**The Object Oriented Approach**

In programming, a thing that is describable and has a certain set of actions can be referred to as an object. An object might represent a real-life entity with a certain number of actions to perform. A dog can be described by using certain states, such as color, breed, age, and so on, and performs certain actions, such as barking, running, wagging its tail, and so on. A table fan can be described by color, speed, direction, and so on, and perform actions such as changing speed, changing direction, rotating, and so on.

In OOP, data and code are bundled together into an entity, which is known as an object. Objects interact with each other. Consider a teacher object and a student object. The teacher might have certain subjects to offer and the student might enroll in these subjects. Hence, if we consider enrolling as an action of the student, then the student object might need to interact with the teacher object regarding the available subjects and register for one or more subjects. Simply put, an object is data that performs actions.

Bundling code into objects has its own benefits, such as your code base becoming modular, which means you can maintain, reuse, and debug your code individually against objects. The implementation of an object (code) remains hidden from the outside world, which means we can hide our data and internal complexities and can interact with the object via a standard set of procedures. For example, in order to use a table fan, you don't need to learn about AC motors or electronic circuitry; rather, you can use the table fan via the provided actions, such as the speed control buttons or rotation control. Hence, hiding such information is another important aspect of OOP.

Such code bundling also differentiates OOP from procedural programming. An object simply contains attributes, also known as data, and a bunch of methods to communicate with that object. These methods are the functions of procedural programming. In OOP, some of these methods can be used to interact with that object, and these methods therefore make up its interface.

There are a good number of famous programming languages, such as C++, Java, PHP, Python, C#, JavaScript, Ruby, Dart, Swift, Objective-C, and so on, that support OOP. Since the introduction of PHP to its most recent version, PHP supports the complete set of object-oriented models. PHP supports class-based object initiation, constructors and destructors, inheritance, property visibility, polymorphism, abstract and final classes, static fields and methods, anonymous classes, interfaces, namespaces, magic methods, object cloning, object comparisons, type hinting, traits, and much more interesting OOP techniques and tools. We will be discussing them in this chapter and will practice the concepts of OOP using different examples.

**OOP Concepts**

The object-oriented approach addresses programming problems using the generalized concepts given in the following list. In this chapter, we are going to discuss these concepts in detail and practice them using a number of exercises so that, by the end of the chapter, we'll be used to working with these concepts:

* Objects are entities with data and interfaces. They may represent a person, a vehicle, a table fan, or maybe a bank account that plays a role in our program. Data and functions (or methods) live together inside an object.
* Classes are templates for object creation. Data is the description of an object, while functions are the behaviors of that object, so such definitions of data and methods can be written using a class. Classes can be referred to as custom data types.
* Data encapsulation is the wrapping up of data and functions into a single unit – that is, a class. Imagine an unbreachable capsule with data and functions encapsulated inside so that the outside world cannot access the data as long as we don't expose methods for them. Such insulation of the data from direct access by the program is called data hiding. In short, declaring a class is the encapsulation of data.
* Data abstraction is the act of representing essential properties and features without giving details. So, the entire entity description remains abstract and the responsibility of detailing the entity can be done via the entity creation process or inheritance. Such abstraction enables everyone to "follow the guidelines and do it your way."
* Inheritance is the process of acquiring properties and behaviors of another class so that common properties and behaviors can be reused in a hierarchical manner.
* Polymorphism is the concept of using the same definition for multiple purposes. For example, flying is a polymorphic behavior, as birds and airplanes have their own different ways of flying.
* Dynamic binding is the linking of a function call to the code that will be executed in response to the function call. With this concept, the code associated with the given function is unknown until the call is made at runtime. Say that multiple objects implemented the same function differently and at runtime, the code matching the object being referenced would be called.
* Message passing is the way that objects interact with each other. It involves specifying the object name, the name of the methods, and the information to be sent. For example, if a car is an object, changing speed is a method on it, and speed in kilometers per hour is the speed parameter to be passed. The outside world will use the car object to send the "change speed" message to that parameter. The following figure depicts the preceding concepts using a vehicle analogy:

A close up of a map

Description automatically generated**Figure 5.3: Vehicle property inheritance diagram**

There are many different types of vehicles, such as cars, buses, motorcycles, airplanes, and many more. Vehicles have general properties such as make, model, color, wheels, engine size, and so on. These are the common properties found in vehicle subtypes or classes too. Since cars, buses, motorcycles, and so on share a common list of properties, those common properties and behaviors come from the parent class, and each subclass adds its very own properties and behaviors. For example, cars have four wheels and motorcycles are two-wheelers, cars have more passenger capacity than a motorcycle, and so on. Therefore, such deviations of vehicle types should be placed into their own vehicle subclasses. Thus, we can inherit common properties and gradually add our own properties using object-oriented concepts.

**Classes**

A class is a blueprint of an object. What data an object should contain and what methods are needed to access that data can be described using a class. A class acts as a template for object creation. Consider a car designed using a blueprint as a guide. Vehicle type, make, model, engine size, color, and so on are defined in the **Car** class along with the methods to retrieve this information, such as get the model name, start the engine, and so on.

A class begins with the **class** keyword followed by the given name and the body enclosed in a pair of curly braces. The body of the class houses class members and they are variables, constants, functions, class variables (also known as class properties or class attributes), and the functions that belong to the class, known as class methods.

Check out the following class declaration:

class ClassName

{

// Class body

}

//or

class ClassName

{

// Class variables declarations

// Class methods declarations

}

A class name starts with letters or underscores followed by any number of alphanumeric characters and underscores. PHP's predefined class names, constants, and reserved keywords – for example, **break**, **else**, **function**, **for**, **new**, and so on – cannot be used as a class name.

This is the list of [reserved words](https://www.php.net/manual/en/reserved.php) in PHP.

In the PHP Standards Recommendations, PSR-1 recommends that a class name is declared in **CamelizedClassName** and class methods are declared in **camelizedMethodName**. Note the camel case of *class name* and the use of lowercase at the start of the method names.Click [PSR-1: Basic Coding Standard](https://www.php-fig.org/psr/psr-1/) for more information.

Let's check out the following simple **Person** class:

class Person

{

public $name = 'John Doe';

function sayHello()

{

echo 'Hello!';

}

}

Here, **class Person {…}** is the **Person** class declaration. A single attribute has been added with the line **public $name = 'John Doe';**, and the body also contains the **sayHello()** member method, which prints a simple string.

In the next section, we will be discussing how we should instantiate a class and what happens in memory when we perform such an instantiation.

**Instantiating a Class**

An object is an instance of a class, so instantiating a class means creating a new object using the class. We can instantiate a class using the **new** keyword, as follows:

$object = new MySimpleClass();

With the instantiation, an object is created in memory with copies of its own attributes. Here, the **$object** variable doesn't hold the actual object; rather, it points to the object. Just to be clear here, the **$object** variable is a pointer to the object and doesn't hold a reference to the object.

The **$object** variable should be of the **MySimpleClass** type as classes are often called as custom data types. Then, the constructor method gets called automatically if one is declared. A class constructor and destructor are two special kinds of methods; for example, **\_\_construct()** and **\_\_destruct()**, which are called automatically with object creation and deletion, respectively.

To access an object's properties and methods, we can use the **->** object operator, as in the following:

$object->propertyName;

$object->methodName();

So, object creation involves memory allocation followed by the constructor method being called automatically. We are going to discuss constructor and destructor methods in later sections.

**Class Attributes**

As we have already seen, class attributes and variables hold data. To write a class attribute in PHP, we need to start with the **public**, **private**, or **protected** keyword, then the rest is the general PHP variable assignment statement. In the previous example, in the **Person** class, the **public $name = 'John Doe';** line was used to assign a person's name; here, the **public** keyword is an access modifier or class member visibility keyword and it has been used so that the attribute can be accessed outside of the class. We will be discussing access modifiers in detail in later sections.

Note that the class structure is compiled before the PHP file execution. Regarding value assignment in class attributes, the value should be static, meaning the value must not be dependent on the runtime. For example, the following class attributes won't work:

public $date = getdate();public $sum = $a + $b;

Here, the attributes are dependent on the **getdate()** function's return and an arithmetic expression evaluation, respectively, as function calling and the arithmetic expression evaluation won't be performed during the class' compile time and can be evaluated at runtime, so such variable initialization won't work in the case of class attributes.

So, class attributes that do not involve in runtime information should be considered a good attribute, such as the following:

public $num = 10

public $str = 'I am a String';

public $arr = array('Apple', 'Mango', 'Banana');

Here, the preceding variables can be evaluated at compile time rather than at runtime.

Non-static class attributes – for example, the **public**, **private**, and **protected** attributes – can be accessed by using the **$this** object context referrer variable with the **->** object operator, as in the following:

class Person

{

  public $name = 'John Doe';

function printName()

{

echo $this->name;

}

}

Also, static properties can be written with the **static** keyword at the start of the variable declaration and can be accessed using the **self** keyword followed by the **::** (double colon) operator. The double colon is also called the scope operator:

class Person

{

public static $name = 'John Doe';

function printName()

{

echo self::$name;

}

}

More on access modifiers and static properties can be found in later sections.

**Class Constants**

Class-specific constants (fixed values that do not change throughout the program) can be written inside a class, as in the following examples:

class SampleClass

{

const ONE = 1;

const NAME = 'John Doe';

}

echo SampleClass::ONE; //1

echo SampleClass::NAME; //John Doe

Note that class constants do not use **$** as it is used in variable declaration and are all the letters are in uppercase. The default visibility of the constant is **public** and they can be accessed with the **::** scope operator from outside of the class.

**Note:** According to the PHP Standards Recommendations, PSR-1, "Class constants MUST be declared in all upper case with underscore separators." Click [PSR-1](https://www.php-fig.org/psr/psr-1/) for more information.

Class constants are allocated memory for a single class and not for every class instance.

Also, you can use such constants using **self::** inside a class, as in the following:

class SampleClass

{

const ONE = 1;

const NAME = 'John Doe';

function printName()

{

echo self::NAME;

}

}

echo SampleClass::NAME; //John Doe

The **self::** operator can be used only inside a class. Since PHP 5.6.0, the constant expression has been added as in the following:

class SampleClass

{

const ONE = 1;

const SUM = self::ONE + 2;

}

echo SampleClass::SUM;//3

Class constants also support access modifiers; for example, **public**, **private**, and so on, which will be demonstrated in the *Access Modifiers* section.

You can use such constants in PHP interfaces, which are another OOP tool to establish a common interface or the standards that classes should implement.

**The $this Variable**

**$this** is the pseudo variable that is available when class member variables or methods are called within an object context. **$this** works when we have instantiated a class and can be used to access the corresponding object's members. So, to access an attribute in an object context, we use **$this->attribute\_name**, and to access a method, we use **$this->methodName()**.

**Note:** For example, a **$name** attribute declared in the class should be accessed with **$this->name**, not with **$this->$name**. Mind the **$** here.

**Class Methods**

Class methods are just functions and act like wrappers on the class data assigned to attributes. Getter and setter are the two most common ways of fetching and assigning data, respectively. Both of these methods simply return and assign data from and to member variables. We might want to prefix the getter and setter methods with **get** and **set** followed by a quick descriptive method name of our choice; for example, **getMyValue()** or **setMyValue()**. Although this is not necessary, this practice improves code readability.

Check out the following getter and setter methods example:

class Person

{

public $name;

function getName()

  {

return $this->name;

}

function setName()

{

$this->name = 'John Doe';

}

}

Here, the key concept of such member methods is to provide a wrapper around the data available in an object.

Along with these, another type of method can often be used that performs certain actions or executions based on the available data within the object:

class Person

{

 public $name;

function getName()

{

return $this->name;

}

function setName()

{

$this->name = 'John Doe';

}

function sayGreetings()

{

if (date('G') < 12)

{

$greetings = 'Good Morning';

}

elseif (date('G') < 17)

{

$greetings = 'Good Afternoon';

}

else

{

$greetings = 'Good Evening';

}

echo "$greetings $this->name! ";

}

}

Here, the **sayGreetings()** method could be an example of a member method that implements an algorithm to identify the current hour and load a greetings string into a local variable, and later on prints the greetings string with the given attribute value assigned at **$name**. The method works for printing greetings – for example, '**Good Morning John Doe**', '**Good Afternoon John Doe**', and '**Good Evening John Doe**' – based on the current hour in 24-hour format, returned by the **date('G')** function.

We also have some manager methods, such as constructor and destructor, to initiate properties of an object and clean up the memory utilized by an object, respectively. In later sections, we will be discussing them in detail.