Year
MAR
JUN
MAY
SEP
AUG
DEC
NOV
OCT
Inbox
Calendars
Today

Table of Contents

Modern PHP Developer

To all aspiring developers, keep learning and stay awesome!

Composer

Package Manager

In general, a block of code forms a method, a group of methods forms a class and a set of classes form a package.

A reusable package can be dropped into any project and be used without any need to add functionality to it.

A package exposes APIs for clients to achieve a single goal.

Packages help our applications achieve DRY (Don't Repeat Yourself), a principle of software development, which reduces repetition of information of all kinds.

In most cases, packages have dependencies. When "Package A" requires "Package B" in order to work, we say "Package A" depends on "Package B". It is quite common to see that a package has a chain of dependencies("Package A" depends on "Package B", "Package B" depends on "Package C", the list goes on).

Imagine there is no such thing as a package manager. What would we need to do in order to get "Package A", which has a dependency of "Package B" to work? First we download source code of "Package A", then discover it depends on "Package B", so we try our best to find source code of "Package B". It might still not work, because we also need to make sure that we download the correct version of "Package B". The story can go on and on. We are only talking about one single dependency here; it would soon turn to be a nightmare if "Package A" has multiple dependencies or there is a chain of dependencies.

We do need a package manager, a package manager that can solve all of these dependency headaches for us.

Composer vs. PEAR

PEAR

Prior to Composer, there was something called PEAR. If you started with PHP early on, you may be aware of PEAR, as it has been in existence since 1999. PEAR is made for the purpose of promoting reusable packages, similar to Composer. However, it has been discouraged by developers due to the following reasons:

- Unlike Composer, PEAR is a system-wide package manager. When you have multiple projects, which share the same dependencies, but each has different versions, this approach causes a lot of confusion and frustration.
- A certain number of up-votes is required in order to have your code accepted into PEAR's repository. This discouragement slows down growth of its repository. At the end of day, developers want to write code, not promote code.

Composer

Composer is an application-level package manager for PHP. It is inspired by NodeJs's NPM and Ruby's Bundler, and is currently the recognized package manager by the community.

The Composer ecosystem consists of two parts: the Composer, which is the command-line utility for installing packages, and the Packagist, the default package repository.

An application-level package manager means it manages dependencies on a per project basis. This makes managing multiple projects easy and keeps your machine clean as it only downloads packages to your project directory.

Everyone is welcome to submit their packages to Packagist. Unlike PEAR, there is no need to get up-votes whatsoever. You do, however, get starts if people like your packages.

Packagist

As mentioned earlier, Packagist(packagist.org) is the default package repository for Composer. As of the time of this writing, September 2015, 69,568 packages are available on Packagist. Next time you need a PHP package, instead of building one from scratch on your own, there is a good chance you can find it on Packagist. As a developer, it is recommended you leverage the power of Packagist as it will save you countless hours and energy.

Now, it's time to get our hands dirty.

Install Composer

We will assume you are a Mac user.

There are two scopes when installing Composer: local scope and global scope. From professional experience, we suggest installing Composer globally on your system. After all, it is very likely we will use Composer to manage dependency for every PHP project. Global installation saves us a lot of hassle.

Global Installation

Run commands below from your Terminal to install Composer globally:

```
curl -sS https://getcomposer.org/installer | php

mv composer.phar /usr/local/bin/composer
```

> If you encounter any errors related to permissions, run commands above in sudo mode (append **sudo** to each command)

Local Installation

Run commands below from your project root directory to install Composer locally:

```
curl -sS https://getcomposer.org/installer | php -- --filename=composer
```

> For a more detailed installation guide on Composer, check out: https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx

Verification

To verify if Composer is installed properly, run command below from the directory Composer is installed (Anywhere if Composer is installed globally).

```
composer about
```

If you see an output similar to the one below, you are ready to go.

```
Composer - Package Management for PHP
Composer is a dependency manager tracking local dependencies of your projects and librari
See https://getcomposer.org/ for more information.
```

Use Composer

Composer is now ready to use. Let's demonstrate its usage through a simple example:

Imagine we have completed an awesome project and we want to generate simulated data, for example, people's names and addresses, to show our client. It would be cool if the data is random and yet makes sense, so the demo would look real. One solution would be to type some fake names and addresses, store them in an array, then pick entries out of the array randomly using **array_rand**. As you have probably realized, this solution sounds tedious and impractical. What happens if we need hundreds of users' data? We need a savior.

It turns out there is a package on Packagist, which does exactly what we need. The best part is that it is even called **Faker**.

Let's install Faker using Composer.

From our project root folder, run command:

```
composer require fzaninotto/faker
```

It will take Composer a few seconds to download the required files. Under the hood, Composer downloads the zip file of Faker from Github. Besides downloading the required package, Composer will also create some internal files, which we will look into later.

Now take a look at our project directory and you should be able to discover some newly created folders and files as shown below:

```
composer.json    composer.lock    vendor
```

composer.json

This file describes the dependencies of your project. It is a simple JSON file and shows you what packages are installed in your project.

Whenever you run `composer require` from command line, composer.json and composer.lock will be automatically updated to reflect package change. Conversely, if you add a package to composer.json file, you run `composer install` to download the new package. If you want to update all packages' versions to the latest specified by their version constraints, you can run `composer update`.

There are three basic commands of Composer:

`composer require`

This command is used to add an individual package to the dependencies. Whenever we need a new package, we can just run it. It is convenient because we do not have to touch the composer.json file at all.

Another usage of this command is to update an existing package's version. For example, we have installed the latest version(1.4.0) of Faker using `composer require fzaninotto/faker` as Composer fetchesthe latest version of a package if we do not specify its version constraint. Since our application is incompatible with 1.4.0, we need to install 1.2.0 in order to run `composer require fzaninotto/faker:1.2 0`. It will download a specific package and update all relevant Composer files accordingly.

`composer install`

This command first looks for composer.lock file, if it is present, exact version of the packages defined, will be installed and composer.son will then be ignored. If it is not present, the command will check packages defined in the composer.json file and download the latest versions of the packages that match the supplied version constraints. Can you spot the difference? When composer.lock is used, exact versions are downloaded, whereas using composer.json, Composer will always attempt to retrieve the latest version of a package that matches the supplied version constraint. When version constraint is defined as an exact number, both actions have the same result. However this is rarely the case.

This command is used when we begin a new project - we define a list of dependencies and run this command to get all packages installed. Or, when kick-start someone else's project, we check out their code from Github and run this command to get all dependencies installed.

In some deployment strategies, we run this command in production to install the application after pulling its source code from repository.

`composer update`

This command only reads from composer.json file which is different from `composer install`. It updates existing packages to the latest versions that match supplied version constraints defined in composer.json. Meanwhile, it downloads any new packages added to composer.json file.

We can use this command to update existing packages' versions, similar to `composer require`. The difference is that `composer require` does not require us to touch the composer.json file manually, it feels more intuitive.

The fact that this command only reads from composer.json brings up a common pitfall, which is running this on production. We should never run `composer update` in production. Here's why:

If your application is working well with Faker 1.2.0 on your local development environment, you push your code to production and run `composer update`. Without your knowledge, the latest version of Faker has already been updated to 1.4.0, so Composer downloads version 1.4.0 of Faker in production, because you have defined its version constraint as 'fzaninotto/faker: 1.*' in composer.json. As a result, your production is now using a different package from your development. This is not the intended outcome.

We recommend deploying composer.lock along with compose.json and running `composer install` in production. This will ensure your production has the same packages as your development.

For more in-depth information about package version constraints and how to define them. Click here.

composer.lock

While composer.json file lets us define packages we need using versions constraints, composer.lock tracks exact versions of packages installed in our project. In other words, it stores the current state of our project. This is a very important point to remember.

The fact that `composer install` reads first from composer.lock, makes it a much safer command to use. Here's why:

If you delete **vendor** completely from the project, this will remove all packages Composer downloaded. Now run `composer install` again and it will obtain the exact versions of packages as it previously did.

This brings up our next point. If we are using a version control system such as Git, should we commit composer.lock?

The answer is "It Depends". Most of time we want to make sure everyone is sharing identical source code at anytime. So we will commit composer.lock. This is very common since most of us work with a team. The rare case of not committing composer.lock is when we develop a package(library), because users rarely need to run `composer install` in our package.

Composer gives us a lot of flexibility in using its commands, however there are a couple of rules we try to follow to prevent liability.

- `composer install` is our friend - use it in production for deployment.

A fair standard Composer workflow:

- Defined some dependencies in composer.json
  - Run `composer install`
- Need an individual package
  - Run `composer require some/package`
- Need multiple packages
  - Define them in composer.json file and run `composer update`
- Want to test out one single newly released package
  - Run `composer require some/package:new-version`
- Ready to test out all the latest versions of packages released
  - Run `composer update`

Autoloading

You have probably used a lot of `include/require` statements. The problem with These statements is that, they make our code cluttered. And the worst part is, whenever we update our directory structure, we end up doing a lot of find&replace work.

The solution is autoloading. It allows you to define paths to search for classes so you do not have to do it manually with `include/require`. But of cause, we should keep in mind that under the hood, autoloading is still using `include/require`.

Now, let's jump back to our awesome project. There is one place we have not really explored yet, and that is the **vendor** directory created by Composer. By default, Composer downloads all packages to this directory.

Composer also generates a **vendor/autoload.php** file, which provides autoloading to us for free, making it really easy to use vendor code.

In our case, we want to use Faker so we can simply include the below file and Faker will be autoloaded.

```
require __DIR__ . '/vendor/autoload.php';
```

Now we can just start using Faker.

```
$faker = Faker\Factory::create();

echo $faker->name;
```

Power of community

You should now have a fair understanding of Composer. Start using it to manage your project's dependencies. We guarantee it will make your and your co-workers' lives much easier. Next time your project needs something, start looking for them on Packagist. Embrace the power of community!

PSR

Prior to PHP Standards Recommendation (PSR), there were no truly uniformed standards for writing PHP code. For instance, for coding style, some people preferred Zend Framework Coding Standard, and some liked PEAR Coding Standards, and still others chose to create their own naming conventions and coding style.

A group of people, representing various popular PHP projects came together in 2009 and formed something called Framework Interoperability Group(FIG). The purpose of FIG is for project representatives to talk about the commonalities between their projects and find ways to work together.

At the time of this writing, there are six accepted PSRs: two of them are about autoloading, two of them are related to PHP coding style and the remaining are about interfaces.

In this chapter, we will discuss each PSR briefly. The purpose of this chapter is to introduce you to the ideas of PSRs. For further details on each one, the respective link are provided.

PSR-0, PSR-4

Both PSR-0 and PSR-4 are standards for autoloading. If you aren't familiar with autoloading, it is basically a way for PHP to include classes without writing cluttered `include/require` statements everywhere.

Let's take a look at the history of autoloading. This will give you a clear picture how autoloading in PHP has involved during the years.

In PHP language, we have to make sure a class's definition is loaded prior to using it. Normally, we will create our PHP classes in their own class files for better organization. Then we will load them with `require` or `include` statements in the files they are being called.

```
include 'manager.php';
$manager = new Manager();
```

This approach quickly raises some issues. Imagine you have tens of external classes to be used in a file and you start writing lines of `require` / `include` statements right at the beginning of a source file. They are ugly and clutter our codebase with repetitive lines of include statements.

Starting in PHP 5, a new magic function was introduced to solve this issue:

```
void __autoload ( string $class )
```

`__autoload` is essentially a helper function, doing what we were doing with include statements. We can define this function anywhere in our codebase, and PHP will automatically use this function to load a class's file when an undefined class is called. This is the last chance to load a class definition before PHP fails with an error.

```
function __autoload($class)
{
    $filename = 'classes/' . $class . '.php';
    if (file_exists($filename)) {
        include_once($filename);
    }
}

$manager = new Manager();
```

`__autoload` quickly became obsolete due to the fact that it can only allows one autoloader function. What this means is that since `autoload` is the one and the only one magic that PHP engine will call, we need to define this particular magic function wherever we want an autoloading feature. Theoretically, this includes every file in a solid object-oriented codebase.

PHP 5.1.2 was shipped with another autoloading function(`spl_autoload_register`) for coping with `__autoload` 's limitation. `spl_autoload_register` is a replacement for `__autoload` , and it provides more flexibility. It works by registering PHP user land functions with the **autoload queue. It effectively creates a queue of autoload functions, and runs through each of them in the order in which they are defined. This means we can have multiple autoloader functions and there is no need for creating** ```autoload``` function in each one of our source files anymore.

Autoloading was such a great idea that every project started to use it. Inevitably everyone created their own version of autoloader as uniform standards were lacking. Clearly, PHP desperately needed a standard for autoloader, which is how PSR-0 was born.

Today, Autoloader standard has evolved significantly. Even PSR-0 is officially depreciated due to some constraints, such as its unfriendiness to Composer.

The latest accepted autoloader standard is PSR-4. You should follow PSR-4 to create your desired autoloader. For specification of PSR-4, please read more from its official page.

PSR-1, PSR-2

PSR-1 and PSR-2 are for PHP coding standards. PSR-1 focuses on the basics, whereas PSR-2 expands upon PSR-1 and provides a more comprehensive coding style guide.

PSR-1 lists a set of simple rules for naming conventions and file structures. Its main purpose is to ensure a high level of technical interoperability between shared PHP codes. In a project that is incorporated with various packages, it can be a mess if each uses different coding standard, which is what PRS-1 was designed to solve.

A quick overview of PSR-1:

- Files MUST use only <?php and <?= tags.
- Files MUST use only UTF-8 without BOM for PHP code.
- Files SHOULD either declare symbols (classes, functions, constants, etc.) or cause side effects (e.g. generate output, change .ini settings, etc.) but SHOULD NOT do both.
- Namespaces and classes MUST follow PSR-0.
- Class names MUST be declared in StudlyCaps.
- Class constants MUST be declared in all upper case with underscore separators.
- Method names MUST be declared in camelCase.

Building on top of PSR-1, PSR-2 provides more comprehensive guidelines with more detailed rules as basic as code indention. It also covers varicose aspects of coding style, from naming conventions to namespace, classes, properties, methods, control structures and closures. It is possible to find any specification you need from PSR-2. Adapting your codebase to this standard for interoperability is highly encouraged.

PSR-3, PSR-7

After autoloading and coding standards, we can finally associate PSR with PHP code. These are PSR-3 and PSR-7. PSR-3 contains a logger interface and PSR-7 contains interfaces for HTTP message interfaces.

PSR-3

PHP desperately needed a standard for Logger interface before PSR-3. Logging is such a common task that every project built its own version of logger. Without a standard, the only way to use a third-party logger was to write a wrapper around it, so it could work with our existing codebases. It was not only a painful process, but also felt wrong, because after all, they were all doing the same type of work: logging. We should be able to switch them around.

PSR-3 provides common interface for logging libraries. As long as they implement the PSR-3 logger interface, they should be theoretically interchangeable with any other PSR-3 logger libraries.

Let's take a look how PSR-3 Logger interface improves our code reusability in a concrete example.

Suppose we have written a simple authentication class User below. It appends an audit message to a log file once an user logins in successfully. It is using our custom logger class, which exposes a single method `addMessage`.

```
class User
{

    private $logger;

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

    public function login($username, $password)
    {
        if ($this->validUsernameAndPassword($username, $password)) {
            $this->logger->addMessage(sprintf('user %s login at %s', $username, date('m/d
        }
    }

    private function validUsernameAndPassword($username, $password)
    {
        // ... ...
    }
}
```

Our custom **Logger** class is injected to **User** class following dependency injection principle, this seems to make our **User** reusable. We can switch to other logger class simply via the constructor. But if we take a look closely, we can't do that, **User** class is still highly coupled to our custom **Logger** class, it is aware of custom method `addMessage` . If we use another third-party logger library in our code, it won't work, because they do not have a method called `addMessage` .

We can modify our code to use PSR-3 Logger interface instead. According to Dependency Inversion principle SOLID, we should depend upon abstractions rather than concretions. PSR-3 Logger interface provides a perfect abstraction for our case.

```
class User
{

    private $logger;

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

    public function login($username, $password)
    {
        if ($this->validUsernameAndPassword($username, $password)) {
            $this->logger->info(sprintf('user %s login at %s', $username, date('m/d/Y h:i
        }
    }

    private function validUsernameAndPassword($username, $password)
    {
        // ... ...
    }
}
```

By changing a few lines of our code, we have replaced our custom logger with **PsrLogLoggerInterface**. Now our code is highly reusable. We can use, switch to, or change to, any third-party logger library that is compliant with with PSR-3 Logger interface.

PSR-7

HTTP messages are essential for web applications. Every action a user takes is a combination of a HTTP request and HTTP response. PSR-7 is the latest accepted standard. It provides abstractions around HTTP messages and the elements composing them. It will have a huge impact on projects that implement details of HTTP messages, since HTTP is a rather complex subject and most of vendors have their own implementation, it is a lot of refactoring for vendors to adapt PSR-7.

As a user of HTTP messages, we can now deal with HTTP messages universally thanks to PSR-7. Similar to PSR-3, PSR-7 makes our lives much easier to build a reusable codebase.

PSR Specifications

We have summarized each PSR briefly and you should now have an understanding of what each PSR is for.

You should refer to the official page whenever you need detailed specifications of each PSR.

- PSR-1: http://www.php-fig.org/psr/psr-1/
- PSR-2: http://www.php-fig.org/psr/psr-2/
- PSR-3: http://www.php-fig.org/psr/psr-3/
- PSR-4: http://www.php-fig.org/psr/psr-4/
- PSR-7: http://www.php-fig.org/psr/psr-7/

Persist with PHP Data Objects

PHP Data Objects, most commonly known as PDO, is a PHP extension built to solve database access problems. It provides a unified interface to access databases.

PDO creates an abstraction layer for data-access, so developers can write portable code without worrying about the underlying database engines. In layman's terms, you use PDO to develop an application using MySQL as the database storage. If you want to switch to PostgreSQL at any point in time, all you need to do is to change the PDO driver. No other code change is required.

PDO consists of three main types of objects: They are PDO object, PDOStatement object and PDOException object. We should not necessarily neglect the PDO Drivers, but these three types of objects together form the main interface of the PDO extension.

Why use PDO?

If you have developed any MySQL database driven application before, but have never tried PDO, you must be wondering what the benefits are to use PDO, especially when comparing it to its two alternatives.

MySQL

The oldest way to interact with MySQL is to use `mysql` extension. It was introduced in PHP 2.0.0, however it was deprecated as of PHP 5.5.0, and has already been removed in PHP 7.0.0. It is not recommended to use this extension at all given the factor that it is not supported in newer PHP versions.

MySQLi

Since PHP 5.0.0, an improved version of `mysql` extension, known as `mysqli` was introduced. It brought a lot of benefits over the `mysql` extension, such as object-oriented interface, prepare statements, multiple statements, transaction support, enhanced debugging capabilities and embedded server support.

The main differences between MySQLi and PDO are:

- PDO supports client-side prepared statements, whereas MySQLi does not. We will discuss client-side prepare statements in details in later sections. It basically means it will emulate prepared statement if the chosen database server does not support it.

- MySQLi supports both object-oriented API and procedural API, whereas PDO religiously uses objected-oriented API.

- The biggest advantage of using PDO is to write portable code. It enables developers to switch databases easily, whereas MySQLi only supports the MySQL database.

Ultimately, we recommend using PDO to build your applications.

- It enables developers to write portable code.
- It encourages object-oriented programming.

Lastly, we want to emphasize that, by no means are you forbidden to use MySQLi.

In the following sections, we will start with some common ways of running queries using PDO. Then we will demonstrate how to perform various MySQL data manipulation statements using PDO. Finally, we will focus on a few PDO APIs, which serve the same purpose but in different ways.

Running PDO Queries

We summarize the different ways of running PDO queries into four categories, classified by number of steps involved from carrying out the query to getting its result. These categories were created to ease the efforts of remembering PDO APIs and include:

- exec
- query fetch
- prepare execute fetch
- prepare bind execute fetch

Establish database connection

Before we get into each category, you will first need to be familiar yourself with establishing a database connection using PDO. This is the absolute the fundamental of PDO, as it is used in every piece of code below:

```
try {
    $dbh = new PDO('mysql:host=localhost;dbname=customers', $user, $pass);
} catch (PDOException $e) {
    die($e->getMessage());
}
```

To establish a database connection, we instantiate a PDO object with three parameters. The first parameter specifies a database source (known as the DSN), which consists of the PDO driver name, followed by a colon, followed by the PDO driver-specific connection syntax. The second and third parameters are database username and password.

An exception will be thrown if the connection fails. You can catch the exception and handle it gracefully. Kudos to exception in this case, we no longer need to put the connection in a `if` statement due to a clean and easy to read code base.

In the following code samples, we will neglect this piece of code, to avoid clutter. Keep in mind, you will always need to make the connection first before proceeding with any PDO operations.

exec

This is the simplest form of running a query. We can use it to run a quick query and normally we do not expect it to return any results.

```
$dbh->exec('INSERT INTO customers VALUES (1, "Andy")');
```

Though PDO::exec does not return the result corresponding to your query, it does return something. Regardless of what query you run with PDO::exec, it returns the number of affected rows on success. It also returns Boolean FALSE on failure.

A caveat when checking the return type: since it PDO::exec returns 0 when there is no row affected, we should always use === comparison operator to verify success of running the method.

```
if (FALSE === $dbh->exec('INSERT INTO customers VALUES (1, "Andy")')) {
    throw new MyException('Invalid sql query');
}
```

If you are building the query string with user input and manually handling security issues as such escaping characters, you should use other alternatives, which we will discuss later.

query fetch

When running query such as **SELECT** statement, we do expect a return of corresponding results. The easiest way of accomplishing this is use:

```
$statement = $dbh->query('SELECT * FROM customers');

while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
    echo $row['id'] . ' ' . $row['name'] . PHP_EOL;
}
```

Note that methods `$dbh->query()` and `$statement->fetch()`, are how we name our categories, by the sequences of calling PDO APIs.

Because PDO::query returns a result set as a PDOStatement object on success (It will return Boolean FALSE on failure, do similar check as PDO::exec if you want to verify). PDOStatement class implements the Traversable interface, which is the base interface for Iterator, meaning it can be used in an iteration statement as such as a `loop`. Naturally, there is a short version of previous code:

```
foreach ($dbh->query('SELECT * FROM customers', PDO::FETCH_ASSOC) as $row) {
   echo $row['id'] . ' ' . $row['name'] . PHP_EOL;
}
```

You might have noticed, when calling either PDO::query or PDOStatement::fetch, we have supplied a flag parameter. This parameter specifies what type of data structure we want from the callee.

A few of the options include:

- PDO::FETCH_ASSOC: returns an associative array indexed by column name.
- PDO::FETCH_NUM: returns an numerically indexed array.
- PDO::FETCH_BOTH (default): returns an array indexed by both column name and 0-indexed column number as returned in your result set. (Combination of PDO::FETCH_ASSOC and PDO::FETCH_NUM).

There are a lot more options. We recommended that you take a quick look at them at PHP Manual. Though this parameter is optional, we should always specify it unless we really want an array indexed by both column name and number. PDO::FETCH_BOTH takes twice as much memory.

prepare execute fetch

We frequently need to accept user's input in order to run a database query. There are two major concerns if we were to use A `query fetch` approach.

First, we will have to make sure the sql query passed to PDO::query is safe. Escaping and quoting the input values must be well taken care of. Second, PDO::query executes an SQL statement in a single function call, which means if we need to run the same query multiple times, it will use multiple times of resources. There is a better way of doing this.

PDO introduces prepare statement for the first time. So what is prepare statement? According to Wikipedia.

> In database management systems, a prepared statement or parameterized statement is a feature used to execute the same or similar database statements repeatedly with high efficiency. Typically used with SQL statements such as queries or updates, the prepared statement takes the form of a template into which certain constant values are substituted during each execution.

Prepare statement solves two concerns raised above. It not only improves the efficiency of running multiple similar queries, but also takes care of escaping and quoting user input values.

Below is how we implement prepare statement using PDO:

```
$users = ['Andy', 'Tom'];

$statement = $dbh->prepare('SELECT * FROM customers where name = :name');

foreach ($users as $user) {
    $statement->execute([':name' => $user]);

    while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
        echo $row['id'];
    }
}
```

Note the steps we have taken here:

- PDO::prepare is used to create a sql query containing a variable parameter. Naming convention for parameters are either named variables prefixed by a colon(:) or a question mark (?).
- PDOStatement::execute is called to execute a query with parameters' value. When ? is used in the prepare statement, they are numbered parameters. We can bind the values using a numeric indexed array. Note in the `foreach` , it uses the same statement to carry out the query after binding the value. It returns Boolean FALSE upon failure. We can use PDOStatement::errorInfo() to get the error information associated with the operation.
- PDOStatement::fetch is used to fetch result with desired data structure.

prepare bind execute fetch

A minor issue you might have OBSERVED in previous code is what happens when there are a lot of parameters in the prepare statement. We can easy create code piece like this:

```
$statement->execute([':name' => $user, ':mobile' => $mobile, ':address' => $address ,':co
```

The list can go on and on. This makes code very difficult to read. However, a more important thing to notice here is that, PHP will cast user input value to match its database field type if they do not match exactly, which is prone to bugs.

Here is where PDOStatement::bindValue comes in to save. The recommended way of running previous is:

```
$users = ['Andy', 'Tom'];

$statement = $dbh->prepare('SELECT * FROM customers where name = :name');

foreach ($users as $user) {
    $statement->bindValue(':name', $user, PDO::PARAM_STR);

    $statement->execute();

    while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
        echo $row['id'];
    }
}
```

Instead of using PDOStatement::execute to bind value to parameter, we used PDOStatement::binValue. It adds a few significant improvements to our code:

- Readability: it makes the code easy to read for other developers, as it indicates the exact data type a parameter should accept.
- Maintainability: The third parameter, which specifies datatype of passing variable, prevents PHP from casting incompatible datatype, which is prone to bug. In the long run, it also makes the code easier to maintain, as future developer will be able to spot the datatype at a glance.

These four techniques are definitely not official: they are just naming conventions made to memorize PDO APIs. There is no need to follow them strictly. In fact, most of time we combine these techniques together.

Data manipulation

Let's put what we have learned into action. In this section, we will use PDO to accomplish some of the most common MySQL tasks.

Sample database table

We will need a database table to play around with.

For demonstration purpose, we will create a very simple database table:

```
CREATE TABLE IF NOT EXISTS `customers` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`name` varchar(100) NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

Insert data

Our first task is to insert some data into the table. For this use case, let's suppose we accept data from a user input form via a POST request, and then we insert whatever data is from the form into the **customers** table.

```
try {
    $dbh = new PDO('mysql:host=localhost;dbname=inventory', 'root', 'root');
} catch (PDOException $e) {
    die($e->getMessage());
}

$name = $_POST['name'];

$statement = $dbh->prepare('INSERT INTO customers (name) VALUES (:name)');

if (false === $statement) {
    throw new Exception('Invalid prepare statement');
}

if (false === $statement->execute([':name' => $name])) {
    throw new Exception(implode(' ', $statement->errorInfo()));
}
```

We are using the **prepare->execute->fetch** techinique in this code sample, except we removed the **fetch** part since we do not expect it to return any result set.

- The first step is to connect to the database as usual.
- Then, we create a prepare statement. Note that we also handle failure case by throwing an exception.
- And lastly, we execute the prepare statement. Failure case is also handled. We are able to output useful information by calling PDOStatement::errorInfo method.

Update data

The second common task is to update existing data. Assume the use case is the same as the previous case, except the user is able to pass in an additional parameter ($id).

```
try {
    $dbh = new PDO('mysql:host=localhost;dbname=inventory', 'root', 'root');
} catch (PDOException $e) {
    die($e->getMessage());
}

$id = $_POST['id'];
$name = $_POST['name'];

$statement = $dbh->prepare('UPDATE customers SET name = :name WHERE id = :id');

if (false === $statement) {
    throw new Exception('Invalid prepare statement');
}

if (false === $statement->execute([':name' => $name, ':id' => $id])) {
    throw new Exception(implode(' ', $statement->errorInfo()));
}
```

As you have probably guessed, beside the additional parameter $id ,the code is identical to previous code sample.

Delete

The third common task is to delete an existing data record (similar use case here). User is able to pass in a single parameter ($id), and the corresponding record should be deleted.

```
try {
    $dbh = new PDO('mysql:host=localhost;dbname=inventory', 'root', 'root');
} catch (PDOException $e) {
    die($e->getMessage());
}

$id = $_POST['id'];

$statement = $dbh->prepare('DELETE from customers WHERE id = :id');

if (false === $statement) {
    throw new Exception('Invalid prepare statement');
}

if (false === $statement->execute([':id' => $id])) {
    throw new Exception(implode(' ', $statement->errorInfo()));
}
```

Again, this is a very similar code sample as the previous one (**prepare->execute->fetch** techinique with **fetch** part). That is the "beautify" of PDO - its object-oriented design makes code easy to write.

Select

Our final task is to select all data records from **customers**, but this time, we won't ask for user's input.

```
try {
    $dbh = new PDO('mysql:host=localhost;dbname=inventory', 'root', 'root');
} catch (PDOException $e) {
    die($e->getMessage());
}

$results = array();

$statement = $dbh->query('SELECT * FROM customers');

if (false === $statement) {
    throw new Exception('Invalid query');
}

while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
    $results[] = $row['name'];
}
```

In this example, we have used the **query->fetch** technique. As we mentioned earlier and it is worth mentioning again, when calling PDOStatement::fetch, it is a good habit to always specify the fetch mode.

Now we have run through some quick samples of using PDO for various tasks. These samples are very simple, yet they have showed us a very easy to use and consistent API provided by PDO.

PDO API

So far, we have demonstrated some common PDO APIs. PDO still provides a lot more than we have shown above. In this section, we will explore PDO APIs for the last time and try to cover as many useful APIs as we can. It is nearly impossible to cover every aspect of this topic but always keep in mind that there is a manual page that can be referenced when you are in doubt. (PDO Manual page)

Fetch methods

We have covered one(PDOStatement::fetch) of the fetching methods used to retrieve result sets. In fact, PDOStatement provides three additional fetching methods.

PDOStatement::fetchAll

Working similarly to PDOStatement::fetch, PDOStatement::fetchAll also accepts a flag as first parameter, which is to specify fetch mode. We should always specify the fetch mode the same way we do for PDOStatement::fetch method. It differs from PDOStatement::fetch in that it returns all result at once.

```
$statement = $dbh->query('SELECT * FROM customers');

$result = $statement->fetchAll(PDO::FETCH_ASSOC);

print_r($result);

// Output
Array
(
    [0] => Array
        (
            [id] => 2
            [name] => TEST2
        )

    [1] => Array
        (
            [id] => 3
            [name] => TEST2
        )

    [2] => Array
        (
            [id] => 4
            [name] => TEST2
        )
)
```

A caveat of this method is that, since it loads all result sets at once, it might result in heavy memory usage depending on how much data is available. You should use this method with attention.

PDOStatement::fetchColumn

A handy method for retrieving data from the desired column is PDOStatement::fetchColumn. It returns a single column from the next row of a result set. It is similar to PDOStatement::fetch, however it returns the next single column only instead of a next array of result reset.

```
$statement = $dbh->query('SELECT id, name FROM customers');

while($result = $statement->fetchColumn(1)) {
    echo $result . PHP_EOL;
}
```

PDOStatement::fetchColumn accepts a single parameter(column name) optionally. The parameter is a 0-indexed number specifying the column to retrieve data from. When this parameter is omitted, it defaults to column number 0.

Two points to not when using this method:

- PDOStatement::fetchColumn will return Boolean FALSE when it reaches the end of the result set, so it should not be used to retrieve Boolean type from the database.
- PDOStatement::fetchColumn moves its pointer one step forward when it is called, so there is no way to retrieve another column from the same row. (Obviously the pointer has already moved to the next row when we call it using different column number).

PDOStatement::fetchObject

This method is an alternative to PDOStatement::fetch() with PDO::FETCH_CLASS or PDO::FETCH_OBJ style. Its purpose is to make our code easier to read when called separately, when this method is called, it will return next result set as a PHP object:

```
$statement = $dbh->query('SELECT id, name FROM customers');

while($object = $statement->fetchObject()) {
    print_r($object);
}

// Output
stdClass Object
(
    [id] => 2
    [name] => TEST2
)
stdClass Object
(
    [id] => 3
    [name] => TEST2
)
```

We can also pass in our custom PHP class as first parameter, PHP will instantiate one instance of our custom PHP object with data retrieved and return it:

```
class MyClass
{

}

$statement = $dbh->query('SELECT id, name FROM customers');

while($object = $statement->fetchObject('MyClass')) {
    print_r($object);
}


// Output
MyClass Object
(
    [id] => 2
    [name] => TEST2
)
MyClass Object
(
    [id] => 3
    [name] => TEST2
)
```

Bind methods:

Previously, we have used PDOStatement::bindValue. This method binds desired value to the placeholder of the query. This method is not the only method for that task though.

bindParam

This method is almost identical to PDOStatement::bindValue and it's no surprise that some people use these two methods interchangeably, however there is a very significant different between these two methods, and it might cost you a fortune if you are not aware of it.

> Unlike PDOStatement::bindValue(), the variable is bound as a reference and will only be evaluated at the time that PDOStatement::execute() is called.

Let's see what it means by example.

```
$user = 'Andy';

$statement = $dbh->prepare('SELECT * FROM customers where name = :name');

$statement->bindValue(':name', $user, PDO::PARAM_STR);

$user = 'Tom';

$statement->execute();

echo $statement->fetchColumn(1);

// Output
Andy
```

```
$user = 'Andy';

$statement = $dbh->prepare('SELECT * FROM customers where name = :name');

$statement->bindParam(':name', $user, PDO::PARAM_STR);

$user = 'Tom';

$statement->execute();

echo $statement->fetchColumn(1);

// Output
Tom
```

Do you spot the difference? These two pieces of code are identical except one is using $statement->bindParam and the other is using $statement->bindValue. The result produced is entirely different.

PDOStatement::bindParam binds variable $user as a reference. At the time of PDOStatement::execute is called, $user variable is changed to 'Tom' whereas PDOStatement::bindParam variable as a value, it remains as 'Andy' from the time PDOStatement::bindValue is called.

Make sure you understand the difference between these two and choose them according to your needs. Switching these two methods without a fair amount of consideration is discouraged.

bindColumn

Different from PDOStatement::bindValue and PDOStatement::bindParam, this method is not a method for binding variable to prepare statement. In fact, it is quite the opposite: it binds columns from resulting set to PHP local variables.

This is an interesting method to observe. Previously, we discussed that a method PDOStatement::fetchObject, can return result set as a defined object. Here, with PDOStatement::bindColumn, we can bind columns from result set to variables.

```
$statement = $dbh->prepare('SELECT id, name FROM customers');

$statement->bindColumn('name', $name);

$statement->execute();

while ($statement->fetch(PDO::FETCH_ASSOC)) {
    echo $name . PHP_EOL;
}
```

The first parameter which specifies the table column, accepts both string column name and 0-indexed number as a value. So the following is valid too.

```
$statement->bindColumn(1, $name);
```

Conditions:

In the last section, we will discuss some tips when working with PDO.

IN clause

Building `IN` clause in a prepare statement is an interesting task. Take a look at following code and imagine this is what we need to build:

```
$users = ['Andy', 'Tom'];

$statement = $dbh->prepare('SELECT * FROM customers where name IN :name');

$statement->execute($user);
```

At first glance, it seems legitimate. Take a closer look. It won't work because prepare statement only accepts scalar types (e.g. string, int and so on).

The ultimate task becomes building a comma separated string containing equal question marks(?) to the binding array variable. This is how we can build a legit `IN` clause string.

```
$users = ['Andy', 'Tom'];

$placeholder = implode(',', array_fill(0, count($users), '?'));

$statement = $dbh->prepare('SELECT * FROM customers where name IN '. $placeholder);

$statement->execute($users);
```

Wildcard characters

When building a `LIKE` clause, we might be tempted to do this:

```
$name = 'Andy';

$statement = $dbh->prepare('SELECT count(*) FROM customers where name LIKE %:name%');

$statement->bindValue(':name', $name);
```

However, that won't work in PDO. We need to shift the wildcard characters to the variable itself:

```
$name = '%Andy%';

$statement = $dbh->prepare('SELECT count(*) FROM customers where name LIKE :name');

$statement->bindValue(':name', $name);
```

Iteration with SPL (Standard PHP Library)

If you have used a `for` loop in PHP, the idea of iteration is most likely not foreign to you. You pass an array to a `for` loop, and perform some logic inside the loop, but did you know that you can actually pass data structures other than arrays to a `for` loop? That's where Iterator comes into play.

Below is summarised definition of an iterator from Wikipedia:

> In computer programming, an iterator is an object that enables a programmer to traverse a container, particularly lists.[...] Note that an iterator performs traversal and also gives access to data elements in a container, but does not perform iteration [...]. An iterator is behaviorally similar to a database cursor.

Some key points to remember here:

- Iterator enables us to traverse a container. It is similar to arrays.
- Iterator does not perform iteration. In our previous example, `for` does the iteration. Other loop types such as `foreach` and `while` do iteration.

Now that we know the definition of Iterator, the concept may still be somewhat obscure, but do not worry, we aren't done yet. We have now established that Iterator works similar to array and it can be loop through in a `for` loop.

It is helpful to understand how array actually works in a `for` loop. Let's take a look at the code below:

```
$data = array(1,2,3,4);

for ($i=0; $i<count($data); $i++) {
    $key = $i;
    $value = $data[$i];
}
```

Here is how an array works in a `for` loop:

- In step 1, we set $i to 0. (`$i=0`)
- In step 2, we check to see $i is less than the length of $data. (`$i<count($data)`)
- In step 3, we increase $i value by 1. (`$i++`)
- In step 4, we can access the key of the current element. (`$key = $i`)
- In step 5, we can also get the value of current element. (`$value = $data[$i]`)

We can abstract the steps as simple functions as below:

- Step 1 = rewind().
- Step 2 = valid().
- Step 3 = next().
- Step 4 = key().
- Step 5 = current().

In abstract level, we can imagine that, as long as an object provides the five functions above, it can be loop through by a `for` loop.

In fact, an iterator is nothing but a class implements all five steps mentioned above. In PHP, The Standard PHP Library(SPL), which is a collection of interfaces and classes that are meant to solve common problems, provides a standard Iterator interface.

```
Iterator extends Traversable {
    /* Methods */
    abstract public mixed current ( void )
    abstract public scalar key ( void )
    abstract public void next ( void )
    abstract public void rewind ( void )
    abstract public boolean valid ( void )
}
```

Your first iterator class

Now that we understand what an iterator is, it's time to build our first one.

Our first iterator represents top 10 stared PHP repositories from Github. We can pass it into a `foreach` and loop through it just like an array. We will name it **TrendingRepositoriesIterator**.

First, we need to make our class implement the Iterator interface.

```
class TrendingRepositoriesIterator implements Iterator
{

    public function rewind()
    {

    }

    public function valid()
    {

    }

    public function next()
    {
    }

    public function key()
    {

    }

    public function current()
    {
    }
}
```

An iterator must always implement the five methods described above. Final code of **TrendingRepositoriesIterator** is as follow:

```
<?php


class TrendingRepositoriesIterator implements Iterator
{
    private $repos = [];
    private $pointer = 0;

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

    public function rewind()
    {
        $this->pointer = 0;
    }

    public function valid()
    {
        return isset($this->repos[$this->pointer]);
    }

    public function next()
    {
        $this->pointer++;
    }

    public function key()
    {
        return $this->pointer;
    }

    public function current()
    {
        return $this->repos[$this->pointer];
    }

    private function populate()
    {
        $client = new GuzzleHttp\Client();

        $res = $client->request('GET', 'https://api.github.com/search/repositories', [
            'query' => ['q' => 'language:php', 'sort' => 'stars', 'order' => 'desc']
        ]);

        $resInArray = json_decode($res->getBody(), true);

        $trendingRepos = array_slice($resInArray['items'], 0, 10);

        foreach ($trendingRepos as $rep) {
            $this->repos[] = $rep['name'];
        };
    }

}
```

- **public function populate()**: We will not go in-depth regarding this function as that will defeat the purpose of this chapter. Basically this function fetches top 10 stared PHP repositories from Github via Github public API and store them into **$repos** property.

- **private $repos**: We use this property to store fetched repositories.
- **private $pointer**: We can use array's internal pointer to do the job, however since we are building our own iterator, we want to retain full control.
- **public function __construct()**: The property fetches target repositories when an object is instantiated.
- **public function rewind()**: We can use this to set pointer to first position.
- **public function valid()**: As long as the value of current pointer is set, it is valid.
- **public function next()**: This is used to increase the pointer by 1 position.
- **public function current()**: We can return value of current pointer through this function.

Let's see the use case of **TrendingRepositoriesIterator**, which can used just like an array:

```
$trendingRepositoriesIterator = new TrendingRepositoriesIterator();

foreach ($trendingRepositoriesIterator as $repository) {
    echo $repository . "\n";
}

// Output
laravel
symfony
CodeIgniter
DesignPatternsPHP
Faker
yii2
composer
WordPress
sage
cakephp
```

Awesome! Now we have written our first iterator and as you can see, it is actually very easy and straightforward.

Why iterator?

You might still wonder why we need to use iterator. Can't we just use array? The answer is yes and no. In most cases, array is sufficient for the job, although iterator does come with some key advantages, which we will share next. Keep in mind, we are by no means suggesting using iterator in all circumstance.

Encapsulation

In our first iterator, **TrendingRepositoriesIterator**, the details of traversing Github repositories is completed hidden from outside. We can update how we get the data, where we get the data from, and how we want to traverse the resources. No change is needed from the client code. This, known as Encapsulation, is one of the key concepts of Object-Oriented Programming.

Additional examples include:

To iterate through MySQl results, we can use:

```
$result = mysql_query("SELECT * FROM books");

// Iterate over the structure
while ( $row = mysql_fetch_array($result) ) {
   // do stuff
}
```

To iterator through content of a text file, we can:

```
$fh = fopen("books.txt", "r");

// Iterate over the structure
while (!feof($fh)) {

   $line = fgets($fh);
   // do stuff with the line here

}
```

With iterator, we can encapsulate the process of traversing the recourse so that the outside world is not aware of the internal operations. In fact, the outside world does not need to know where we get the data from or how it is traversed in a loop. All they need to know is that, they can iterate through it as simply as:

```
$bookIterator = new BookIterator();
foreach($bookIterator as $book) {
    // do stuff with $book
}
```

Encapsulation is a very powerful concept and it enables us to write clean code.

Efficient memory usage

Efficient memory usage is a key benefit of iterator.

In our **TrendingRepositoriesIterator** class, we can actually fetch resource dynamically, meaning we will only fetch data from Github API when the `next()` method is called. This technique is called Lazy Loading. It helps us save a very significant amount of memory as value is only generated when it is needed.

Easy to add additional functionalities

Another benefit of using iterator is that we can decorate it to add additional functionalities. Take our **TrendingRepositoriesIterator** class for example. We want to exclude "laravel" from the resource. One obvious method is to update our original class, although that is of course not what we would do here.

We can decorate the original iterator using SPL's **CallbackFilterIterator** and no change is needed for **TrendingRepositoriesIterator** at all.

```
$trendingRepositoriesIterator = new TrendingRepositoriesIterator();

$newTrendingRepositoriesIterator = new CallbackFilterIterator($trendingRepositoriesIterat
    return $value != 'laravel';
});

foreach ($newTrendingRepositoriesIterator as $repository) {
    echo $repository . "\n";
}

// Output
symfony
CodeIgniter
DesignPatternsPHP
Faker
yii2
composer
WordPress
sage
cakephp
```

The cool part of this is that there is no duplication of objects. The callback fires only when **TrendingRepositoriesIterator** hits the `next()` method, and then the logic it will be applied accordingly. This is a great way to save memory as well as boost performance.

SPL Iterators

Now that we understand the power and benefits of using iterators, it is good practice to use iterators to solve suitable problems. However if we were to write iterators by ourselves whenever we encounter a new problem, it would be very time consuming since it does require us to implement a set of pre-defined functions.

Luckily PHP has done a good job of providing a set of iterators for solving some common problems. In the following sections, we will work through a set of common iterators provided by SPL. As a refresher, SPL standards for Standard PHP Library was built to provide a collection of interfaces and classes that are meant to solve common problems.

ArrayObject vs SPL ArrayIterator

In PHP, array is one of the eight primitive types. PHP provides 79 functions for handling array related tasks (reference). It is completely suitable to use array, however there are times, depending on how much you embrace Object-Oriented programming, that you may want to use array as an object. In this case, PHP provides two classes to make array a first class citizen in Object-Oriented code.

ArrayObject

The first option we have is ArrayObject. This class allows objects to work as arrays.

Let's take a look at its class signature:

```
ArrayObject implements IteratorAggregate , ArrayAccess , Serializable , Countable{
 ...
 public ArrayIterator getIterator ( void )
 ...
}
```

As we have seen above, ArrayObject implements IteratorAggregate. What is IteratorAggregate? It is an interface to create an external iterator. In simple terms, it is a quick way to create an iterator, instead of implementing Iterator interfaces with five methods: rewind,valid,current,key and valu, IteratorAggregate allows you to delegate that task to another iterator. All you need to do is implement a single method getIterator().

```
IteratorAggregate extends Traversable {
    abstract public Traversable getIterator ( void )
}
```

ArrayObject implements IteratorAggregate. It creates an external ArrayIterator for iterator feature.

As ArrayObject implements IteratorAggregate, we can use it in a `foreach` loop just as an array.

```
$books = array(
    'Head First Design Patterns',
    'Clean Code: A Handbook of Agile Software Craftsmanship',
    'Domain-Driven Design: Tackling Complexity in the Heart of Software',
    'Agile Software Development, Principles, Patterns, and Practices',
);

$booksAsArrayObject = new ArrayObject($books);

foreach ($booksAsArrayObject as $book) {
    echo $book . "\n";
}

// Output
Head First Design Patterns
Clean Code: A Handbook of Agile Software Craftsmanship
Domain-Driven Design: Tackling Complexity in the Heart of Software
Agile Software Development, Principles, Patterns, and Practices
```

The primary reason we want to use ArrayObject is to use array in Object-Oriented fashion.

```
$books = array(
    'Head First Design Patterns',
    'Clean Code: A Handbook of Agile Software Craftsmanship',
    'Domain-Driven Design: Tackling Complexity in the Heart of Software',
    'Agile Software Development, Principles, Patterns, and Practices',
);

$booksAsArrayObject->append('The Pragmatic Programmer: From Journeyman to Master');
// --- vs ---
$books[] = 'The Pragmatic Programmer: From Journeyman to Master';
```

ArrayIterator

ArrayIterator works similar to ArrayObject.

Let's take look at its class signature as well:

```
ArrayIterator implements ArrayAccess , SeekableIterator , Countable , Serializable {

}
```

It is almost identical to ArrayObject in terms of interfaces they implement. The only difference is, instead of ArrayIterator interface ArrayObject implements, it implements SeekableIterator.

We use ArrayIterator the same way as we use ArrayObject in a `foreach` loop:

```
$books = array(
    'Head First Design Patterns',
    'Clean Code: A Handbook of Agile Software Craftsmanship',
    'Domain-Driven Design: Tackling Complexity in the Heart of Software',
    'Agile Software Development, Principles, Patterns, and Practices',
);

$booksAsArrayIterator = new ArrayIterator($books);

foreach ($booksAsArrayIterator as $book) {
    echo $book . "\n";
}

// Output
Head First Design Patterns
Clean Code: A Handbook of Agile Software Craftsmanship
Domain-Driven Design: Tackling Complexity in the Heart of Software
Agile Software Development, Principles, Patterns, and Practices
```

Use array in Object-Oriented fashion:

```
$books = array(
    'Head First Design Patterns',
    'Clean Code: A Handbook of Agile Software Craftsmanship',
    'Domain-Driven Design: Tackling Complexity in the Heart of Software',
    'Agile Software Development, Principles, Patterns, and Practices',
);

$booksAsArrayIterator = new ArrayIterator($books);

$booksAsArrayIterator->append('The Pragmatic Programmer: From Journeyman to Master');
// --- vs ---
$books[] = 'The Pragmatic Programmer: From Journeyman to Master';
```

Comparison

You may be wondering when to use ArrayObject and when to use ArrayIterator. It is important to know the difference and the relationship between ArrayObject and ArrayIterator.

As we have already discovered in the ArrayObject section, ArrayObject actually creates ArrayIterator as an external iterator. It is fair to say ArrayIterator does what ArrayObject does, and it provides more functionality, specifically seeking to a position. This is accomplished by implementing SeekableIterator.

Besides moving a pointer from top to bottom as iterator, it allows you to randomly jump to a position.

```
$books = array(
    'Head First Design Patterns',
    'Clean Code: A Handbook of Agile Software Craftsmanship',
    'Domain-Driven Design: Tackling Complexity in the Heart of Software',
    'Agile Software Development, Principles, Patterns, and Practices',
);

$booksAsArrayIterator = new ArrayIterator($books);

$booksAsArrayIterator->seek(3);

echo $booksAsArrayIterator->current();

// Output
Agile Software Development, Principles, Patterns, and Practices
```

At last, ArrayIterator is part of SPL whereas ArrayObject is not.

Iterating the File System

It is a very common task to list outthe content of a given directory. PHP provides lots of functions for handling a file system. One of them is `scandir()`.

Suppose we are given a task to list out all files in a given directory as below:

```
---books
|   ---book_item_1.txt
|   ---book_item_2.txt
|   ---book_item_3.txt
|    ---book_item_4.txt
```

We can accomplish it through `scandir()` as shown below:

```
$books = scandir("books");
foreach($books as $book) {
    echo $book . "\n";
}

// Output
.
..
book_item_1.txt
book_item_2.txt
book_item_3.txt
book_item_4.txt
```

These are two virtual directories("." and "..") you'll find in each directory of the file system.

As this chapter is about iterators, we are going to introduce some iterators for handling filesystem. Hopefully in your next project, you will be able to utilize some of them. Three iterators come in handy: DirectoryIterator, FilesystemIterator and RecursiveDirectoryIterator.

Before we look into each one of them, it is useful to take a look at their inherit relationship:

```
DirectoryIterator extends SplFileInfo
FilesystemIterator extends DirectoryIterator
RecursiveDirectoryIterator extends FilesystemIterator
```

DirectoryIterator

The DirectoryIterator class provides a simple interface for viewing the contents of filesystem directories.

To accomplish the same task, we can use DirectoryIterator:

```
$books = new DirectoryIterator('books');

foreach($books as $book) {
    echo $book->getFilename() . "\n";
}

// Output
.
..
book_item_1.txt
book_item_2.txt
book_item_3.txt
book_item_4.txt
```

The only parameter needed to create a DirectoryIterator object is a directory's path. Compared to `scandir` function, instead of the file name as a string, DirectoryIterator returns an object. The object holds various information relating to a file, which we can make use.

FilesystemIterator

To accomplish the same task by using FilesystemIterator, we can use:

```
$books = new FilesystemIterator('books');

foreach($books as $book) {
    echo $book->getFilename() . "\n";
}

// Output
book_item_1.txt
book_item_2.txt
book_item_3.txt
book_item_4.txt
```

This looks almost the same as DirectoryIterator, except that FilesystemIterator has automatically filtered out the two virtual directories.

Are they really the same? We can use a simple method to tell the differences:

```
$books = new DirectoryIterator('books');
foreach($books as $key=>$value) {
    echo $key . ' is a type of '. gettype($key) . "\n";
    echo $value . ' is a type of '. get_class($value) . "\n";
}

echo '--------------------------'."\n";

$books = new FilesystemIterator('books');
foreach($books as $key=>$value) {
    echo $key . ' is a type of '. gettype($key) . "\n";
    echo $value . ' is a type of '. get_class($value) . "\n";
}
```

The result of running above script from CLI is:

```
0 is a type of integer
. is a type of DirectoryIterator
1 is a type of integer
.. is a type of DirectoryIterator
2 is a type of integer
book_item_1.txt is a type of DirectoryIterator
3 is a type of integer
book_item_2.txt is a type of DirectoryIterator
4 is a type of integer
book_item_3.txt is a type of DirectoryIterator
5 is a type of integer
book_item_4.txt is a type of DirectoryIterator

--------------------------------

books/book_item_1.txt is a type of string
books/book_item_1.txt is a type of SplFileInfo
books/book_item_2.txt is a type of string
books/book_item_2.txt is a type of SplFileInfo
books/book_item_3.txt is a type of string
books/book_item_3.txt is a type of SplFileInfo
books/book_item_4.txt is a type of string
books/book_item_4.txt is a type of SplFileInfo
```

Now we can see they are actually quite different internally:

- DirectoryIterator returns an integer as the key and a DirectoryIterator as the value in a loop.
- FilesystemIterator returns a string of full path as the key and a SplFileInfo object as the value in a loop.

In fact, FilesystemIterator comes with a bit more flexibility. When creating a FilesystemIterator object, it accepts a directory's path as the first parameter similar to DirectoryIterator. Moreover, you can optionally pass a second parameter as a flag. This flag is able to configure various aspects of this function.

- FilesystemIterator::CURRENT_AS_PATHNAME: This flag will make FilesystemIterator return file path instead of SplFileInfo object as the value.
- FilesystemIterator::CURRENT_AS_FILEINFO: This flag will make FilesystemIterator return SplFileInfo object as the value. This is the default behavior. You don't have to set it explicitly.
- FilesystemIterator::CURRENT_AS_SELF: This flag will make FilesystemIterator return FilesystemIterator itself as the value.
- FilesystemIterator::KEY_AS_PATHNAME: This flag will make FilesystemIterator return file path as the key. This is the default behavior. You don't have to set it explicitly.
- FilesystemIterator::KEY_AS_FILENAME: This flag will make FilesystemIterator return file name and extension instead of file path as the key.
- FilesystemIterator::FOLLOW_SYMLINKS: This flag will make RecursiveDirectoryIterator::hasChildren() follow symlinks.
- FilesystemIterator::NEW_CURRENT_AND_KEY: This flag helps set two other flags(FilesystemIterator::KEY_AS_FILENAME and FilesystemIterator::CURRENT_AS_FILEINFO) at once.
- FilesystemIterator::SKIP_DOTS: This flag will make FilesystemIterator ignore virtual directories ("." and "..").
- FilesystemIterator::UNIX_PATHS: This flag will make FilesystemIterator use Unix style directory separator() despite what system the PHP script runs on.

Peeking ahead with CachingIterator

In this section, we will introduce an iterator with the ability of peeking into next element in an iteration. This feature enables us to do a lot useful things such as, executing something different when iterator reaches the end of the list.

The class with this great power is CachingIterator.

Let's first take a look at it class signature, then, we will go into details of its usage.

```
CachingIterator extends IteratorIterator
```

CachingIterator inherits from IteratorIterator. What is IteratorIterator? It is simply a wrapper around another iterator, under the hood. It will forward the five Itertator methods(`rewind()` , `current()` , `key()` , `valid()` , `next()`) calls to the iterator it wraps around. We can also retrieve the inner iterator by calling method `getInnerIterator()` .

Due to the nature of this class, the inner iterator's pointer always moves one step ahead of CachingIterator, and CachingIterator provides a method `hasNext()` to tell us if it reaches the end of the list. That is how CachingIterator peeks ahead.

Now, let's put it into action.

```
$books = array(
    'Head First Design Patterns',
    'Clean Code: A Handbook of Agile Software Craftsmanship',
    'Domain-Driven Design: Tackling Complexity in the Heart of Software',
    'Agile Software Development, Principles, Patterns, and Practices',
);

$booksAsCachingIterator = new CachingIterator(new ArrayIterator($books));

foreach ($booksAsCachingIterator as $book) {
    echo 'current book - ' . $book . PHP_EOL;
    if ($booksAsCachingIterator->hasNext()) {
        echo 'next book - ' . $booksAsCachingIterator->getInnerIterator()->current() . PH
        echo '----------------------------' . PHP_EOL;
    }
}
```

Result of running above script in a CLI:

```
current book - Head First Design Patterns
next book - Clean Code: A Handbook of Agile Software Craftsmanship

----------------------------
current book - Clean Code: A Handbook of Agile Software Craftsmanship
next book - Domain-Driven Design: Tackling Complexity in the Heart of Software

----------------------------

current book - Domain-Driven Design: Tackling Complexity in the Heart of Software
next book - Agile Software Development, Principles, Patterns, and Practices

----------------------------

current book - Agile Software Development, Principles, Patterns, and Practices
```

Similar to other iterators, to create an CachingIterator instance, we pass in an iterator as the first parameter to the class contractor. As we can see, the real magic behind peeking ahead is provided by method `hasNext()`. This method is able to tell us if there is an immediate next element.

Beside the first parameter, CachingIterator also optionally accepts a second parameter as a flag.

- CachingIterator::CALL_TOSTRING: It will return __toString of the current element as value. This is the default behavior.
- CachingIterator::CATCH_GET_CHILD: It will capture all exceptions thrown when accessing children.
- CachingIterator::TOSTRING_USE_KEY: It will return the key value when casting the iterator to a string in a loop.

```
$books = array(
    'Head First Design Patterns',
    'Clean Code: A Handbook of Agile Software Craftsmanship',
    'Domain-Driven Design: Tackling Complexity in the Heart of Software',
    'Agile Software Development, Principles, Patterns, and Practices',
);

$booksAsCachingIterator = new CachingIterator(new ArrayIterator($books), CachingIterator:

foreach ($booksAsCachingIterator as $key=>$book) {
    echo $booksAsCachingIterator . PHP_EOL;
}

// Output
0
1
2
3
```

- CachingIterator::TOSTRING_USE_CURRENT: It will return the current value when casting the iterator to a string in a loop.

```
$books = array(
    'Head First Design Patterns',
    'Clean Code: A Handbook of Agile Software Craftsmanship',
    'Domain-Driven Design: Tackling Complexity in the Heart of Software',
    'Agile Software Development, Principles, Patterns, and Practices',
);

$booksAsCachingIterator = new CachingIterator(new ArrayIterator($books), CachingIterator:

foreach ($booksAsCachingIterator as $key=>$book) {
    echo $booksAsCachingIterator . PHP_EOL;
}

// Output
Head First Design Patterns
Clean Code: A Handbook of Agile Software Craftsmanship
Domain-Driven Design: Tackling Complexity in the Heart of Software
Agile Software Development, Principles, Patterns, and Practices
```

- CachingIterator::TOSTRING_USE_INNER: It will return the inner iterator casted to a string when casting the iterator to a string in a loop. If we set this flag in the same code as previous example, it will throw an exception. This is because ArrayIterator does not implement the `__toString()` method.

- CachingIterator::FULL_CACHE: A CachingIterator won't have a key word "caching" in its name if it is not able to do some sort of cache. When this flag is set, it will cache the results, should they ever be iterated for future use.

Generator

You are now convinced by the benefits of iterators. They encapsulate the details of traversing and they are much more efficient than creating in-memory arrays. However, everything has its price. To create an iterator, we still have to implement the SPL Iterator interface. You might be terrified of iterators and not want to implement those five methods contracted by Iterator interface. It is time consuming and sometimes even complex to implement them.

Starting from PHP 5.5, you won't be intimidated any more. PHP introduces something, Generators, which provide an easy way to implement simple iterators without the overhead or complexity of implementing a class that implements the Iterator interface.

What is exactly a generator? A generator is like a normal PHP function, except that it has a special keyword , "yield", in it.

Below is a simple example of a generator function. We won't have such a generator in the real world application - it is here for demonstration only:

```
function booksGenerator()
{
    $books = array(
        'Head First Design Patterns',
        'Clean Code: A Handbook of Agile Software Craftsmanship',
        'Domain-Driven Design: Tackling Complexity in the Heart of Software',
        'Agile Software Development, Principles, Patterns, and Practices',
    );

    foreach ($books as $book) {
        yield $book;
    }
}

foreach (booksGenerator() as $book) {
    echo $book . PHP_EOL;
}

// Output
Head First Design Patterns
Clean Code: A Handbook of Agile Software Craftsmanship
Domain-Driven Design: Tackling Complexity in the Heart of Software
Agile Software Development, Principles, Patterns, and Practices
```

Internally PHP realizes a generator function when it spots the `yield` keyword. When a generator function is called for the first time, PHP creates a Generator object. This Generator object is an instance of an internal class Generator and Generator class implements the Iterator interface. This way, users are able to create iterators without writing the contracted code, all thanks to PHP generator.

The `yield` is called when we need to provide the step values. Think of it as `return` in a function or `current` method in a regular iterator.

Let's turn one of our first iterator class TrendingRepositoriesIterator to a generator function:

```
function trendingRepositoriesGenerator()
{
    $client = new GuzzleHttp\Client();

    $res = $client->request('GET', 'https://api.github.com/search/repositories', [
        'query' => ['q' => 'language:php', 'sort' => 'stars', 'order' => 'desc']
    ]);

    $resInArray = json_decode($res->getBody(), true);

    $trendingRepos = array_slice($resInArray['items'], 0, 10);

    foreach ($trendingRepos as $rep) {
        yield $rep['name'];
    };
}
```

It turns out to be much less code with a generator. We can also use it in a foreach loop, in the same way as we did with TrendingRepositoriesIterator:

```
foreach (trendingRepositoriesGenerator() as $repo) {
    echo $repo . PHP_EOL;
}
```

Note that generators themselves do not provide anything special - they just make creating iterators simpler. In other words, they are definitely not replacements for iterators.

Exception handling

Since PHP 5 was released, Exception is added to PHP as an object-oriented programming language feature. By definition, an Exception is an exceptional event during program execution. In PHP, an Exception is simply an object (an instance of Exception class). When an exception occurs, PHP will halt current execution flow and look for an handler, and then it will continue its execution by the handler's code. If no handler is found, a PHP Fatal Error will be issued with an "Uncaught Exception ..." message and the program terminates.

When to use Exception

Exception is good for handling exceptional cases of your program, however it is not the solution for all error cases. Sometimes it is perfectly fine to return a boolean FALSE. Sometimes you are much better off throwing exceptions instead of returning weird error codes. Therefore it is very important to understand when to use Exception and when not to.

By now, we all know an exception should be thrown when an exceptional situations occurs. But if exceptional situation seems rather arbitrary, what qualifies as an "exceptional" situation?

Here is a good rule of thumb: since exceptional situations don't happen frequently, if you supply correct values to your function and remove the thrown exception, if the function then fails, the exception is used incorrectly.

Let's take a look at some concrete examples:

- If you create a function to save a user's input to database, when a database connection fails, an exception should be thrown.
- For the same function, you create a validator for checking a user's input. When an invalid value is supplied, you should not throw an exception. Invalid value is a rather frequent case for a validator class.

A good use case of Exception

Here we have an example of returning error codes to indicate error cases:

```
class User
{
    ...

    public function login()
    {
        if ($this->invalidUsernameOrPassword()) {
            return -2;
        }

        if ($this->tooManyLoginAttempts()) {
            return -1;
        }

        return 1;
    }

    public function redirectToUserPage()
    {
        ...
    }
}
```

Client code might be something similar to the below:

```
$user = new User();
$result = $user->login();

if (-2 == $result) {
    log('invalid username or password');
} else if (-1 == $result) {
    log('too many login attempts');
} else if (1 == $result) {
    $user->redirectToUserPage();
}
```

Here we can spot a couple of problems with error codes:

- Error codes do not contain error related information by themselves, which make them very hard to maintain.
- Error codes result in number of `if/else` statements in client's code, depending on how many there are. (Conditional statements should be eliminated as much as possible, in order to make our code clean).

Let's refactor the code to use exceptions:

```
class User
{
    ...

    public function login()
    {
        if ($this->invalidUsernameOrPassword()) {
            throw new InvalidCredentialException('Invalid username or password');
        }

        if ($this->tooManyLoginAttempts()) {
            throw new LoginAttemptsException('Too many login attempts');
        }
    }

    public function redirectToUserPage()
    {
        ...
    }
}
```

And client code can be refactored as:

```
try {

    $user = new User();
    $user->login();
    $user->redirectToUserPage();

} catch (InvalidCredentialException $e) {
    log($e->getMessage());
} catch (LoginAttemptsException $e) {
    log($e->getMessage());
}
```

As we can see, by using exceptions, the second code sample conveys messages about the errors much more clearly. Beyond that, in the client code, by eliminating conditional statements, the code become self-explanatory.

A case of misusing Exception

One common way of misusing exceptions is using them to control the flow of application logic. It is not only confusing, but also slows down your code. To emphasize again, exception is used to raise exceptional situations.

Below is one example of of misusing exceptions, which is discouraged.

```
function register($email, $role) {
    try {

        if ($role == 'member') {
            throw new CreateMemberException();
        } else if ($role == 'admin') {
            throw new CreateAdminException();
        }

    } catch (CreateMemberException $e) {

        // code for creating member account

    } catch (CreateAdminException $e) {

        // code for creating admin account

    }
}
```

The function `register()` uses exceptions to delegate account creation tasks. This is obviously against rules of exceptions. Though PHP does not stop you, you should religiously forbid yourself from doing this.

How to use Exception

Four key words are associated with Exception. They are: `throw` , `try` , `catch` and `finally` . An exception object is thrown(`throw`) in a method when an exceptional event happens. Clients that call the method will normally place the method in a `try` block and `catch` it with some handling code. A `finally` block ensures the code inside the block will always be executed.

Throw

All exceptions in PHP are a class or subclass of **Exception**. It takes three optional parameters in its constructor.

```
public __construct ([ string $message = "" [, int $code = 0 [, Exception $previous = NULL
```

- $message: The exception message. This message provides some human readable information. And normally supply this parameter when instantiating an exception.
- $code: It is useful for identifying types of exceptions that belong to the same class.
- $previous: The exception before current one. This is normally used when we want to re-throw an exception in a `catch` block.

Below is an example of PHP syntax to throw an exception

```
throw new Exception('some error message');
```

The keyword here is `throw` . Note that we first need to initiate an exception object.

Catch

When we need to catch exceptions, we place the callee in a try-catch block as below:

```
<?php
try {
    methodThatThrowsExceptions();
} catch (Exception $e) {
    // handle exception gracefully
}
?>
```

The `catch` block is where we place our handler code. Detailed exception handling implementation varies depending on application design. For example, we could try to recover the exception as much as we can, and if that is not possible, we could redirect users to a customer support page. If we leave it unhanded, PHP will eventually terminate the program and leave users with a page of meaningless error message, which is discouraged.

Exception bubble effect

If you are using some kind of frameworks, exceptions are likely handled even if you never actually create any handler for them. That is because exceptions bubble up, and your framework will eventually handle them. A simple example of exception bubble effect is:

```
function methodA()
{
    throw new Exception('error from methodA');
}

function methodB()
{
    methodA();
}

function methodC()
{
    try {
        methodB();
    } catch (Exception $e) {
        // handle error gracefully
    }
}
```

In the sample code, when **methodC** is called, it calls **methodB**, which directly invokes **methodA**. An exception is thrown in **methodA**, since **methodB** does not handle it. It then bubbles up to **methodC**, which gracefully handles the exception. In this example, though **methodC** does not call **methodA** directly, because exception bubbles up the stack, it is still handled gracefully at the end.

Multiple catch blocks

A method might contain different exceptions: some might be directly thrown by themselves and some might be bubbled up from their underlying stack. The `catch` block is designed to handle multiple exceptions, so we can have multiple catch blocks for handling different exceptions. A caveat here is that the position matters.

In multiple catch blocks, PHP picks the first block that matches the thrown exception's type. A good rule for positioning catch blocks is from a more specific one to a less specific one.

Let's see it in an example.

```
class ExceptionA extends Exception{}

class ExceptionB extends ExceptionA{}

try {

    methodThatThrowsExceptionA();

} catch (ExceptionA $e) {

} catch (ExceptionB $e) {

} catch (Exception $e) {

}
```

In the sample code, it is obvious ExceptionA catch block will get picked. Now let's change the method to throw ExceptionB.

```
class ExceptionA extends Exception{}

class ExceptionB extends ExceptionA{}

try {

    methodThatThrowsExceptionB();

} catch (ExceptionA $e) {

} catch (ExceptionB $e) {

} catch (Exception $e) {

}
```

Which catch block do you think it will be picked? The answer is still ExceptionA. Because ExceptionA is a parent class of ExceptionB, when ExceptionB is thrown, ExceptionA catch block comes first and matches the thrown exception's type, giving ExceptionB is an instance of ExceptionA.

This can be fixed by positioning them from the the more specific type to a less specific type, as shown below:

```
class ExceptionA extends Exception{}

class ExceptionB extends ExceptionA{}

try {

    methodThatThrowsExceptionB();

} catch (ExceptionB $e) {

} catch (ExceptionA $e) {

} catch (Exception $e) {

}
```

trace message

Since exceptions can be thrown anywhere in your program, it is very important to find the root cause. Exception provides various APIs to make it easy to trace where the exception comes from.

Seven public methods are extracted from PHP manual file:

```
final public string getMessage ( void )
final public Exception getPrevious ( void )
final public mixed getCode ( void )
final public string getFile ( void )
final public int getLine ( void )
final public array getTrace ( void )
final public string getTraceAsString ( void )
```

And we can use them to trace details of the thrown exception:

- Exception::getMessage — Gets the Exception message
- Exception::getPrevious — Returns previous Exception
- Exception::getCode — Gets the Exception code
- Exception::getFile — Gets the file in which the exception occurred
- Exception::getLine — Gets the line in which the exception occurred
- Exception::getTrace — Gets the stack trace
- Exception::getTraceAsString — Gets the stack trace as a string

Let's put it in action.

For demonstration purpose, let's pretend we have a `createAccount()` method, which throws an Exception when an email address is invalid.

```
function createAccount($email)
{
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        throw new Exception('In valid email address');
    }

    return sprintf('account creation email is sent to %s', $email);
}
```

What information can we get if we trigger the exception?

```
function linSeparator()
{
    return PHP_EOL.'----------------------------------'.PHP_EOL;
}

try {

    createAccount('test');

} catch (Exception $e) {

    echo $e->getMessage();
    echo linSeparator();

    echo $e->getPrevious();
    echo linSeparator();

    echo $e->getCode();
    echo linSeparator();

    echo $e->getFile();
    echo linSeparator();

    print_r($e->getTrace());
    echo linSeparator();

    echo $e->getTraceAsString();

}
```

Running the script from CLI, we get the following:

```
In valid email address

---------------------------------



---------------------------------
0

---------------------------------
/Users/xu/Desktop/Exception/trace.php

---------------------------------
Array
(
    [0] => Array
        (
            [file] => /Users/xu/Desktop/Exception/trace.php
            [line] => 19
            [function] => createAccount
            [args] => Array
                (
                    [0] => test
                )

        )

)

---------------------------------
#0 /Users/xu/Desktop/Exception/trace.php(19): createAccount('test')
```

Beside obvious trace information, we can also tell that, the default code is 0 and previous exception is null when instantiating an exception object.

```
public __construct ([ string $message = "" [, int $code = 0 [, Exception $previous = NULL
```

One more point we want to address here is that an exception is created when it is instantiated, not when it is thrown. So Exception APIs will give you information related to the time an exception is instantiated.

For example, in the method below, Exception::getLine will return 2.

```
function methodThrowException()
{
    $exception = new Exception('error from methodThrowException'); // line 2

    throw $exception; // line 3
}
```

finally

Before PHP 5.5, there was not `finally` block in PHP. A problem surfaced. If we wanted to ensure that one piece of code always ran regardless of which `catch` block got picked, we had to put that piece of code into each one of `catch` blocks.

To solve this problem, `finally` block was introduced since PHP 5.5. Code inside `finally` block will always be executed after `catch` block. We can even use `try` / `finally` only without `catch` .

This block is a place for us to do some clean up jobs. Tasks such as rolling back database transactions, closing database connections, releasing file locks and so on. Its usage is pretty straightforward.

For example, to use it alongside `try` / `catch` blocks:

```
try {

    createAccount('test');

} catch (Exception $e) {

    echo $e->getMessage();

} finally {

    echo 'Close Database Connection';
}
```

Use `try` / `finally` block only:

```
try {

    createAccount('test');

} finally {

    echo 'Close Database Connection';
}
```

Create your first custom exception

Throwing custom exception allows client code to handle the error case in a recognized manner. For example, when a database exception is thrown, it is reasonable to shut down the process completed. However, in the case of an invalid user input, we might just want to log an error message.

By creating custom exceptions, we express error cases of our code proactively. This not only helps the clients to avoid pitfalls, but also gives them enough information to handle the error cases confidently.

As all exceptions in PHP 5.x use **Exception** as the base, we are actually extending **Exception** to create our custom exception. In this example, let's revisit our previous code.

```
class User
{
    ...

    public function login()
    {
        if ($this->invalidUsernameOrPassword()) {
            throw new InvalidCredentialException('Invalid username or password');
        }

        if ($this->tooManyLoginAttempts()) {
            throw new LoginAttemptsException('Too many login attempts');
        }
    }

    public function redirectToUserPage()
    {
        ...
    }
}
```

We have two custom exceptions here (**InvalidCredentialException** and **LoginAttemptsException**). They should actually be under one type. And they will be assign different messages.

Since **InvalidCredentialException** and **LoginAttemptsException** are error cases for an invalid login runtime error, it is reasonable to create an exception called **InvalidLoginException**, and use it for the two error cases above.

It is simple enough to create a custom exception with only one line of code.

```
class InvalidLoginException extends Exception {}
```

We can refactor previous code to use the newly created exception class:

```
class User
{
    ...

    public function login()
    {
        if ($this->invalidUsernameOrPassword()) {
            throw new InvalidLoginException('Invalid username or password');
        }

        if ($this->tooManyLoginAttempts()) {
            throw new InvalidLoginException('Too many login attempts');
        }
    }

    public function redirectToUserPage()
    {
        ...
    }
}
```

A little trick

A potential issue might appear shortly, if we are using **InvalidLoginException** with too many different messages. The issue is easy to illustrate.

Imagine somewhere in our code, we need to throw another **InvalidLoginException** when an user account is blocked. We will throw the exact **InvalidLoginException**, but with different messages. The same thing happens again, and we will repeat the same actions. The list adds up. Now imagine doing this for different types of exceptions. We would lose track as the developer.

So here is a little trick: we shift the exception creation task to **InvalidLoginException** class.

```
class InvalidLoginException extends Exception
{
    public static function invalidUsernameOrPassword() {
        return new static('Invalid username or password');
    }

    public static function tooManyLoginAttempts() {
        return new static('Too many login attempts');
    }
}
```

Now the client code becomes:

```
class User
{
    ...

    public function login()
    {
        if ($this->invalidUsernameOrPassword()) {
            throw InvalidLoginException::invalidUsernameOrPassword();
        }

        if ($this->tooManyLoginAttempts()) {
            throw InvalidLoginException::tooManyLoginAttempts();
        }
    }

    public function redirectToUserPage()
    {
        ...
    }
}
```

When the instantiation of exception is shifted to a function block, we gain much more space and freedom to do more interesting stuff, compared to a single line within the `if` block previously.

By keeping all of them in a centralized location, which is the exception class itself, not only does it create a more maintainable code base, but also give clients an opportunity to take a quick glance what exact exception they would expect.

SPL exceptions

Creating your own custom exception is great, but it does take some mental energy to come out as a meaningful name. Naming is hard, arguably one of the hardest thing in programming.

The Standard PHP Library (SPL) provides a set of standard Exceptions. We should use them for our own benefit. They cover a list of common error cases, which can save us energy if we were to come out on our own. Additionally, we can also expand from these standard Exceptions to make them more specific to our own domain.

In this sections, we will go through 14 SPL Exceptions, explaining them in simplest terms, so that you can use them next time in your own project.

```
-LogicException (extends Exception)
    --BadFunctionCallException
    --BadMethodCallException
    --DomainException
    --InvalidArgumentException
    --LengthException
    --OutOfRangeException
-RuntimeException (extends Exception)
    --OutOfBoundsException
    --OverflowException
    --RangeException
    --UnderflowException
    --UnexpectedValueException
```

There are two main categories of SPL Exceptions. They are **LogicException** and **RuntimeException**, Under each one of them, there are a few sub exception classes describing more specific error cases.

Logic Errors

LogicExcetpion

It's not hard to tell from its name that, **LogicException** covers error cases related to logics. As it is a parent class for a few more specific exceptions, it is a bit generic. When your code is returning or receiving something that is not logic, there is a logic error. When an error case is determined to be a logic error, use **LogicException** if you cannot find a better fit from its subclasses.

BadFunctionCallException

When an non-existing function get called, or wrong parameters are supplied to a function, this exception gets thrown. As this exception covers function scope, not method in a class, it is mostly thrown by PHP.

BadMethodCallException

When an non-existing method of a class gets called, or wrong parameters are supplied to that method, an **BadFunctionCallException** can be thrown. While this exception is similar to to **BadFunctionCallException**, it is designed for class scope.

DomainException

Domain here refers to the business our code works for. When a parameter is valid by its datatype, but invalid to the domain, a **DomainException** can be thrown.

For example, in an universal image manipulation function, `transformImage($imageType)` , **DomainException** should be thrown when $imageType contains an invalid image type. To this domain, an invalid image type is a domain error.

InvalidArgumentException

This one is simple, as its name says: it should be thrown when an invalid argument is supplied.

PHP5 introduces type hinting, however it does not yet work for scalar types such as int, string yet. To make it work, we throw **InvalidArgumentException** when a scalar type does not meet the requirement.

LengthException

We can use this exception when length of something is invalid. For example, password needs to be at least 8 characters.

OutOfRangeException

Use this exception when an invalid index is accessed. The keyword here is range.

RuntimeException

RuntimeException is a name derived from compile language, such as Java. In Java, there are two main categories of exception: checked exceptions and runtime exceptions. A complier won't compile the code until all checked exception are handled (in a `catch` block). Runtime exception can only be detected at run time and is not required to place these exceptions in a `catch` block.

Since PHP is not a compile language, we can think of its "compile time" as the time we write the code, and its "run time" as the code execution time. "Compile time" exceptions can be detected in development time, for example invalid datatype parameter.

To avoid confusion, keep in mind that logic exceptions discussed above are for "compile time".

RuntimeException's subclasses cover more specific scenarios. Use this exception if you cannot find a better fit from its subclasses.

OutOfBoundsException

This exception is used when an invalid index is called. Not to be confused with OutOfRangeException, OutOfBoundsException is a run time exception.

For example, when a user creates an array data structure and when an invalid index is called, an OutOfBoundsException should be thrown. Whereas trying to obtain the day of week using 8 should throw an OutOfRangeException.

```
$booksList = array();
$bookList[5];     // OutOfBoundsException

$dayOfWeek = $calendar->day(8); // OutOfRangeException
```

OverflowException

When a container with limited capacity is asked to fill more than it can hold, this exception can be thrown.

RangeException

This exception is for generic error cases related to range at "run time".

UnderflowException

Opposite of OverflowException is UnderflowException. When an empty container is asked to remove an element, this exception can be thrown.

UnexpectedValueException

As simple as its name says, when an unexpected value is raised or accessed, we throw this exception.

That is all the exceptions provided by PHP SPL. We should always throw the most accurate exception for an error case. Inevitably, one exception may fit under multiple exceptions in which case, it is fine to pick one.

An meaningful exception message goes a long way towards a maintainable project.

Test Driven Development

If you have not heard of Test Driven Development(TDD), you should begin to familiarise yourself with it. Though PHP community is a bit late on TDD practice compared to other languages such as Ruby, once the benefits TDD were realized, it has become almost essential for a modern PHP developer.

TDD is a software development technique. The basic idea behind TDD is that, we create tests before we actually code any thing. Writing test against no code is more of a mindset shift than anything else. It is opposite of traditional coding habit, where we create code first, then manually run the unit to make sure it does what we intended manually. The benefits that TDD brings to us are enormous. At first it forces us to think about code design before we create any concrete code, then it allows us to refactor our code base without worrying about the side effect. It makes our code easy to maintain in the long run.

TDD consists of three phases: which are Red, Green and Refactor.

Red phase

In red phase, as the developer, we will plan out what the code will look like without actually writing it. This is to say, we will design our class or class methods, without implementing its details. Initially this phase is hard, it requires us to change our traditional habit of coding. But once we get used to this process, we will naturally adapt to it and realize that it helps us design better code. It is about changing our mindset, as we should focus on the input and output of the API, instead of the details of the code. The result of this phase is successful creation of red test.

Green phase

In green phase, it is all about writing the quickest piece of code to pass the tests. In this phase, we should not spend too much making the code clean or refactoring. Though we all want to write the most beautiful piece of code, that is not the task at hand in this phase. The result of this phase is green tests.

Refactor phase

In refactor phase, we focus on making the code clean. Since we have tests created above to guard bugs from side effects, we gain confidence for carrying out refactor. If by chance, a bug is introduced from refactoring, our tests will report it as soon as it appears. So the natural way of refactor is to run the test as soon as you have modified any code.

PHPUnit

TDD lets us test drive our development cycle. When practicing TDD in PHP, obviously we need to define the kind of test we will do. The most common test in TDD is Unit Test which tests the smallest testable parts of an application it considers a unit, which is typically a class method.

Now imagine writing unit tests manually and building an automated method to run them. It is definitely a lot of work. Fortunately, there are already unit testing frameworks out there for us to use. Among a number of unit testing frameworks, PHPUnit is the most popular one and it is widely used in the PHP community.

Getting started with PHPUnit

Installation

The easiest way to install PHPUnit is via Composer. Open up your terminal and in your project folder, simply run `composer require phpunit/phpunit`. By default, the bin file of PHPUnit will be placed into vendor/bin folder, so we can run `vendor/bin/phpunit` directly from our project's root folder.

Your first unit test

Time to create your first unit test! Before doing so, we need a class to test. Let's create a very simple class called Calculator and write a test for it.

Create a file with the name of **Calculator.php** and copy the code below to the file. This **Calculator** class only has an Add function. :

```
<?php
class Calculator
{

    public function add($a, $b)
    {
        return $a + $b;
    }

}
```

Create the test file **CalculatorTest.php**, and copy the code below to the file. We will explain each function in details.

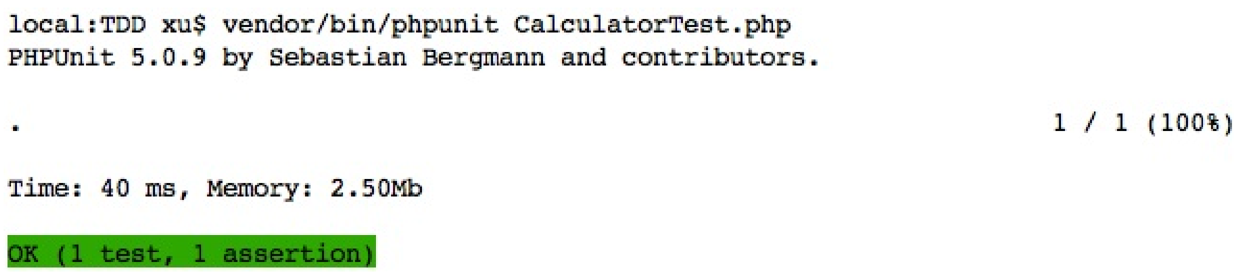
```
<?php
require 'Calculator.php';

class CalculatorTest extends PHPUnit_Framework_TestCase
{
    private $calculator;

    protected function setUp()
    {
        $this->calculator = new Calculator();
    }

    protected function tearDown()
    {
        $this->calculator = NULL;
    }

    public function testAdd()
    {
        $result = $this->calculator->add(1, 2);
        $this->assertEquals(3, $result);
    }

}
```

- Line 2: Includes class file **Calculator.php**. This is the class that we are going to test against, so make sure you include it.
- Line 8: `setUp()` is called before each test runs. Keep in mind that it runs before each test, which means, if you have another test function, it too will run `setUp()` before.
- Line 13: Similar to `setUp()` , `tearDown()` is called after each test finishes.
- Line 18: `testAdd()` is the test function for add function. PHPUnit will recognize all functions prefixed with test as a test function and run them automatically. This function is actually very straightforward: we first call Calculator.add function to calculate the value of 1 plus 2. Then we check to see if it returns the correct value by using PHPUnit function assertEquals.

The last part of the task is to run PHPUnit and make sure it passes all tests. Navigate to the folder where you have created the test file and run the commands below from your terminal:

`vendor/bin/phpunit CalculatorTest.php`

You should be able to see the successful message as below:



```
local:TDD xu$ vendor/bin/phpunit CalculatorTest.php
PHPUnit 5.0.9 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 40 ms, Memory: 2.50Mb

OK (1 test, 1 assertion)
```

Data Provider

When to use data provider

When we write a function, we want to make sure it passes a series of edge cases. The same applies to tests. This means we will need to write multiple tests to test the same function using different sets of data. For instance, if we want to test our **Calculator** class using different data, without data provider, we would have multiple tests as shown below:

```
<?php
require 'Calculator.php';

class CalculatorTest extends PHPUnit_Framework_TestCase
{
        private $calculator;

        protected function setUp()
        {
            $this->calculator = new Calculator();
        }

        protected function tearDown()
        {
            $this->calculator = NULL;
        }

        public function testAdd()
        {
            $result = $this->calculator->add(1, 2);
            $this->assertEquals(3, $result);
        }

        public function testAddWithZero()
        {
            $result = $this->calculator->add(0, 0);
            $this->assertEquals(0, $result);
        }

        public function testAddWithNegative()
        {
            $result = $this->calculator->add(-1, -1);
            $this->assertEquals(-2, $result);
        }

}
```

In this case, we can use data provider function in PHPUnit to avoid duplication in our tests.

How to use data provider

A data provider method returns a variety of arrays or an object that implements the **Iterator** interface. The test method will be called with the contents of the array as its arguments.

Some key points to keep in mind when using data provider are:

- Data provider method must be public.
- Data provider returns an array of a collection data.
- Test method use annotation(@dataProvider) declares its data provider method.

Once we know the key points, it is actually quite straightforward to use data provider. First, we create a new public method, which returns an array of a collection data as arguments of the test method.Then, we add annotation to the test method to tell PHPUnit which method will provide arguments.

Add data provider to our first unit test

Let's modify our tests above using data provider.

```php
<?php
require 'Calculator.php';

class CalculatorTest extends PHPUnit_Framework_TestCase
{
    private $calculator;

    protected function setUp()
    {
        $this->calculator = new Calculator();
    }

    protected function tearDown()
    {
        $this->calculator = NULL;
    }

    public function addDataProvider() {
        return array(
            array(1,2,3),
            array(0,0,0),
            array(-1,-1,-2),
        );
    }

    /**
     * @dataProvider addDataProvider
     */
    public function testAdd($a, $b, $expected)
    {
        $result = $this->calculator->add($a, $b);
        $this->assertEquals($expected, $result);
    }

}
```

- Line 18: Add a data provider method. Take note that a data provider method must be declared as public.
- Line 27: Use annotation to declare the test method's data provider method.

Now, run our test again and it should pass. As you can see, we have utilized data provider to avoid code duplication. Instead of writing three test methods for essentially the same method, we now have only one test method.

Test Double

When to use test double

As mentioned in the first part of this series. One of PHPUnit's powerful features is test double. It is very common in our code that a method of a class calls another class's method. In this case, there is a dependency between these two classes. In particular, the caller class has a dependency on the calling class, but as we already know from part 1, unit test should test the smallest unit of functionality. In this case, it should test only the caller function. To solve this problem, we can use test double to replace the calling class. Since a test double can be configured to return predefined results, we can focus on testing the caller function.

Types of test doubles

Test double is a generic term for objects we use, to replace real production ready objects. In our experience, it is very useful to categorize test doubles by their purpose. It not only makes it easy for us to understand the test case, but also make our code friendly to other parties.

Accordingly to Martin Fowler's post, there are five types of test double:

- Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- Fake objects actually have working implementations, but usually take some shortcuts, which make them not suitable for production.
- Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- Mocks are pre-programmed with expectations that form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they received all the calls they were expecting.

How to create test double

PHPUnit's method getMockBuilder can be used to create any similar user defined objects. Combining with its configurable interface, we can use it to create basically all five types of test doubles.

Add test double to our first unit test

It is meaningless to use test double in our calculator test case, since currently the Calculator class has no dependency on other classes, however, to demonstrate how to use test double in PHPUnit, we will create a stub Calculator class and test it.

Let's add a test case called testWithStub to our existing class:

```
public function testWithStub()
{
    // Create a stub for the Calculator class.
    $calculator = $this->getMockBuilder('Calculator')
                       ->getMock();

    // Configure the stub.
    $calculator->expects($this->any())
               ->method('add')
               ->will($this->returnValue(6));

    $this->assertEquals(6, $calculator->add(100,100));
}
```

- getMockBuilder() method creates a stub similar to our Calculator object.
- getMock() method returns the object.
- expects() method tells the stub to be called any number of times.
- method() method specifies which method it will be called.
- will() method configures the return value of the stub.

We have introduced some basic usage of PHPUnit, which provides almost all the features we would need to create unit tests. You should always try to find more information from its official manual as you needed.

TDD by example

In this section, we will demonstrate the process behind TDD through a very simple example. You should concentrate on how the three phases of TDD are carried out in this example.

Suppose we are given a task of building a price calculator for our e-commerce system. The class we are going to develop will be **PriceCalculator**. Let's first setup the project's folder and file structure as well as its dependencies.

As usual, we will use **Composer** as our package manager and PSR-4 as our code standard. The only third party dependency is **PHPUnit**. To set things up, we will create a folder **src** for placing our source files, and a folder **tests** for placing test files. We will also create **src/PriceCalculator.php** and **tests/PriceCalculatorTest.php** respectively. Finally, we will create a **composer.json** file as below:

```
{
    "require": {
        "phpunit/phpunit": "^5.0"
    },
    "autoload": {
        "psr-4": {
            "Dilab\\Order\\": "src"
        }
    }
}
```

This file tells **Composer** to download PHPUnit and tells autoloader that our source code follows PRS-4 standard.

By running command **composer install**, we should end up with a folder structure as below:

```
.
+-- src
|   +-- PriceCalculator.php
+-- tests
|   +-- PriceCalculatorTest.php
+-- vendor
|   +-- dependency-1
|   +-- dependency-2
|   +-- dependency-3
|   +-- dependency-xxx
+-- composer.json
+-- composer.lock
```

The final piece we need for the setup is a **phpunit.xml** file to config PHPUnit. Let's create it as the folder root.

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         bootstrap="vendor/autoload.php"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false"
         syntaxCheck="false">
    <testsuites>
        <testsuite name="Test Suite">
            <directory suffix=".php">./tests/</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Our final folder structure should be as shown below:

```
.
+-- src
|   +-- PriceCalculator.php
+-- tests
|   +-- PriceCalculatorTest.php
+-- vendor
|   +-- dependency-1
|   +-- dependency-2
|   +-- dependency-3
|   +-- dependency-xxx
+-- composer.json
+-- composer.lock
+-- phpunit.xml
```

Red phase

At this phase we will plan how our API will look like and create failing test. In this example, the required API method is very simple. We just want a method that accepts an array as its parameter and calculate the total price. We will name this method `total`.

Let's create some tests in `tests/PriceCalculatorTest.php` file before we write any source code.

```
<?php
```

```
namespace Dilab\Order\Test;

use Dilab\Order\PriceCalculator;

class PriceCalculatorTest extends \PHPUnit_Framework_TestCase
{
    private $PriceCalculator;

    public function setUp()
    {
        parent::setUp();
        $this->PriceCalculator = new PriceCalculator();
    }

    public function tearDown()
    {
        parent::tearDown();
        unset($this->PriceCalculator);
    }

    /**
     * @test
     */
    public function object_can_created()
    {
        $priceCalculator = new PriceCalculator();
        $this->assertInstanceOf('Dilab\Order\PriceCalculator', $priceCalculator);
    }

    /**
     * @test
     */
    public function should_sum_price()
    {
        $items = [
            ['price' => 100],
            ['price' => 200],
        ];

        $result = $this->PriceCalculator->total($items);

        $this->assertEquals(300, $result);
    }

    /**
     * @test
     */
    public function empty_items_should_return_zero()
    {
        $items = [];

        $result = $this->PriceCalculator->total($items);
        $this->assertEquals(0, $result);
    }
}
```

We have created three tests for **PriceCalculator**:

- `public function object_can_created()` : This test assures the object can be instantiated. Some may argue that this is unnecessary, but from a TDD point of view, we like to have such a simple test. When this test is passed, we can naturally move on to ones testing its real behaviour.
- `public function should_sum_price()` : This method tests whether `total` method does its job as described.
- `public function empty_items_should_return_zero()` : This method tests an edge case, where there is no item in the order. In such case, `total` method should return zero.

Now if we run `vendor/bin/phpunit` from terminal, we should get error as expected as below:

```
Fatal error: Class 'Dilab\Order\PriceCalculator' not found in tests/PriceCalculatorTest.p
```

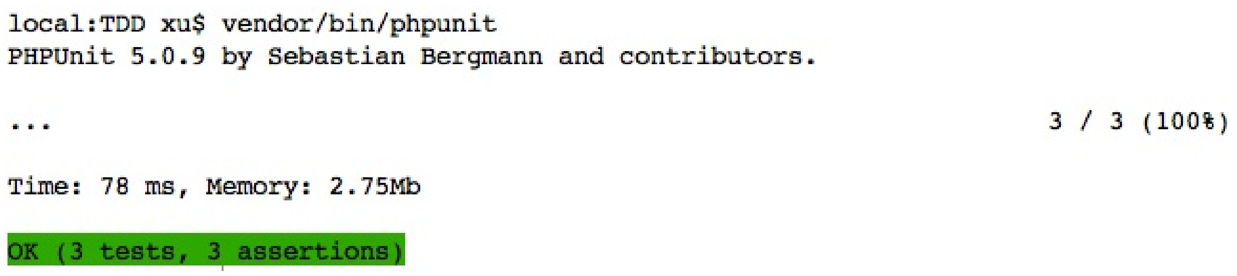
Green phase

The task of this phase is to make the failing tests above pass with the easiest but not necessarily the best code. The ultimate goal of this phase is the green message.

The implementation is fairly easy. All we need to do is to sum up the value with a `foreach` loop.

```
<?php

namespace Dilab\Order;

class PriceCalculator
{

    public function total($items)
    {
        $total = 0;
        foreach ($items as $item) {
            $total += $item['price'];
        }
        return $total;
    }

}
```

Now if we run `vendor/bin/phpunit` from terminal, we should get a green message as below:



```
local:TDD xu$ vendor/bin/phpunit
PHPUnit 5.0.9 by Sebastian Bergmann and contributors.

...                                                              3 / 3 (100%)

Time: 78 ms, Memory: 2.75Mb

OK (3 tests, 3 assertions)
```

Refactoring phase

This is the final phase of TDD, which we believe is the most valuable part of TDD. In this phase, we will take a look at the code we have written previously, and think of ways to make it cleaner and better.

We are using a `foreach` loop inside `total` method. It loops through `$items` array and returns the sum of the each individual element. This is actually a perfect use case of `array_reduce` method. Function `array_reduce` iteratively reduces the array to a single value using a callback function. Let's refactor our code by replacing `foreach` loop with `array_reduce`.

```
public function total($items)
{
    return array_reduce($items, function ($carry, $item) {
       return $carry + $item['price'];
    }, 0);
}
```

If we run our tests again and they all still pass, we are good to go. Because we need to run the tests constantly to make sure refactoring does not break anything, it is important to keep our code fast.

We have cleaned up our code from five lines to two lines. There is no more temporary variable. The method has become easier to debug. There might not be apparent benefits for doing so in this example, but imagine this in a large scale project, even cleaning up one line of code could potentially make development easier.

This is the end of TDD. To emphasize again, the spirit of TDD is to let tests drive our development. Using PHPUnit in a project does not necessarily make it a TDD driven project. It is the three phases processes involved in the development that make it TDD.