2
Object-oriented Programming with PHP

This chapter introduces the readers to the basic features of object-oriented programming with PHP and then provides an overview of the common design patterns. Later, we will go over how error handling and exception handling are performed in PHP. PHP has traditionally not been an **object-oriented programming** (**OOP**) language until PHP 5 when the language was revamped for a great deal to support the OOP features.

PHP in programming

PHP is a scripting language that is often used to build dynamic web applications. PHP inherits its programming style from C and Java. PHP comes with powerful libraries and strong community support, making it one of the favorite languages that developers use for building web applications. We will be utilizing the PHP libraries that were installed in *Bonus chapter 1*, *Installation of PHP, MariaDB, and Apache* to execute our scripts. Let us look at the three ways in which PHP scripts can be executed:

- Via the PHP shell
- Via the command line
- Using a web server such as Apache

The PHP shell is commonly used as a playground to test small scripts, and the shell can get tedious when working with bigger scripts. Executing files via command line is the second option where the PHP scripts will live inside the files. As we will be using object-oriented design to build our scripts, we will skip the first method and use the second and third methods to execute our scripts.

Object-oriented programming

Object-oriented programming is a popular programming paradigm where concepts are grouped into reusable objects that carry their own attributes and behaviors. An attribute can be described as a variable that is in the object, which is used to hold data pertaining to that object, while a behavior describes what an object can do. Let us consider the example of a `User` object; a user will have a name, an age, and an address, and these will be the attributes for a user. As the `User` object stores the address, we could have a behavior to allow and facilitate address retrieval; the behaviors are referred to as class methods. An object is a complex data structure that can have one or more types of attributes and one or more types of behaviors. The attributes and behaviors are defined in a class, and an object is an instance of a class. Therefore, the instances carry the same attributes and behaviors of that class. Though there could be multiple objects of the same class, the data stored in each object would be stored in different memory locations.

OOP is not a new concept and has been around for a very long period. OOP allows us to group our code based on behaviors and attributes and also allows us to organize our code for better reusability. Basic object-oriented features such as objects and classes were introduced into PHP 3 in 1998. In 2000, PHP 4 was released with a better support for object-oriented features but the implementation of objects was still an issue, as the object referencing was handled similar to value types. So a whole object would be copied when an object had to be passed in as a parameter to a function. As the object had to be copied, recreated, and stored several times, this led to poor scalability for big applications, extra usage of memory, and unnecessary overhead on the resources. PHP 5, which was shipped in 2004, arrived with a far superior object model that was better at handling objects, thereby increasing the performance and scalability of the web applications. Now that we have a basic idea about OOP, let us dive into understanding classes and objects.

[Sublime text is used as the text editor for this series. It is recommended to use a text editor or IDE of your choice. A few popular IDEs are Eclipse for PHP and the NetBeans IDE for PHP development.]

Classes and objects

In our first example, let us define a class. For creating a class, we would at least need one piece of information, the unique name for the class. In PHP, a class definition begins with the keyword `class`, and the keyword is followed by the unique name of the class. This is followed by a pair of curly braces and any class attributes and/or methods are enclosed into these curly braces.

There are two rules for naming a class in PHP mentioned as follows:

- It should begin with a letter or an underscore
- It can only contain letters, numbers, or underscores

It is a good practice to have the name of the class as part of the filename (for example, `class.Students.php`) or the actual filename itself (for example, `Students.php`). It is also common to use camel case, which is a common practice for naming the classes, class attributes, and class methods where multiple words are compounded into one, and the first letter of each word is written in upper case. Consider the following code snippet as an example showing a `Students` class:

```
<?php
    class Students
    {

        public function __construct()
            /**
            * Code to be executed
            *upon object instantiation
            */
        }
    }
?>
```

In this example, we are creating a `Students` class with a constructor using the `__construct` keyword. A constructor is the first function that is triggered upon the object instantiation. A constructor is commonly used for any bootstrapping purposes such as importing configurations and/or performing setup operations. The `__construct()` keyword is reserved by the PHP engine to identify the constructors in a class. Therefore, a constructor cannot be declared outside a class. The PHP engine treats a constructor like any other function; so, if a constructor is declared more than once, we will receive an error. Now that we understand how to create a class, let us continue with our `Students` class and add a few class attributes and class methods. After adding the attributes and methods, let us instantiate an object to access the attributes and methods. To instantiate an object, we will use the `new` keyword. We would need at least two pieces of information for the object instantiation; the first piece is the name of the object and the second is the name of class for which an object is being instantiated. Consider the following code example:

```
<?php
    class Students
    {
        public $first_name;
        public $last_name;
        public $address;

        public function __construct($first_name
          , $last_name, $address){

            $this->first_name = $first_name;
            $this->last_name = $last_name;
            $this->address = $address;
        }

        public function greeting(){
            return "Hello ".$this->first_name."\n";
        }

        public function getAddress(){
            return $this->address."\n";
        }
    }


    $student = new Students("John", "Doe", "3225 Woodland Park
      St");
    echo $student->greeting();
    echo $student->getAddress();
?>
```

In this example, we have added three class attributes or properties to store the first name, the last name, and the address of the student. We are initializing our properties via the constructor and the value for our properties will be passed in after the object instantiation. We have also added two class methods that would print out a greeting and return the address of the student. We are using the `$this` keyword to access our properties, and it can also be used to access the class methods as it is a reference to the `Students` object. We are using the `->` notation to access the properties or the methods of a class. After defining the class, we are instantiating an object of the `Students` class, and are calling the `$student` object. During the instantiation, we are passing in the arguments that are expected by our constructor, and these values are assigned to the properties. After the object instantiation, we are invoking the class methods using our `$student` object.

The output for the previous code will be as follows:

```
Hello John
3225 Woodland Park St
```

Static properties and methods

It is not always necessary to instantiate an object to access the properties or methods of a class. A class can also have static methods and properties that are bound to the class, rather than the object. To access a static method or a static property, we will use the scope resolution operator (`::`). To create a static property or static method, we will append the `static` keyword ahead of the variable. A static property or method will be commonly used to instantiate a database connection or a connection to a remote service that can add significant overhead. For a method to be accessible, the object of the class has to be created, during which a virtual method and member table are created for that class. While accessing static methods, we can avoid this overhead of creating a virtual method and member table for the class. Static methods are commonly used for high-performance systems.

We have successfully built our first class, instantiated the class, accessed the class methods, and discussed static methods and properties. During this process, we have already come across one of the four principles of OOP, which is abstraction. Abstraction is a concept about exposing the behavior and properties and hiding the particular code that performs that behavior. In our previous example, we have abstracted all the functionalities that a `student` object can have into the `Students` class, and have accessed those functionalities by creating an object of that class. The other three principles that we would look at are encapsulation, inheritance, and polymorphism.

Encapsulation

With abstraction, we have seen how to hide the underlying implementation that provides properties and methods. With encapsulation, let us see how we can expose a specific set of methods and properties, while hiding or restricting access to another set of properties and/or methods based on who is accessing this functionality. In our last example, we have used the keyword **public** while declaring the properties to define the access to those properties.

The access to the properties and methods in a class can be defined using **public**, **protected**, or **private** keywords as shown in the following table:

Visibility	Description	Comment
public	A public variable or a method can be accessed by anybody who can access the class.	All properties and methods are public by default.
protected	A protected variable or a method can only be accessed by the class members and the class members of a child class.	
private	A private variable or a method can only be accessed by the internal class members.	

Inheritance

Inheritance is commonly used when we want to create classes that would reuse the properties and/or class methods that are similar to existing classes. It is common to have this abstract high level functionality in a class that is referred to as the **base** or the **parent** class and then group this functionality into multiple different subclasses or child classes that would use properties or methods from that base class. In PHP, we use the `extends` keyword to inherit the functionality of a base class. There are at least two pieces of information that we would need for inheritance; the first is the parent class or base class and the second is the child class or subclass. In PHP, a child class can only have one parent class. Let's go over an example of inheritance where we create an `Animal` class and then inherit the functionality from the `Animal` base class to a `Dog` subclass. For this example, we will be housing these classes in different files, so the `Animal` class will go into `Animal.php` and the `Dog` subclass will go into `Dog.php`. Consider the following example where the `Animal` class will go into `Animal.php`:

```
<?php

class Animal{
    public $name;

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

    public function greet(){
        return "Hello ".$this->name."\n";
    }
}

?>
```

Consider the following example where the `Dog` subclass will go into `Dog.php`:

```
<?php

require('Animal.php');

class Dog extends Animal{
    public function run(){
        return $this->name." likes to run \n";
    }
}

$dog = new Dog("scooby");
echo $dog->greet();
echo $dog->run();

?>
```

In this example, we have two classes: the first class is the `Animal` class that has a public property `$name` and a public method `greet`. The value of `$name` can be set when an object is created for the `Animal` class or for a subclass that extends the `Animal` class. In the greet method, we are greeting the animal. In the `Dog` class, we begin by requiring the file that contains the `Animal` class in order to inherit from that class.

The `require` keyword is used to insert the content of a PHP file into another file and the PHP engine explicitly verifies that the content has only been added once.

After requiring the `Animal.php` file, we are creating the `Dog` subclass and are using the `extends` keyword to inherit the functionality of the `Animal` class. As dogs love to run, let us add a `run` method into the `Dog` class. After we have defined the `Dog` class, we will go forward and create an object of the `Dog` class and pass in a name to it. Though we have not declared and defined a constructor for the `Dog` class or the `greet` method in the `Dog` class, they would be available as we are inheriting the method from the `Animal` base class and both the constructor and the `greet` method are `public`.

The output for the previous code snippets is as follows:

```
Hello scooby
scooby likes to run
```

Magic methods

Before we move onto polymorphism, let us go over a few magic methods that are provided by PHP to perform operations such as property overloading, method overloading, and representing an object as text. These magic methods are collections of special methods that respond to an event. The magic methods follow a unique naming convention where the name is prepended by two underscores ("__"); one of the magic method that we have already worked with is the `__construct()` method that is automatically fired upon object instantiation. A few other magic methods that we would work with are `__destruct()`, `__get()`, `__set()`, `__toString()`, `__clone()`, and `__call()`.

It is recommended to refer to the official PHP documentation at `php.net` to understand and read more about these and more magic methods that are made available by PHP.

Constructors and destructors

We have already seen a couple of examples of how a constructor can be used to bootstrap an object upon instantiation. Constructors are commonly used to initialize properties and run any setup operations that are required to get the operations started. A destructor is called when the object is about to be destroyed. A destructor is used to perform clean-up operations such as destroying any open resource connections to the database or destroying an open file handle that has been created by the object. Consider the following example of `ConsDesc.php`:

```
<?php

    classConsDesc{
        /**
        * Constructor
        */
        public function __construct(){
            // initialize variables
            // log the time zone of the user
            // open database connections
            // open file handles
        }

        /**
        * Destructor
        */
        public function __destruct(){
            // close database connections
            // close file handles
        }
    }

?>
```

In this example, we are creating a class with a constructor and a destructor, and mentioning the common functionalities that can be part of these magic methods.

> In PHP, overloading refers to dynamic generation of a property or a method that has not yet been declared or is visible in the current scope.

Property overloading

Property overloading is performed when code attempts to access or set a missing property. For example, let us add a `$type` property with `protected` visibility to our `Animal` class. As the `$type` property is of a `protected` type, we will not be able to set a value after object instantiation. However, the value of the `$type` property should be available with read-only capabilities. To accomplish this task, we will use the `__get()` magic method provided by PHP as shown in the following example in `Animal.php`:

```
<?php

class Animal{
    public $name;
    protected $type;

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


    public function greet(){
        return "Hello ".$this->name."\n";
    }
}

?>
```

The only change that has been made to the `Animal` class that we built in the last section was to add the `$type` property with `protected` visibility. Now let us extend the `Animal` class and set the type based on the type of animal that we are creating the class for, as shown in the `Dog.php` file present in the code bundle:

```
<?php
require('Animal.php');

class Dog extends Animal{
    protected $type=__CLASS__;

    public function __get($property){
        if(property_exists($this, $property)){
            return $this->$property."\n";
        }
        else{
            return $property." does not exist \n";
        }
    }

    public function run(){
        return $this->name." likes to run \n";
    }

}

$dog = new Dog("scooby");
echo $dog->type;
echo $dog->greet();
echo $dog->run();

?>
```

In this example, we are extending the `Animal` class and setting the value of the `$type` protected property using `__CLASS__`, which is a magic constant. Then we are using the `__get()` magic method to return the value of `$type`. In our magic method, we are using a conditional statement to verify that the requested property exists within the scope of the current class. After checking to see if the property exists, we are gracefully allowing the execution to either return the value of the property on success, or display a message if the property does not exist.

The `__CLASS__` keyword is a magic constant that returns the name of the class it is in.

The output of the previous code snippets is as follows:

```
Dog
Hello scooby
scooby likes to run
```

We have looked at the `__get()` method that allows us to retrieve the value of a property; now let us look at the `__set()` method that allows us to change the value of a property. The setter magic method is commonly used to set values into an overloaded property, and take two arguments, the first being the name of the property and the second being the value to be assigned to that property. It is important to note that a setter magic method can expose protected and private class properties if the necessary checks are not performed. Let us build a `User` class that would store its overloaded properties into a `private` class property and let us see that the data can be populated via the setter method and retrieved via the getter method in `User.php` as shown in the following example:

```
<?php

    class User{
        private $data = array();

        public function __set($key, $value){
            $this->data[$key] = $value;
        }

        public function __get($key){
            if(array_key_exists($key, $this->data)){
                return $this->data[$key];
            }
        }
    }

    $user = new User();
    $user->first_name = "John";
    $user->last_name = "Doe";
    echo $user->first_name.' '.$user->last_name."\n";
?>
```

In this example, we are building a `User` class with a private class property. This private class property can store an array of dynamically declared properties; this type of dynamic property declaration is done while dealing with schema-less data architectures or architectures that support multiple schemas.

Here the `__get()` method is as important as the `__set()` method as if we try and access the first name and last name properties for the `$user` object without defining the `__get()` method, we would get an `Undefined Property` error.

The output of the previous code is as follows:

```
John Doe
```

Method overloading

PHP provides the `__call()` method to handle the concept of method overloading. This magic method is fired when the code attempts to call for methods that are either not accessible due to the scope or do not exist. This magic method can be either used to provide minimal access to such inaccessible methods or could be used to print out an error message gracefully. The `__call()` method takes two arguments, the first argument is the name of that function and the second argument could either be a single value or an array of values that have to be passed into that function.
Let us build a `MyMath` class with a `__call()` magic method in `MyMath.php` shown as follows:

```
<?php
    classMyMath{
        public $a=0;
        public $b=0;

        public function __construct($a, $b){
            $this->a = $a;
            $this->b = $b;
        }

        public function add(){
            return $this->a + $this->b."\n";
        }

        public function __call($name, $arguments){
            return "A function with name: ".$name." does not
              exist\n";
        }
    }

    $math = new MyMath(5,6);
    echo $math->add();
    echo $math->subtract();

?>
```

In this example, we are working with a class that takes two values upon object instantiation and returns the sum of those values when the `add()` method is called. This class does not have a `subtract` method but, when the `subtract` method is called by the object, the execution is gracefully passed into the `__call()` magic method. In this example, we are returning an error statement that says that the function does not exist. We could also use PHP's `call_user_func_array()` method to invoke any other functions that could serve this purpose; this is another example of code abstraction.

The output of the previous code is as follows:

```
11
A function with name: subtract does not exist
```

Representing an object as text

The last magic method that we will work with is the `__toString()` magic method, which is invoked when an object is treated like a string. In this case, the code tries to print out the object and at that point the PHP engine looks for an implementation of the `__toString()` method. If we do not have the `__toString()` magic method and if we print the object, we would receive a notice that would say that object could not be converted to a string. We will be going over this method while working on the factory design pattern.

Polymorphism

Polymorphism, as the name suggests, is a principle that allows different classes to have common methods that have the same name and signature but provide different functionality. It is a practice of sharing common programming functionality among classes in a single project. Let us take the example of a cat, a dog, and a duck. All of them are animals; cats and dogs belong to the family of mammals, while a duck belongs to the family of birds. Though all of them are animals, when representing them as objects, they share a few common features and those common features can live in the `Animal` base class. Let us consider a common feature such as talking or speaking, while a human can speak, a dog barks, a cat meows, and a duck quacks. So if the common feature is communication, the way communication is implemented among them is different. Let us use the concept of polymorphism to tackle this example and represent it in objects and classes in `Animal.php` as shown in the following example:

```
<?php

class Animal{
    public $name;
    protected $type;

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

    public function greet(){
        return "Hello ".$this->name."\n";
    }

    public function run(){
        return $this->name."  runs \n";
    }

    public function communicate(){
        return $this->name." says rrrrrr";
    }
}

?>
```

Consider the following example in the `Dog.php` file extending the `Animal` base class:

```
<?php
require('Animal.php');

class Dog extends Animal{
    protected $type=__CLASS__;

    public function __get($property){
        if(property_exists($this, $property)){
            return $this->$property."\n";
        }
        else{
            return $property." does not exist \n";
        }
    }

    public function run(){
        return $this->name." likes to run \n";
    }
    public function communicate(){
        return $this->name." says bow wow \n";
    }
}
```

```
$dog = new Dog("scooby");
echo $dog->type;
echo $dog->greet();
echo $dog->run();
echo $dog->communicate();

?>
```

Let us discuss this example before we look at our new `Cat` class. We have added the `run()` and `communicate()` methods to our `Animal` base class and are overriding these methods in the `Dog` subclass. My dog loves to run, so I am overriding the `run` method in the base class as it is too generic; by overriding this method in my subclass, I am making sure that my subclass functionality is implemented.

The output of the previous code snippet is as follows:

```
Dog
Hello scooby
scooby likes to run
scooby says bow wow
```

Upon execution, the name of the class is printed as our `__get()` magic method is fired. Then the `greet()` method is executed, and the outputs of the two methods that were overridden in the subclass are printed. Scooby really loves to run and is quite a talker. Now let us look at the implementation of these methods in our `Cat` class in `Cat.php` as shown in the following example:

```
<?php

require('Animal.php');

class Cat extends Animal{
    protected $type=__CLASS__;

    public function __get($property){
        if(property_exists($this, $property)){
            return $this->$property."\n";
        }
        else{
            return $property." does not exist \n";
        }
    }

    public function run(){
        return $this->name." hates to run \n";
    }
```

```
        public function communicate(){
            return $this->name." says meow \n";
        }
    }

    $cat = new Cat("cuddles");
    echo $cat->type;
    echo $cat->greet();
    echo $cat->run();
    echo $cat->communicate();

    ?>
```

Our `Cat` class is similar to the `Dog` class, except for the implementation of the `run()` and `communicate()` method. We are using the `greet()` method from the `Animal` base class. Let us execute this code and examine the output shown as follows:

```
Cat
Hello cuddles
cuddles hates to run
cuddles says meow
```

Upon execution, the name of the class is fired by our getter method and the `greet()` method is fired next. After the `greet()` method, the overridden implementations of the `run()` and `communicate()` method are fired. Unlike Scooby, Cuddles is very lazy and hates to run.

Interfaces

In our previous example, we noticed that the `run()` method and the `communicate()` method are not being used by the `Animal` class, but are only present in that class so the subclasses can inherit those methods and override them. As we start building bigger applications, we will be dealing with different types of data with different types of functionality. Though the underlying functionality is different, the high-level description and aim could still be the same, as in case of `run` and `communicate`. So, using multiple names for the same functionality would lead to confusion and inconsistency. To avoid this confusion, we can utilize the object interfaces provided by PHP 5. Unlike a regular class, an object interface would only specify the methods that the class must implement, but doesn't provide a specific implementation for those methods, unlike the `Animal` class in the previous example. All methods in an interface are `public` by default. Let us move the `run()` method and the `communicate()`method into the interface as the `Animal` class is not the apt location for those methods.

Creation of an interface would at least need two pieces of information, the first one being a unique name of the interface and the second the signature of at least one method. It is a common convention to prefix the names of interfaces with the letter "I" in uppercase. The interface, like a class, will be stored in a separate file with the same name as the interface. Let us build our first interface and see how we cascade the changes into the rest of our files. To implement an interface, we use the `implements` keyword followed by the name of the interface in `IAnimal.php` shown as follows:

```
<?php

    interfaceIAnimal{
        function run();
        function communicate();
    }

?>
```

Now that we have moved the `run()` method and the `communicate()` method into our interface into our `Animal` class, which is implementing the interface, those methods have to be implemented in `Animal.php` as shown in the following example:

```
<?php
require_once('IAnimal.php');

class Animal implements IAnimal{
    public $name;
    protected $type;

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

    public function greet(){
        return "Hello ".$this->name."\n";
    }
    public function run(){
        return $this->name." likes to run \n";
    }

    public function communicate(){
        return $this->name." says bow wow \n";
    }

}

?>
```

Now let us extend the `Animal` class in the `Dog` class in `Dog.php` as shown in the following example:

```
<?php
require('Animal.php');

class Dog extends Animal{
    protected $type=__CLASS__;

    public function __get($property){
        if(property_exists($this, $property)){
            return $this->$property."\n";
        }
        else{
            return $property." does not exist \n";
        }
    }

}

$dog = new Dog("scooby");
echo $dog->type;
echo $dog->greet();
echo $dog->run();
echo $dog->communicate();

?>
```

We begin by requiring the file that carries the interface; once the snippet is ingested into our `Dog.php` file, the interface can be implemented by using the `implements` keyword.

Abstract classes

One thing that we have noticed here is that we are not creating an object for the `Animal` class, and are only using this class to house the definitions of common functionality among animals. PHP 5 arrives with a concept of abstract classes. Abstract classes cannot be instantiated and should only be used to provide functional direction and behavior to the subclasses. To create an abstract class, add the keyword `abstract` before the `class` keyword in `Animal.php` as shown in the following example:

```
<?php

abstract class Animal{
    public $name;
```

```
        protected $type;

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

        public function greet(){
            return "Hello ".$this->name."\n";
        }
    }

    ?>
```

The output would still remain the same as before but one thing that we have noticed is that our code looks a lot cleaner, consistent, and scalable. If we add a new animal, we would have to extend the `Animal` abstract class, implement the `IAnimal` interface, and add any custom functionality that the new animal requires. Now that we are at a good place with the OOP concepts, let us look at a few popular design patterns.

Design patterns

Design patterns are generalized solutions to common problems that have already been solved by programmers. These patterns are reusable solutions that have been built by software development teams and the community of software developers. Design patterns are often agnostic to the programming languages as they focus on the problem, rather than the implementation of the solution. The popular design patterns that we will look at are the **factory** pattern and the **singleton** pattern.

The factory pattern

The factory pattern is one of the most commonly used design patterns. As the name suggests, there will be a class that is dedicated just to the generation of objects. Rather than directly creating an object, we will use the factory class to create objects for a class. This is done in order to minimize the amount of changes that have to be made across the project due to a change in the class for which the factory is being used. Let us look at the following example in `DesignPattern-Factory.php` to simplify this:

```
    <?php
        class Car{
            private $make;
            private $model;
            public function __construct($make, $model){
                $this->make = $make;
                $this->model = $model;
            }

            public function __toString(){
                return "The make is ".$this->make." and the model is
    ".$this->model." \n";
            }
        }

        classCarFactory{
            public static function create($make, $model){
                return new Car($make, $model);
            }
        }
        $car = CarFactory::create("Audi", "Q5");
        echo $car;
    ?>
```

In this example, we begin by building the `Car` class that has `$make` and `$model` as `private` properties. The values for these properties will be assigned upon the object's instantiation. We are using the `__toString()` magic method to print out the make and the model of the `car` object that is being built. After the `Car` class, we have the `CarFactory` class that has the `static` function that creates and returns the `Car` object.

The output of the previous code is as follows:

```
The make is Audi and the model is Q5
```

The singleton pattern

The singleton pattern is commonly used while establishing a connection to the database or while working with a remote service, as they can add significant overhead that affect the operations of the application. In a singleton pattern, the instantiation is restricted to a single instance, thereby avoiding overhead of multiple instances. We will need at least three pieces of information to use the singleton pattern: the first is a unique name for the class, the second piece would be a private constructor that would free us from creating multiple objects, and the third being a static method that would return the instance. Consider the following code in `DesignPattern-Singleton.php`:

```
<?php
    class DB{
        private static $singleton;

        private function __construct(){}

        public static getInstance(){
            if(self::$singleton){
                self::$singleton = new DB();
            }

            return self::$singleton;
        }

    }

    $db = new DB::getInstance();
?>
```

In this example, we are using the singleton pattern to restrict object instantiation by using a private constructor. We are using the static `getInstance()` method to instantiate an object to the `DB` class. As this is a static method, it is bound to the class.

> This is not a full-fledged example and this should only be used as a high-level concept. We will be adding the connection parameters to this class in *Chapter 3*, *Advanced Programming with PHP*, while working with the MariaDB database.

Error handling

Error handling is a very important part of the software development life cycle. Errors can be briefly divided into three categories:

- Syntax errors
- Runtime or external errors
- Logic errors

A syntax error could be caused due to a missing semicolon or a missing closing brace. A runtime or an external error could be caused due to a missing file that has been added to our script using `require` or `require_once`, a broken file handle, or a broken database connection. At runtime, if the execution is expecting a specific set of resources and if they are not provided, we would receive runtime errors.

Logic errors are caused due to wrong interpretations of requirements or just faulty code. They are referred to as bugs, and are common part of the software development life cycle. PHP is shipped with a built-in support for error handling, and groups these errors into different severity levels. It is a common practice to log the errors that are generated, and based on the severity of the error, it is also advised to send notifications as required. PHP's core settings are located in the `php.ini` file that is located in the server configurations folder, a constant name is assigned to each of the severity levels. Let us look at the different levels of errors that are provided by PHP as shown in the following table:

Constant	**Description**
`E_ERROR`	These are fatal runtime errors and the execution cannot be recovered from these errors. Until the error is not fixed, this error will not subside. For example, a script is expecting a connection to a database that does not exist.
`E_WARNING`	These are non-fatal runtime warnings that do not halt the execution. For example, a file that has been included via `include` or `include_once` does not exist.
`E_PARSE`	These are fatal errors that occur when a script cannot be parsed due to a syntax error. For example, the script is missing a semicolon or an apostrophe.
`E_NOTICE`	These are nonfatal notices that are encountered by the script.
`E_DEPRECATED`	These are nonfatal notices that are provided by PHP to warn about code that will not be available in future versions.
`E_STRICT`	These are runtime notices that suggest changes that will ensure the best interoperability and forward compatibility with future versions of PHP.
`E_ALL`	These are all errors and warnings supported by PHP.
`E_CORE_ERROR`	These are fatal run-time errors that are generated in the PHP core engine.
`E_COMPILE_ERROR`	These are fatal compile-time errors generated by the PHP core engine.

These are the common types of errors, warnings, and notices that we come across while scripting with PHP. In order to display them during execution, we will have to update the `php.ini` file as shown in the following table:

Setting	**Description**	**Note**
`error_reporting`	This setting controls the types of errors, warnings, and notices that have been triggered.	`E_ALL`, `~E_DEPRECATED` and `~E_STRICT` are the recommended settings for production.
`display_errors`	This setting controls whether or not to display errors to the screen.	The `display_errors` setting should only be turned on in the development and testing environments. This has to be turned off for staging and production environments.
`log_errors`	This setting controls whether the errors are logged or not.	It is recommended to turn this setting on in all environments.
`report_memleaks`	This setting helps in tracking the memory leaks in the application.	This setting will need `E_WARNING` to be included in error reporting.
`html_errors`	This setting displays the error with a better HTML format making it easier to read.	It is recommended to turn this setting on in all environments.

PHP also contains the `trigger_error()` function that allows the user to fire a custom error from a script. There are three levels of custom errors that can be generated by using the `trigger_error()` function:

- `E_USER_NOTICE`
- `E_USER_WARNING`
- `E_USER_ERROR`

Consider the following code in `Trigger_error.php`:

```
<?php
    $value = 0;

    if($value>0){
        while($value < 10){
            echo $value;
            $value++;
        }
    }
    else{
        trigger_error("Value is not greater than 0");
    }

?>
```

In this example, we are initializing the `$value` variable to `0`, and are checking to see if the value is greater than zero. This condition will certainly fail, as this is an example of a logic error. In our `else` block, we are using the `trigger_error()` method to fire a PHP notice to indicate this logic error.

The output of the previous code is as follows:

```
PHP Notice:  Value is not greater than 0 in
  /var/www/chapter4/Trigger_error.php on line 12
```

Let us look at an alternative for not printing out warnings, notices, and errors. PHP provides the error suppression operator (`@`) to hide any warnings, errors, or notices that will be printed out on the page. This is not a recommended practice and should only be used to avoid printing them onto the page, as shown in the following code in `error_suppression.php`:

```
<?php
include("fileDoesNotExist");

function add($a, $b){
    return $a+$b."\n";
}

echo add(5,4);
?>
```

When this script is executed, a PHP warning would be fired as the `include` function cannot locate the file; however, as it is just a warning, the execution will continue and the output will be the value returned by the `add` function.

The warning generated will be:

```
PHP Warning:  include(fileDoesNotExist): failed to open stream: No
such file or directory in /var/www/chapter4/error_suppression.php on
line 3
PHP Warning:  include(): Failed opening 'fileDoesNotExist' for
inclusion (include_path='.:/usr/share/php:/usr/share/pear') in /var/
www/chapter4/error_suppression.php on line 3
```

The output generated will be:

```
9
```

Now let us prepend the suppression operator to the include statement as shown in the following code in `error_suppression.php`:

```
<?php

@include("fileDoesNotExist");

function add($a, $b){
    return $a+$b."\n";
}

echo add(5,4);

?>
```

After we have added the suppression operator, the script executes smoothly and the warning is not fired onto the output shown as follows:

```
9
```

Summary

In this chapter, we have gone through the concepts of Object-oriented programming such as classes and objects, abstraction, encapsulation, inheritance, magic methods, polymorphism, interfaces, and abstract classes. We have also covered the basics of design patterns and error handling.

In the next chapter, we will go over the new features that are part of PHP 5.4 and 5.5; we will focus on another aspect of error handling, exception handling, and how PHP 5.5 has made exception handling better. Then we will continue to focus on avoiding errors and exceptions by setting up unit tests for our code.