}
```

Di sini proses pembuatan object dari class DB di lakukan dengan perintah `$DB = DB::getInstance()`. Hasil fungsi `var_dump($DB)` memperlihatkan bahwa variabel `$DB` sudah berisi object dari class DB.

Inilah cara pembuatan **singleton pattern**, dimana class DB hanya bisa di-instansiasi satu kali saja. Atau lebih tepatnya, constructor class DB hanya bisa diakses 1 kali sepanjang kode program. Sebagai pembuktian, kita akan coba panggil method `DB::getInstance()` beberapa kali:

04.DB_constructor_singleton/index.php

```
1 <?php
2 require 'DB.php';
3
4 $DB = DB::getInstance();
5 $DB = DB::getInstance();
```

```
6 $DB = DB::getInstance();
```

Hasil kode program:

Koneksi dibuat

Teks "Koneksi dibuat" hanya muncul 1 kali. Teks ini berasal dari perintah echo yang sebelumnya kita sisip ke dalam constructor class DB. Ini membuktikan bahwa constructor dari class DB hanya dijalankan 1 kali saja, meskipun terdapat 3 buah perintah untuk membuat object class DB.

Dengan demikian, class DB menjadi lebih efisien. Kita bisa memastikan bahwa koneksi ke database MySQL hanya akan dilakukan 1 kali sepanjang kode program, tidak peduli berapa kali class DB di-instansiasi.

Sebelum kita lanjut ke materi berikutnya, silahkan hapus kembali perintah echo "Koneksi dibuat
" dari constructor class DB. Dan untuk menghemat tempat, saya tidak menulis lagi kode program untuk constructor class DB serta method getInstance(). Sepanjang sisa pembahasan, kedua method ini dianggap sudah tersedia di dalam class DB.

11.5. Class DB: Method runQuery

Kita sudah berhasil membuat koneksi ke database MySQL, serta memastikan koneksi hanya dilakukan 1 kali saja dengan teknik *singleton pattern*. Sekarang kita akan buat method untuk memproses query, yakni runQuery().

Method runQuery() saya rancang sebagai method "generik". Maksudnya, method ini akan dipakai sebagai dasar untuk method-method lain. Method runQuery() hanya berfungsi untuk menjalankan query saja, belum sampai meng-generate perintah query secara otomatis.

Berikut kode program yang dipakai untuk membuat method runQuery():

05.DB_method_runQuery/DB.php

```
1 <?php
2 class DB{
3
4     // kode program untuk property dan constructor class DB
5     // ...
6     // ...
7
8     public function runQuery($query, $bindValue = []){
9         try {
10             $stmt = $this->_pdo->prepare($query);
11             $stmt->execute($bindValue);
12         }
13         catch (PDOException $e){
14             die("Koneksi / Query bermasalah: ".$e->getMessage().
15                 " (".$e->getCode()."")");
16         }
17     }
18 }
```

```

16     }
17     return $stmt;
18 }
19
20 }

```

Method `runQuery()` menerima 2 buah argument. Argument pertama, yakni `$query` dipakai untuk menampung perintah query dalam bentuk string, misalnya `'SELECT * FROM barang'` sedangkan argument kedua yakni `$bindValue` dipakai untuk menampung nilai prepared statement dalam bentuk array. Argument `$bindValue` memiliki nilai default berupa array kosong sehingga bersifat opsional (boleh tidak ditulis).

Baris pertama method `runQuery()` langsung disambut dengan block try-catch. Di baris 10 saya membuat variabel `$stmt` yang dipakai untuk menampung hasil perintah `$this->_pdo->prepare($query)`. Hasilnya, variabel `$stmt` akan berisi sebuah **PDOStatement object**.

Jika anda lihat kembali constructor class `DB`, property `$this->_pdo` saya buat untuk menampung hasil koneksi ke MySQL. Dengan demikian perintah `$this->_pdo->prepare($query)` berfungsi menjalankan sebuah query MySQL.

Di baris 11 terdapat perintah `$stmt->execute($bindValue)`. Perintah ini dipakai untuk proses **bind** dan **execute** dari PDO prepared statement. Nilai yang ada di dalam array `$bindValue` akan berpasangan dengan tanda tanya atau *placeholder* "?" yang nantinya terdapat di string `$query`.

Selanjutnya terdapat block **catch** di baris 13 – 16 yang dipakai untuk menampilkan pesan error jika terdapat kesalahan perintah query. Terakhir di baris 17 saya mengembalikan isi variabel `$stmt`.

Method `runQuery()` ini mungkin cukup susah dipahami karena kita belum lihat bagaimana bentuk prakteknya. Berikut contoh pemanggilan method `runQuery()` dari file `index.php`:

05.DB_method_runQuery/index.php

```

1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->runQuery('SELECT * FROM barang');
6 $tabelBarang = $result->fetchAll(PDO::FETCH_OBJ);
7
8 echo "<pre>";
9 print_r($tabelBarang);
10 echo "</pre>";

```

Hasil kode program:

```

Array
(
    [0] => stdClass Object
)

```

```

(
    [id_barang] => 1
    [nama_barang] => TV Samsung 43NU7090 4K
    [jumlah_barang] => 5
    [harga_barang] => 5399000
    [tanggal_update] => 2019-03-11 17:35:28
)
[1] => stdClass Object
(
    [id_barang] => 2
    [nama_barang] => Kulkas LG GC-A432HLHU
    [jumlah_barang] => 10
    [harga_barang] => 7600000
    [tanggal_update] => 2019-03-11 17:35:28
)
...
...
)

```

Perintah di baris 2 – 3 dipakai untuk membuat object dari class `DB`. Object ini kemudian disimpan ke dalam variabel `$DB`.

Di baris 4 saya mengakses method `runQuery()` dengan perintah `$result = $DB->runQuery('SELECT * FROM barang')`. Artinya, string 'SELECT * FROM barang' dikirim sebagai argument pertama untuk method `runQuery()`. Di dalam method `runQuery()`, string ini akan dijalankan sebagai perintah query MySQL dan menghasilkan sebuah **PDO Statement object**.

PDO Statement object ini saya tampung ke dalam variabel `$result`. Untuk menampilkan data tabel, di baris 5 saya menjalankan perintah `$tabelBarang = $result->fetchAll(PDO::FETCH_OBJ)`. Hasilnya, variabel `$tabelBarang` berisi seluruh tabel barang dalam bentuk array object.

Jika saya ingin menampilkan 1 baris pertama dari tabel `barang` dengan perintah `echo`, bisa menggunakan kode berikut:

05.DB_method_runQuery/index.php

```

1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->runQuery('SELECT * FROM barang');
6 $tabelBarang = $result->fetchAll(PDO::FETCH_OBJ);
7
8 echo $tabelBarang[0]->id_barang." | ";
9 echo $tabelBarang[0]->nama_barang." | ";
10 echo $tabelBarang[0]->jumlah_barang." | ";
11 echo $tabelBarang[0]->harga_barang." | ";
12 echo $tabelBarang[0]->tanggal_update." | ";

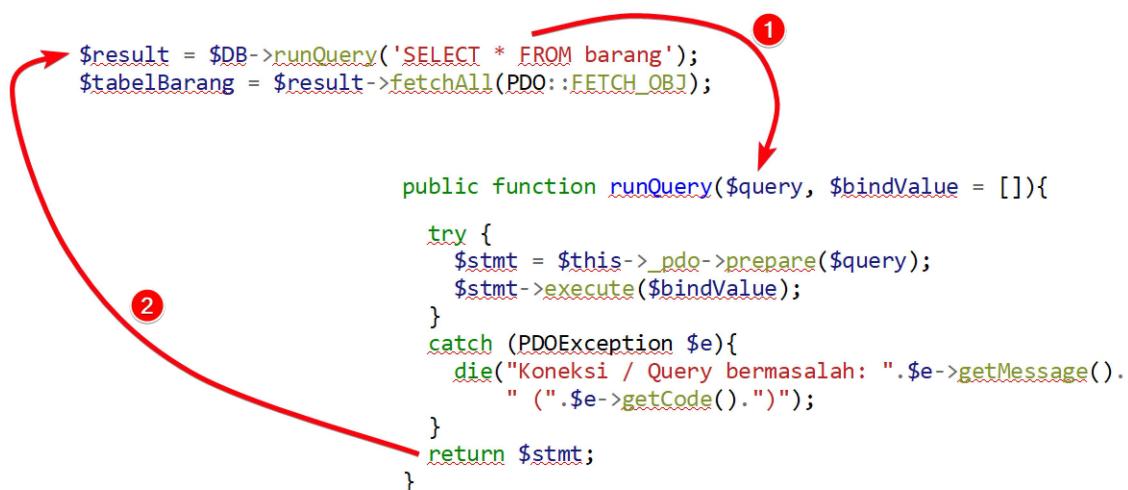
```

Hasil kode program:

```
1 | TV Samsung 43NU7090 4K | 5 | 5399000 | 2019-03-11 17:35:28 |
```

Intinya adalah bagaimana cara menampilkan array tabel yang dipanggil dengan metode `fetchAll(PDO::FETCH_OBJ)`. Penjelasan lebih lanjut sudah kita bahas dalam bab tentang PDO.

Saya sangat sarankan anda untuk meluangkan waktu sejenak memahami alur proses pemanggilan method `runQuery()` ini. Misalnya bagaimana string 'SELECT * FROM barang' di kirim dan diterima oleh method `runQuery()`.



Ilustrasi pemanggilan method `runQuery()`

Di sini method `runQuery()` hanya dipakai untuk menjalankan query saja. Pada saat method dijalankan, string 'SELECT * FROM barang' akan dikirim ke dalam variabel `$query` sebagai argument pertama (1). Kemudian method `runQuery()` akan memproses query tersebut dan menghasilkan **PDO statement object** yang disimpan ke dalam variabel `$stmt`. Variabel `$stmt` kemudian di-return ke dalam variabel `$result` (2).

Tapi tunggu dulu, bagaimana dengan perintah `$stmt->execute($bindValue)`? Meskipun agak 'aneh', prepared statement tetap bisa dipakai untuk query yang tidak memiliki data *placeholder*. Karena dalam contoh ini kita menjalankan query tanpa nilai *placeholder*, yang akan dijalankan adalah sebagai berikut:

```
try {
    $stmt = $this->_pdo->prepare('SELECT * FROM barang');
    $stmt->execute([]);
}
```

Sekarang mari kita panggil method `runQuery()` dengan data placeholder:

06.DB_method_runQuery_prepared/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->runQuery('SELECT * FROM barang WHERE id_barang = ?', [4]);
6 $tabelBarang = $result->fetchAll(PDO::FETCH_OBJ);
7
8 echo "<pre>";
9 print_r($tabelBarang);
10 echo "</pre>";
```

Hasil kode program:

```
Array
(
    [0] => stdClass Object
        (
            [id_barang] => 4
            [nama_barang] => Printer Epson L220
            [jumlah_barang] => 14
            [harga_barang] => 2099000
            [tanggal_update] => 2019-03-11 17:35:28
        )
)
```

Di baris 5 method `runQuery()` saya panggil dengan 2 buah argument. Argument pertama berupa string `'SELECT * FROM barang WHERE id_barang = ?'` dan argument kedua berupa array `[4]`. Dalam perintah query saya menambahkan *clausa* `WHERE id_barang = ?`, di sini terdapat tanda tanya `'?` yang nilainya akan diambil dari argument kedua, yakni 4.

Dalam method `getQuery()`, perintah yang dijalankan adalah sebagai berikut:

```
try {
    $stmt = $this->_pdo->prepare('SELECT * FROM barang WHERE id_barang = ?');
    $stmt->execute([4]);
}
```

Artinya saya ingin mengambil data di tabel barang yang memiliki kolom `id_barang = 4`.

Agar lebih rapi, argument method `getQuery()` bisa saya tulis sebagai variabel terpisah:

06.DB_method_runQuery_prepared/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $query = 'SELECT * FROM barang WHERE id_barang = ?';
6 $arr = [4];
7
8 $result = $DB->runQuery($query, $arr);
9 $tabelBarang = $result->fetchAll(PDO::FETCH_OBJ);
10
```

```
11 echo "<pre>";
12 print_r($tabelBarang);
13 echo "</pre>";
```

Di sini saya memisahkan string query ke dalam variabel \$query di baris 5 dan data placeholder ke variabel \$arr di baris 6.

Sebagai latihan, bisakah anda menjalankan sebuah query `INSERT` menggunakan method `getQuery()`? Data yang diinput ke tabel barang adalah sebagai berikut:

- ✓ nama_barang = 'Cosmos CRJ-8229 - Rice Cooker'
- ✓ jumlah_barang = 4
- ✓ harga_barang = 299000

Syarat tambahan: gunakan prepared statement. Silahkan anda coba rancang kode programnya.

Baik, berikut kode yang saya gunakan:

07.DB_method_runQuery_insert/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $query = 'INSERT INTO barang (nama_barang, jumlah_barang, harga_barang)
6           VALUES (?,?,?)';
7 $arr = ['Cosmos CRJ-8229 - Rice Cooker',4,299000];
8
9 // jalankan proses insert
10 $DB->runQuery($query,$arr);
11
12 // tampilkan semua tabel barang
13 $result = $DB->runQuery('SELECT * FROM barang');
14 $tabelBarang = $result->fetchAll(PDO::FETCH_OBJ);
15
16 echo "<pre>";
17 print_r($tabelBarang);
18 echo "</pre>";
```

Hasil kode program:

```
Array
(
    ...
    ...
    [5] => stdClass Object
        (
            [id_barang] => 6
            [nama_barang] => Cosmos CRJ-8229 - Rice Cooker
            [jumlah_barang] => 4
            [harga_barang] => 299000
            [tanggal_update] => 2019-03-12 07:59:53
        )
)
```

```
)  
)
```

Method `runQuery()` pada dasarnya bisa menerima query apapun. Jika query tersebut berbentuk prepared statement, maka untuk setiap tanda tanya "?" harus berpasangan dengan data placeholder.

Dalam variabel `$query` terdapat 3 buah tanda tanya, maka akan berpasangan dengan 3 buah nilai di dalam variabel `$arr`. Di baris 13 – 18 saya menampilkan kembali isi seluruh tabel barang untuk memastikan query `INSERT` sudah berhasil berjalan.

Sedikit tambahan, dalam query `INSERT` di baris 5 saya tidak menginput nilai tanggal untuk kolom `tanggal_update`. Namun ketika ditampilkan, kolom ini sudah terisi sendiri.

Pada saat pembuatan tabel barang (dari file `generate_tabel_barang.php`), kolom `tanggal_update` di set sebagai `timestamp`. Di dalam MySQL, kolom dengan tipe data `timestamp` otomatis berisi tanggal server sekarang apabila tidak diinput. Ini kurang lebih sama seperti kolom `id_barang` yang di set sebagai `AUTO_INCREMENT`, dimana nilainya juga otomatis di generate oleh MySQL.

11.6. Class DB: Method `getQuery`

Method `getQuery()` merupakan sedikit tambahan dari method `runQuery()`. Method ini saya rancang agar langsung mengembalikan data tabel dalam bentuk array object, yakni hasil dari method `fetchAll(PDO::FETCH_OBJ)`. Secara internal, method `getQuery()` juga memanggil method `runQuery()`.

Berikut kode program dari method `getQuery()`:

08.DB_method_getQuery/DB.php

```
1 <?php  
2 class DB{  
3  
4     // kode program untuk property dan constructor class DB  
5     // kode program untuk method runQuery()  
6     // ...  
7  
8     public function getQuery($query,$bindValue = []){  
9         return $this->runQuery($query,$bindValue)->fetchAll(PDO::FETCH_OBJ);  
10    }  
11 }
```

Seperti yang terlihat, di dalam method `getQuery()` ini saya memanggil ulang method `runQuery()`, namun dengan tambahan langsung disambung dengan memanggil method `fetchAll(PDO::FETCH_OBJ)`.

Dengan penulisan seperti ini, maka method `getQuery()` secara khusus dipakai untuk query

SELECT. Berikut cara penggunaan method ini dari file index.php:

08.DB_method_getQuery/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 // tampilkan semua tabel barang
6 $tabelBarang = $DB->getQuery('SELECT * FROM barang WHERE id_barang = ?',[2]);
7
8 echo "<pre>";
9 print_r($tabelBarang);
10 echo "</pre>";
```

Hasil kode program:

```
Array
(
    [0] => stdClass Object
        (
            [id_barang] => 2
            [nama_barang] => Kulkas LG GC-A432HLHU
            [jumlah_barang] => 10
            [harga_barang] => 7600000
            [tanggal_update] => 2019-03-12 07:59:50
        )
)
```

Hasil akhir dari method `getQuery()` sudah langsung berbentuk array object yang saya simpan ke dalam variabel `$tabelBarang`. Untuk query `SELECT`, akan lebih praktis jika menggunakan method `getQuery()`, karena kita tidak perlu memanggil lagi method `fetchAll(PDO::FETCH_OBJ)`.

Namun kelebihannya, method `getQuery()` "sudah terkunci" untuk menampilkan hasil dalam bentuk array object. Jika anda ingin mengambil nilai tabel dalam bentuk *associative array* atau *numeric array* maka silahkan pakai method `runQuery()` dan jalankan method `fetchAll(PDO::FETCH_ASSOC)` atau `fetchAll(PDO::FETCH_NUM)` secara manual.

11.7. Class DB: Method get

Kita sudah membuat 2 buah method dasar, yakni `runQuery()` dan `getQuery()`. Kali ini saya akan masuk ke method yang berbentuk query builder, dimana kita tidak perlu lagi menulis query secara langsung. Method ini saya beri nama `get()`, yang berfungsi untuk mengambil semua isi dari sebuah tabel. Method `get()` ini cuma butuh 1 argument, yakni nama tabel yang ingin ditampilkan.

Berikut cara pengaksesan method `get()` dari file index.php:

09.DB_method_get/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 // tampilkan semua tabel barang
6 $tabelBarang = $DB->get('barang');
7
8 echo "<pre>";
9 print_r($tabelBarang);
10 echo "</pre>";
```

Hasil kode program:

```
Array
(
    [0] => stdClass Object
        (
            [id_barang] => 1
            [nama_barang] => TV Samsung 43NU7090 4K
            [jumlah_barang] => 5
            [harga_barang] => 5399000
            [tanggal_update] => 2019-03-12 07:59:50
        )
    ...
    ...
)
```

Perhatikan baris ke 6, itulah cara pemanggilan method get(). Jika dipanggil \$DB->get('barang'), maka akan mengembalikan array object dari tabel barang. Atau jika dipanggil \$DB->get('user'), akan mengembalikan array object dari tabel user.

Silahkan anda coba rancang method get() ini di dalam class DB.

Baik, berikut kode yang saya pakai untuk membuat method get():

09.DB_method_get/DB.php

```
1 <?php
2 class DB{
3
4     // kode program untuk property dan constructor class DB
5     // kode program untuk method runQuery() dan getQuery()
6     // ...
7
8     public function get($tableName){
9         $query = "SELECT * FROM {$tableName}";
10        return $this->getQuery($query);
11    }
12 }
```

Isi dari method get() hanya 2 baris, dimana pada baris 9 saya membuat variabel \$query yang

berisi string "SELECT * FROM {\$tableName} ". Variabel \$tableName sendiri berasal dari argument method.

Di baris 10, terdapat perintah return \$this->getQuery(\$query). Artinya, method get() akan mengembalikan nilai hasil pemanggilan method getQuery(\$query).

Silahkan anda pahami sebentar alur proses pemanggilan method get() karena terdapat "pemanggilan berantai". Di dalam method get(), prosesnya diserahkan ke dalam method getQuery(). Kemudian di dalam method getQuery(), proses kembali diserahkan ke dalam method runQuery().

Meskipun terkesan rumit, tapi pemanggilan berantai seperti ini membuat kode program menjadi lebih efisien karena kita bisa menghindari adanya kode program yang berulang.

Saya bisa saja menulis method get() sebagai berikut:

```
1 <?php
2 class DB{
3
4     public function get($tableName,$bindValue = []){
5         $query = "SELECT * FROM {$tableName}";
6         try {
7             $stmt = $this->_pdo->prepare($query);
8             $stmt->execute($bindValue);
9         }
10        catch (PDOException $e){
11            die("Koneksi / Query bermasalah: ".$e->getMessage().
12                " (".$e->getCode()." )");
13        }
14        return $stmt->fetchAll(PDO::FETCH_OBJ);
15    }
16 }
```

Dengan penulisan seperti ini, method get() bisa berdiri sendiri, tidak bergantung dengan method lain seperti getQuery().

Namun kode program untuk class DB akan banyak yang sama. Proses exception misalnya, sudah kita buat di dalam method runQuery(). Akan lebih efisien jika proses exception ini diserahkan ke method tersebut agar cukup di satu tempat saja, tidak perlu ditulis berulang di setiap method. Begitu juga halnya dengan method fetchAll(PDO::FETCH_OBJ) yang juga sudah tersedia di dalam method getQuery().

Nantinya kita akan buat sekitar 8 method lagi, yang jika tidak dipisah maka proses penanganan exception perlu ditulis ulang di setiap method.

Skill untuk bisa memecah sebuah method menjadi method-method kecil seperti ini memang sangat berguna, tapi hanya bisa dilakukan dengan banyaknya latihan **re-factoring** kode program, yakni menulis ulang kode yang sudah jadi dan membuatnya lebih efisien lagi.

Sedikit bocoran, seluruh method untuk class DB ini saya tulis ulang sekitar 3 kali. Setiap kali

penulisan, saya mencoba membuat class DB lebih efisien lagi dengan cara memecah method untuk mengurangi kode yang berulang.

11.8. Class DB: Method select

Method `select()` yang akan kita buat ini masih berhubungan dengan method `get()`. Saya ingin merancang sebuah mekanisme agar bisa memilih kolom mana saja yang akan ditampilkan. Dengan kode yang ada sekarang, method `get()` selalu menampilkan semua kolom yang ada di tabel.

Untuk cara kerja method `select()` ini saya terinspirasi dengan *query builder* bawaan framework **Code Igniter**. Untuk menentukan kolom yang akan diambil, kita akan panggil method `select()` terlebih dahulu.

Berikut konsep penggunaan method `select()` dari file `index.php`:

10.DB_method_get_select/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 // pilih kolom nama_barang
6 $DB->select('nama_barang');
7 $tabelBarang = $DB->get('barang');
8
9 echo "<pre>";
10 print_r($tabelBarang);
11 echo "</pre>";
```

Hasil kode program:

```
Array
(
    [0] => stdClass Object
        (
            [nama_barang] => TV Samsung 43NU7090 4K
        )

    ...
    ...

    [5] => stdClass Object
        (
            [nama_barang] => Cosmos CRJ-8229 - Rice Cooker
        )
)
```

Di baris 6 terdapat perintah `$DB->select('nama_barang')`. Method `select()` ini akan men-set

"sesuatu" di class DB, sehingga ketika dipanggil method `$DB->get('barang')`, akan berisi kolom `nama_barang` saja.

Bagaimana cara membuatnya?

Pertama, di dalam class DB saya akan tambah sebuah private property `$_columnName`. Property ini dipakai untuk menampung nama kolom dari hasil pemanggilan method `select()`:

```
private $_columnName = "*";
```

Property `$_columnName` berisi nilai awal karakter bintang '*' agar ketika method `select()` tidak dipanggil, maka tampilkan semua kolom.

Berikutnya, buat method `select()` yang dipakai untuk mengubah isi property `$_columnName`:

```
public function select($columnName){  
    $this->_columnName = $columnName;  
}
```

Artinya, ketika method `select('nama_barang')` dipanggil, string 'nama_barang' akan menimpa isi property `$_columnName`.

Terakhir, kita perlu modifikasi method `get()` sebagai berikut:

10.DB_method_get_select/DB.php

```
1  public function get($tableName){  
2      $query = "SELECT {$this->_columnName} FROM {$tableName}";  
3      $this->_columnName = "*";  
4      return $this->getQuery($query);  
5  }
```

Perubahannya ada di string `$query`. Setelah perintah `SELECT`, terdapat pemanggilan variabel `$this->_columnName`. Dalam perintah query MySQL, bagian ini adalah tempat untuk menulis nama kolom.

Misalnya jika kita ingin menampilkan kolom `nama_barang` dari tabel `barang`, query yang harus ditulis adalah '`SELECT nama_barang FROM barang`'. Atau jika ingin menampilkan seluruh kolom, maka querynya adalah '`SELECT * FROM barang`'. Bagian nama kolom inilah yang kita ambil dari property `$this->_columnName`.

Di baris 3 terdapat tambahan perintah `$this->_columnName = "*"`. Ini dipakai untuk me-reset ulang isi property `$_columnName` agar kembali menjadi '*'. Karena kalau tidak, pemanggilan method `get()` berikutnya akan terpengaruh dengan pemanggilan method `select()` sebelumnya.

Dengan tambahan ini, method `get()` menjadi lebih fleksibel karena kita bisa menentukan nama kolom yang ingin diambil. Jika method `select()` tidak dipanggil, artinya tampilkan seluruh kolom.

Berikut contoh pemanggilan method `select()` dan `get()`:

```
// ambil seluruh kolom
$tabelBarang = $DB->get('barang');

// ambil kolom nama_barang
$DB->select('nama_barang');
$tabelBarang = $DB->get('barang');

// ambil kolom nama_barang, id_barang dan harga_barang
$DB->select('nama_barang,id_barang,harga_barang');
$tabelBarang = $DB->get('barang');

// ambil seluruh kolom
$tabelBarang = $DB->get('barang');
```

Khusus untuk pemanggilan yang terakhir, akan menampilkan seluruh kolom karena setiap kali method `get()` dipanggil nilai property `$_columnName` akan di reset ulang menjadi tanda bintang `"*"`.

Agar lebih "canggih" lagi, saya ingin menerapkan konsep **method chaining** ke dalam method `select()`. Maksudnya agar kita bisa memanggil method `select()` dan `get()` dengan cara berikut:

11.DB_method_get_select_chaining/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->select('nama_barang')->get('barang');
6 $tabelBarang = $DB->select('harga_barang, nama_barang')->get('barang');
```

Di baris 5 dan 6, saya menyambung pemanggilan method `select()` dan `get()`. Untuk mendapatkan fitur seperti ini, kita hanya perlu menambah 1 baris ke dalam method `select()`:

11.DB_method_get_select_chaining/DB.php

```
1 <?php
2 class DB{
3     // ...
4     // ...
5
6     public function select($columnName){
7         $this->_columnName = $columnName;
8         return $this;
9     }
10 }
11 }
```

Yup, hanya perlu menambahkan baris perintah `return $this` di baris 8, maka secara otomatis kita sudah bisa melakukna *method chaining*.

11.9. Class DB: Method orderBy

Method `orderBy()` di rancang agar kita bisa mengatur urutan baris tampilan tabel. Dalam MySQL, ini bisa dilakukan dengan tambahan clausa '`ORDER BY <nama_kolom> ASC`' atau '`ORDER BY <nama_kolom> DESC`'.

Teknik yang akan saya pakai sama seperti method `select()`, yakni dengan memanggil method `orderBy()` sebelum menjalankan query `get()`. Berikut contoh penggunaannya dari file `index.php`:

12.DB_method_get_orderBy/index.php

```

1  <?php
2  require 'DB.php';
3  $DB = DB::getInstance();
4
5  $DB->select('id_barang,nama_barang');
6  $DB->orderBy('id_barang','DESC');
7  $tabelBarang = $DB->get('barang');
8
9  echo "<pre>";
10 print_r($tabelBarang);
11 echo "</pre>";

```

Hasil kode program:

```

Array
(
    [0] => stdClass Object
        (
            [id_barang] => 6
            [nama_barang] => Cosmos CRJ-8229 - Rice Cooker
        )

    [1] => stdClass Object
        (
            [id_barang] => 5
            [nama_barang] => Smartphone Xiaomi Pocophone F1
        )

    ...
    [5] => stdClass Object
        (
            [id_barang] => 1
            [nama_barang] => TV Samsung 43NU7090 4K
        )
)

```

Di baris 5 saya menjalankan method `select()` untuk menentukan kolom apa saja yang akan diambil. Dalam contoh ini kolom tersebut adalah '`id_barang, nama_barang`'.

Di baris 6 terdapat pemanggilan method `orderBy('id_barang', 'DESC')`. Method ini diisi dengan 2 buah argument. Argument pertama berupa nama kolom yang menjadi patokan pengurutan, yakni `id_barang`. Argument kedua berupa metode pengurutan, apakah akan diurut secara menaik ('`ASC`') atau secara menurun ('`DESC`').

Terakhir pada baris 7 terdapat perintah untuk menjalankan method `$DB->get('barang')`.

Hasil kode program dari baris 5 – 7 akan menghasilkan query MySQL sebagai berikut:

```
SELECT id_barang, nama_barang FROM barang ORDER BY id_barang DESC
```

Dan seperti yang terlihat, array `$tabelBarang` ditampilkan dari `id_barang` 6, 5, 4 sampai 1, yakni di urutkan menurun berdasarkan kolom `id_barang`.

Bagaimana cara pembuatannya? Kurang lebih mirip seperti pembuatan method `select()`. Pertama, saya butuh sebuah private property `$_orderBy`:

```
private $_orderBy = "";
```

Property ini tidak memiliki nilai awal karena akan diisi dari dalam method `orderBy()`:

```
public function orderBy($columnName, $sortType = 'ASC'){
    $this->_orderBy = "ORDER BY {$columnName} {$sortType}";
    return $this;
}
```

Di sini property `$_orderBy` diisi dengan string "`ORDER BY {$columnName} {$sortType}`". Variabel `$columnName` berasal dari argument pertama, dan variabel `$sortType` berasal dari argument kedua. Khusus untuk variabel `$sortType`, memiliki nilai opsional '`ASC`' yang akan dipakai jika method `orderBy()` dipanggil tanpa menyertakan nilai argument `$sortType`.

Di baris terakhir terdapat perintah `return $this` untuk membuat efek *method chaining* seperti di method `select()` sebelumnya.

Langkah terakhir adalah memodifikasi method `get()`:

```
12.DB_method_get_orderBy/DB.php
```

```
1 public function get($tableName){
2     $query = "SELECT {$this->_columnName} FROM {$tableName} {$this->_orderBy}";
3     $this->_columnName = "*";
4     $this->_orderBy = "";
5     return $this->getQuery($query);
6 }
```

Perubahan ada di baris ke-2, yakni isi dari variabel `$query`. Terdapat tambahan variabel `->{$this->_orderBy}` di akhir string. Kemudian di baris 4 terdapat perintah `$this->_orderBy = ""` yang dipakai untuk me-reset ulang isi property `$_orderBy`.

Dengan penambahan ini, kita sudah bisa memakai method `orderBy()` untuk mengatur urutan

tampilan kolom. Karena juga mendukung *method chaining*, pemanggilan method bisa dirangkai sebagai berikut :

```
$tabelBarang = $DB->select('harga_barang, nama_barang')->orderBy('harga_barang')
->get('barang');
```

Yang artinya sama dengan pemanggilan query:

```
SELECT harga_barang, nama_barang FROM barang ORDER BY harga_barang ASC
```

Karena method `orderBy()` dipanggil tanpa menyertakan argument kedua, maka akan memakai nilai default `ASC`.

11.10. Class DB: Method get (condition)

Class DB yang kita rancang sudah memiliki *query builder* untuk proses pengambilan tabel, pemilihan nama kolom serta pengurutan. Sekarang kita akan tambah dengan kondisi `WHERE`.

Namun daripada membuat sebuah method baru, saya akan modifikasi method `get()` agar bisa menerima argument berupa sebuah kondisi `WHERE`. Kondisi `WHERE` ini akan diinput sebagai argument kedua. Selain itu karena kita menerapkan prepared statement, maka butuh argument ketiga untuk data *placeholder*.

Berikut contoh pemanggilan method `get` dengan tambahan kondisi `WHERE`:

13.DB_method_get_condition/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->get('barang', 'WHERE id_barang = ?',[2]);
```

Perintah di baris ke-5 akan diproses sebagai: `SELECT * FROM barang WHERE id_barang = 2`

Dan berikut modifikasi untuk method `get()` di class DB:

13.DB_method_get_condition/DB.php

```
1 <?php
2 class DB{
3
4     public function get($tableName, $condition = "", $bindValue =[]){
5         $query = "SELECT {$this->_columnName} FROM {$tableName} {$condition}
6             {$this->_orderBy}";
7         $this->_columnName = "*";
8         $this->_orderBy = "";
9         return $this->getQuery($query, $bindValue);
10    }
11 }
```

Method `get()` mendapat 2 tambahan argument, yakni `$condition` dan `$bindValue`. Variabel `$condition` akan ditambah ke dalam string `$query`, yakni setelah `[$tableName]` dan sebelum `[$this->_orderBy]`. Sedangkan variabel `$bindValue` langsung diteruskan sebagai argument kedua untuk pemanggilan method `getQuery()` di baris 9.

Perhatikan bahwa argument `$condition` dan `$bindValue` memiliki nilai default, yang berarti keduanya bersifat opsional. Jika kedua argument tidak ditulis, maka itu berarti tanpa kondisi WHERE. Dari halaman `index.php`, kita tetap bisa memanggil method `get()` dengan hanya 1 argument saja:

13.DB_method_get_condition/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->get('barang'); // ambil semua isi tabel barang
```

Lebih jauh lagi, method `get()` sudah bisa menangani pembuatan query yang cukup kompleks, seperti contoh berikut:

13.DB_method_get_condition/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $DB->select('harga_barang, nama_barang');
6 $DB->orderBy('harga_barang', 'DESC');
7 $tabelBarang = $DB->get('barang', 'WHERE harga_barang > ?',[5000000]);
```

Perintah query di baris 5 – 7 sama dengan berikut:

```
SELECT harga_barang, nama_barang FROM barang
WHERE harga_barang > 5000000 ORDER BY harga_barang DESC
```

Sebagai latihan, silahkan anda coba panggil method `select()`, `orderBy()` dan `get()` di atas menjadi 1 baris perintah dengan teknik *method chaining*.

11.11. Class DB: Method `getWhere`

Pengambilan data tabel dengan kondisi WHERE cukup sering kita lakukan, oleh karena itu saya akan buat method khusus untuk keperluan ini, yakni `getWhere()`. Method `getWhere()` ini merupakan pengembangan dari method `get()` dengan kondisi WHERE yang baru saja kita buat.

Dalam method `get()`, kondisi WHERE harus ditulis lengkap seperti contoh berikut:

```
$tabelBarang = $DB->get('barang', 'WHERE harga_barang > ?',[5000000]);
```

Untuk method `getWhere()`, saya ingin query tersebut tidak lagi tulis, tapi cukup datanya saja:

```
$tabelBarang = $DB->getWhere('barang',[ 'harga_barang' , '>' ,5000000]);
```

Di sini, argument kedua berbentuk array dengan 3 komponen, yakni nama **kolom kondisi**, **operator**, dan **nilai**. Nantinya, kode program di dalam method `getWhere()` lah yang akan meng-generate perintah query.

Contoh lain, query `SELECT * FROM barang WHERE id_barang = 3` bisa dijalankan dengan perintah:

```
$tabelBarang = $DB->getWhere('barang',[ 'id_barang' , '=' ,3]);
```

Bagaimana rancangan method `getWhere()` ini? Berikut kode program yang saya gunakan:

14. DB_method_getWhere

```
1  <?php
2  class DB{
3      // ...
4      // ...
5
6      public function getWhere($tableName, $condition){
7          $queryCondition ="WHERE {$condition[0]} {$condition[1]} ? ";
8          return $this->get($tableName,$queryCondition,[ $condition[2] ]);
9      }
10 }
```

Method `getWhere()` butuh 2 buah argument yang ditampung ke dalam variabel `$tableName` dan `$condition`.

Di baris 7, variabel `$queryCondition` dipakai untuk membuat ulang cluasa `WHERE` menggunakan nilai dari `[$condition[0]]` dan `[$condition[1]]` serta sebuah tanda tanya "?". Ini diperlukan karena kita akan menjalankan kondisi menggunakan PDO prepared statement.

Sebagai contoh, ketika dipanggil perintah berikut:

```
$tabelBarang = $DB->getWhere('barang',[ 'harga_barang' , '>' ,5000000]);
```

String 'barang' akan ditampung ke dalam variabel `$tableName`, sedangkan array `['harga_barang' , '>' , 5000000]` ditampung ke dalam `$condition`. Untuk mengakses setiap element array `$condition`, kita harus menggunakan index `$condition[0]`, `$condition[1]` dan `$condition[2]`.

Dengan demikian, perintah "WHERE {\$condition[0]} {\$condition[1]} ?" akan dikonversi menjadi: "WHERE harga_barang > ?".

Terakhir, di baris 8 proses query di alihkan ke method `get()` dengan mengirim 3 buah argument, yakni `$tableName`, `$queryCondition`, dan `[$condition[2]]`. Dalam contoh kita, perintah di baris 8 ini akan dikonversi sebagai berikut:

```
return $this->get('barang', "WHERE harga_barang > ?", [5000000]);
```

Sisa pemrosesan selanjutnya akan dikerjakan oleh method `get()`. Inilah salah satu keuntungan dari teknik memecah method, dimana method yang satu akan saling memanggil method lain untuk bersama-sama memecahkan sebuah masalah.

Keuntungan lain dengan memanggil method `get()` secara internal, adalah method `getWhere()` secara otomatis juga bisa memanfaatkan *method chaining* milik method `get()`, seperti contoh berikut:

14.DB_method_getWhere/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->select('nama_barang,jumlah_barang')
6             ->getWhere('barang',[ 'id_barang' , '=' , 5]);
7
8 echo "<pre>";
9 print_r($tabelBarang);
10 echo "</pre>";
```

Hasil kode program:

```
Array
(
    [0] => stdClass Object
        (
            [nama_barang] => Smartphone Xiaomi Pocophone F1
            [jumlah_barang] => 25
        )
)
```

Perintah di baris 5 akan diproses oleh class `DB` sebagai:

```
SELECT nama_barang, jumlah_barang FROM barang WHERE id_barang = 5
```

Dengan class `DB`, kita tidak perlu menulis manual query ini, tapi cukup mengakses method yang tersedia.

11.12. Class DB: Method `getWhereOnce`

Dalam banyak situasi, kadang kita ingin mengambil data pertama dari hasil query `SELECT`, atau kita yakin bahwa hasil query tersebut hanya akan mengembalikan 1 data saja. Dengan method `getWhereOnce()` yang sudah tersedia, ini bisa dilakukan namun dengan sedikit kendala.

Bisakah anda menebak apa yang salah dari kode berikut?

15.DB_method_getWhere_problem/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->getWhere('barang',[ 'id_barang' , '=' ,5]);
6
7 echo $tabelBarang->nama_barang;
```

Hasil kode program:

Warning: Attempt to read property "nama_barang" on array in index.php on line 7

Error ini terjadi karena `$tabelBarang->nama_barang` tidak bisa diakses. Cara pengaksesan yang benar adalah `$tabelBarang[0]->nama_barang`. Tambahan angka atau index [0] berhubungan dengan metode yang dipakai untuk mengambil data.

Secara internal, method `getWhere()` menggunakan method `getQuery()` untuk mengambil data, yakni menggunakan method `fetchAll(PDO::FETCH_OBJ)` dari PDO. Hasilnya, variabel `$tabelBarang` berisi **array dari object**, bukan hanya object saja.

Dalam contoh di atas variabel `$tabelBarang` hanya berisi 1 baris data, tetapi saja kita harus tulis index [0] seperti `$tabelBarang[0]->nama_barang`.

Saya memutuskan membuat method khusus untuk keperluan ini, yakni `getWhereOnce()`. Method `getWhereOnce()` akan mengembalikan data langsung berbentuk object, bukan lagi array dari object. Dengan demikian, kita tidak harus menulis index [0] setiap kali ingin menampilkan hasil tabel.

Dengan method `getWhereOnce()`, pemanggilan di atas bisa dijalankan sebagai berikut:

16.DB_method_getWhereOnce/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->getWhereOnce('barang',[ 'id_barang' , '=' ,5]);
6
7 echo $tabelBarang->nama_barang;
```

Hasil kode program:

Smartphone Xiaomi Pocophone F1

Dan berikut kode program untuk membuat method `getWhereOnce()` dalam class DB:

16.DB_method_getWhereOnce/DB.php

```
1 <?php
2 class DB{
```

```

3 // ...
4 // ...
5
6 public function getWhereOnce($tableName, $condition){
7     $result = $this->getWhere($tableName,$condition);
8     if (!empty($result)) {
9         return $result[0];
10    } else {
11        return false;
12    }
13 }
14
15 }
```

Prinsip kerja dari method `getWhereOnce()` sebenarnya hanya dengan memanggil method `getWhere()[0]`. Namun ada sedikit tambahan pemeriksaan kondisi.

Di baris 7, hasil pemanggilan method `getWhere()` saya simpan ke dalam variabel `$result`. Kemudian isinya diperiksa menggunakan kondisi `if(!empty($result))`. Jika variabel `$result` berisi suatu nilai, maka kembalikan hasil dari `$result[0]`. Namun jika isi variabel `$result` ternyata kosong, yang berarti data tidak ditemukan, maka kembalikan nilai `false`.

Dengan penulisan seperti ini, method `getWhereOnce()` hanya mengembalikan 1 baris pertama dari hasil query, meskipun query tersebut bisa mengambil lebih dari 1 baris data. Sebagai contoh, jika yang dijalankan adalah perintah berikut:

```
$tabelBarang = $DB->getWhereOnce('barang',[ 'harga_barang' , '>' ,5000000]);
```

Maka variabel `$tabelBarang` hanya akan berisi baris pertama dari barang yang harganya lebih dari 5000000. Meskipun sebenarnya ada banyak barang dengan harga lebih dari 5000000 di dalam tabel.

Method `getWhereOnce()` ini cocok dipakai untuk query yang hanya mengembalikan 1 baris saja, seperti query dengan kondisi `WHERE id_barang = 5`. Query ini hanya akan berisi 1 baris karena kolom `id_barang` sudah di set sebagai `PRIMARY KEY` yang tidak bisa diisi nilai ganda.

Sebenarnya method `getWhereOnce()` bisa saja ditulis seperti ini:

```

1 public function getWhereOnce($tableName, $condition){
2     return $this->getWhere($tableName,$condition)[0];
3 }
```

Namun kode ini akan error jika method `getWhere()` tidak mengembalikan apa-apa, yakni kondisi dimana data tidak ditemukan di dalam tabel. Karena itulah kita perlu membuat sebuah pemeriksaan `if(!empty($result))` terlebih dahulu.

11.13. Class DB: Method getLike

Method `getLike()` saya rancang untuk menjalankan query `SELECT` dengan kondisi `LIKE`. Dalam bahasa SQL, perintah `LIKE` dipakai dalam proses pencarian.

Konsep pembuatan method `getLike()` ini mirip seperti method `getWhere()`, dimana saya akan rancang string tambahan, lalu memanggil method `get()` secara internal.

Berikut contoh pemanggilan method `getLike()` dari halaman `index.php`:

17.DB_method_getLike/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->getLike('barang', 'nama_barang', '%kulkas%');
6
7 echo "<pre>";
8 print_r($tabelBarang);
9 echo "</pre>";
```

Hasil kode program:

```
Array
(
    [0] => stdClass Object
        (
            [id_barang] => 2
            [nama_barang] => Kulkas LG GC-A432HLHU
            [jumlah_barang] => 10
            [harga_barang] => 7600000
            [tanggal_update] => 2019-03-12 07:59:50
        )
)
```

Method `getLike()` perlu 3 buah argument, yakni nama tabel, kolom yang akan dicari, dan pola pencarian. Pemanggilan method `getLike('barang', 'nama_barang', '%kulkas%')` di baris 5 akan meng-generate query berikut:

```
SELECT * FROM barang WHERE nama_barang LIKE '%kulkas%'
```

Query di atas akan mencari semua data barang yang memiliki kata "kulkas" di kolom `nama_barang`.

Berikut kode yang saya pakai untuk membuat method `getLike()` dalam class DB:

17.DB_method_getLike/DB.php

```
1 <?php
2 class DB{
3     // ...
```

```

4 // ...
5
6 public function getLike($tableName, $columnLike, $search){
7     $queryLike = "WHERE {$columnLike} LIKE ?";
8     return $this->get($tableName,$queryLike,[[$search]]);
9 }
10 }
```

Ketika dijalankan perintah `$DB->getLike('barang', 'nama_barang', '%kulkas%')`, maka di dalam method `getLike()`, variabel `$queryLike` akan berisi string `"WHERE nama_barang LIKE ?"`. Tanda tanya "?" diperlukan karena di dalam class DB kita memproses semua query sebagai *prepared statement*. Nilai pengganti untuk tanda tanya ini (nilai placeholder), yakni pola pencarian ditampung ke dalam argument `$search`.

Selanjutnya, proses bisa kita alihkan ke method `get()` dengan mengirim argument berupa `$tableName`, `$queryLike`, dan `[$search]`. Khusus untuk argument `$search`, sengaja menggunakan tanda kurung siku "[..]" agar dikonversi menjadi tipe data array. Ini diperlukan karena argument ini akan menjadi inputan untuk method `execute()` di dalam method `get()`. Dalam PDO prepared statement, method `execute()` harus menerima inputan dalam bentuk array, meskipun isinya hanya 1 data saja.

Dengan memanggil method `get()` secara internal, maka method `getLike()` otomatis juga mendukung berbagai method tambahan seperti `select()`, `orderBy()` serta teknik method *chaining* sebagaimana yang kita rancang untuk method `get()`. Berikut contohnya:

17.DB_method_getLike/index.php

```

1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $tabelBarang = $DB->select('nama_barang,id_barang')
6             ->getLike('barang','nama_barang','%k%');
7
8 echo "<pre>";
9 print_r($tabelBarang);
10 echo "</pre>";
```

Hasil kode program:

```

Array
(
    [0] => stdClass Object
        (
            [nama_barang] => TV Samsung 43NU7090 4K
            [id_barang] => 1
        )

    [1] => stdClass Object
        (
```

```
[nama_barang] => Kulkas LG GC-A432HLHU
[id_barang] => 2
)

[2] => stdClass Object
(
    [nama_barang] => Cosmos CRJ-8229 - Rice Cooker
    [id_barang] => 6
)
)
```

Pemanggilan method `getLike()` di baris 5–6 akan menjalankan query berikut:

```
SELECT nama_barang, id_barang FROM barang WHERE nama_barang LIKE '%k%'
```

Query ini akan mencari seluruh barang dimana kolom `nama_barang` mengandung huruf 'k', ini cocok dengan data TV Samsung 43NU7090 4K, Kulkas LG GC-A432HLHU dan Cosmos CRJ-8229 - Rice Cooker.

11.14. Class DB: Method check

Method `check()` saya rancang untuk memeriksa apakah sebuah data ada di dalam tabel. Ini sering dipakai untuk proses validasi form, misalnya memeriksa apakah sebuah username sudah ada di dalam tabel atau belum.

Teknik yang akan saya pakai adalah dengan menjalankan method `rowCount()` bawaan PDO. Method `rowCount()` akan mengembalikan jumlah baris dari hasil query `SELECT`. Jika method ini menghasilkan nilai 0 maka artinya tidak ada data. Jika hasilnya 1 atau lebih, artinya data tersebut ada di dalam tabel.

Berikut contoh penggunaan method `check()` dari halaman `index.php`:

18.DB_method_check/index.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->check('barang', 'id_barang', '4');
6 echo $result;
```

Hasil kode program:

1

Angka 1 di sini berarti dalam tabel barang hanya terdapat 1 baris yang memiliki `id_barang` = 4.

Berikut contoh lainnya:

18.DB_method_check/index.php

Case Study: Database Query Builder

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->check('barang', 'id_barang', '10');
6 echo $result;
```

Hasil kode program:

```
0
```

Angka 0 berarti dalam tabel barang tidak ditemukan baris yang memiliki `id_barang = 10`.

Berikut kode program yang saya pakai untuk membuat method `check()`:

```
18.DB_method_check/DB.php
```

```
1 <?php
2 class DB{
3     // ...
4     // ...
5
6     public function check($tableName, $columnName, $dataValues){
7         $query = "SELECT {$columnName} FROM {$tableName} WHERE {$columnName} = ? ";
8         return $this->runQuery($query, [$dataValues])->rowCount();
9     }
10 }
```

Method `check()` butuh 3 buah argument, yakni nama tabel, nama kolom dan nilai. Ketiganya ditampung ke dalam variabel `$tableName`, `$columnName`, dan `$dataValues`. Di baris 7, saya menulis seluruh query yang dibutuhkan dari argument ini.

Sebagai contoh, jika yang dipanggil adalah:

```
$result = $DB->check('barang', 'id_barang', '10');
```

Maka variabel `$query` akan berisi string:

```
$query = "SELECT id_barang FROM barang WHERE id_barang = ?"
```

Tanda tanya "?" di akhir string diperlukan karena kita menggunakan prepared statement. Kemudian variabel `$query` ini akan dikirim ke method `runQuery()`, beserta variabel `$dataValues`.

Hasil pemanggilan method `runQuery()` berbentuk **PDO Statement object** yang langsung disambung dengan method `rowCount()` di akhir baris 8.

Method `check()` ini bisa langsung dipakai untuk pengecekan kondisi, seperti contoh berikut:

```
18.DB_method_check/index.php
```

```
1 <?php
```

```
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 if ($DB->check('barang','id_barang','4')) {
6     echo "ID barang 4 tersedia";
7 }
```

Hasil kode program:

```
ID barang 4 tersedia
```

Di baris 5 saya langsung menulis `if ($DB->check('barang','id_barang','4'))`, ini bisa dilakukan karena PHP akan mengkonversi angka menjadi tipe data boolean.

Method `check()` akan mengembalikan nilai 0 jika data tidak ditemukan, yang akan dikonversi menjadi boolean **false**. Namun jika method `check()` mengembalikan angka selain 0, itu akan dikonversi menjadi **true**.

Method `check()` ini menjadi method terakhir untuk proses `SELECT`, berikutnya kita akan masuk ke method untuk memproses query `INSERT`, `UPDATE` dan `DELETE`. Ketiga query ini menjadi tantangan tersendiri karena harus kita proses sebagai prepared statement.

11.15. Class DB: Method insert

Sesuai dengan namanya, method `insert()` dipakai untuk memproses query `INSERT`. Method `insert()` yang akan kita rancang ini menjadi salah satu method yang paling rumit di dalam class `DB`. Ini karena jumlah data yang diinput bisa berbeda-beda, serta kita juga tetap akan menggunakan prepared statement.

Berikut contoh penggunaan method `insert()`:

```
19.DB_method_insert/insert_method.php
```

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $DB->insert('barang',[  
6     'nama_barang' => 'Philips Blender HR 2157',  
7     'jumlah_barang' => 11,  
8     'harga_barang' => 629000  
9 ]);
```

Method `insert()` saya rancang dengan 2 buah argument, yang pertama berupa nama tabel yang akan ditambahkan, dan kedua berupa *associative array* yang berisi pasangan nama kolom dan nilai untuk kolom tersebut.

Tugas kita adalah, bagaimana cara mengonversi pemanggilan method `insert()` di baris 5 – 9 menjadi query berikut:

```
INSERT into barang (nama_barang, jumlah_barang, harga_barang) VALUES ('Philips  
Blender HR 2157', 11, 629000)
```

Untuk keperluan ini kita perlu "membongkar" isi associative array. Dan akan sedikit rumit karena kita butuh query versi *prepared statement*. Maksudnya, query yang diperlukan oleh method `insert()` adalah sebagai berikut:

```
INSERT into barang (nama_barang, jumlah_barang, harga_barang) VALUES (?, ?, ?)
```

Dimana nilai dari setiap *placeholder* akan dikirim terpisah, yakni dalam bentuk array ['Philips
Blender HR 2157', 11, 629000]

Proses pembuatan method `insert()` akan saya bahas secara bertahap. Nantinya kita butuh berbagai function bawaan PHP seperti `array_keys()`, `array_values()`, `implode()`, serta `str_repeat()`.

Kita akan mulai dengan memecah associative array. Dalam PHP, associative array adalah array yang key atau indexnya berupa string seperti contoh berikut:

```
1 $foo = [  
2   'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',  
3   'jumlah_barang' => 4,  
4   'harga_barang' => 299000  
5 ];
```

Untungnya PHP menyediakan function bawaan untuk mengambil key dan value secara terpisah, yakni menggunakan function `array_keys()` dan `array_values()`. Sesuai namanya, function `array_keys()` dipakai untuk mengambil nilai **key** dari sebuah array, sedangkan function `array_values()` dipakai untuk mengambil nilai atau **value** dari sebuah array.

Berikut contoh penggunaannya:

19.DB_method_insert/insert_function_1.php

```
1 <?php  
2 $foo = [  
3   'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',  
4   'jumlah_barang' => 4,  
5   'harga_barang' => 299000  
6 ];  
7  
8 $dataKeys = array_keys($foo);  
9 $dataValues = array_values($foo);  
10  
11 print_r($dataKeys);  
12 echo "<br>";  
13 print_r($dataValues);
```

Hasil kode program:

```
Array ( [0] => nama_barang [1] => jumlah_barang [2] => harga_barang )
Array ( [0] => Cosmos CRJ-8229 - Rice Cooker [1] => 4 [2] => 299000 )
```

Terlihat kita sudah bisa memisahkan antara key dan value dari associative array \$foo. Seluruh key dari array \$foo tersimpan ke dalam variabel \$dataKeys, dan untuk value dari variabel \$foo tersimpan ke dalam variabel \$dataValues.

Agar pembahasan kita lebih sederhana, saya akan buat sebuah fungsi insert(). Di dalam fungsi insert() ini kita akan coba rancang kode program untuk meng-generate query INSERT, dan jika sudah berhasil, baru nanti dipindahkan ke dalam class DB:

19.DB_method_insert/insert_function_2.php

```
1 <?php
2 insert('barang',[ 
3     'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',
4     'jumlah_barang' => 4,
5     'harga_barang' => 299000
6 ]);
7
8 function insert($tableName, $data){
9     $dataKeys = array_keys($data);
10    $dataValues = array_values($data);
11 }
```

Di awal kode program saya memanggil fungsi insert() dengan argument yang sama persis seperti method insert() sebelumnya. Argument pertama berupa nama tabel dan argument kedua berbentuk associative array.

Kode program untuk function insert() itu sendiri ada di baris 8 – 10. Argument \$tableName akan menampung nama tabel, dan argument \$data dipakai untuk menampung associative array.

Fungsi insert() ini sangat pas sebagai simulasi method insert() untuk class DB nanti. Tujuan akhir kita adalah bagaimana meng-generate sebuah query INSERT dalam bentuk prepared statement.

Jika function insert() dipanggil sebagai berikut:

```
insert('barang',[ 
    'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',
    'jumlah_barang' => 4,
    'harga_barang' => 299000
]);
```

Saya ingin nanti bisa men-generate query:

```
'INSERT INTO barang (nama_barang, jumlah_barang, harga_barang) VALUES (?,?,?,?)';
```

Saat ini kita sudah berhasil mengumpulkan semua **key** dari array `$data` yang disimpan ke dalam variabel `$dataKeys`. Langkah berikutnya adalah memproses array `$dataKeys` ini menjadi sebuah string.

Sekarang variabel `$dataKeys` berisi array dengan 3 element:

```
['nama_barang', 'jumlah_barang', 'harga_barang']
```

Kita akan rancang kode program agar array ini bisa menjadi string: '(`nama_barang`, `jumlah_barang`, `harga_barang`)'. Untuk proses konversi dari array menjadi string, PHP sudah menyediakan fungsi bawaan yakni `implode()`. Berikut hasilnya:

19.DB_method_insert/insert_function_3.php

```
1 <?php
2 insert('barang',[ 
3     'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',
4     'jumlah_barang' => 4,
5     'harga_barang' => 299000
6 ]);
7
8 function insert($tableName, $data){
9     $dataKeys = array_keys($data);
10    $dataValues = array_values($data);
11
12    $result= implode(', ', $dataKeys);
13
14    echo($result);
15 }
```

Hasil kode program:

```
nama_barang, jumlah_barang, harga_barang
```

Di baris 12, saya menjalankan fungsi `implode(' ', '$dataKeys')`. Hasilnya, seluruh array akan disatukan menjadi string dengan tanda koma sebagai pemisah. Sekarang tinggal menambahkan string tanda kurung di awal dan akhir:

19.DB_method_insert/insert_function_4.php

```
1 <?php
2 insert('barang',[ 
3     'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',
4     'jumlah_barang' => 4,
5     'harga_barang' => 299000
6 ]);
7
8 function insert($tableName, $data){
9     $dataKeys = array_keys($data);
10    $dataValues = array_values($data);
11
12    $result= '('.implode(', ', $dataKeys).')';
13 }
```

```
13     echo($result);
14 }
```

Hasil kode program:

```
(nama_barang, jumlah_barang, harga_barang)
```

Di baris 12 saya menambah awalan "(" dan akhiran ")", agar string berada di dalam tanda kurung. Dengan demikian, string ini kita bisa rangkai sebagai berikut:

19.DB_method_insert/insert_function_5.php

```
1 <?php
2 insert('barang',[  
3     'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',  
4     'jumlah_barang' => 4,  
5     'harga_barang' => 299000  
6 ]);  
7  
8 function insert($tableName, $data){  
9     $dataKeys = array_keys($data);  
10    $dataValues = array_values($data);  
11  
12    echo "INSERT INTO {$tableName} (" . implode(', ', $dataKeys) . ')';  
13 }
```

Hasil kode program:

```
INSERT INTO barang (nama_barang, jumlah_barang, harga_barang)
```

Sip, bagian awal query sudah selesai. Selanjutnya adalah bagaimana cara membuat tanda tanya sejumlah data yang akan diinput.

Sebagaimana yang sudah kita pelajari, prepared statement menggunakan placeholder berupa tanda tanya "?". Jumlah tanda tanya ini harus sesuai dengan jumlah data yang diinput. Kita bisa mengetahui jumlah data ini dari total element yang ada di dalam array \$data.

Untuk mengetahui jumlah element dari sebuah array, PHP menyediakan fungsi `count()`. Dalam contoh kita, `count($data)` akan menghasilkan angka 3. Artinya untuk tempat placeholder perlu 3 buah tanda tanya.

Untuk membuat 3 buah tanda tanya secara dinamis, bisa menggunakan perulangan `for`, atau bisa juga menggunakan fungsi `str_repeat()` bawaan PHP. Berikut hasil penggunaannya:

19.DB_method_insert/insert_function_6.php

```
1 <?php
2 insert('barang',[  
3     'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',  
4     'jumlah_barang' => 4,  
5     'harga_barang' => 299000  
6 ]);
```

Case Study: Database Query Builder

```
7
8  function insert($tableName, $data){
9    $dataKeys = array_keys($data);
10   $dataValues = array_values($data);
11
12   echo str_repeat('?', ', count($data));
13 }
```

Hasil kode program:

```
? , ? , ? ,
```

Fungsi `str_repeat()` akan mengulang karakter yang diinput sebagai argument pertama sebanyak nilai di argument kedua. Perintah di baris 12 akan diproses sebagai `echo str_repeat('?', ', 3)`. Hasilnya, string `'? ,'` akan di ulang sebanyak 3 kali.

Namun ada sedikit masalah. Kita harus menghapus tanda koma terakhir, yakni agar hasilnya menjadi:

```
? , ? , ?
```

Untuk ini saya akan memakai sedikit "trik". Daripada mengulang tanda tanya dan koma `'? ,'` sejumlah data yang ada, saya akan kurangi 1 dan menulis manual tanya tanya terakhir.

Perintahnya menjadi sebagai berikut:

```
echo str_repeat('?', ', count($data)-1).'?' ;
```

Sehingga yang akan diproses adalah: `str_repeat('?', ', 2)` yang ditambah dengan 1 karakter `'?'` di bagian akhir.

Placeholder ini juga perlu tambahan tanda kurung di awal dan akhir string:

19.DB_method_insert/insert_function_7.php

```
1  <?php
2  insert('barang',[  
3    'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',  
4    'jumlah_barang' => 4,  
5    'harga_barang' => 299000  
6  ]);  
7
8  function insert($tableName, $data){  
9    $dataKeys = array_keys($data);  
10   $dataValues = array_values($data);  
11
12   echo '('.str_repeat('?', ', count($data)-1) . '?)';  
13 }
```

Hasil kode program:

```
(? , ? , ? )
```

Done. Kita sudah berhasil membuat karakter placeholder secara dinamis. Berikutnya tinggal merangkai semua string menjadi sebuah perintah query **INSERT**:

19.DB_method_insert/insert_function_8.php

```
1 <?php
2 insert('barang',[  
3     'nama_barang' => 'Cosmos CRJ-8229 - Rice Cooker',  
4     'jumlah_barang' => 4,  
5     'harga_barang' => 299000  
6 ]);  
7  
8 function insert($tableName, $data){  
9     $dataKeys = array_keys($data);  
10    $dataValues = array_values($data);  
11    $placeholder = ('.str_repeat('?',', count($data)-1) . '?');  
12  
13    echo "INSERT INTO {$tableName} (".$implode(', ', $dataKeys).")  
14        VALUES {$placeholder}";  
15 }
```

Hasil kode program:

```
INSERT INTO barang (nama_barang, jumlah_barang, harga_barang) VALUES (?, ?, ?)
```

Inilah hasil akhir dari perancangan fungsi **insert()**, dimana kita membuat query **INSERT prepared statement** yang dihasilkan secara dinamis. Untuk uji coba, mari test dengan data yang berbeda:

19.DB_method_insert/insert_function_9.php

```
1 <?php
2 insert('user',[  
3     'username' => 'Andi',  
4     'email' => 'andi@gmail.com',  
5     'umur' => 15,  
6     'sekolah' => 'SMA N 7 lumut ijo',  
7     'alamat' => 'Jl. Perintis no 9'  
8 ]);  
9  
10 function insert($tableName, $data){  
11     $dataKeys = array_keys($data);  
12     $dataValues = array_values($data);  
13     $placeholder = ('.str_repeat('?',', count($data)-1) . '?');  
14  
15     echo "INSERT INTO {$tableName} (".$implode(', ', $dataKeys).")  
16         VALUES {$placeholder}";  
17     echo "<br>";  
18     print_r($dataValues);  
19 }
```

Hasil kode program:

```
INSERT INTO user (username, email, umur, sekolah, alamat) VALUES (?, ?, ?, ?, ?)

Array ( [0] => Andi
       [1] => andi@gmail.com
       [2] => 15 [3] => SMA N 7 lumut ijo
       [4] => Jl. Perintis no 9 )
```

Argument kedua pada saat pemanggilan fungsi insert di baris 2 – 8 berisi 5 buah element. Hasilnya, query `INSERT` juga akan memiliki lima buah tanda tanya untuk *placeholder*. Ini sesuai dengan apa yang kita inginkan.

Dan, tiba saatnya membuat method `insert()` di dalam class `DB`. Berikut kode yang diperlukan:

19.DB_method_insert/DB.php

```
1 <?php
2 class DB{
3     // ...
4     // ...
5
6     public function insert($tableName, $data){
7         $dataKeys = array_keys($data);
8         $dataValues = array_values($data);
9         $placeholder = (''.str_repeat('?', ', count($data)-1) . '?');
10
11        $query = "INSERT INTO {$tableName} (".implode(', ', $dataKeys)."
12                      VALUES {$placeholder}";
13        $this->runQuery($query,$dataValues);
14    }
15 }
```

Isi method `insert()` ini sama seperti yang kita rancang sebelumnya. Sebagai tambahan, terdapat pemanggilan method `$this->runQuery($query,$dataValues)` di baris akhir. Method `runQuery()` lah yang akan selanjutnya memproses query `INSERT`.

Mari kita jalankan dengan proses insert yang sebenarnya:

19.DB_method_insert/insert_method.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $DB->insert('barang',[[
6     'nama_barang' => 'Philips Blender HR 2157',
7     'jumlah_barang' => 11,
8     'harga_barang' => 629000
9 ]);
10
11 // tampilkan semua tabel barang
12 $tabelBarang = $DB->get('barang');
13
14 echo "<pre>";
```

```
15 print_r($tabelBarang);
16 echo "</pre>";
```

Hasil kode program:

```
Array
(
    ...
    ...
    [6] => stdClass Object
        (
            [id_barang] => 7
            [nama_barang] => Philips Blender HR 2157
            [jumlah_barang] => 11
            [harga_barang] => 629000
            [tanggal_update] => 2019-03-14 14:23:56
        )
)
```

Setelah pemanggilan method `insert()` di baris 5, saya juga menjalankan method `get('barang')` di baris 12 untuk menampilkan isi dari tabel barang. Seperti yang terlihat, barang 'Philips Blender HR 2157' sukses diinput ke dalam tabel barang.

11.16. Class DB: Method count

Jika anda perhatikan, method `insert()` yang baru saja kita buat tidak mengembalikan nilai apapun. Ini bisa diatasi dengan menulis perintah `return true` di akhir method `insert()`. Ini tidak wajib ditulis karena tidak berpengaruh apa-apa ke dalam method `insert()`.

Jika pun ternyata query yang ditulis salah atau tidak sesuai, block kode **try-catch** di method `runQuery()` akan langsung menghentikan proses yang ada.

Selain itu kadang kita butuh kepastian mengenai jumlah baris yang baru saja di input (*affected rows*). Untuk hal ini, saya akan merancang method `count()`. Method `count()` pada dasarnya mirip seperti method `check()`, dimana secara internal akan mengakses method `rowCount()` bawaan PDO.

Berikut contoh pemanggilan method `count()` dari file `index.php`:

```
20.DB_method_insert_count/index.php

1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->insert('barang',
6     'nama_barang' => 'Mouse Gaming Razer Basilisk',
7     'jumlah_barang' => 25,
8     'harga_barang' => 1250000
```

```
9  ]);  
10  
11 if($result) {  
12     echo "Terdapat ".$DB->count()." data yang ditambah";  
13     // Terdapat 1 data yang ditambah  
14 }
```

Di baris 5, saya menyimpan hasil pemanggilan method `insert()` ke dalam variabel `$result`. Jika query berjalan sebagaimana mestinya, variabel `$result` akan berisi nilai boolean **true**.

Nilai dari `$result` kemudian di periksa dalam sebuah kondisi `if` di baris 11. Jika isinya boolean **true** (query `INSERT` berhasil dijalankan), maka perintah `echo` di baris 12 akan di proses. Di sini terdapat pemanggilan method `$DB->count()` yang akan berisi jumlah baris yang diinput oleh query `INSERT` sebelumnya.

Untuk membuat sistem seperti, kita perlu modifikasi class `DB` dengan beberapa penambahan perintah. Pertama, saya akan buat sebuah private property `$_count` di awal class `DB`:

```
private $_count = 0;
```

Property ini dipakai untuk menampung hasil dari pemanggilan method `rowCount()` bawaan PDO.

Kemudian, saya akan modif method `insert()` sebagai berikut:

```
1  public function insert($tableName, $data){  
2      $dataKeys = array_keys($data);  
3      $dataValues = array_values($data);  
4      $placeholder = (''.str_repeat('?', ', ', count($data)-1) . '?');  
5  
6      $query = "INSERT INTO {$tableName} (".$implode(', ', $dataKeys).")  
7          VALUES {$placeholder}";  
8      $this->_count = $this->runQuery($query, $dataValues)->rowCount();  
9      return true;  
10 }
```

Perubahannya ada di baris 8 dan 9.

Di baris 8, pemanggilan method `runQuery()` saya sambung dengan method `rowCount()`, kemudian hasilnya disimpan ke dalam property `$this->_count`. Dengan demikian, setiap kali method `insert()` dijalankan, property `$this->_count` juga akan berisi sebuah nilai, yakni jumlah kolom yang terdampak, atau *affected rows*.

Di baris 9 terdapat perintah `return true` agar method `insert()` mengembalikan nilai **true** jika sukses dijalankan.

Agar isi property `$_count` ini bisa diakses, kita perlu sebuah method getter untuk mengambil nilainya:

```
1  public function count(){  
2      return $this->_count;
```

```
3 }
```

Sekarang, setiap kali method `insert()` dijalankan, jumlah baris yang di insert bisa diakses dari method `count()`. Nantinya, method `count()` ini juga bisa dipakai untuk mencari info jumlah baris hasil query lain seperti `UPDATE` dan `DELETE`.

11.17. Class DB: Method update

Membuat method `update()` punya tantangan tersendiri yang lebih rumit daripada method `insert()`. Untuk query `INSERT`, kita hanya perlu informasi seputar nama tabel dan data yang akan di input. Sedangkan query `UPDATE` butuh 3 hal, yakni nama tabel, data yang akan diubah, serta kondisi yang dipakai untuk mencari baris yang akan di update.

Namun karena sebelumnya sudah selesai membuat method `insert()`, pembuatan method `update()` akan terasa sedikit mudah karena menggunakan teknik yang mirip.

Langsung saja kita lihat contoh pemanggilan method `update()` ini:

21.DB_method_update/update_method.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $DB->update('barang',
6             ['nama_barang' => 'Smartphone iPhone XR',
7              'harga_barang' => 17999000],
8             ['id_barang', '=', 5]);
9
```

Di baris 5 terdapat pemanggilan method `$DB->update()`. Method `update()` butuh 3 buah argument. Argument pertama, yang dalam contoh ini berupa string 'barang', merujuk ke nama tabel yang akan di update, yakni tabel barang.

Argument kedua berupa *associative array* yang berisi pasangan nama kolom dan nilai baru. Dalam contoh ini saya ingin mengubah nilai untuk kolom 'nama_barang' menjadi 'Smartphone iPhone XR' dan kolom 'harga_barang' menjadi 17999000.

Argument ketiga berbentuk array dengan 3 buah element. Ini dipakai untuk menentukan kondisi `WHERE` dari data yang akan di update. Dalam contoh ini, array `['id_barang', '=', 5]` sama artinya dengan kondisi `WHERE id_barang = 5`.

Secara keseluruhan, pemanggilan method `$DB->update()` di baris 5 akan menjalankan query berikut:

```
UPDATE barang SET nama_barang = 'Smartphone iPhone XR', harga_barang = 17999000
WHERE id_barang = 5
```

Namun tugas kita akan lebih berat, karena harus membuat versi *prepared statement* sebagai berikut:

```
UPDATE barang SET nama_barang = ?, harga_barang = ? WHERE id_barang = ?
```

Nantinya data *placeholder* akan di simpan dalam array tersendiri dengan nilai:

```
['Smartphone iPhone XR', 17999000, 5]
```

Baik, mari kita mulai proses pembuatan method `insert()` untuk class DB.

Sama seperti proses pembuatan method `insert()`, kita akan pakai fungsi `update()` terlebih dahulu untuk mempermudah proses perancangan query. Setelah query berhasil di-generate, baru kemudian pindahkan ke dalam class DB:

21.DB_method_update/update_function_1.php

```
1 <?php
2 update('barang',
3     ['nama_barang' => 'Smartphone iPhone XR',
4      'harga_barang' => 17999000],
5     ['id_barang', '=', 5]);
6
7 function update($tableName, $data, $condition){
8     // ...
9 }
```

Masalah pertama yang akan kita pecahkan adalah bagaimana memproses pemanggilan fungsi `update()` di baris 2 agar bisa menghasilkan string berikut:

```
UPDATE nama_tabel SET nama_kolom_1 = ?, nama_kolom_2 = ?
```

Untuk nama tabel, sudah bisa langsung diakses karena tersimpan di dalam argument pertama, yakni variabel `$tableName`. Dengan demikian awal string bisa dibuat sebagai berikut:

```
$query = "UPDATE {$tableName} SET ";
```

Kemudian untuk membuat pasangan nama kolom dan tanya tanya '?' placeholder, saya susun dengan sebuah perulangan `foreach`:

```
foreach ($data as $key => $val){
    $query .= "$key = ?, ";
}
```

Kita perlu sebuah perulangan karena jumlah kolom yang akan di update tidak bisa ditentukan, bisa 2, 3 atau 10 sesuai dengan array yang ditulis dalam argument kedua.

Perulangan di atas akan dijalankan sejumlah element yang ada di dalam array `$data`. Dalam setiap perulangan, variabel `$query` akan disambung dengan nilai `"$key = ?, "`. Variabel `$key` ini merujuk ke nama key dari setiap element `$data`.

Berikut kode program fungsi `update()` kita sejauh ini:

21.DB_method_update/update_function_1.php

```
1 <?php
2 update('barang',
3     ['nama_barang' => 'Smartphone iPhone XR',
4      'harga_barang' => 17999000],
5     ['id_barang', '=', 5]);
6
7 function update($tableName, $data, $condition){
8     $query = "UPDATE {$tableName} SET ";
9     foreach ($data as $key => $val){
10         $query .= "{$key} = ?, " ;
11     }
12     echo $query;
13 }
```

Hasil kode program:

```
UPDATE barang SET nama_barang = ?, harga_barang = ?,
```

Sepintas query sudah sesuai. Namun ada masalah dengan tanda koma dan sebuah spasi di akhir string. Dua karakter ini berasal dari proses perulangan `foreach`. Untuk menghapusnya, saya akan memakai fungsi `substr()` bawaan PHP:

21.DB_method_update/update_function_2.php

```
1 <?php
2 update('barang',
3     ['nama_barang' => 'Smartphone iPhone XR',
4      'harga_barang' => 17999000],
5     ['id_barang', '=', 5]);
6
7 function update($tableName, $data, $condition){
8     $query = "UPDATE {$tableName} SET ";
9     foreach ($data as $key => $val){
10         $query .= "{$key} = ?, " ;
11     }
12     $query = substr($query, 0, -2);
13     echo $query;
14 }
```

Hasil kode program:

```
UPDATE barang SET nama_barang = ?, harga_barang = ?
```

Di baris 12, fungsi `substr($query, 0, -2)` akan menghapus 2 karakter terakhir yang terdapat di dalam string `$query`.

Langkah berikutnya adalah menyambung string `$query` dengan kondisi `WHERE`, yakni mencari baris mana yang akan di update. Berikut kode program yang saya pakai:

21.DB_method_update/update_function_3.php

```
1 <?php
2 update('barang',
3     ['nama_barang' => 'Smartphone iPhone XR',
4      'harga_barang' => 17999000],
5     ['id_barang', '=', 5]);
6
7 function update($tableName, $data, $condition){
8     $query = "UPDATE {$tableName} SET ";
9     foreach ($data as $key => $val){
10         $query .= "{$key} = ?, ";
11     }
12     $query = substr($query, 0, -2);
13     $query .= " WHERE {$condition[0]} {$condition[1]} ?";
14     echo $query;
15 }
```

Hasil kode program:

```
UPDATE barang SET nama_barang = ?, harga_barang = ? WHERE id_barang = ?
```

Proses penambahan kondisi WHERE cukup sederhana karena nama kolom dan operator sudah tersedia di element ke-1 dan ke-2 argument \$condition.

Sampai di sini proses pembuatan query sudah selesai, kita akan masuk ke perancangan data sebagai pengganti *placeholder*.

Data untuk nilai *placeholder* berasal dari dua buah array, yakni array \$data dan \$condition. Contoh berikut memperlihatkan isi dari kedua argument ini:

21.DB_method_update/update_function_4.php

```
1 <?php
2 update('barang',
3     ['nama_barang' => 'Smartphone iPhone XR',
4      'harga_barang' => 17999000],
5     ['id_barang', '=', 5]);
6
7 function update($tableName, $data, $condition){
8     print_r($data);
9     echo "<br>";
10    print_r($condition);
11 }
```

Hasil kode program:

```
Array ( [nama_barang] => Smartphone iPhone XR [harga_barang] => 17999000 )
Array ( [0] => id_barang [1] => = [2] => 5 )
```

Hasil akhir yang kita perlukan adalah sebuah array yang berisi nilai gabungan dari kedua argument, yakni dalam bentuk:

```
[Smartphone iPhone XR, 17999000, 5]
```

Isinya berupa seluruh *value* dari array `$data`, serta element ke-3 dari array `$condition`. Ini akan berpasangan dengan tanda tanya *placeholder* dari query prepared statement yang sudah kita siapkan sebelumnya.

Untuk mengambil nilai atau *value* dari associative array `$data` caranya cukup mudah, yakni menggunakan fungsi `array_values()` sebagaimana yang kita pakai pada saat pembuatan method `insert()`:

```
$dataValues = array_values($data);
```

Sekarang variabel `$dataValues` sudah berisi nilai atau *value* dari associative array `$data`. Kita akan tambah 1 nilai lagi, yakni element ke-3 dari array `$condition`.

Untuk keperluan ini saya akan memakai fungsi `array_push()` bawaan PHP. Fungsi `array_push()` berguna untuk menambah 1 nilai baru ke dalam sebuah array. Nilai baru ini akan berada di posisi paling akhir:

```
array_push($dataValues,$condition[2]);
```

Perintah ini artinya, tambah nilai yang tersimpan di `$condition[2]` ke posisi terakhir array `$dataValues`.

Berikut kode program lengkap proses pengambilan value array:

21.DB_method_update/update_function_5.php

```
1 <?php
2 update('barang',
3     ['nama_barang' => 'Smartphone iPhone XR',
4      'harga_barang' => 17999000],
5     ['id_barang','=',5]);
6
7 function update($tableName, $data, $condition){
8     $dataValues = array_values($data);
9     array_push($dataValues,$condition[2]);
10
11    print_r($dataValues);
12 }
```

Hasil kode program:

```
Array ( [0] => Smartphone iPhone XR [1] => 17999000 [2] => 5 )
```

Akhirnya, variabel `$dataValues` sudah berisi semua nilai yang kita butuhkan. Mari test seluruh fungsi `update()` dengan data yang berbeda:

21.DB_method_update/update_function_6.php

```
1 <?php
```

Case Study: Database Query Builder

```
2 update('user',[  
3     'username' => 'Andi',  
4     'email' => 'andi@gmail.com',  
5     'umur' => 15,  
6     'sekolah' => 'SMA N 7 lumut ijo',  
7     'alamat' => 'Jl. Perintis no 9'],  
8     ['id_user','=' ,85]);  
9  
10 function update($tableName, $data, $condition){  
11     $query = "UPDATE {$tableName} SET ";  
12     foreach ($data as $key => $val){  
13         $query .= "{$key} = ?, " ;  
14     }  
15     $query = substr($query,0,-2);  
16     $query .= " WHERE {$condition[0]} {$condition[1]} ?";  
17  
18     $dataValues = array_values($data);  
19     array_push($dataValues,$condition[2]);  
20  
21     echo $query;  
22     echo "<pre>";  
23     print_r($dataValues);  
24     echo "</pre>";  
25 }
```

Hasil kode program:

```
UPDATE user SET username = ?, email = ?, umur = ?, sekolah = ?, alamat = ? WHERE  
id_user = ?
```

```
Array  
(  
    [0] => Andi  
    [1] => andi@gmail.com  
    [2] => 15  
    [3] => SMA N 7 lumut ijo  
    [4] => Jl. Perintis no 9  
    [5] => 85  
)
```

Dalam contoh ini saya ingin meng-update tabel `user` dengan 5 buah data, yakni `username`, `email`, `umur`, `sekolah` dan `alamat`. Kondisi update adalah kolom `id_user` = 5.

Seperti yang terlihat, query prepared statement berhasil di generate dan disimpan ke dalam variabel `$query`. Kemudian terdapat 6 buah data placeholder yang disimpan dalam variabel `$dataValues`.

Karena proses pembuatan query sudah selesai, tinggal memindahkannya ke dalam class DB:

```
21.DB_method_update/DB.php
```

```
1 <?php  
2 class DB{
```

Case Study: Database Query Builder

```
3
4 // ...
5 // ...
6
7 public function update($tableName, $data, $condition){
8     $query = "UPDATE {$tableName} SET ";
9     foreach ($data as $key => $val){
10         $query .= "{$key} = ?, ";
11    }
12    $query = substr($query,0,-2);
13    $query .= " WHERE {$condition[0]} {$condition[1]} ?";
14
15    $dataValues = array_values($data);
16    array_push($dataValues,$condition[2]);
17
18    $this->runQuery($query,$dataValues)->rowCount();
19 }
20
21 }
```

Isi method `update()` ini tinggal di copy dari hasil percobaan fungsi `update()` sebelumnya. Dan sama seperti method `insert()`, di baris 18 proses menjalankan query dialihkan ke method `runQuery()`. Mari kita test proses update yang sebenarnya:

21.DB_method_update/update_method.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $DB->update('barang',
6             ['nama_barang' => 'Smartphone iPhone XR',
7              'harga_barang' => 17999000],
8             ['id_barang', '=', 5]);
9
10 // tampilkan semua tabel barang
11 $tabelBarang = $DB->getWhere('barang',[ 'id_barang', '=', 5]);
12
13 echo "<pre>";
14 print_r($tabelBarang);
15 echo "</pre>";
```

Hasil kode program:

```
Array
(
    [0] => stdClass Object
        (
            [id_barang] => 5
            [nama_barang] => Smartphone iPhone XR
            [jumlah_barang] => 25
            [harga_barang] => 17999000
            [tanggal_update] => 2019-03-16 07:50:09
        )
)
```

```
)
```

Hasilnya, tabel barang dengan `id_barang = 5` sukses di update.

Sentuhan terakhir, saya ingin menambah kode program untuk mengetahui jumlah baris yang di update. Caranya juga sama seperti method `insert()`, yakni mengakses method `rowCount()` bawaan PDO dan menyimpannya ke dalam private property `$_count`:

22.DB_method_update_count/DB.php

```
1 <?php
2 class DB{
3
4     // ...
5     // ...
6
7     public function update($tableName, $data, $condition){
8         $query = "UPDATE {$tableName} SET ";
9         foreach ($data as $key => $val){
10             $query .= "{$key} = ?, ";
11         }
12         $query = substr($query,0,-2);
13         $query .= " WHERE {$condition[0]} {$condition[1]} ?";
14
15         $dataValues = array_values($data);
16         array_push($dataValues,$condition[2]);
17
18         $this->_count = $this->runQuery($query,$dataValues)->rowCount();
19         return true;
20     }
21 }
```

Perubahan dari kode sebelumnya ada di baris 18 dan 19, dimana terdapat perintah untuk meng-update isi property `$_count`, serta perintah `return true`.

Dengan tambahan kode ini, kita bisa mendapat info mengenai jumlah baris kolom yang di update:

22.DB_method_update_count/DB.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->update('barang',
6                         ['nama_barang' => 'Dummy Product',
7                          'harga_barang' => 999999],
8                         ['id_barang','>',3]);
9
10 if($result) {
11     echo "Terdapat ".$DB->count()." data yang diubah";
12     // Terdapat 4 data yang diubah
```

```
13 }
```

Dalam kode program ini kondisi update saya tulis sebagai `['id_barang', '>', 3]`. Artinya, seluruh barang yang memiliki `id_barang` lebih dari 3 akan di update. Dari hasil pemanggilan method `$DB->count()` terlihat bahwa ada 4 baris data yang berhasil di update.

Jika kode di atas saya jalankan sekali lagi, hasilnya menjadi "Terdapat 0 data yang diubah". Ini terjadi karena method `DB->count()` hanya akan mengembalikan jumlah tabel yang terdampak (*affected rows*). Jika sebuah query `UPDATE` tidak melakukan perubahan apapun, maka hasilnya 0, meskipun query tersebut sukses dijalankan.

Karena alasan ini pula kita tidak bisa berpatokan kepada hasil `DB->count()` mengenai sukses atau tidaknya sebuah query. Jika hasilnya 0, bukan berarti itu query gagal dijalankan, tapi hanya tidak ada baris tabel yang berubah.

11.18. Class DB: Method delete

Method `delete()` ini menjadi method terakhir yang akan kita buat untuk class DB. Sesuai dengan namanya, method `delete()` dipakai untuk menghapus sebuah baris tabel.

Berikut contoh pemanggilannya dari halaman `index.php`:

23.DB_method_delete/index_1.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->delete('barang',[ 'id_barang' , '=' ,4]);
6
7 if($result) {
8     echo "Terdapat ".$DB->count()." data yang dihapus";
9 }
```

Method `delete()` saya rancang dengan 2 buah argument. Argumen pertama berupa nama tabel yang akan dihapus, dan argument kedua berupa kondisi WHERE untuk proses penghapusan.

Berikut kode yang dipakai untuk membuat method `delete()` di class DB:

23.DB_method_delete/DB.php

```
1 <?php
2 class DB{
3
4     // ..
5     // ..
6     public function delete($tableName, $condition){
7         $query = "DELETE FROM {$tableName} WHERE {$condition[0]} {$condition[1]} ? ";
8         $this->_count = $this->runQuery($query,[ $condition[2] ])->rowCount();
```

```
9     return true;
10    }
11 }
```

Proses pembuatan query `DELETE` cukup sederhana, dimana saya merancang sebuah string prepared statement di baris 7. Semua data yang diperlukan tinggal diambil dari argument `$tableName` dan `$condition`. Di baris 8, proses lanjutan diserahkan ke pada method `runQuery()` serta meng-update isi property `$this->_count`.

Sehingga ketika method `delete()` dipanggil dengan kode berikut:

```
$DB->delete('barang',[ 'id_barang' , '=' ,4]);
```

Akan di translate menjadi query: "DELETE FROM barang WHERE id_barang = ?".

Sebagai latihan, bisakah anda membuat kode program untuk menghapus isi tabel barang yang memiliki `id_barang` kurang dari 5?

Berikut kode program yang dibutuhkan:

23.DB_method_delete/index_2.php

```
1 <?php
2 require 'DB.php';
3 $DB = DB::getInstance();
4
5 $result = $DB->delete('barang',[ 'id_barang' , '<' ,5]);
6
7 if($result) {
8     echo "Terdapat ".$DB->count()." data yang dihapus";
9     // Terdapat 4 data yang dihapus
10 }
```

Hasilnya, 4 buah data yang memiliki `id_barang` < 5 akan di hapus dari tabel barang.

Method `delete()` ini menutup studi kasus kita dalam membuat sebuah library query builder MySQL. Meskipun terasa rumit, tapi ini masih sebuah query builder versi sederhana.

Cukup banyak batasan query MySQL yang belum bisa kita proses, sebagai contoh class `DB` ini tidak menyediakan method query builder untuk lebih dari 1 kondisi, seperti `SELECT * FROM barang WHERE id_barang = 2 AND id_barang = 5`. Termasuk query yang kompleks seperti `JOIN`.

Jika anda tertarik, silahkan kembangkan class `DB` ini lebih jauh lagi. Nantinya class ini bisa dipakai untuk berbagai project. Daripada membuat ulang semua kode program untuk mengakses database MySQL, kita tinggal mengcopy isi class `DB`. Inilah prinsip dari `code reuse`, atau filosofi **DRY** (*don't repeat yourselves*) yang bisa diterapkan dari pemrograman berorientasi objek.