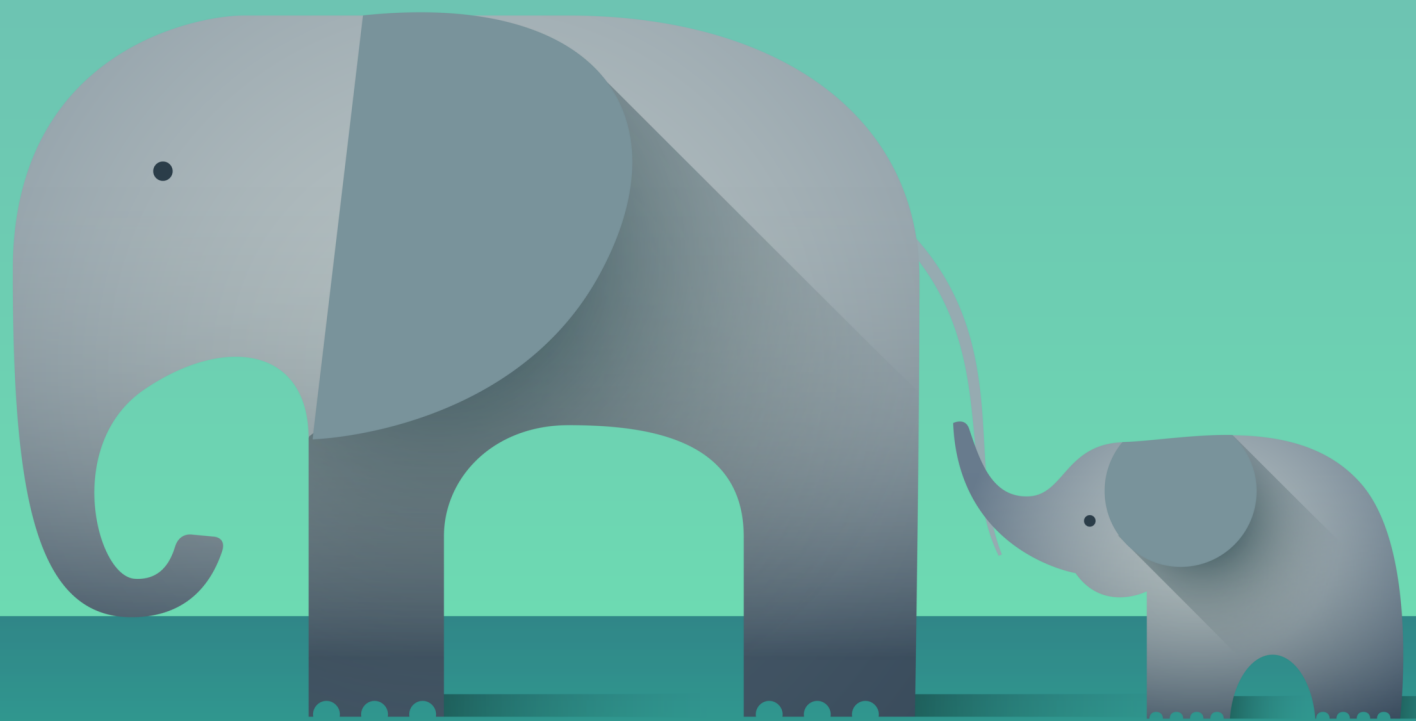# The essentials of object oriented PHP

Learn and practice

Joseph Benharosh

The essentials of Object Oriented PHP

Learn, practice, and apply

Joseph Benharosh

To my parents, Rachel & Zion Benharosh.

Contents

CONTENTS

CONTENTS

Introduction

Why bother learning object oriented PHP?

Object oriented programming came late to PHP. It has been around in other languages, like C++, Ruby, Python, and JavaScript for much longer, but its demand in PHP is becoming stronger. With that demand for the programming style there is an increasing demand for developers who can program in object oriented PHP. And those developers typically earn 60% more than programmers who know only procedural PHP.

Why?

That is a valid question. After all, most PHP code is used in WordPress, and WordPress is not written with object oriented code at its core. In addition, there are great examples of applications that are written in a procedural style of PHP. But it does not tell the full story. Consider this:

- Procedural programming is inefficient.
- PHP frameworks, like Laravel, Symfony, and CodeIgniter rely exclusively on object oriented PHP.
- CMS systems, like Drupal 8 and October are object oriented at their core.
- Every day, additional parts of WordPress are becoming object oriented.

Let's expand on the first point as it is the most important one - everything else flows from it, i.e. object oriented programming is gaining popularity because it is more efficient. As already stated, there are great applications written in a procedural style of PHP code. But even the best end up in spaghetti code where functions and variables are mixed together. Add in HTML, CSS, and JavaScript to the mix and the code becomes something that is, well, inefficient.

Object oriented code is more organized. Distinct units are created for a single purpose. This means it is easier to find things in the code (ending those days of spending hours trawling through lines of spaghetti code to find a single element). It is also easier to add, replace, or remove parts of code, as it only has to be done once. With procedural programming, on the other hand, it is necessary to change every instance.

Finally, object oriented programming gives the opportunity to developers to integrate code libraries from other resources without worrying about name collisions, even though some of the functions in the code might have the same name.

So, back to the original question, why bother learning object oriented PHP? The answer is:

- It is increasingly common.
- Developers with this skill earn more.
- The code they write is better.

Programming in object oriented PHP requires a different way of thinking. That is why it is important to get an understanding of the fundamentals.

The purpose of this book

The purpose of this book is to unlock the secrets of object oriented PHP. With these secrets programmers can improve their skills and reap the rewards and benefits of being able to program this way.

At the heart of all of this is **the principle of modularity**. This principle is what makes object oriented PHP possible. It essentially means separating the code of an application into much smaller and easier to use parts. It makes the code more flexible, as well as easier to change, update, debug, and develop.

It also saves huge amounts of time. With modularity, programmers can change the code of an application simply by working on the specific, relevant sections, instead of looking at the whole thing. Adding, removing, or changing functions is therefore quicker.

Time is also saved because modularity allows the work of other programmers to be easily incorporated into a project. This means developers can use tried and tested functions written elsewhere, and they can put them into the code without heavy adaptation.

Finally, modularity saves time because it better facilitates multiple people working on a single project. Without it, a small change to the code from one programmer can have repercussions across many workstations, but modularity solves this.

Take a blog as an example. This is a simple application but it might still have several classes. This can include a user class, a database class, a posts class, and a mail class. With modularity and object oriented PHP, each of these classes has its own section in the code. This section can be handled independently of the others. This means it is possible to add, remove, or replace parts of the code without affecting everything else. In the blog example, it is possible to change the mail class by replacing it with another library. With modularity, this can be done even though the user class depends on the mail class for sending emails to users of the blog. Each of the classes is treated separately.

This is why almost all big websites on the Internet use object oriented programming - it saves time and many headaches.

The benefit of the ability to more efficiently work with multiple programmers on a single project should not be understated either. This again saves time, as well as headaches, errors, and arguments

among programmers. In particular, there is no risk of name collisions. This is where more than one function or variable in the program has the same name. Modularity means that each of these functions or variables resides in an independent module, so name collisions are not an issue.

So, the purpose of this book is to instill in programmers the principle of modularity. Everything is based on this from classes to namespaces to methods to code libraries. Modularity is a way of organizing code, and it gives object oriented PHP most of its power.

What does the book cover?

This book covers the fundamentals of programming in object oriented PHP. This starts with the basics of writing in an object oriented style. That style applies to everything, including classes, objects, methods, and properties. It also covers how code in a class can be approached from within the class and, crucially, from outside it.

For example:

- Creating classes
- Adding properties to a class
- Creating objects from a class
- Setting an object's properties
- Adding methods to a class

The book also shows developers how to write streaming code that is easier to digest and understand because it more closely resembles human language.

In addition, the book covers a number of other skills, hints, tips, and tricks to make object oriented programming more efficient. This includes:

- Protecting code inside classes from the outside
- Object inheritance and how it reduces repetition, therefore saving time
- Abstracting classes and methods so that code is organized better
- Other code organizing tips
- Constructing applications using interfaces and polymorphism
- Integrating code from other sources and libraries with the help of namespaces

And it covers the intimidating concept of dependency injection in a practical and easy-to-understand way. Once finished with the dependency injection chapter, object oriented programming becomes even faster.

Who is this book for?

Developers who want to achieve the objectives stated earlier will benefit from this book, i.e. those who want to develop a new skill that is in increasing demand in order to become more employable, and to write better code.

But this book is not for everyone. For a start, it is not for experienced, object oriented PHP developers. This book covers the fundamentals, and it is not intended as a development resource for those who already code in this way.

Similarly, it is not for those with little or no knowledge of programming. It covers the basics of object oriented programming, but it is not a beginner's guide. There is an expectation that readers will have a good foundation of programming knowledge before starting.

There are a couple of exceptions to this, though. This includes those whose programming foundation is based in HTML and CSS. That is not enough programming knowledge to properly benefit from this book.

So, who is it for then?

Obviously, it is for procedural PHP developers who want to improve their skills and take their programming development to a new level. It is also for programmers who have a very basic understanding of object oriented PHP, but who want to learn it properly and get more practice.

Others who will benefit are those programmers who have a background in other objected oriented languages like C++ and Java.

So, in summary, the book is for existing programmers who want to learn object oriented PHP as a new skill.

Getting the most from this book

As already established, this book is designed for existing programmers who want to expand their skill set with object oriented PHP. As a result, it is written in a learn-then-practice form.

Every principle, idea, or process is described in a simple and concise way. It is easy to read, as there is no superfluous text. The flow is also structured so that the reader develops hooks to hang additional knowledge on as their understanding grows and develops.

After the concise description of concepts, code examples are given to bring those concepts to life. This is where proper understanding develops as practical applications are demonstrated.

Throughout the text, practical example questions are asked of the reader. And that brings us to the first tip for reading this book - read it actively, not passively.

It is not a passive learning tool. This book is designed to give developers knowledge and skills in object oriented PHP quickly. That is achieved through active learning where programmers think for themselves while reading the explanations and examples.

This principle of active learning is carried through to the next stage, which is practice. Readers are given the opportunity to practice the ideas, concepts, and processes that have just been explained. It is important to the learning process that time is given to these practice elements, so this is the next tip - serious learning cannot be done without practice.

Finally, solutions are given for the practice exercises so that the reader can see their success and gain confidence.

Each principle of object oriented PHP is taught using this process:

- Concise explanations and learning
- Simple but practical examples
- Questions to encourage independent thinking and active learning
- Practice exercises
- Solutions to those practice exercises

The punchy text is easy to digest so that readers can go through sections quickly if they want. However, there is also enough detailed information and examples if the reader needs to spend more time getting an understanding of a particular concept.

So, to get the most from this book, approach it with an open mind and be ready to get involved. By doing so, you will get a good understanding of object oriented PHP with sufficient knowledge and confidence to apply it practically in your work.

How to create classes and objects?

Object-oriented programming is a programming style in which it is customary to group all of the variables and functions of a particular topic into a single class. Object-oriented programming is considered to be more advanced and efficient than the procedural style of programming. This efficiency stems from the fact that it supports better code organization, provides modularity, and reduces the need to repeat ourselves. That being said, we may still prefer the procedural style in small and simple projects. However, as our projects grow in complexity, we are better off using the object oriented style.

With this chapter, we are going to take our first steps into the world of object oriented PHP by learning the most basic terms in the field:

- **class**
- **object**
- **method**
- **property**

How to create classes?

In order to create a **class**, we group the code that handles a certain topic into one place. For example, we can group all of the code that handles the users of a blog into one class, all of the code that is involved with the publication of the posts in the blog into a second class, and all the code that is devoted to comments into a third class.

To name the class, it is customary to use a singular noun that starts with a capital letter. For example, we can group a code that handles users into a `User` class, the code that handles posts into a `Post` class, and the code that is devoted to comments into a `Comment` class.

For the example given below, we are going to create a `Car` class into which we will group all of the code which has something to do with cars.

```
class Car {
    // The code
}
```

- We declare the class with the `class` keyword.
- We write the name of the class and capitalize the first letter.
- If the class name contains more than one word, we capitalize each word. This is known as **upper camel case**. For example, `JapaneseCars`, `AmericanIdol`, `EuropeTour`, etc.
- We circle the class body within curly braces. We put our code within the curly braces.

How to add properties to a class?

We call **properties** to the variables inside a class. Properties can accept values like strings, integers, and booleans (true/false values), like any other variable. Let's add some properties to the `Car` class.

```php
class Car {
    public $comp;
    public $color = "beige";
    public $hasSunRoof = true;
}
```

- We put the `public` keyword in front of a class property.
- The naming convention is to start the property name with a lower case letter.
- If the name contains more than one word, all of the words, except for the first word, start with an upper case letter. For example, $color or $hasSunRoof.
- A property can have a default value. For example, $color = 'beige'.
- We can also create a property without a default value. See the property $comp in the above example.

How to create objects from a class?

> We can create several objects from the same class, with each object having its own set of properties.

In order to work with a class, we need to create an **object** from it. In order to create an object, we use the `new` keyword. For example:

```php
$bmw = new Car ();
```

- We created the object $bmw from the class `Car` with the `new` key word.
- The process of creating an object is also known as **instantiation**.

We can create more than one object from the same class.

```php
$bmw = new Car ();
$mercedes = new Car ();
```

In fact, we can create as many objects as we like from the same class, and then give each object its own set of properties.

Objects, what are they good for?

While in the procedural style of programming, all of the functions and variables sit together in the **global scope** in a way that allows their use just by calling their name, the use of classes makes anything inside the classes hidden from the global scope. That's because the code inside the classes is encapsulated within the **class scope**, outside the reach of the global scope. So, we need a way to allow the code from the global scope to use the code within the class, and we do this by creating objects from a class.

I say objects, and not object, because we can create as many objects as we would like from the same class and they all will share the class's methods and properties. See the image below:
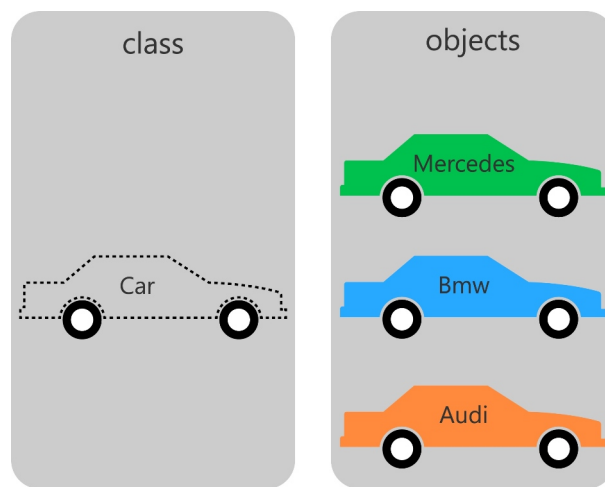
Classes and objects

From the same Car class, we created three individual objects with the name of: Mercedes, Bmw, and Audi.

Although all of the objects were created from the same class and thus have the class's methods and properties, they are still different. This is not only, because they have different names, but also because they may have different values assigned to their properties. For example, in the image above, they differ by the color property - the Mercedes is green while the Bmw is blue and the Audi is orange.

> **The message to take home is**:
>
> A class holds the methods and properties that are shared by all of the objects that are created from it.
>
> Although the objects share the same code, they can behave differently because they can have different values assigned to them.

How to get an object's properties?

Once we create an object, we can get its properties. For example:

```
echo $bmw -> color;
echo $mercedes -> color;
```

- In order to get a property, we write the class name, and then dash greater than (->), and then the property name.
- Note that the property name does not start with the $ sign; only the object name starts with a $.

Result:

beige

beige

How to set an object property?

In order to set an object property, we use a similar approach.

For example, in order to set the color to 'blue' in the bmw object:

```
$bmw -> color = 'blue';
```

and in order to set the value of the $comp property for both objects:

```
$bmw -> comp = "BMW";
$mercedes -> comp = "Mercedes Benz";
```

Once we set the value of a property, we can get its value.

In order to get the color of the $bmw, we use the following line of code:

```
1   echo $bmw -> color;
```

Result:

blue

We can also get the company name and the color of the second car object.

```
echo $mercedes -> color;
echo $mercedes -> comp;
```

Result:

beige

Mercedes Benz

How to add methods to a class?

The classes most often contain functions. A function inside a class is called a **method**. Here we add the **method** `hello()` to the class with the prefix `public`.

```
class Car {
    public $comp;
    public $color = "beige";
    public $hasSunRoof = true;

    public function hello()
    {
        return "beep";
    }
}
```

- We put the `public` keyword in front of a method.
- The naming convention is to start the function name with a lower case letter.
- If the name contains more than one word, all of the words, except for the first word, start with an upper case letter. For example, `helloUser()` or `flyPanAm()`.

We can approach the methods similar to the way that we approach the properties, but we first need to create at least one object from the class.

```php
$car1 = new Car ();
$car2 = new Car ();

echo $car1 -> hello();
echo $car2 -> hello();
```

Result:

beep

beep

Here is the full code that we have written during this chapter:

```php
<?php
// Declare the class
class Car {
    // Properties
    public $comp;
    public $color = "beige";
    public $hasSunRoof = true;

    // Method that says hello
    public function hello()
    {
        return "beep";
    }
}

// Create an instance
$bmw = new Car ();
$mercedes = new Car ();

// Get the values
echo $bmw -> color; // beige
echo "<br />";
echo $mercedes -> color; // beige
echo "<hr />";

// Set the values
$bmw -> color = "blue";
$bmw -> comp = "BMW";
$mercedes -> comp = "Mercedes Benz";
```

```php
// Get the values again
echo $bmw -> color; // blue
echo "<br />";
echo $mercedes -> color; // beige
echo "<br />";
echo $bmw -> comp; // BMW
echo "<br />";
echo $mercedes -> comp; // Mercedes Benz
echo "<hr />";

// Use the methods to get a beep
echo $bmw -> hello(); // beep
echo "<br />";
echo $mercedes -> hello(); // beep
```

Conclusion

In this chapter we have done our first steps into the world of object oriented PHP by learning about **classes** and about the **objects** that can be created out of them. In the next chapter we will learn about the **$this** keyword that enables us to approach properties and methods from within the class.

Let's practice what we have just learned

1.1 Which of these definitions best explains the term 'class'?

A: A collection of variables and functions working with these variables.

B: Sets and gets the class's own properties and methods.

C: Is the embodiment of a real life object.

1.2 Which of these definitions best explains the term 'object'?

A: An **object** gives us the ability to work with the class, and to have several instances of the same class.

B: A variable within a class.

C: A function within a class.

1.3 Which of these definitions best explains the term 'property'?

A: Groups the code that is related to one topic.

B: A variable within a class.

C: A function within a class.

1.4 Which of these definitions best explains the term 'method'?

A: A function within a class.

B: A variable within a class.

C: The embodiment of a real action.

Coding exercise

Almost every application or blog handles users. Whether it's the registration process, the login and logout, sending reminders to users who lost their passwords, or changing the passwords on demand, all of the code that handles users can be grouped into a single class. In our example, we call the class that handles users, User, in agreement with the prevailing naming convention.

Let's write a user class with the tools we have just acquired. This class will contain the first and last name of each user and will be able to say hello to anyone who uses our application.

1.5 Write what you think should be the class name, the names of the properties for the first and last name, and the name of the method that returns hello.

class name: _____

class properties: (1) _____ , **(2)** _____

class method: _____

1.6 Write the class User, and add the properties. That's how we start to write the class:

```
class User {
    // Your code here
}
```

1.7 Add the method that says hello to the class.

1.8 Create the first instance, and call it $user1. Use the new keyword to create an object from the class.

1.9 Set the values for the first and last name to $user1.

$firstName = 'John'

$lastName = 'Doe'

1.10 Get the user first and last name, and print it to the screen with echo.

1.11 Use the hello method with the first and last name variables in order to say hello to the user.

1.12 Add another object, call it $user2, give it a first name of 'Jane' and last name of 'Doe', then say hello to the user.

Suggested solutions

The $this keyword

In the [first chapter of this book](#), we learned how to group the code related to a certain topic into one single class. As an example, we wrote a `Car` class that groups all the code that handles cars.

```php
class Car {

    public $comp;
    public $numWheels = 4;
    public $hasSunRoof = true;

    public function hello()
    {
        return "beep";
    }
}
```

We also created two objects out of the class in order to be able to use its code:

```php
$bmw = new Car ();
$mercedes = new Car ();
```

The $this keyword

> The $this keyword indicates that we use the class's own methods and properties, and allows us to have access to them within the class's scope.

The `$this` keyword allows us to approach the class properties and methods from within the class using the following syntax:

```php
$this -> propertyName;
$this -> methodName();
```

- Only the `this` keyword starts with the $ sign, while the names of the properties and methods do not start with it.

The `$this` keyword indicates that we use the class's own methods and properties, and allows us to have access to them within the class's scope.

Let's illustrate what we have just said on the `Car` class. We will enable the `hello()` method to approach the class's own properties by using the `$this` keyword.

We use:

```
$this -> comp
```

in order to approach the class $comp property. We also use:

```
$this -> color
```

in order to approach the class $color property.

That's what the code looks like:

```php
class Car {
    // The properties
    public $comp;
    public $color = "beige";
    public $hasSunRoof = true;

    // The method can now approach the class properties
    //   with the $this keyword
    public function hello()
    {
        return "Beep I am a <i>" . $this -> comp . "</i>, and I am <i>" .
            $this -> color;
    }
}
```

Let us create two objects from the class:

```php
$bmw = new Car();
$mercedes = new Car ();
```

and set the values for the class properties:

```
$bmw -> comp = "BMW";
$bmw -> color = "blue";



$mercedes -> comp = "Mercedes Benz";
$mercedes -> color = "green";
```

We can now call the `hello` method for the first car object:

```
echo $bmw -> hello();
```

Result

Beep I am a *BMW*, and I am *blue.*

And for the second car object:

```
echo $mercedes -> hello();
```

Result

Beep I am a *Mercedes Benz*, and I am *green.*

This is the full code that we have written in this chapter:

```php
class Car {
    // The properties
    public $comp;
    public $color = "beige";
    public $hasSunRoof = true;

    // The method that says hello
    public function hello()
    {
        return "Beep I am a <i>" . $this -> comp .
            "</i>, and I am <i>" . $this -> color ;
    }
}



// We can now create an object from the class
$bmw = new Car();
$mercedes = new Car();

// Set the values of the class properties
$bmw -> color = "blue";
$bmw -> comp = "BMW";
$mercedes -> comp = "Mercedes Benz";

// Call the hello method for the the $bmw object
echo $bmw -> hello();
```

Conclusion

In this chapter, we have learned how to use the $this keyword in order to get the class's own properties and methods from within the class. In the next chapter, we will learn how to chain methods and properties.

Let's practice what we have just learned

2.1 Which keyword would you use in order to approach the class properties and methods from within the class?

A: The `new` keyword.

B: The `class` keyword.

C: The `$this` keyword.

Coding exercise

In the previous chapter, we wrote the `hello()` method inside the `User` class. In the following exercise, we will add to the `hello()` method the ability to approach the class properties with the `$this` keyword.

First, let's remind ourselves what the `User` class looks like:

```php
class User {
    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user
    public function hello()
    {
        return "hello";
    }
}
```

Wouldn't it be nicer if we could allow the `hello()` method the ability to get the class's properties, so that it would be able to say hello to the user name (for example, "hello, John Doe")?

2.2 Add to the `hello()` method the ability to approach the `$firstName` property, so the `hello()` method would be able to return the string "hello, $firstName".

2.3 Create a new object with the first name of 'Jonnie' and last name of 'Roe'.

2.4 Echo the `hello()` method for the `$user1` object, and see the result.

Suggested solutions

Chaining methods and properties

In [the previous chapter](), we learned to use the $this keyword to approach properties and methods within the scope of the class. In this chapter we will learn that, when a class's methods return the $this keyword, they can be chained together to create much more streaming code.

For instance, in our `Car` class, let's say that we want to measure how much fuel we have in our car's tank. The amount of fuel in the tank is dependent upon the number of miles we have driven in our car, as well as the amount of fuel that we put in the tank.

In order to achieve our goal, we are going to put a public property $tank to our class that represents the number of gallons of fuel that we have in the car's tank.

```php
class Car {
    public $tank;
}
```

We must also add two methods to our `Car` class:

1. The `fill()` method adds gallons of fuel to our car's tank.
2. The `ride()` method calculates how much fuel we consume when we ride a certain distance, and then subtracts it from the tank. In our example, we assume that the car consumes 1 gallon of fuel every 50 miles.

```php
class Car {

    public $tank;

    // Add gallons of fuel to the tank when we fill it
    public function fill($float)
    {
        $this-> tank += $float;
    }

    // Subtract gallons of fuel from the tank as we ride the car
    public function ride($float)
    {
        $miles = $float;
```

```
        $gallons = $miles/50;
        $this-> tank -= $gallons;
    }
}
```

As we would like our code to look elegant, we will chain the methods and properties. Note the arrows in the code.

```
$tank = $car -> fill(10) -> ride(40) -> tank;
```

In words: How much fuel do we have left in our tank after putting in 10 gallons, and driving 40 miles?

In order to perform the chaining, the methods should return the $this keyword.

In order for us to be able to perform the chaining, the methods should return the object and, since we are inside the class, the methods should return the $this keyword.

In the code below, we can see how each method returns the $this keyword in order to allow the chaining.

```
class Car {

    public $tank;

    // Add gallons of fuel to the tank when we fill it
    public function fill($float)
    {
        $this-> tank += $float;

        return $this;
    }

    // Subtract gallons of fuel from the tank as we ride the car
    public function ride($float)
    {
        $miles = $float;
        $gallons = $miles/50;
        $this-> tank -= $gallons;

        return $this;
    }
}
```

Now, we can create an object from the `Car` class with the name of `$bmw` and find out the number of gallons of fuel left in our car's tank after we have filled the tank with 10 gallons of fuel and driven 40 miles.

```php
// Create an object from the Car class
$bmw = new Car();

// Add 10 gallons of fuel, then ride 40 miles
//   and get the number of gallons in the tank
$tank = $bmw -> fill(10) -> ride(40) -> tank;

// Print the results to the screen
echo "The number of gallons left in the tank: " . $tank . " gal.";
```

Result:

The number of gallons left in the tank: 9.2 gal.

Conclusion

In this chapter, we learned that we can chain properties and methods to provide for a fluent code and improve its readability. In the next chapter, we will learn how to restrict access to the properties and methods inside our classes.

Let's practice what we have just learned

Let's now add some methods to represent register and mail functionality to the User class that we had written in the previous chapters. These methods would echo a string as placeholder for their actual purpose.

This is the User class that we have generated in the previous chapter:

```php
class User {
    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user $firstName.
    // The user $firstName property can be approached
    //    with the $this keyword.
    public function hello()
    {
        return "hello, " .  $this -> firstName;
    }
}
```

3.1 Add a register() method to the class that echoes the name of the user plus the verb "registered".

Use the $this keyword to approach the class's own properties.

3.2 Add a mail() method to the class that echoes the string "emailed".

3.3 Add return $this to the register() method so that it can be chained to any other method in the class.

3.4 Create a new $user1 object with the first name of 'Jane' and the last name of 'Roe'. For this object, chain the mail() method to the register() method and see the result.

Notice that each method we want to chain to should return the $this keyword. So, if we would like to write the next line of code that chains together 3 methods:

```
$user1 -> register() -> mail() -> hello();
```

We would have to return the $this keyword from both the register() and mail() methods, but we won't have to return the $this keyword from the hello() method because it concludes the chain.

Suggested solutions

Access modifiers: public vs. private

In the previous chapters, we used the **public access modifier** in front of the methods and properties in our classes without any explanations. The public access modifier is only one of several modifiers that we use. In this chapter, we will learn about another modifier called the **private access modifier**.

While the **public** access modifier allows a code from outside or inside the class to access the class's methods and properties, the **private** modifier prevents access to a class's methods or properties from any code that is outside the class.

The public access modifier

The following example should already be familiar to you. In the example, the class's property and method are defined as `public`, so the code outside the class can directly interact with them.

```php
class Car {

    // Public methods and properties
    public $model;

    public function getModel()
    {
        return "The car model is " . $this -> model;
    }
}

$mercedes = new Car();

// Here we access a property from outside the class
$mercedes -> model = "Mercedes";

// Here again we access another method from outside the class
echo $mercedes -> getModel();
```

Result:

The car model is Mercedes

The private access modifier

We can prevent access to the properties and methods inside our classes if we define them with the `private` access modifier instead of the `public` access modifier.

In the following example, we define the property `$model` as private, and when we try to set its value from outside the class, we encounter a fatal error.

```php
class Car {

    // Private
    private $model;

    public function getModel()
    {
        return "The car model is " . $this -> model;
    }
}

$mercedes = new Car();

// We try to access a private property from outside the class.
$mercedes -> model = "Mercedes";
echo $mercedes -> getModel();
```

Result:

Fatal error: Cannot access private property Car::$model

How to access a private property?

We saw that we have no access to private properties from outside the class, but we still have to somehow set and get the properties' values. In order to interact with private properties, we use public methods because they can interact with both the code outside of the class's scope as well as the code inside the class. The public methods that can interact in this manner are commonly divided into two kinds of methods:

- **Setters** that set the values of the private properties.
- **Getters** that get the values of the private properties.

In the following example, we will see that we can get and set the value of a private property, $carModel, through the use of setter and getter methods. We will use the setModel() method in order to set the value of the car model, and the getModel() method to get the value of the property.

```php
class Car {

    // The private access modifier denies access to the method
    //    from outside the class's scope
    private $model;

    // The public access modifier allows the access to the method
    //    from outside the class
    public function setModel($model)
    {
        $this -> model = $model;
    }

    public function getModel()
    {
        return "The car model is  " . $this -> model;
    }
}

$mercedes = new Car();

// Set the car's model
$mercedes -> setModel("Mercedes");

// Get the car's model
echo $mercedes -> getModel();
```

Result:

The car model is Mercedes

Why do we need access modifiers?

We need **access modifiers** in order to limit the modifications that code from outside the classes can do to the classes' methods and properties. Once we define a property or method as private, only methods that are within the class are allowed to approach it. So, in order to interact with private methods and properties, we need to provide public methods. Inside these methods, we can put logic that can validate and restrict data that comes from outside the class.

In our example, we can validate that only certain car models can make their way, and be assigned to the $model property, by defining the allowed alternatives for models in the getModel() method. For this purpose, we define inside the getModel() method an array of allowed car models, and check that only these models are assigned to the $model property.

```php
class Car {

    // The private access modifier denies access to the
    //   method from outside the class's scope
    private $model;

    // The public access modifier allows the access to
    //   the method from outside the class
    public function setModel($model)
    {
        // Validate that only certain car models are assigned
        //   to the $carModel property
        if(in_array($model,$allowedModels))
        {
            $this -> model = $model;
        }
    }

    public function getModel()
    {
        return "The car model is  " . $this -> model;
    }
}

$mercedes = new Car();

// Set the car's model
$mercedes -> setModel("Mercedes");

// Get the car's model
echo $mercedes -> getModel();
```

Conclusion

So far, we have learned about two access modifiers: **public**, which allows outside functions to modify the code inside a class, and **private**, that prevents any code from outside the class to change the properties and methods which it protects. We saw that, in order to modify private methods and properties, we can use public methods that have the privilege to interact with the code outside the scope of the class.

In the next chapter, we will learn about **magic methods and constants**. These are kinds of candies that PHP provides, but they should be used with much caution.

Let's practice what we have just learned

4.1 We use the private access modifier in order to:

A : Limit the access to a class.

B : Limit the access to properties.

C : Limit the access to methods.

D : B+C

Coding exercise

Let's return to the User class that we developed in the previous chapters, but let's now define the $firstName of the user as a private property.

This is the User class:

```
class User {
    // Your code goes here
}
```

4.2 Create a new class property with the name of $firstName, and prevent any code from outside the class from changing the property value by using the appropriate access modifier (private or protected).

4.3 Create a method to set the $firstName property value. Remember to use the right access modifier (public/private).

4.4 Now, create a method to return the $firstName value.

4.5 Create a new user object with the name of $user1, set its name to 'Joe' and make it return its name.

Suggested solutions

Magic methods and constants unveiled

Object Oriented PHP offers several **magic methods and constants** that enable a lot of programming with very little code, but which suffer from a few disadvantages like a slight reduction in the performance time as well as in the level of code clarity.

There is much dispute about the use of magic methods, so in this chapter we explain the parts of the field which are widely agreed, but ignore the parts in dispute.

The __construct() magic method

The names of magic methods always start with two underscores, and the __construct() magic method is no exception. We use __construct() in order to do something as soon as we create an object out of a class. A method of this kind is called a **constructor**. Usually we use the constructor to set a value to a property.

In our simple example, we should set the value of the $model property as soon as we create the object, so we add a constructor inside the class that sets the value of the $model property.

```php
class Car {
    private $model;

    // A constructor method
    public function __construct($model)
    {
        $this -> model = $model;
    }
}
```

In order to use the constructor, we have to pass an argument to the class with the name of the model as soon as we create the object, but if we try to create a new object without assigning the value that the constructor needs, we will encounter an error.

```php
$car1 = new Car();
```

Result:

Warning: Missing argument 1 for Car::__construct()

In order to avoid such an error, we have to assign a value to the constructor. Thus, for the sake of example, we assign the value "Mercedes" to the constructor by writing it within the brackets of the newly created object.

```php
$car1 = new Car("Mercedes");
```

Now, let's add the method getCarModel() in order to echo the car model from the object that we have just created.

```php
class Car {
    private $model;

    // The constructor
    public function __construct ($model)
    {
        $this -> model = $model;
    }

    public function getCarModel()
    {
        return ' The car model is: ' . $this -> model;
    }
}

// We pass the value of the variable
//   once we create the object
$car1 = new Car("Mercedes");

echo $car1 -> getCarModel();
```

Result:

The car model is: Mercedes.

How to write a constructor method without risking an error?

When we try to create an object that has a constructor method, we run the risk of our code breaking if we don't pass a value to the constructor. In order to avoid this risk, we can define a default value for the properties that we would like to set through the constructor. The default value may be the most reasonable choice for the property, zero, an empty string, or even a null.

If we use a null as the default value, we can use a condition to assess if a value was passed and then, only in that case, assign the value to the property.

In the example below, we give a default value of null to the $model property and, only if a value is passed to the constructor, we assign this value to the property. In any other case, the $model property has a default value of "N/A" string.

```php
class Car {
    // The $model property has a default value of "N/A"
    private $model = "";

    // We don't have to assign a value to the $model property
    //   since it already has a default value
    public function __construct($model = null)
    {
        if($model)
        {
            $this -> model = $model;
        }
    }

    public function getCarModel()
    {
        return ' The car model is: ' . $this -> model;
    }
}

// We create the new Car object
//   without passing a value to the model
$car1 = new Car();

echo $car1 -> getCarModel();
```

Even though we created the object without passing a value to the model property, we didn't cause an error because the model property in the constructor has a default value of null.

Result:

The car model is: N/A

On the other hand, let's see what happens when we define the model once we create the object. In the example below, we assign the value "Mercedes" to the $model property as soon as we create the object.

```php
class Car {
    private $model = '';

    // The constructor
    public function __construct($model = null)
    {
        if($model)
        {
            $this -> model = $model;
        }
    }

    public function getCarModel()
    {
        return ' The car model is: ' . $this -> model;
    }
}

// We create the new Car object
//   with the value of the model
$car1 = new Car('Mercedes');

echo $car1 -> getCarModel();
```

Result:

The car model is: Mercedes

Magic constants

In addition to magic methods, the PHP language offers several magic constants.

For example, we may use the magic constant __CLASS__ (magic constants are written in uppercase letters and prefixed and suffixed with two underlines) in order to get the name of the class in which it resides.

Let's take a look at the following example in which we use the __CLASS__ magic constant in the getter method:

```php
class Car {
    private $model = '';

    // The constructor
    public function __construct($model = null)
    {
        if($model)
        {
            $this -> model = $model;
        }
    }

    public function getCarModel()
    {
        // We use the __class__ magic constant
        //   in order to get the class name
        return " The <b>" . __class__ . "</b> model is: " . $this -> model;
    }
}

$car1 = new Car('Mercedes');

echo $car1 -> getCarModel();
```

Result:

The **Car** model is: Mercedes

Other magic constants that may be of help are:

__LINE__ to get the line number in which the constant is used.

__FILE__ to get the full path or the filename in which the constant is used.

__METHOD__ to get the name of the method in which the constant is used.

Conclusion

In this chapter, we have learned about **magic methods and constants**, and how to use a constructor to set the values of properties as soon as we create objects out of classes. In the next chapter, we will learn about inheritance, a principle of object oriented programming that can save us from repeating ourselves.

Let's practice what we have just learned

Let's return to the User class that we developed in the previous chapters. However, instead of using setter methods, we will set the values for the first and last name through the constructor.

This is the user class with the private properties of $firstName and $lastName:

```php
class User {
    private  $firstName;
    private $lastName;
}
```

5.1 Add to the class a constructor method to set a value to the $firstName property as soon as the object is created.

So far in the chapter, we have used the constructor method to set the value of a single property, but we can use a constructor in order to set the values of more than one property. In our exercise, we will use the constructor method to set the value of the $firstName as well as the $lastName.

5.2 Add to the constructor the ability to set the value of the $lastName property (remember that you can pass to a method more than parameter).

5.3 Add to the class a getFullName() public method that returns the full name.

5.4 Create a new object, $user1, and pass to the constructor the values of the first and last name. The first name is "John" and the last name is "Doe" (you may choose your preferred combination of first and last name).

5.5 Get the full name, and echo it to the screen.

Suggested solutions

Inheritance in object oriented programming

One of the main advantages of object-oriented programming is the ability to reduce code duplication with **inheritance**. Code duplication occurs when a programmer writes the same code more than once, a problem that **inheritance** strives to solve. In inheritance, we have a parent class with its own methods and properties, and a child class (or classes) that can use the code from the parent. By using inheritance, we can create a reusable piece of code that we write only once in the parent class, and use again as much as we need in the child classes.

How can a class inherit the code of another class?

> Inheritance allows us to write the code only once in the parent, and then use the code in both the parent and the child classes.

In order to declare that one class inherits the code from another class, we use the **extends** keyword. Let's see the general case:

```
class Parent {
    // The parent's class code
}

class Child extends Parent {
    // The  child can use the parent's class code
}
```

The child class can make use of all the non-private methods and properties that it inherits from the parent class. This allows us to write the code only once in the parent, and then use it in both the parent and the child classes.

In the example given below, the `SportsCar` class inherits the `Car` class, so it has access to all of the `Car`'s methods and properties that are not private. This allows us to write the `setModel()` and `hello()` public methods only once in the parent, and then use these methods in both the parent and the child classes.

```php
// The parent class
class Car {
    // Private property inside the class
    private $model;

    // Public setter method
    public function setModel($model)
    {
        $this -> model = $model;
    }


    public function hello()
    {
        return "beep! I am a <i>" . $this -> model . "</i><br />";
    }
}



// The child class inherits the code
//   from the parent class
class SportsCar extends Car {
    // No code in the child class
}

// Create an instance from the child class
$sportsCar1 = new SportsCar();

// Set the value of the class' property.
// For this aim, we use a method that we created in the parent
$sportsCar1 -> setModel('Mercedes Benz');

// Use another method that the child class inherited
//   from the parent class
echo $sportsCar1 -> hello();
```

Result:

beep! I am a Mercedes Benz

How can a child class have its own methods and properties?

Just as a child class can use the properties and methods of its parent class, it can have properties and methods of its own as well. However, while a child class can use the code it inherited from the parent, the parent class is not allowed to use the child class's code.

In the example given below, we will add to the child class some code of its own by adding the $style property as well as the driveItWithStyle() method:

```php
// The parent class has its properties and methods
class Car {
    // A private property or method can be used only by the parent
    private $model;

    // Public methods and properties can be used
    //   by both the parent and the child classes
    public function setModel($model)
    {
        $this -> model = $model;
    }

    public function getModel()
    {
        return $this -> model;
    }
}


// The child class can use the code it inherited from the parent class,
//   and it can also have its own code
class SportsCar extends Car {

    private $style = 'fast and furious';

    public function driveItWithStyle()
    {
        return 'Drive a '  . $this -> getModel() . ' <i>' .
            $this -> style . '</i>';
    }
}
```

```
// Create an instance from the child class
$sportsCar1 = new SportsCar();

// Use a method that the child class inherited from the parent class
$sportsCar1 -> setModel('Ferrari');

// Use a method that was added to the child class
echo $sportsCar1 -> driveItWithStyle();
```

Result:

Drive a Ferrari fast and furious.

The protected access control modifier

> When we declare a property or a method as protected, we can approach it from both the parent and the child classes.

In a previous chapter, we learned that we can use the public access modifier to allow access to a class's methods and properties from both inside and outside the class. We also learned that those methods and properties that are private can only be used from inside the class. In this chapter, we will learn about a third modifier - the **protected** modifier, which allows code usage from both inside the class and from its child classes.

The first example demonstrates what might happen when we declare the $model property in the parent as private, but still try to access it from its child class.

What do you think might happen when we try to call a private method or property from outside the class?

Here is the code:

```
// The parent class
class Car {
    // The $model property is private, thus it can be accessed
    //   only from inside the class
    private $model;

    // Public setter method
    public function setModel($model)
    {
        $this -> model = $model;
```

```php
    }
}


// The child class
class SportsCar extends Car{
    // Tries to get a private property
    //   that belongs to the parent
    public function hello()
    {
        return "beep! I am a <i>" . $this -> model . "</i><br />";
    }
}



 //Create an instance from the child class
$sportsCar1 = new SportsCar();

// Set the class model name
$sportsCar1 -> setModel('Mercedes Benz');

// Get the class model name
echo $sportsCar1 -> hello();
```

Result:

Notice: Undefined property: SportsCar::$model

We get an error because the `hello()` method in the child class is trying to approach a private property, $model, that belongs to the parent class.

We can fix the problem by declaring the $model property in the parent as protected, instead of private, because when we declare a property or a method as protected, we can approach it from both the parent and the child classes.

```php
// The parent class
class Car {
    // The $model property is now protected, so it can be accessed
    //      from within the class and its child classes
    protected $model;

    // Public setter method
    public function setModel($model)
    {
        $this -> model = $model;
    }
}


// The child class
class SportsCar extends Car {
    // Has no problem to get a protected property
    //   that belongs to the parent
    public function hello()
    {
        return "beep! I am a <i>" . $this -> model . "</i><br />";
    }
}



// Create an instance from the child class
$sportsCar1 = new SportsCar();

// Set the class model name
$sportsCar1 -> setModel('Mercedes Benz');

// Get the class model name
echo $sportsCar1 -> hello();
```

Result:

beep! I am a Mercedes Benz

Now it works, because we can access a protected code that belongs to a parent from a child class.

How to override the parent's properties and methods in the child class?

In the same way that the child class can have its own properties and methods, it can **override** the properties and methods of the parent class. When we override the class's properties and methods, we rewrite a method or property that exists in the parent again in the child, but assign to it a different value or code.

In the example given below, we create a `hello()` method in the parent class that returns the string `"beep"` and override it in the child class with a method by the same name that returns a different string, `"Hallo"`.

Here is the code:

```php
// The parent class has hello method that returns "beep"
class Car {
    public function hello()
    {
        return "beep";
    }
}


 // The child class has hello method that returns "Hallo"
class SportsCar extends Car {
    public function hello()
   {
        return "Hallo";
    }
}



 // Create a new object
$sportsCar1 = new SportsCar();

// Get the result of the hello method
echo $sportsCar1 -> hello();
```

Result:

Hallo

The result reflects the fact that the `hello()` method from the parent class was overridden by the child method with the same name.

How to prevent the child class from overriding the parent's methods?

In order to prevent the method in the child class from overriding the parent's methods, we can prefix the method in the parent with the **final** keyword.

In the example given below, we declare the `hello()` method in the parent as `final`, but still try to override it in the child class. What do you think might happen if we try to override a method that was declared as `final`?

```php
// The parent class has hello method that returns "beep"
class Car {
    final public function hello()
    {
        return "beep";
    }
}


// The child class has hello method that tries
//   to override the hello method in the parent
class SportsCar extends Car {
    public function hello()
    {
        return "Hallo";
    }
}



// Create a new object
$sportsCar1 = new SportsCar();

// Get the result of the hello method
echo $sportsCar1 -> hello();
```

Result:

Fatal error: Cannot override final method Car::hello()

Since we declared the hello method as `final` in the parent, we cannot override the method in the child class.

> We use inheritance in order to reduce code duplication by using code from the parent class in the child classes.

Conclusion

In this chapter we have learned one of the principles of object oriented programming, the concept of **inheritance**. We use inheritance in order to reduce code duplication by using code from the parent class in the child classes.

In the next chapter, we will learn about abstract classes and methods, which are classes that cannot be instantiated and methods that have no bodies.

Let's practice what we have just learned

6.1 Which keyword do we use in order to declare that one class inherits from another class?

A : `extends` in the parent

B : `extends` in the child

C : `extends` in both the child and the parent

6.2 Which keyword do we use in order to declare that a method or property can only be used within the parent class and its child classes?

A : private

B : protected

C : public

6.3 We learned about three access control modifiers (public, private, and protected) that we use to allow/restrict access to the code.

In the following table, we will use "V" to mark that the code can be accessed from a certain level, and "X" if it cannot be accessed.

Fill in the table with the right values.

For example, public code can be accessed from within the class, from the code within the child classes, and from the global scope, while private code cannot be accessed from the global scope (*).

Modifier	Class	Child class	Global scope *
Public	✓	✓	✓
Protected			
Private			✗

Comparison between access control modifiers

(*) The global scope is outside the classes.

Coding exercise

In the following example, we will create an Admin class, which is a child class of the User class.

6.4 Create a user class.

6.5 Add to the class a private property with the name of $username.

6.6 Create a setter method that can set the value of the $username.

6.7 Create a class Admin that inherits the User class.

6.8 Now, let's add to the Admin class some code. First, add a public method by the name of expressYourRole, and make it return the string: "Admin".

6.9 Add to the Admin class another public method, sayHello, that returns the string "Hello admin, XXX" with the username instead of XXX.

6.10 Create an object $admin1 out of the class Admin, set its name to "Balthazar", and say hello to the user. Do you see any problem?

6.11 What do you think is the cause of the problem?

6.12 How will you fix this problem?

Suggested solutions

Abstract classes and methods

We use **abstract classes** when we want to commit the programmer (either oneself or someone else) to write a certain class method, but we are only sure about the name of the method, and not the details of how it should be written. To take an example, circles, rectangles, octagons, etc. may all look different, but are all 2D shapes nonetheless, and thus all possess the traits of area and circumference. So, it makes perfect sense to group the code that they have in common into one parent class. In this parent class, we would have the two properties of area and circumference, and we may very well have a method that calculates the area (which might be problematic since different shapes require different calculations). In these kinds of cases, when we need to commit the child classes to certain methods that they inherit from the parent class but we cannot commit about the code that should be used in the methods, we use abstract classes.

> We use abstract classes and methods when we need to commit the child classes to certain methods that they inherit from the parent class but we cannot commit about the code that should be written inside the methods.

An **abstract class** is a class that has at least one **abstract method**. Abstract methods can only have names and arguments, and no other code. Thus, we cannot create objects out of abstract classes. Instead, we need to create child classes that add the code into the bodies of the methods, and use these child classes to create objects.

How to declare classes and methods as abstract?

In order to declare a class as **abstract**, we need to prefix the name of the class with the `abstract` keyword.

See the following example:

```
abstract class Car { }
```

We put the abstract methods that are also declared with the `abstract` keyword within the abstract class. Abstract methods inside an abstract class don't have a body, only a name and parameters inside parentheses.

In the example given below, we create a public abstract method, `calcNumMilesOnFullTank()`, that is the skeleton for methods that we will create in the child classes. Once created, these methods will return the number of miles a car can be driven on a full tank of gas.

```php
// Abstract classes are declared with the abstract keyword,
//   and contain abstract methods
abstract class Car {
    abstract public function calcNumMilesOnFullTank();
}
```

It is important to know that once we have an abstract method in a class, the class must also be abstract.

Can we have non abstract methods inside an abstract class?

An abstract class can have non abstract methods. In fact, it can even have properties, and properties couldn't be abstract.

Let's add to our example the protected property, $tankVolume, and public method with the name of setTankVolume().

```php
abstract class Car {
    // Abstract classes can have properties
    protected $tankVolume;

    // Abstract classes can have non abstract methods
    public function setTankVolume($volume)
    {
        $this -> tankVolume = $volume;
    }

    // Abstract method
    abstract public function calcNumMilesOnFullTank();
}
```

How to create child classes from an abstract class?

Since we cannot create objects from abstract classes, we need to create child classes that inherit the abstract class code. Child classes of abstract classes are formed with the help of the extends keyword, like any other child class. They are different in that they have to add the bodies to the abstract methods.

> The child classes that inherit from abstract classes must add bodies to the abstract methods.

Let's create a child class with the name of Honda, and define in it the abstract method that it inherited from the parent, calcNumMilesOnFullTank().

```php
class Honda extends Car {
    // Since we inherited abstract method,
    //   we need to define it in the child class,
    //   by adding code to the method's body.
    public function calcNumMilesOnFullTank()
    {
        $miles = $this -> tankVolume*30;
        return $miles;
    }
}
```

We can create another child class from the Car abstract class and call it Toyota, and here again define the abstract method calcNumMilesOnFullTank() with a slight change in the calculation. We will also add to the child class its own method with the name of getColor() that returns the string "beige".

```php
class Toyota extends Car {
    // Since we inherited abstract method,
    //   we need to define it in the child class,
    //   by adding code to the method's body.
    public function calcNumMilesOnFullTank()
    {
        return $miles = $this -> tankVolume*33;
    }

    public function getColor()
    {
        return "beige";
    }
}
```

Let's create a new object, $toyota1, with a full tank volume of 10 Gallons, and make it return the number of miles on full tank as well as the car's color.

```php
$toyota1 = new Toyota();
$toyota1 -> setTankVolume(10);

echo $toyota1 -> calcNumMilesOnFullTank();
echo $toyota1 -> getColor();
```

Result:

330

beige

Conclusion

In this chapter, we had our first encounter with the concept of abstraction, which enables us to commit the child classes to methods names, and not to methods bodies. In the next chapter, we are going to revisit the concept of abstraction, but this time through the use of interface.

Let's practice what we have just learned

7.1 Which keyword is used to declare a class or method as abstract?

A : abstract

B : inherit

C : abstract

D : extends

7.2 Which keyword is used to declare a child class that inherits from an abstract class?

A : abstract

B : inherit

C : abstract

D : extends

7.3 What are the main reasons for using an abstract class in the code?

A : To enable code inheritance in PHP.

B : To commit the child classes to certain methods.

C : To make a good impression on your boss and colleagues.

D : To encapsulate the code of the parent from any code outside the parent class.

7.4 Can we create objects from abstract classes?

Coding exercise

In the following example, we will create an abstract `User` class and two child classes (`Admin` and `Viewer` classes) that inherit from the abstract class.

7.5 Create an abstract class with the name of `User`, which has an abstract method with the name of `stateYourRole`.

7.6 Add to the class a protected variable with the name of `$username`, and public setter and getter methods to set and get the `$username`.

7.7 Create an `Admin` class that inherits the abstract `User` class.

7.8 Which method should be defined in the class?

7.9 Define the method `stateYourRole()` in the child class and make it return the string "admin";

7.10 Create another class, `Viewer` that inherits the `User` abstract class. Define the method that should be defined in each child class of the `User` class.

7.11 Create an object from the `Admin` class, set the username to "Balthazar", and make it return the string "admin".

Suggested solutions

Interfaces - the next level of abstraction

Interfaces resemble abstract classes in that they include abstract methods that the programmer must define in the classes that inherit from the interface. In this way, interfaces contribute to code organization because they commit the child classes to the methods that they should implement. The use of interfaces become very helpful when we work in a team of programmers and want to ensure that all the programmers write the methods that they should work on, or even in the case of a single programmer that wants to commit himself to write certain methods in the child classes.

An interface commits its child classes to abstract methods that they should implement.

How to declare and implement an interface?

We declare an interface with the `interface` keyword and, the class that inherits from an interface with the `implements` keyword. Let's see the general case:

```
interface interfaceName {
    // Abstract methods
}
```

```
class Child implements interfaceName {
    // Defines the interface methods,
    //   and may have its own methods.
}
```

In the simple example given below, we will create an interface for the classes that handle cars, which commits all its child classes to `setModel()` and `getModel()` methods.

```
interface Car {
    public function setModel($name);

    public function getModel();
}
```

Interfaces, like abstract classes, include abstract methods and constants. However, unlike abstract classes, interfaces can have only public methods, and cannot have variables.

The classes that implement the interfaces must define all the methods that they inherit from the interfaces, including all the parameters. So, in our concrete class with the name of `miniCar`, we add the code to all the abstract methods.

```php
class miniCar implements Car {
    private $model;

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

    public function getModel()
    {
        return $this -> model;
    }
}
```

Can we implement more than one interface in the same class?

We can implement a number of interfaces in the same class.

We can implement a number of interfaces in the same class, and so circumvent the law that prohibits the inheritance from more than one parent class. In order to demonstrate multiple inheritance from different interfaces, we create another interface, `Vehicle`, that commits the classes that implement it to a boolean `$hasWheels` property.

```php
interface Vehicle {
    public function setHasWheels($bool);

    public function getHasWheels();
}
```

Now, our child class can implement the two interfaces.

```php
// The class implements two interfaces
class miniCar implements Car, Vehicle {

    private $model;
    private $hasWheels;

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

    public function getModel()
    {
        return $this -> model;
    }

    public function setHasWheels($bool)
    {
        $this -> hasWheels = $bool;
    }

    public function getHasWheels()
    {
        return ($this -> hasWheels)? "has wheels" : "no wheels";
    }
}
```

What are the differences between abstract classes and interfaces?

We saw that abstract classes and interfaces are similar in that they provide abstract methods that can be implemented only in the child classes. However, they still have the following differences:

- Interfaces can include abstract methods and constants, but cannot contain concrete methods and variables.
- All the methods in the interface must be in the `public` visibility scope.
- A class can implement more than one interface, while it can inherit from only one abstract class.

Let's summarize these differences in the following table:

	interfaces	Abstract classes
the code	-abstract methods -constants	-abstract methods -constants -concrete methods -concrete variables
access modifiers	-public	-public -protected -private etc.
number of parents	The same class can implement more than 1 interface	The child class can inherit only from 1 abstract class

Conclusion

In this chapter, we saw how to create and use **interfaces**, and learned about the differences between abstract classes and interfaces. In the next chapter, we are going to use our knowledge of abstract classes to improve the consistency of our code.

Let's practice what we have just learned

8.1 Which keyword should we use in order to implement an interface?

A : extends

B : public

C : private

D : implements

8.2 Which of the following code types are allowed in an interface?

A : methods

B : properties

C : constants

D : A+C

8.3 Which of the following access modifiers is allowed for abstract methods in interfaces?

A : public

B : protected

C : private

D : A + B + C

8.4 Which of the following is valid inheritance?

A : A class can inherit from an abstract class and two interfaces.

B : A class can inherit from a parent class and one interface.

C : A class can inherit from a parent class and more than one interface.

D : A class can inherit from 2 abstract classes.

E : A + B + C

Coding exercise

In this chapter, we saw that a class can implement more than one interface. In the concluding example, we will go one step further by letting the same child class inherit from both a parent class and from two interfaces.

8.5 Create a `User` class with a protected `$username` property and methods that can set and get the `$username`.

8.6 Create an Author interface with the following abstract methods that can give the user an array of authorship privileges. The first method is `setAuthorPrivileges()`, and it gets a parameter of `$array`, and the second method is `getAuthorPrivileges()`.

8.7 Create an `Editor` interface with methods to set and get the editor's privileges.

8.8 Now, create an `AuthorEditor` class that extends the `User` class, and implements both the `Author` and the `Editor` interfaces.

8.9 Create in the `AuthorEditor` class the methods that it should implement, and the properties that these methods force us to add to the class.

For example, in order to implement the public method `setAuthorPrivileges()`, we must add to our class a property that holds the array of authorship privileges, and name it `$authorPrivilegesArray` accordingly.

8.10 Create an object with the name of `$user1` from the class `AuthorEditor`, and set its username to "Balthazar".

8.11 Set in the `$user1` object an array of authorship privileges, with the following privileges: "write text", "add punctuation".

8.12 Set in the `$user1` object an array with the following editorial privileges: "edit text", "edit punctuation".

8.13 Use the following code to get the `$user1` name and privileges:

```php
$userName  = $user1 -> getUsername();
$userPrivileges = array_merge($user1 -> getAuthorPrivileges(),
    $user1 -> getEditorPrivileges());

echo $userName . " has the following privileges: ";
foreach($userPrivileges as $privilege)
{
    echo " {$privilege},";
}
echo ".";
```

Suggested solutions

Polymorphism in PHP

In this chapter, we are going to learn a naming convention that can help us write code which is much more coherent and easy to use. According to the **Polymorphism** (Greek for "many forms") principle, methods in different classes that do similar things should have the same name.

> According to the **Polymorphism** principle, methods in different classes that do similar things should have the same name.

A prime example is of classes that represent geometric shapes (such as rectangles, circles and octagons) that are different from each other in the number of ribs and in the formula that calculates their area, but they all have in common an area that needs to be calculated. In such case, the **polymorphism principle** says that all the methods that calculate the area (and it doesn't matter for which shape or class) should have the same name. For example, we can write in each class that represents a shape a `calcArea()` method that calculates the area with a different formula. Now, whenever the end users would like to calculate the area for the different shapes, the only thing that they need to know is that a method with the name of `calcArea()` is used to calculate the area. So, they don't have to pay any attention to the technicalities of how actually each method works. The only thing that they need to know is the name of the method and what it is meant to do.

How to implement the polymorphism principle?

> In order to implement the polymorphism principle, we can choose between abstract classes and interfaces.

In order to ensure that the classes do implement the polymorphism principle, we can choose between one of the two options of either abstract classes or interfaces.

In the example given below, an interface with the name of `Shape` commits all the classes that implement it to define an abstract method with the name of `calcArea()`.

```php
interface Shape {
    public function calcArea();
}
```

In accordance, the `Circle` class implements the interface by putting into the `calcArea()` method the formula that calculates the area of circles.

```php
class Circle implements Shape {
    private $radius;

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

    // The method calcArea calculates the area of circles
    public function calcArea()
    {
        return $this -> radius * $this -> radius * pi();
    }
}
```

The rectangle class also implements the Shape interface but defines the function calcArea() with a calculation formula that is suitable for rectangles.

```php
class Rectangle implements Shape {
    private $width;
    private $height;

    public function __construct($width, $height)
    {
        $this -> width = $width;
        $this -> height = $height;
    }

    // calcArea calculates the area of rectangles
    public function calcArea()
    {
        return $this -> width * $this -> height;
    }
}
```

Now, we can create objects from the concrete classes:

```php
$circ = new Circle(3);
$rect = new Rectangle(3,4);
```

We can be sure that all of the objects calculate the area with the method that has the name of calcArea(), whether it is a rectangle object or a circle object (or any other shape), as long as they implement the Shape interface.

Now, we can use the calcArea() methods to calculate the area of the shapes:

```
echo $circ -> calcArea();
echo $rect -> calcArea();
```

Result:

28.274333882308

12

Conclusion

In this chapter, we have learned how to implement the **polymorphism principle** in PHP. In the next chapter, we will learn about type hinting that specifies the expected data type (arrays, objects, interface, etc.) for an argument in a function declaration.

Let's practice what we have just learned

9.1 Which of these sentences best defines the **polymorphism principle** in PHP?

A : Methods that serve the same functionality in different classes should have the same name.

B : The same function can have many names.

C : Is the Greek term for inheritance.

D : Can be implemented in most of the modern programming languages.

Coding example

In our coding example let's return to the User class that we used in the previous chapters.

In order to implement the **polymorphism principle**, we are going to create an abstract User class that commits the classes that inherit from it to calculate the number of scores that a user has depending on the number of articles that he has authored or edited. On the basis of the User class, we are going to create the Author and Editor classes, and both will calculate the number of scores with the method calcScores(), although the calculated value will differ between the two classes.

This is the skeleton for the abstract User class:

```php
abstract class User {
    protected $scores          = 0;
    protected $numberOfArticles = 0;

    // The abstract and concrete methods

}
```

9.2 Add to the class concrete methods to set and get the number of articles:

1. setNumberOfArticles($int)
2. getNumberOfArticles()

$int stands for an integer.

9.3 Add to the class the abstract method: calcScores(), that performs the scores calculations separately for each class.

9.4 Create an `Author` class that inherits from the `User` class. In the `Author` create a concrete `calcScores()` method that returns the number of scores from the following calculation:

```
numberOfArticles * 10 + 20
```

9.5 Also create an `Editor` class that inherits from the `User` class. In the `Editor` create a concrete `calcScores()` method that returns the number of scores from the following calculation:

```
numberOfArticles * 6 + 15
```

9.6 Create an object, `$author1`, from the `Author` class, set the number of articles to 8, and echo the scores that the author gained.

9.7 Create another object, `$editor1`, from the `Editor` class, set the number of articles to 15, and echo the scores that the editor gained.

Suggested solutions

Type hinting

With **Type hinting** we can specify the expected data type (arrays, objects, interface, etc.) for an argument in a function declaration. This practice can be most advantageous because it results in better code organization and improved error messages.

This chapter explains the subject of **type hinting** for arrays and objects in PHP5. It also explains the subject of type hinting for basic data types (integers, floats, strings, and booleans) which is only supported in PHP7.

How to do array type hinting?

When we would like to force a function to get only arguments of the type **array**, we can put the keyword `array` in front of the argument name, with the following syntax:

```php
function functionName (array $argumentName)
{
    // The code block
}
```

In the following example, the `calcNumMilesOnFullTank()` function calculates the number of miles a car can be driven on a full tank of gas by using the tank volume as well as the number of miles per gallon (mpg). This function accepts only array as an argument, as we can see from the fact that the argument name is preceded by the `array` keyword.

```php
// The function can only get array as an argument.
function calcNumMilesOnFullTank(array $models)
{
    foreach($models as $item)
    {
        echo $carModel = $item[0];
        echo " : ";
        echo $numberOfMiles = $item[1] * $item[2];
        echo "<br />";
    }
}
```

First, let's try to pass to the function an argument which is not an array to see what might happen in such a case:

```
calcNumMilesOnFullTank("Toyota");
```

Result:

> 🪲 Catchable fatal error: Argument 1 passed to calcNumMilesOnFullTank() must be of the type array, string given

This error is a precise description of what went wrong with our code. From it, we can understand that the function expected an array variable, and not a string.

Let's rewrite the code and pass to the function an array with the expected items, including the model names, the tank volumes, and the mpg (miles per gallon).

```
$models = array(
    array('Toyota', 12, 44),
    array('BMW', 13, 41)
);

calcNumMilesOnFullTank($models);
```

Result:

Toyota : 528

BMW : 533

Now it's working because we passed to the function the array that it expected to get.

How to do object type hinting?

Type hinting can also be used to force a function to get an argument of type Object. For this purpose, we put the name of the class in front of the argument name in the function.

In the following example, the class's constructor can only get objects that were created from the `Driver` class. We ensure this by putting the word `Driver` in front of the argument name in the constructor.

```php
class Car {
    protected $driver;

    // The constructor can only get Driver objects as arguments.
    public function __construct(Driver $driver)
    {
        $this -> driver = $driver;
    }
}


class Driver {}


$driver1 = new Driver();
$car1    = new Car($driver1);
```

Does PHP support type hinting to basic data types?

It depends. Whereas PHP5 doesn't allow type hinting for basic data types (integers, floats, strings and booleans), PHP7 does support **scalar type hinting**.

PHP5 does not support type hinting to basic data types like integers, booleans or strings. So, when we need to validate that an argument belongs to a basic data type, we can use one of PHP's "is_" family functions. For example:

- **is_bool** - to find out whether a variable is a boolean (true or false).
- **is_int** - to find out whether a variable is an integer.
- **is_float** - to find out whether a variable is a float (3.14, 1.2e3 or 3E-10).
- **is_null** - to find out whether a variable is null.
- **is_string** - to find out whether a variable is a string.

On the other hand, PHP7 **does** support **scalar type hinting**. The supported types are: integers, floats, strings, and booleans.

The following code example can only work in PHP7.

```php
class car
{
    protected $model;
        protected $hasSunRoof;
        protected $numberOfDoors;
        protected $price;

        // string type hinting
        public function setModel(string $model)
        {
            $this->model = $model;
        }

        // boolean type hinting
        public function setHasSunRoof(bool $value)
        {
            $this->hasSunRoof = $value;
        }

        // integer type hinting
        public function setNumberOfDoors(int $value)
        {
            $this->numberOfDoors = $value;
        }

        // float type hinting
        public function setPrice(float $value)
        {
            $this->price = $value;
        }
}
```

Conclusion

In this chapter, we explained **type hinting** for the array and object data types as well as for the scalar data types (integers, floats, strings, and booleans), but did not explain how to do it for interfaces. However, the use of type hinting for interfaces can yield many benefits, as you can learn in the next chapter.

Let's practice what we have just learned

10.1 Type hinting can validate that the types of values that are passed to functions belong to the following data types:

A : booleans, integers, floats, and strings.

B : arrays, objects, and interfaces.

C : Type hinting for complex types (arrays and objects) is supported in both PHP 5 and 7, while scalar type hinting (for integers, floats, strings, and booleans) is only supported in PHP7.

Suggested solutions

Type hinting for interfaces

In the previous chapter we learned that **type hinting** contributes to code organization and improves error messages. In this chapter, we will learn how to implement type hinting for interfaces.

In the first part of the chapter we will see that the use of type hinting for objects is not always sufficient, and in the second part we will learn how to fix the problem by the use of type hinting for interfaces.

Why type hinting for objects may not be sufficient?

We will base this chapter on the following imaginary scenario: A manager in a car rental company that rents only BMWs hired a programmer to write a program that calculates the price for a full tank of gas for each and every car that the company owns. In accordance with these demands, the programmer writes a class to which he decides to call `Bmw` that holds the code for the task. This class has the data about the BMWs including the license plate number, the car model and, most importantly, it has a method that calculates the volume of the fuel tank.

The function that calculates the volume is called `calcTankVolume`. This method calculates the tank volume by multiplying the base area (the square of the base ribs' length) by the height. The height, as well as the length of the base ribs and the license plate number, is introduced to the class through the constructor.

```php
class Bmw {

    protected $model;
    protected $rib;
    protected $height;

    // The properties are introduced to the class through the constructor
    public function __construct($model, $rib, $height)
    {
        $this -> model = $model;
        $this -> rib = $rib;
        $this -> height = $height;
    }

    // Calculate the tank volume for rectangular tanks
    public function calcTankVolume()
```

```
    {
        return $this -> rib * $this -> rib * $this -> height;
    }
}
```

Outside of the class, the programmer writes a function to calculate the price for a full tank of gas by multiplying the tank volume by the price per gallon. However, since he doesn't want the function to get any argument other than those that belong to the `Bmw` class, he uses type hinting:

```
// Type hinting ensures that the function gets only Bmw objects as arguments
function calcTankPrice(Bmw $bmw, $pricePerGalon)
{
    return $bmw -> calcTankVolume() * 0.0043290 * $pricePerGalon . "$";
}
```

Now, he can easily calculate how much a full tank of gas costs for BMWs. for a car with the license plate number of '62182791', a rib length of 14" and height of 21", and using gas priced at 3 dollars per gallon.

```
$bmw1 = new Bmw('62182791', 14, 21);
calcTankPrice($bmw1, 3);
```

Result:

53.454492$

Type hinting for interfaces

After a short time, the manager decides to introduce to his fleet of cars a brand new Mercedes, but the problem is that the `calcTankPrice()` function can only perform calculations for BMWs. It turns out that, while BMWs have a rectangular shaped gas tank, Mercedes have a cylindrical shaped gas tank. So, our programmer is summoned once again to write another class that can handle this new task at hand. To this end, our programmer now writes the following class with the name of `Mercedes` that can calculate the tank volume for cylindrical shaped tanks.

```php
class Mercedes {

    protected $model;
    protected $radius;
    protected $height;

    public function __construct($model, $radius, $height)
    {
        $this -> model = $model;
        $this -> radius = $radius;
        $this -> height = $height;
    }

    // Calculate the volume of cylinders
    public function calcTankVolume()
    {
        return $this -> radius * $this -> radius * pi() * $this -> height;
    }
}
```

When our programmer completed the task of writing the `Mercedes` class, he tried to calculate the price of a full tank of gas for a Mercedes.

```php
$mercedes1 = new Mercedes('12189796', 7, 28);
echo calcTankPrice($mercedes1, 3);
```

But the result wasn't quite what he had expected:

Result:

Catchable fatal error: Argument 1 passed to calcTankPrice() must be an instance of Bmw, instance of Mercedes given

This error message is the result of not passing the right object to the function, since he tried to pass a `Mercedes` object to the `calcTankVolume()` function, while the function can only accept objects that belong to the `Bmw` class.

First, our programmer attempted to solve the problem by not using any type hinting, but then he understood that a better solution would be the use of **type hinting for interfaces**. Here I use interface in its wider meaning that includes both abstract classes and real interfaces.

So, first he created an abstract class with the name of `Car` that both the `Bmw` and the `Mercedes` (and, in fact, any other car model) can inherit from.

```php
abstract class Car {
    protected $model;
    protected $height;

    abstract public function calcTankVolume();
}
```

Then he re-factored the Bmw and Mercedes classes so they inherit from the Car class:

```php
class Bmw extends Car {

    protected $rib;

    public function __construct($model, $rib, $height)
    {
        $this -> model = $model;
        $this -> rib        = $rib;
        $this -> height = $height;
    }

    // Calculate a rectangular tank volume
    public function calcTankVolume()
    {
        return $this -> rib * $this -> rib * $this -> height;
    }
}


class Mercedes extends Car {

    protected $radius;

    public function __construct($model, $radius, $height)
    {
        $this ->model = $model;
        $this -> radius = $radius;
        $this -> height = $height;
    }

    // Calculates the volume of cylinder
    public function calcTankVolume()
    {
```

```php
        return $this -> radius * $this -> radius * pi() * $this -> height;
    }
}
```

Since both the classes inherit from the same interface, he chose wisely to type hint the function `calcTankPrice()` with the `Car` interface, so that the function can get any object that belong to this interface.

```php
// Type hinting ensures that the function gets only objects
//   that belong to the Car interface
function calcTankPrice(Car $car, $pricePerGalon)
{
    echo $car -> calcTankVolume() * 0.0043290 * $pricePerGalon . "$";
}
```

Now, let's see the result when we try to use the function `calcTankPrice()` on both a `Bmw` and a `Mercedes` objects:

```php
$bmw1 = new Bmw('62182791', 14, 21);
echo calcTankPrice($bmw1, 3);

$mercedes1 = new Mercedes('12189796', 7, 28);
echo calcTankPrice($mercedes1, 3);
```

Result:

53.454492$

55.977413122858$

> Whenever we need to do type hinting to more than one related classes, we should be using interface type hinting.

The message to take home from this chapter is that whenever we need to do type hinting to more than one related classes, we should be using interface type hinting.

Conclusion

In this chapter, we have learned how to implement **type hinting** for interfaces. In the next chapter we will learn about static methods and properties that are used without the need of creating an object.

Let's practice what we've just learned

In this chapter we used an example for type hinting with interface in the form of abstract class. Now, we are going to practice what we have learned with type hinting to interface per se.

11.1 Create a `User` interface with set and get methods for both a `$username` property, as well as for a `$gender` property.

11.2 Now, create a `Commentator` class to implement the `User` interface.

11.3 Create function to add "Mr." or "Mrs." to the username. When writing the function make sure to type hint it correctly, so it can only get the types it is expected to.

11.4 Run the code against a user with the name of "Jane" and against another user with the name of "Bob".

Suggested solutions

Static methods and properties

In certain cases, we might want to approach methods and properties of a class without the need to first create an object out of the class. This can be achieved by defining the methods and properties of a class as **static**. Even though the use of static methods and properties is **not** considered a good practice, there are cases in which their use is quite handy and justified.

> We can access static methods and properties without the need to first create an object, but their use should be limited.

We have already learned about the three access modifiers called public, protected and private. This chapter is devoted to the fourth modifier, **static**, that allows access to classes' properties and methods without the need to create objects out of the classes.

How to define methods and properties as static?

In order to define methods and properties as **static**, we use the reserved keyword static. In the following example, the class Utilis has a static public property with the name of $numCars, and so the property's name is preceded by both the static as well as the public keywords.

```php
class Utilis {
    // Static methods and properties
    //   are defined with the static keyword
    static public $numCars = 0;
}
```

How to approach static methods and properties?

In order to approach static methods and properties we use the **scope resolution operator** (::). In the example given below, in order to work with the static property of $numCars, we use the following code:

```php
// Set the number of cars
Utilis::$numCars = 3;

// Get the number of cars
echo Utilis::$numCars;
```

Result:

3

Pay attention to the use of the $ sign right after the scope resolution operator.

How to approach the static methods from within the class?

In the same manner that we used the $this keyword to approach the class's own properties and methods from within the class, we use the reserved keyword self to approach static methods and properties.

In the example given below, the method addToNumCars() gets and sets the number of cars from the static $numCars property within the same class by using the self keyword.

```php
class Utilis {
    static public $numCars = 0;

    static public function addToNumCars($int)
    {
        $int = (int)$int;
        self::$numCars += $int;
    }
}
```

Now, we can use the code we have written so far to set and get the number of cars.

```php
echo Utilis::$numCars;

Utilis::addToNumCars(3);
echo Utilis::$numCars;

Utilis::addToNumCars(-1);
echo Utilis::$numCars;
```

Result:

0

3

2

When to use static properties and methods?

> The main cases in which we consider the use of static methods and properties are for utilities.

The use of static properties and methods is not a good practice. However, in some cases, the ability to use a property or a method without the need to first create an object out of a class can be advantageous. The main cases in which we consider the use of static methods and properties are when we need them as counters and for utility classes.

Use case 1: as counters

We use static properties as counters since they are able to save the last value that has been assigned to them. For example, the method `add1ToCars()` adds 1 to the `$numberOfCars` property each time the method is called.

```php
class Utilis {
    // Hold the number of cars
    static public $numberOfCars = 0;

    // Add 1 to the number of cars each time the method is called
    static public function add1ToCars()
    {
        self::$numberOfCars++;
    }
}


echo Utilis::$numberOfCars;

Utilis::add1ToCars();
echo Utilis::$numberOfCars;

Utilis::add1ToCars();
echo Utilis::$numberOfCars;
```

```
Utilis::add1ToCars();
echo Utilis::$numberOfCars;
```

Result:

0

1

2

3

Use case 2: for utility classes

It is very common to use static methods for **utility classes**. The sole purpose of utility classes is to provide services to the main classes. Utility methods can perform all kinds of tasks, such as: conversion between measurement systems (kilograms to pounds), data encryption, sanitation, and any other task that is not more than a service for the main classes in our application.

The example given below is of a static method with the name of `redirect` that redirects the user to the URL that we pass to it as an argument.

```
class Utilis {
    // The method uses PHP's header function
    //   to redirect the user
    static public function redirect($url)
    {
        header("Location: $url");
        exit;
    }
}
```

In order to use the `redirect()` method, all we need to do is call it from the scope of the `Utilis` class without having the need to create an object. It's as simple as this:

```
Utilis::redirect("http://www.phpenthusiast.com");
```

Why static should be used with caution?

Whenever you use static, be sure to use it for utilities and not for convenience reasons.

If you are considering the use of static methods and properties because they are convenient and can be approached without the need to first create an object, please be cautious, because static methods suffer from the following two major disadvantages:

- Static methods and properties present globals to the code that can be approached from anywhere, and globals are something that we should avoid as much as possible.
- You will have a hard time performing automated testing on classes that use static methods.

So, whenever you use static, be sure to use it for utilities and not for convenience reasons.

Conclusion

In this chapter, we have learned how and when to use **static methods and properties**, but most important when not to use it. In the next chapter, we are going to learn how to include code the object oriented way with traits.

Let's practice what we have just learned

12.1 The use of static methods and appropriate is appropriate in the following case:

A : Whenever we want to use a class's method without instantiation.

B : For utilities such as conversion between units of measurements, sanitation, encryption, etc.

C : When we would like to unit test our code.

D : When our code is static.

Suggested solutions

Traits and code inclusion

PHP allows a class to inherit from only a single parent class, but sometimes we cannot escape the necessity to use code from more than one resource. In these cases, it is common to chain more and more parents. However, this can easily get quite complicated and it is considered a good practice not to exceed beyond an inheritance tree of three generations. Another common practice is to include the code that we would like to share between the classes by using the 'include' or 'require' PHP built-in functions. This practice, though valid, is hackish by nature.

This is why we need to consider the use of a **trait**, a new feature that was introduced into PHP version 5.4, as it allows a class to get its code from more than one trait. In fact, it allows a class to use as many traits as it needs.

How do traits work?

Traits resemble classes in that they group together code elements under a common name, and with the following syntax:

```php
trait TraitName {
    // The trait's code
}
```

In the example given below, a trait with the name of `Price` has a method `changePriceByDollars()` that calculates the new price from the old price and the price change in dollars.

```php
trait Price {
    public function changePriceByDollars($price, $change)
    {
        return $price + $change;
    }
}
```

Once we create a trait, we can use it in other classes with the `use` keyword. In the example given below, both the classes `Bmw` and `Mercedes` use the `Price` trait.

```
class Bmw {
    use Price;
}

class Mercedes {
    use Price;
}
```

In order to see our code in action, let's create objects from the classes and then, let's make use of the `changePriceByDollars()` method that they got from the trait.

```
$bmw1 = new Bmw();

// Add 3000$
echo $bmw1 -> changePriceByDollars(45000, +3000);


$mercedes1 = new Mercedes();

// Subtract 2100$
echo $mercedes1 -> changePriceByDollars(42000, -2100);
```

Result:

48000

39900

This example can teach us that we can use the code from the traits inside the classes in pretty much the same way that we can include a block of code into each of the classes.

Is it possible for a class to use more than one trait?

A class can use more than one trait, after all, this is what traits were invented for.

Let's present to our code another trait with the name of `SpecialSell` that has a method with the name of `annonunceSpecialSell`, and make the `Mercedes` class use both traits.

```php
// The first trait
trait Price {
    public function changePriceByDollars($price, $change)
    {
        return $price + $change;
    }
}


// The second trait
trait SpecialSell {
    public function annonunceSpecialSell ()
    {
        return __CLASS__ . " on special sell";
    }
}


// The Mercedes class uses both traits
class Mercedes {
    use Price;
    use SpecialSell;
}


$mercedes1 = new Mercedes();

// Subtract 2100$
echo $mercedes1 -> changePriceByDollars(42000, -2100);
echo $mercedes1 -> annonunceSpecialSell();
```

Result:

39900

Mercedes on special sell

How is a trait different from inheritance?

Traits use a special form of inheritance that enables them to include the code from the traits in the classes.

- Traits use a form of inheritance that is known as **horizontal inheritance** in which the code from the trait is included in the classes in which it is used. It is pretty much like using 'require' or 'include' in the classes to include code from the outside, albeit not hackish.

- In a trait it is possible to put concrete (real) methods, abstract methods, properties and even constants.
- While the same class can use more than one trait, it can only inherit from one class.
- Traits do not respect the visibility scope, thus allowing a trait's methods to access private properties and methods in the class that uses them.

In the example given below, the trait's method `changePriceByDollars()` is allowed to interact with the private `$price` property.

```php
trait Price {
    // The method needs the $price property.
    public function changePriceByDollars($change)
    {
        return $this -> price += $change;
    }
}


class Mercedes {
    use Price;

    // The $price is private
    private $price;

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

    public function getPrice()
    {
        return $this -> price;
    }
}


$mercedes1 = new Mercedes(42000);
echo $mercedes1 -> getPrice();
$mercedes1 -> changePriceByDollars(-2100);
echo $mercedes1 -> getPrice();
```

Result:

42000

39900

We can see that the trait's methods has access to private methods within the classes that use them.

What are the advantages of using traits?

Traits allow us to use code from more than one resource in a class and, by doing so, we are able to circumvent the limitation of single class inheritance.

What are the disadvantages of using traits?

When using a trait, we should be on the lookout for code duplication and for naming conflicts that are the result of calling the methods in different traits with the same name.

Traits may be easily misused, resulting in a code that looks like a patchwork, instead of consisting of well-designed and well-thought of class hierarchies through the use of inheritance.

In which scenarios is it preferable to use traits?

In some cases, the use of traits can save the day and prove to be preferable to using inheritance. Think of a scenario in which number of classes implement the same interface and so share its code, but only a few of the classes (and not all of them) need a certain piece of that code. It is reasonable to use inheritance for those methods that are shared by all of the classes, while leaving it better of to traits to hold the methods that are needed in only some of the classes. For example, if we have three classes that inherit the `Car` interface (`Mercedes`, `Bmw`, and `Audi`) and we need the method that handles special sell to work in only two of these classes (`Mercedes` and `Bmw`), we will use a trait so that the code will be shared only by the classes that need it.

> The message to take home is that traits are a tool that allows code inclusion from different resources and so could be quite handy when used appropriately.

Conclusion

In this chapter, we learned that **traits** can be used to share code between different classes, but that they are best reserved for only those cases in which the use of inheritance is limited. Such cases include the need for inheritance from several classes, and a piece of code which is shared between several (but not all) of the classes that implement an interface.

In the next chapter, we will learn how to group together related code under a namespace and how we can approach code within a namespace.

Let's practice what we have just learned

13.1 Which of the following keywords is used to define a trait?

A : defines

B : traits

B : trait

13.2 Which of the following types can be found within a trait?

A : abstract methods

B : concrete methods and properties

C : constants

D : all of the above

13.3 Is it appropriate to use traits in a scenario in which all of the child classes of an interface need the same methods?

Coding exercise

In the following exercise, we are going to write a simple program that uses an interface. In it, we are going to have three classes that inherit from the interface User: `Author`, `Commentator`, and `Viewer`. Only the first two classes have the need to use a `Writing` trait.

13.4 Write an interface with the name of `User`.

13.5 Write three classes to implement the `User` interface: `Author`, `Commentator` and `Viewer`.

13.6 Add a trait with the name of `Writing` that contains an abstract method with the name of `writeContent()`.

13.7 Use the trait `Writing` in the `Author` class, and implement its abstract method by making it return the string "Author, please start typing an article…".

13.8 Use the trait `Writing` in the `Commentator` class, and implement its abstract method by making it return the string "Commentator, please start typing your comment…".

13.9 Run the code you have just written and see the result.

Suggested solutions

Namespaces and code integration

As our projects grow in complexity, we have no choice but to integrate code from different resources. It can be code that is written by other programmers in our team or is already present in code libraries that were developed elsewhere, and can do exactly what we need in a fraction of the time that it would take for developing our own library.

In spite of its many benefits, we might encounter some problems when we integrate code from different resources. A huge problem, for example, is that of **name collision**, which occurs when different classes have the same name or have the same name for their methods. The problem is solved, as of PHP 5.3, by using **namespaces**.

In this chapter, we are going to learn how to define a class as belonging to a namespace and, then how to use the namespaced class in our code.

The directory structure

We'll start by creating an **index.php** file in the root directory of our site.

Next, we will create a class with the name of `CarIntro` and place it in the **src/Car** folder, as follows:

/src/Car/CarIntro.php

The following diagram can help us understand the folder structure:

The folder structure
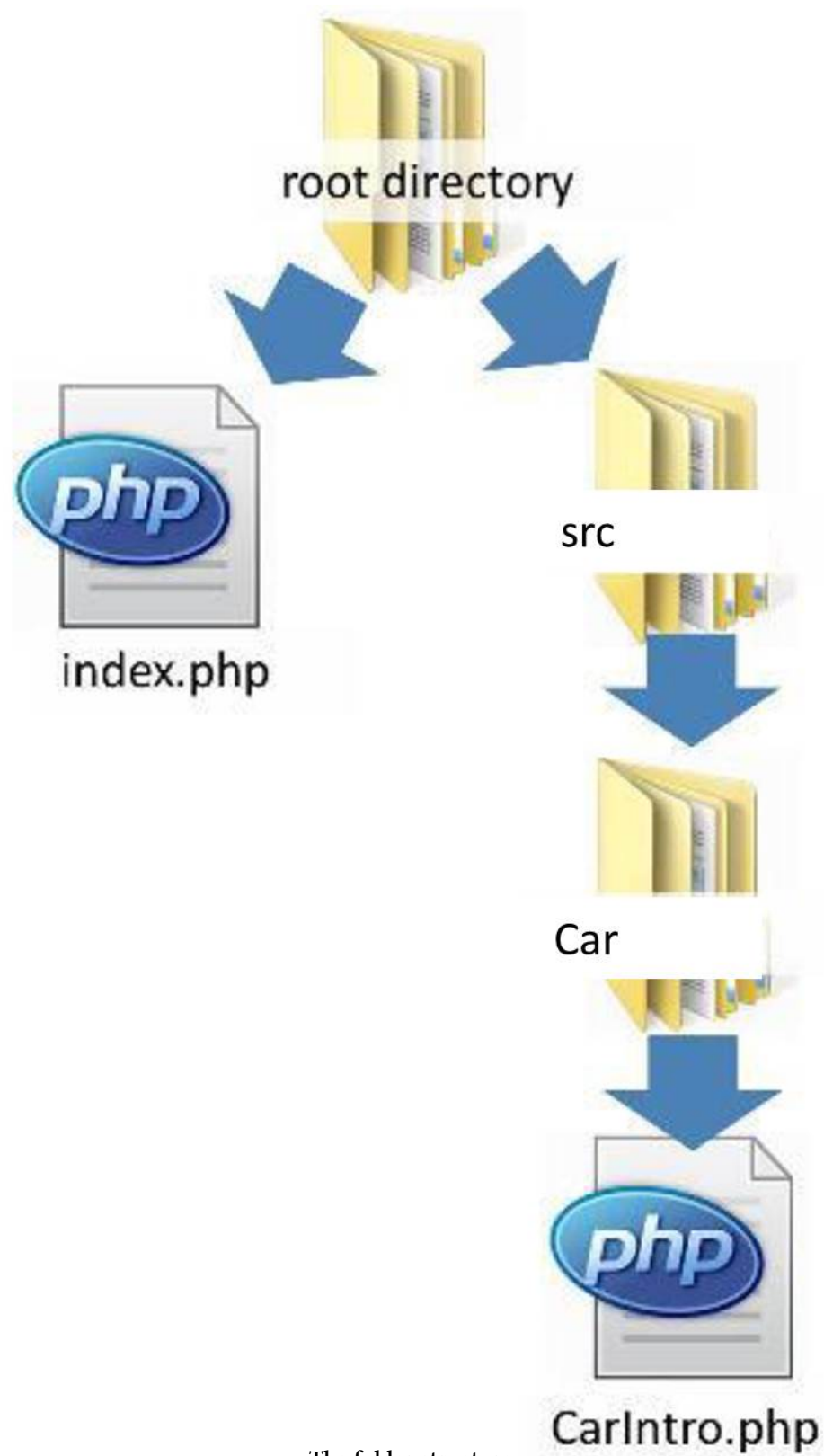
Now, we can write the `CarIntro` class within the **CarIntro.php** file:

```php
class CarIntro {
    public function sayHello()
    {
        return "beep!";
    }
}
```

Pay attention to the fact that we call our classes with the same name as our files. This naming convention would pay off later when we'd like to autoload our classes.

How to define a namespace?

We define the namespace at the very top of the page with the `namespace` keyword.

For example, we will give the class the `Acme\Car` namespace:

```php
<?php  namespace Acme\Car;

class CarIntro {
    public function sayHello()
    {
        return "beep!";
    }
}
```

- It is customary to give to the `src` directory the `Acme` namespace.
- You can replace the `Acme` namespace with your brand name.
- It is a good practice to imitate the directory structure with the namespace. So, every class within the directory: `src/Car` will get the `Acme\Car` namespace.

So far, we have learned how to namespace our classes, and in the next part we'll see how to use the namespaced classes.

How to use a class that belongs to a namespace?

After we have namespaced the `CarIntro` class, we need to some how use it. For instance, we may want to use the class in the **index.php** file. So, in **index.php**, we need to do two things: First, we need to include the library and then import the namespace with the `use` keyword.

```php
<?php
// 1. Require the namespaced file
require "src/Car/CarIntro.php";

// 2. Import the namespace
use Acme\Car\CarIntro;
```

Pay attention that in order to use a class we add the name of the class to the namespace. So, in order to import the CarIntro class from the Acme\Car namespace, we use Acme\Car\CarIntro.

Now, that we have imported the class, we can create an object out of it:

```php
<?php
// 1. Require the namespaced file
require "src/Car/CarIntro.php";

// 2. Import the namespace
use Acme\Car\CarIntro();

// 3. Use the class from the context of the namespace
$carIntro = new Acme\Car\CarIntro();
echo $carIntro -> sayHello();
```

Result:

beep!

How to alias a class from a namespace with a friendly name?

Since calling a class from a namespace might be a drag, we can use only the last part of the namespace in order to call the class.

In the example given below, after including the file and importing the namespace, we call the class from the context of the namespace, but this time by using only the name of the class.

```php
<?php
// 1. Require the file
require "src/Car/CarIntro.php";

// 2. Import the namespace
use Acme\Car\CarIntro;

// 3. Call the class from the context of the namespace,
//      this time by using only the class name
$carIntro = new CarIntro();
echo $carIntro -> sayHello();
```

Result:

beep!

We can also give the class a friendly name (also known as aliasing) by using the as keyword with the following syntax:

```php
use OriginalClassName as AliasName
```

In our example, let's replace the original class name of CarIntro with the friendly name of Intro, and then use this friendly name when working with the class methods.

```php
<?php
// 1. Require the file
require "src/Car/CarIntro.php";

// 2. Import the namespace and give it a friendly name
use Acme\Car\CarIntro as Intro;

// 3. Call the class from the namespace with the friendly name
$carIntro = new Intro();
echo $carIntro -> sayHello();
```

Result:

beep!

How to call a class from the global namespace?

Prior to PHP 5.3, there was no such thing as a namespace in PHP, so we may very well encounter some libraries that are written without the use of namespaces and with their code in the **global namespace**.

For example, let's now add to our code, a file with the name of **CarPrice.php** which is found in the global scope since we haven't given it a namespace. The file holds the code for the CarPrice class and is found in the following path:

src/Car/CarPrice.php

```php
<?php
class CarPrice {
    public function getQuote()
    {
        return 19500;
    }
}
```

This is a diagram that depicts the folder structure after we have added the new class to the **Car** directory:
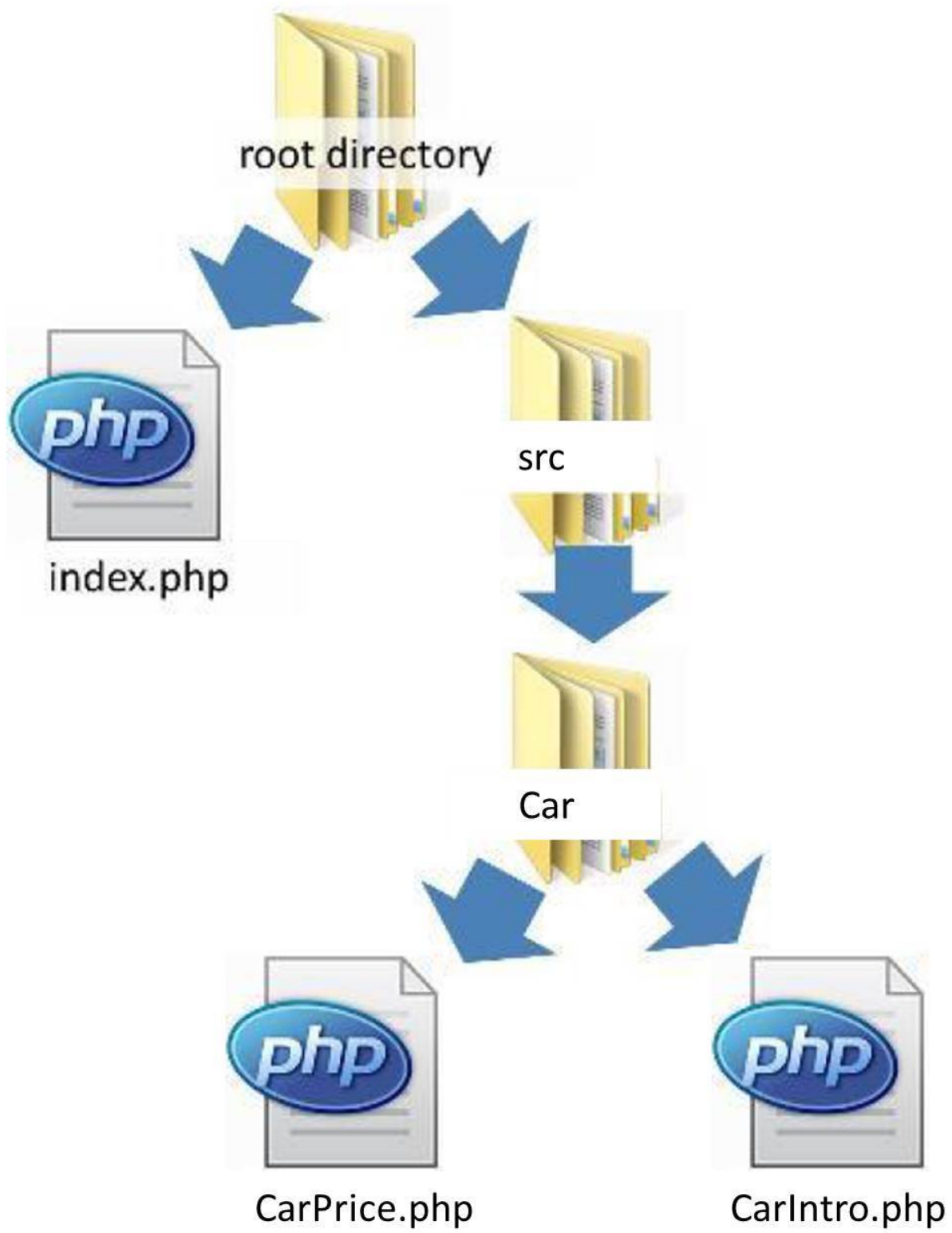
The folder structure

In order for classes from the global scope to be used securely within a namespace, we need to precede the name of such a class with a backslash.

For example, in the **index.php** file we need to put a backslash in front of the `CarPrice` class because we want to use it from the global namespace:

```php
require "src/Car/CarIntro.php";
require "src/Car/CarPrice.php";

use Acme\Car\CarIntro as Intro;

$carIntro = new Intro();
echo $carIntro -> sayHello();

// We use the backslash in front of the class name in order to call a class
//   from the global namespace
$carPrice = new \CarPrice();
echo $carPrice -> getQuote();
```

Result:

beep!

19500

- Some other common instances in which there is a need to use the backslash before the class name are for the built in PHP extensions PDO and DateTime, and for PHP's generic empty class, stdClass.

Can more than one namespace be imported into the same file?

Each class that we import needs to have its own namespace.

Each class that we need to import needs to have its own namespace. In our example, let's add to the class `CarPrice` the namespace `Acme\Car`.

src/Car/CarPrice.php:

```php
<?php
namespace Acme\Car;

class CarPrice {
    public function getQuote()
    {
        return 19500;
    }
}
```

All the classes within the same directory need to have the same namespace.

Since we follow the good practice of giving the classes a namespace which follows the directory structure all the classes within the same directory need to have the same namespace. In our example, all the classes in the **src/Car** directory have the Acme\Car namespace.

We can now use the CarPrice class in **index.php**, alongside the CarIntro class:

```php
// 1. Require both classes
require "src/Car/CarIntro.php";
require "src/Car/CarPrice.php";

// 2. Import the namespaces and give them a friendly name
use Acme\Car\CarIntro as Intro;
use Acme\Car\CarPrice as Price;

// 3. Use the methods in the context of their namespace
$carIntro = new Intro();
echo $carIntro -> sayHello();

$carPrice = new Price();
echo $carPrice -> getQuote();
```

Result:

beep!

19500

Conclusion

In this chapter, we learned how to give classes a **namespace**, and how to use the the namespaced classes in our scripts.

In the next chapter, we will learn how to loosen the coupling between classes by using **dependency injection**.

Let's Practice what we have just learned

14.1 What are the main reasons for using namespaces?

A : In order to confuse novices.

B : To avoid name collision.

C : To group related classes, interfaces, functions and constants.

D : B+C

14.2 Which keyword is used to define a namespace?

A : use

B : namespace

C : namespaces

D : none of the above

14.3 Which keyword is used to import a namespace?

A : use

B : namespace

C : namespaces

D : none of the above

Suggested solutions

Appendix: Group use declarations in PHP7

In PHP7, we can group use declarations. When we group use declarations, we utilize the keyword use only once in order to import all of the classes, instead of using it repeatedly for each class that we want to import.

In PHP7, we can still import the classes in the traditional way:

```php
use Acme\Car\CarIntro as Intro;
use Acme\Car\CarPrice as Price;
```

However, we can also import the classes with only one use:

```php
use Acme\{
Car\CarIntro as Intro ,
Car\CarPrice as Price
};
```

And if we would like to also import a user class, we can still do so with the traditional syntax:

```php
use Acme\Car\CarIntro as Intro;
use Acme\Car\CarPrice as Price;
use Acme\User;
```

But also the PHP7 way:

```php
use Acme\{
Car\CarIntro as Intro ,
Car\CarPrice as Price
User
};
```

Group use declarations is an alternative syntax that was introduced with PHP7, and it will not damage or break your code in any way.

Dependency injection

One of the less understood subjects for fresh PHP programmers is that of **dependency injection**. But don't worry! This chapter will explain the subject to you in the simplest way possible by first explaining the problem and then showing you how to solve it.

The problem: tight coupling between classes

When class **A** cannot do its job without class **B**, we say that class **A** is dependent on class **B**. In order to perform this dependency, many programmers create the object from class **B** in the constructor of class **A**.

For example, if we have the classes of `Car` and `HumanDriver`, the `Car` class is dependent on the `HumanDriver` class, so we might be tempted to create the object from the `HumanDriver` class in the constructor of the `Car` class.

```php
class HumanDriver {
    // Method to return the driver name.
    public function sayYourName($name)
    {
        return $name;
    }
}



// Car is dependent on HumanDriver
class Car {
    protected $driver;

    // We create the driver object in the constructor,
    //   and use this object to populate the $driver variable.
    public function __construct()
    {
        $this->driver = new HumanDriver();
    }

    // A getter method that returns the driver object
    //   from within the car object
```

```
    public function getDriver()
    {
        return $this -> driver;
    }
}
```

Although creating the human driver object inside the Car class might look like the right thing to do, we are going to face a real problem the first time that we need to switch dependencies. For example, if the technological advancements dictate us a car that is driven by a robot, we will find our self in a problem since the Car class only knows how to handle human drivers.

The problem that stems from directly creating the driver object inside the Car class is also known as **tight coupling** between classes, something that should be avoided as much as possible. In fact, **tight coupling** is considered to be a bad practice for the following reasons:

- The programmers need to be concerned with more than one class when they introduce new classes to their code. In our example, if we would like to change the class on which the Car class is dependent from HumanDriver to RobotDriver, except for introducing the new class to our code, we also need to change the Car class so it can handle the new type of driver.
- We will have difficulties in performing unit testing since it is meant to check one class at a time.

In fact, when we do **tight coupling** between classes, we violate a fundamental principle of well designed code called the **"single responsibility principle"** (SRP), according to which a class should have only one responsibility. In order to respect this principle, the only code that the Car class needs to handle is that of cars, without messing with any other code.

The solution: dependency injection

Now that we understand why creating an object from one class inside the constructor of another class is unfavorable, let's see how to fix the problem by using **dependency injection**.

First, let's rewrite the Car class so it can set its own $driver property that is passed as a parameter to the constructor.

```php
// The Car class gets the driver object injected
//   to its constructor
class Car {
    protected $driver;

    // The constructor sets the value of the $driver
    public function __construct($driver)
    {
        $this -> driver = $driver;
    }

    // A getter method that returns the driver object
    //   from within the car object
    public function getDriver()
    {
        return $this -> driver;
    }
}
```

We can now perform the **dependency injection** by first creating the Driver object and then injecting this object into the newly created Car object through the constructor.

```php
$humanDriver = new HumanDriver();
$car = new Car($humanDriver);
```

And Voila! We just performed **dependency injection**.

Let's test the code:

```php
$humanDriver = new HumanDriver();
$car = new Car($humanDriver);
echo $car -> getDriver() -> sayYourName("Bob");
```

Result:

Bob

Now that we have seen how dependency injection works, let's see how easy it is to switch the dependency from HumanDriver to RobotDriver, saving us from changing the code in the Car class.

First, let's write the RobotDriver class with the sayYourName method.

```php
class RobotDriver {

    public function sayYourName($name)
    {
        return $name;
    }
}
```

We can leave the Car class without any changes since it depends on a driver but does not require any commitment to the type of the driver. The driver can be a robot, human, alien, etc. It really doesn't matter as long as we supply a driver object.

```php
class Car {
    protected $driver;

    // The constructor sets the value of the $driver
    public function __construct($driver)
    {
        $this -> driver = $driver;
    }

    // A getter method that returns the driver object
    //   from within the car object
    public function getDriver()
    {
        return $this -> driver;
    }
}
```

Now, we can create the driver object from the RobotDriver and inject it to the Car class without too much of a hassle.

```php
$robotDriver = new RobotDriver();
$car     = new Car($robotDriver);
echo $car -> getDriver() -> sayYourName("R2-D2");
```

Result:

R2-D2

> It is easy to switch dependencies if only we adopt the good practice of injecting the objects from the outside.

The message to take home from this example would be that it is easy to switch dependencies if only we avoid the bad practice of creating the objects on which the main class depends inside the constructor, and adopt the good practice of injecting the objects on which the class depends from the outside.

Why is it a good idea to type hint the injected objects?

Though type hinting is not necessary for dependency injection, it is favorable to use it in order to make it clear on which type of objects does the main class depend. In our example, we expect any driver that uses the Car object to also have a sayYourName method. For this purpose, let's create a Driver interface, and make both the humanDriver as well as the robotDriver classes implement the interface.

```php
// The Driver interface
interface Driver {
        public function sayYourName($name);
}

// The HumanDriver implements the interface
class HumanDriver implements Driver {
    public function sayYourName($name)
    {
        return $name;
    }
}

// The RobotDriver implements the interface
class RobotDriver implements Driver {
    public function sayYourName($name)
    {
        return $name;
    }
}
```

Now, we need to type hint the $driver parameter that we want to inject to the Car's constructor since we want the injected object to implement the Driver interface.

```php
// The Car class gets the driver object injected
//   to its constructor
class Car {
    protected $driver;

    // The constructor sets the value of the $driver
    public function __construct(Driver $driver)
    {
        $this -> driver = $driver;
    }

    public function getDriver()
    {
            return $this -> driver;
    }
}
```

Let's test everything by making the Car's drivers say their names for both a human driver as well as a robot driver.

```php
$humanDriver = new HumanDriver();
$car = new Car($humanDriver);
echo $car -> getDriver() -> sayYourName("Bob");

$robotDriver = new RobotDriver();
$car = new Car($robotDriver);
echo $car -> getDriver() -> sayYourName("R2-D2");
```

And the result is:

Bob

R2-D2

> It is favorable to type hint the injected objects in order to make it clear on which type of objects the main class is dependent.

The take-home message is that it is favorable to type hint the injected objects in order to make it clear on which type of objects the main class is dependent.

How to make a dependency injection through a setter method?

We may also choose to inject the objects on which the class is dependent through a setter method.

Let's start again by writing the HumanDriver class with exactly the same code as in the previous example.

```php
interface Driver {
        public function sayYourName($name);
}

class HumanDriver implements Driver {

    public function sayYourName($name)
    {
        return $name;
    }
}
```

We will equip the Car class with a setDriver() method that is responsible for setting the $driver value.

```php
class Car {
    protected $driver;

    // The setDriver method sets
    //   the value for the driver property
    public function setDriver(Driver $driver)
    {
        $this -> driver = $driver;
    }

    public function getDriver()
    {
        return $this -> driver;
    }
}
```

Now, we can create the driver object outside the car, inject it to the car with the setDriver() method, and echo the driver's name.

```php
$driver = new HumanDriver();
$car    = new Car();

// Inject the driver to the car object
$car -> setDriver($driver);

echo $car -> getDriver() -> sayYourName("Bob");
```

Result:

Bob

Conclusion

> When we use **dependency injection** we pass the objects on which our class depends to the class from the outside.

When we use **dependency injection** we pass the objects on which our class depends to the class from the outside, instead of creating these objects within the class. By doing so, we make our code more maintainable and flexible. It is also advisable to type hint the injected objects in order to make it clear on which type of objects does the main class depend.

Let's practice what we have just learned

In this exercise, we are going to write an `Article` class that is dependent upon an `Author` class, and introduce the author object through dependency injection into the constructor.

The `Article` class has the following properties: `$title`, for the title of the article, and `$Author`, to hold the author object.

```php
class Article {
    protected $title;
    protected $author;
}
```

15.1 Add to the `Article` class, a constructor method that gets the parameters of: `$title` and `$author` and sets the class's properties accordingly.

15.2 Write to the class, a getter method that returns the article's title.

15.3 Add to the class, a getter method that returns the author's object.

15.4 Write another class with the name of `Author`, that has a protected property `$name` and setter and getter methods that handle the property.

15.5 Now, first create the `Author` object, call it `$author1`, and then set its name to 'Joe'.

15.6 Create the article object, call it `$article1`, and pass to it the title "To PHP and Beyond" and the `$author1` object that you have just created.

15.7 Write the code that returns the string: "To PHP and Beyond by Joe".

15.8 Improve your code by type hinting the $author object that you pass as a parameter to the article's constructor. **Clue:** You need to first create an Authors interface, make the Author class implement the interface, and then type hint the $author object that you inject to the article's constructor.

Suggested solutions

What are exceptions and how to handle them?

Exception handling is an elegant way to handle errors which are beyond the program's scope. For example, if our application fails to contact the database, we use exception handling to contact another data source or to provide instructions to the users that they need to contact technical support.

> **Exception handling** is an elegant way to handle errors which are beyond the program's scope.

In this chapter, we are going to learn how and when to use exception handling and, even more important, when not to use it.

In order to explain the need for exception handling, we will try to see what happens when a program is fed with faulty data. In our example, we will develop the class `FuelEconomy` that calculates with the method `calculate()` the fuel efficiency of cars by dividing the distance traveled by the gas consumption.

```php
class FuelEconomy {
    // Calculate the fuel efficiency
    public function calculate($distance, $gas)
    {
        return $distance/$gas;
    }
}
```

We will feed the class with the nested array `$dataFromCars`, in which each nested array has two values: the first value is for the distance traveled and the second value is for the gas consumption. While the first and last nested arrays contain legitimate data, the second nested array provides zero gas consumption and the third nested array has negative gas consumption.

```php
$dataFromCars = array(
    array(50,10),
    array(50,0),
    array(50,-3),
    array(30,5)
);
```

Here is what happens when we feed the array to the class:

```php
foreach($dataFromCars as $data => $value)
{
    $fuelEconomy = new FuelEconomy();
    echo $fuelEconomy -> calculate($value[0],$value[1]) . "<br />";
}
```

Result:

5

 Warning: Division by zero

-16.666666666667

6

From the result, we can see what might happen when we feed the class with faulty data. On the second line, we got a nasty warning for trying to divide by zero while, on the third line, we got a negative value (which is not what we would expect for gas efficiency). These are the kinds of errors that we would like to suppress by using exception handling.

How to throw an exception?

When handling exceptions, we make use of the `Exception` class, which is a built-in PHP class. We throw the exception with the command `throw new Exception()` that creates the exception object. Between the brackets of the `throw new Exception()` command, we write our own custom message that we would like to appear in the case of the exception.

In the example given below, whenever the user tries to feed the class with a gas consumption value that is less than or equal to zero, a custom message is thrown instead of an error.

```php
class FuelEconomy {
    // Calculate the fuel efficiency
    public function calculate($distance, $gas)
    {
        if($gas <= 0 )
        {
            // Throw custom error message, instead of an error
            throw new Exception("The gas consumption cannot be less than
                or equal to zero. You better drink coffee or take a break.");
        }

        return $distance/$gas;
    }
}
```

If we run the code as it is written now, we'll get an even nastier error because, in order to handle exceptions, we need to not just throw the exception but also to catch it.

How to catch an exception?

In order catch an exception, we need two blocks:

1. A **try block**.
2. A **catch block**.

While the **try block** runs the normal data, the faulty data is handled by the **catch block**. Inside the catch block, we use methods that the PHP Exception class provides.

In the example given below, the custom error message that we throw from the **try block** is being caught by the **catch block** that, in turn outputs the error message with the getMessage() method (which is a method that the Exception class provides).

```php
class FuelEconomy {
    // Calculate the fuel efficiency
    public function calculate($distance, $gas)
    {
        if($gas <= 0 )
        {
            // Throw custom error message, instead of an error
            throw new Exception("The gas consumption cannot be less than
                or equal to zero. You better drink coffee or take a break.");
```

```php
        }

            return $distance/$gas;
    }
}

// The data to feed the class with
$dataFromCars = array(
    array(50,10),
    array(50,0),
    array(50,-3),
    array(30,5)
);

// Create the object from the class and feed it
//   with the array values
foreach($dataFromCars as $data => $value)
{
    // Try block handles the normal data
    try
    {
        $fuelEconomy = new FuelEconomy();
        echo $fuelEconomy -> calculate($value[0],$value[1]) . "<br />";
    }
    // Catch block handles the exceptions
    catch (Exception $e)
    {
        // Echo the custom error message
        echo "Message: " . $e -> getMessage() . "<br />";
    }
}
```

Result:

5

Message: The gas consumption cannot be less than or equal to zero. You better drink coffee or take a break.

Message: The gas consumption cannot be less than or equal to zero. You better drink coffee or take a break.

6

Thanks to exception handling, we now get an elegant error message instead of the nasty one that we got when not using exception handling.

Are there any other methods that can help us handle exceptions?

The Exception class is a PHP built-in class that offers several helpful methods. We have already used the getMessage() method to get a custom error message, yet there are other beneficial methods that the Exception class provides. Of those, the most beneficial are:

- getFile() that returns the path to the file in which the error occurred.
- getLine() that returns the line in which the exception occurred.

We can now use the getFile() and getLine() methods to add the path and line in which the exception has occurred to the custom message from the above example.

```php
foreach($dataFromCars as $data => $value)
{
    try
    {
        $fuelEconomy = new FuelEconomy();
        echo $fuelEconomy -> calculate($value[0],$value[1]) . "<br />";
    }
    catch (Exception $e)
    {
        echo "Message: " . $e -> getMessage() . "<br />";

        // Output the path to the file
        echo "File: " . $e -> getFile() . "<br />";

        // Output the line in the code
        echo "Line: " . $e -> getLine() . "<br />";
        echo "<hr />";
    }
}
```

Result:

5

Message: The gas consumption cannot be less than or equal to zero. You better drink coffee or take a break.

File: C:\wamp\www\index.php

Line: 9

Message: The gas consumption cannot be less than or equal to zero. You better drink coffee or take a break.

File: C:\wamp\www\index.php

Line: 9

6

The result above demonstrates that exception handling can provide more than custom messages. It can provide beneficial data for tracing the exceptions.

What about writing the exceptions to a log file?

In the same way that we can echo the exception messages to the user, we can also write them into a log file with the error_log() command. For example:

```php
error_log($e -> getFile());
```

In our example, let's now add the ability to write errors into the log file whenever an exception is thrown:

```php
foreach($dataFromCars as $data => $value)
{
    try
    {
        $fuelEconomy = new FuelEconomy();
        echo $fuelEconomy -> calculate($value[0],$value[1]) . "<br />";
    }
    catch (Exception $e)
    {
        // Echo the error message to the user
        echo "Message: " . $e -> getMessage() . "<br />";
        echo "<hr />";

        // Write the error into a log file
        error_log($e -> getMessage());
        error_log($e -> getFile());
        error_log($e -> getLine());
    }
}
```

When to use exception handling?

> The sole purpose of exception handling is to handle external problems, such as math errors, logical errors, or technical difficulties.

Exception handling is a great tool that handles specific errors that are outside the scope of a program. These errors might be math errors, logical errors or technical difficulties, examples of which are as follows:

- Math errors, e.g., trying to divide by zero.
- Logical errors happen when trying to do something that doesn't make sense in the context of the program (like assigning a negative value to gas consumption).
- Technical problems are most often caused by hardware failure (like the computer running out of memory) but they can also be the fault of external services (like communication failures or problems with an external API).

It is important to stress that, although exception handling is a great tool for handling errors which are outside the scope of the program in an elegant way, it is not a way to hide bugs or badly written code. It is also not a fancy way to replace if-then and switch statements. The sole purpose of exception handling is to handle external problems, such as math errors, logical errors, or technical difficulties.

Conclusion

In this chapter, we have learned that **exceptions** are problems that a program might encounter but is not meant to solve, and that exceptions are best handled by try - catch blocks that catch, throw, and handle the exceptions with the `Exception` class.

Let's practice what we've just learned:

16.1 In which of the following cases would you use exception handling?

A : To suppress errors and bugs in a badly written code.

B : When trying to handle an external error which the program cannot or is not meant to handle, like losing the database connection.

C : Instead of if..then when you try to impress your boss and colleagues with your object oriented programming skills.

Coding example

So far, in this chapter, we have seen how to handle a single exception. Now, in the following exercise, we'll handle more than one exception.

This is the User class with the methods that set and get the user name and age.

```php
class User {
    private $name;
    private $age;

    public function setName($name)
    {
        // Trim the white spaces
        $name = trim($name);
        $this -> name = $name;
    }

    public function setAge($age)
    {
        // Cast into an integer type
        $age = (int)$age;
        $this -> age = $age;
    }

    public function getName()
    {
        return $this -> name;
    }
```

```php
    public function getAge()
    {
        return $this -> age;
    }
}
```

16.2 Add to the `setName()` method a code that throws exception whenever the user's name is shorter than 3 letters. What is a suitable message that can be thrown?

16.3 Add to the `setAge()` method a code that throws exception whenever the user's age is less than or equal to 0.

This is the nested array that you can feed the class with.

```php
$dataForUsers = array(
    array("Ben",4),
    array("Eva",28),
    array("li",29),
    array("Catie","not yet born"),
    array("Sue",1.5)
);
```

16.4 Write a foreach loop (like the one we have already used in this chapter) that feeds the array to the class, and handles the exceptions by echoing the custom error messages as well as the file path and line number in which the exception was thrown.

16.5 Run the code and see the result.

Suggested solutions

PDO - PHP database extension

PDO (PHP Data Objects) is a PHP extension through which we can access and work with databases. Though **PDO** is similar in many aspects to mySQLi, it is better to work with for the following reasons:

- It is better protected against hackers.
- It is consistent across databases, so it can work with MySQL as well as other types of databases (SQLite, Oracle, PostgreSQL, etc.)
- It is object oriented at its core.

In this chapter you will find recipes for 4 basic functions that we perform with the database: insertion, selection, update, and deletion. The recipes are intended to work with MySQL, but we can easily switch it with another database.

How to connect with the MySQL database through PDO?

It is considered good practice to wrap the database connection within a try-catch block so that, if anything goes wrong, an exception will be thrown. We can customize the error message but, in order to keep things simple, we'll settle with the error message that **PDO** provides.

In order to connect to the database, we'll need the database name, username, and password.

```php
// DB credentials.
define('DB_HOST','localhost');
define('DB_USER','your user name');
define('DB_PASS','your user password');
define('DB_NAME','your database name');

// Establish database connection.
try
{
    $dbh = new PDO("mysql:host=".DB_HOST.";dbname=".DB_NAME,
    DB_USER, DB_PASS,
    array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES 'utf8'"));
}
```

```
catch (PDOException $e)
{
    exit("Error: " . $e->getMessage());
}
```

* You can find the code for the sql table that is used in the examples in this chapter's supplement section.

How to use PDO to insert data into the database?

1) Write a regular SQL query but, instead of values, put named placeholders. For example:

```
$sql = "INSERT INTO `users`
        (`name`, `phone`, `city`, `date_added`)
        VALUES
        (:name,:phone,:city,:date)";
```

The use of placeholders is known as **prepared statements**. We use prepared statements as templates that we can fill later on with actual values.

2) Prepare the query:

```
$query = $dbh -> prepare($sql);
```

3) Bind the placeholders to the variables:

```
$query->bindParam(':name',$name);
```

You can add a third parameter which filters the data before it reaches the database:

```
$query->bindParam(':name',$name,PDO::PARAM_STR);
$query->bindParam(':phone',$phone,PDO::PARAM_INT);
$query->bindParam(':city',$city,PDO::PARAM_STR);
$query->bindParam(':date',$date,PDO::PARAM_STR);
```

- PDO::PARAM_STR is used for strings.
- PDO::PARAM_INT is used for integers.
- PDO::PARAM_BOOL allows only boolean (true/false) values.
- PDO::PARAM_NULL allows only NULL datatype.

4) Assign the values to the variables.

```php
$name  = "Joe";
$phone = "1231234567";
$city  = "New York";
$date  = date('Y-m-d');
```

5) Execute the query:

```php
$query -> execute();
```

6) Check that the insertion really worked:

```php
$lastInsertId = $dbh->lastInsertId();

if($lastInsertId>0)
{
    echo "OK";
}
else
{
    echo "not OK";
}
```

If the last inserted id is greater than zero, the insertion worked.

All together now:

```php
$sql = "INSERT INTO `users`
        (`name`, `phone`, `city`, `date_added`)
        VALUES
        (:name,:phone,:city,:date)";

$query = $dbh -> prepare($sql);

$query->bindParam(':name',$name,PDO::PARAM_STR);
$query->bindParam(':phone',$phone,PDO::PARAM_INT);
$query->bindParam(':city',$city,PDO::PARAM_STR);
$query->bindParam(':date',$date);

// Insert the first row
$name  = "Joe";
$phone = "1231234567";
$city  = "New York";
```

```php
$date   = date('Y-m-d');

$query -> execute();

$lastInsertId = $dbh->lastInsertId();

if($lastInsertId>0) { echo "OK"; } else { echo "not OK"; }


// Insert the second row
$name  = "Joseph";
$phone = "037234561";
$city  = "Tel Aviv";
$date  = date('Y-m-d');

$query -> execute();

$lastInsertId = $dbh->lastInsertId();

if($lastInsertId>0){echo "OK";} else {echo "not OK";}
```

The same task is better performed within a loop:

```php
$sql = "INSERT INTO `users`
        (`name`, `phone`, `city`, `date_added`)
        VALUES
        (:name,:phone,:city,:date)";

$query = $dbh -> prepare($sql);

$query->bindParam(':name',$name,PDO::PARAM_STR);
$query->bindParam(':phone',$phone,PDO::PARAM_INT);
$query->bindParam(':city',$city,PDO::PARAM_STR);
$query->bindParam(':date',$date,PDO::PARAM_STR);


// Provide the data to the loop within an array
$date = date('Y-m-d');
$userData = array(
    array("Joe","1231234567","New York",$date),
    array("Joseph","037234561","Tel Aviv",$date)
);
```

```php
// Perform the query within a loop
foreach($userData as $key => $value)
{
    $name = $value[0];
    $phone = $value[1];
    $city = $value[2];
    $date = $value[3];

    $query -> execute();

    $lastInsertId = $dbh->lastInsertId();

    if($lastInsertId>0){echo "OK";} else {echo "not OK";}
}
```

How to use PDO to read from the database?

Reading data from the database is not so different than inserting data, with steps 1 to 5 being almost identical while the sixth step is different. Then there are two more steps (7 and 8).

1) Write the regular select statement and again, instead of values, put named placeholders. For example:

```
$sql = "SELECT * FROM users WHERE city = :city";
```

2) Prepare the query:

```
$query = $dbh -> prepare($sql);
```

3) Bind the parameters:

```
$query -> bindParam(':city', $city, PDO::PARAM_STR);
```

4) Define the bound value:

```
$city = "New York";
```

5) Execute the query:

```
$query -> execute();
```

6) Assign the data which you pulled from the database (in the preceding step) to a variable.

```
$results = $query -> fetchAll(PDO::FETCH_OBJ);
```

Here I used the parameter PDO::FETCH_OBJ that returns the fetched data as an object. If you'd like to fetch the data in the form of an array, use: PDO::FETCH_ASSOC.

7) Make sure that you were able to retrieve the data from the database, by counting the number of records.

```
if($query -> rowCount() > 0){}
```

8) In case that the query returned at least one record, we can echo the records within a foreach loop:

```php
if($query -> rowCount() > 0)
{
    foreach($results as $result)
    {
        echo $result -> name . ", ";
        echo $result -> city . ", ";
        echo $result -> date_added;
    }
}
```

All together now:

```php
$sql = "SELECT * FROM users WHERE city = :city";

$query = $dbh -> prepare($sql);

$query -> bindParam(':city', $city, PDO::PARAM_STR);

$city = "New York";

$query -> execute();

$results = $query -> fetchAll(PDO::FETCH_OBJ);

if($query -> rowCount() > 0)
{
    foreach($results as $result)
    {
        echo $result -> name . ", ";
        echo $result -> city . ", ";
        echo $result -> date_added;
    }
}
```

How to use PDO to update the database?

1) Write the regular update statement and again, instead of values, assign the named placeholders. For example:

```
$sql = "UPDATE `users`
SET `city`= :city, `phone` = :tel
WHERE `id` = :id";
```

2) Prepare the query:

```
$query = $dbh->prepare($sql);
```

3) Bind the parameters:

```
$query -> bindParam(':city', $city, PDO::PARAM_STR);
$query -> bindParam(':tel' , $tel , PDO::PARAM_INT);
$query -> bindParam(':id'  , $id  , PDO::PARAM_INT);
```

4) Define the bound values:

```
$tel  = '06901234567';
$city = 'Paris';
$id   = 1;
```

5) Execute the query:

```
$query -> execute();
```

6) Check that the query has been performed and that the database has been successfully updated.

```php
if($query -> rowCount() > 0)
{
    $count = $query -> rowCount();
    echo $count . " rows were affected.";
}
else
{
    echo "No affected rows.";
}
```

All together now:

```php
$sql = "UPDATE users
SET `city`= :city, `phone` = :tel
WHERE `id` = :id";

$query = $dbh->prepare($sql);

$query -> bindParam(':city', $city, PDO::PARAM_STR);
$query -> bindParam(':tel' , $tel , PDO::PARAM_INT);
$query -> bindParam(':id'  , $id  , PDO::PARAM_INT);

$tel = '02012345678';
$city = 'London';
$id = 1;

$query -> execute();

if($query -> rowCount() > 0)
{
    $count = $query -> rowCount();
    echo $count . " rows were affected.";
}
else
{
    echo "No affected rows.";
}
```

How to delete records?

1) Write the delete statement:

```
$sql = "DELETE FROM `users` WHERE `id`=:id";
```

2) Prepare the query:

```
$query = $dbh -> prepare($sql);
```

3) Bind the parameters:

```
$query -> bindParam(':id', $id, PDO::PARAM_INT);
```

4) Define the bound values:

```
$id = 1;
```

5) Execute the query:

```
$query -> execute();
```

6) Check that the query has been performed and that the records have been successfully deleted from the database.

```
if($query -> rowCount() > 0)
{
    $count = $query -> rowCount();
    echo $count . " rows were affected.";
}
else
{
    echo "No affected rows.";
}
```

All together now:

```php
$sql   = "DELETE FROM `users` WHERE `id`=:id";

$query = $dbh -> prepare($sql);

$query -> bindParam(':id', $id, PDO::PARAM_INT);

$id = 1;

$query -> execute();

if($query -> rowCount() > 0)
{
    $count = $query -> rowCount();
    echo $count . " rows were affected.";
}
else
{
    echo "No affected rows.";
}
```

How to close the database connection?

PHP automatically closes the database connection but, if the need arises, we can deliberately close the connection with the following line of code:

```php
$dbh = null;
```

Conclusion

Now that you've seen how easy it is to work with **PDO**, I hope that you'll consider using it in your projects instead of the outdated mysql and mysqli extensions.

Let's practice what we have just learned

Coding example

In our coding example we'll integrate the User class with a database class.

17.1 Write a class with the name of Db that can connect with the database.

The class will be able to create the connection with the database, and will have a method that can get the connection for the use of other classes.

17.2 Write a User class that can get the database connection, and save it into a private property with the name of $dbCon.

17.3 Write into the User class the following methods that can work with the database:

insert method to insert a new user to the users table.

getAll that returns all the users.

getUserById to return only a single user.

updateUser to update user by id.

and a delete method to delete a single user.

17.4 Test your code.

Suggested solutions

Supplement

The SQL code for the users table:

```sql
CREATE TABLE IF NOT EXISTS users (
  id int(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  name varchar(60) DEFAULT NULL,
  phone varchar(14) DEFAULT NULL,
  city varchar(60) DEFAULT NULL,
  date_added date DEFAULT NULL,
  PRIMARY KEY (id)
)
```

How to use Packagist and Composer to integrate existing code libraries into your PHP apps?

Are you tired of having to reinvent the code in any project in which you work? Would you like to know where the PHP community stores all the code libraries that are written worldwide? Have you ever wondered where you can get a code library to manage your website users or to handle your shopping cart? Would you rather write less code and still achieve more?

> While **Packagist** is a huge repository of PHP code libraries, **Composer** is a dependency manager which allows us to download the code libraries from **Packagist** straight to our project.

If you are a PHP programmer and answered yes to at least one of these questions, you should familiarize yourself with **Packagist** and **Composer** that allow us to easily find PHP libraries, download them to our website, and manage them as well. While **Packagist** is a huge repository of PHP code libraries, **Composer** is a dependency manager which allows us to automatically download the code libraries from **Packagist** straight to our project along with all of their dependencies (the code packages on which they are dependent).

How to install composer?

Installing **Composer** on Linux and Mac is done via the command line and is pretty damn easy:

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

As you can see in the following link: getcomposer[1].

Installing Composer on Windows machines is also not particularly difficult using a dedicated Wizard (see the documentation[2]).

---

[1]https://getcomposer.org
[2]https://getcomposer.org

We examine the installation by typing the following command into the command line:

```
$ composer
```

If **Composer** is properly installed, we'll see all the composer commands with a short description of what they do.


```
$ composer


   _____
  / ____/___  ____ ___  ____  ____  _____  _____
 / /   / __ \/ __ `__ \/ __ \/ __ \/ ___/ _ \/ ___/
/ /___/ /_/ / / / / / / /_/ / /_/ (__  )  __/ /
\____/\____/_/ /_/ /_/ .___/\____/____/\___/_/

Composer version 1.0-dev (4ca9e602a9a0ef7639ad143b6366b71638c77029) 2015-01-16 20:51:31

Usage:
 [options] command [arguments]

Options:
 --help (-h)          Display this help message.
 --quiet (-q)         Do not output any message.
 --verbose (-v|vv|vvv) Increase the verbosity of messages: 1 for normal output, 2 for more verbose o
utput and 3 for debug.
 --version (-V)       Display this application version.
 --ansi               Force ANSI output.
 --no-ansi            Disable ANSI output.
 --no-interaction (-n) Do not ask any interactive question.
 --profile            Display timing and memory usage information
 --working-dir (-d)   If specified, use the given directory as working directory.

Available commands:
 about          Short information about Composer
 archive        Create an archive of this composer package
 browse         Opens the package's repository URL or homepage in your browser.
 clear-cache    Clears composer's internal package cache.
 clearcache     Clears composer's internal package cache.
 config         Set config options
 create-project Create new project from a package into given directory.
 depends        Shows which packages depend on the given package
 diagnose       Diagnoses the system to identify common errors.
 dump-autoload  Dumps the autoloader
 dumpautoload   Dumps the autoloader
 global         Allows running commands in the global composer dir ($COMPOSER_HOME).
 help           Displays help for a command
 home           Opens the package's repository URL or homepage in your browser.
 info           Show information about packages
 init           Creates a basic composer.json file in current directory.
 install        Installs the project dependencies from the composer.lock file if present, or falls
back on the composer.json.
 licenses       Show information about licenses of dependencies
 list           Lists commands
 remove         Removes a package from the require or require-dev
 require        Adds required packages to your composer.json and installs them
 run-script     Run the scripts defined in composer.json.
 search         Search for packages
 self-update    Updates composer.phar to the latest version.
 selfupdate     Updates composer.phar to the latest version.
 show           Show information about packages
 status         Show a list of locally modified packages
 update         Updates your dependencies to the latest version according to composer.json, and up
dates the composer.lock file.
 validate       Validates a composer.json
```

Composer commands

At this stage, we expect to see a composer.phar file in the root directory of the website.

Installing the package

In order to install the packages (the code libraries that we want to download from **Packagist** to our project), the packages need to be registered to a composer.json file within the **root directory** of the website.

We'll get back to the `composr.json` file later, but we first need to find a package to install in the **Packagist** repository, which is found at packagist.org[3].

In the packagist.org[4] website, type the name of the package that you want to download in the search box. For the sake of this chapter, we'll search for the "faker" package which generates fake data for our projects (something that can be most beneficial during the development stage of a website).

You will be shown a number of packages from which you need to choose the package that suits your needs the best.
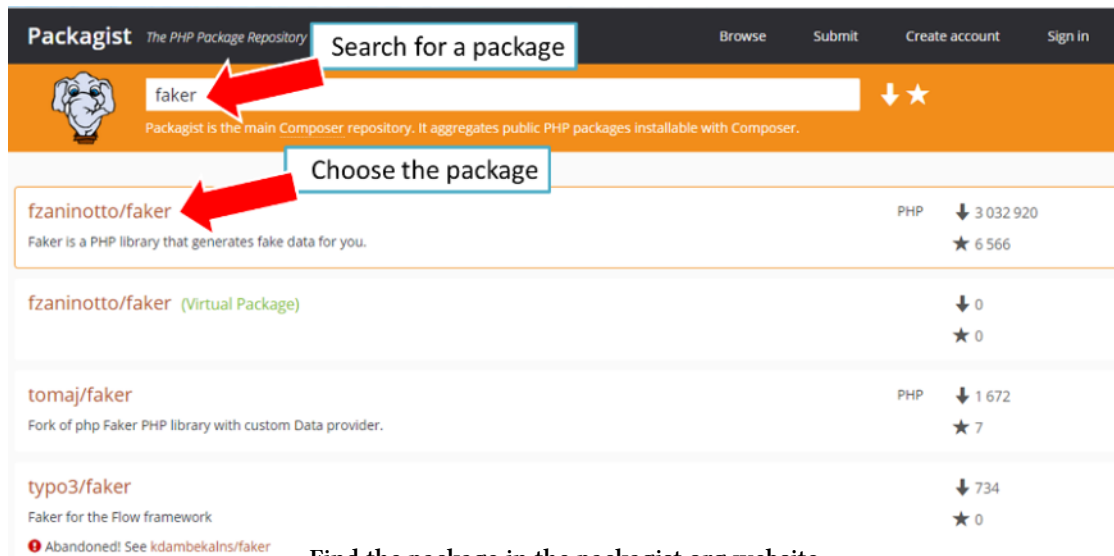
Find the package in the packagist.org website.

Click on a package to open the package page. Here you can find relevant information about the package, including the latest version and dependencies. You need to copy the command to download the latest stable version (see the picture below).

---

[3]https://packagist.org
[4]https://packagist.org

Faker package page on packagist

Back to the terminal, you need to navigate to the root directory of your app or website, and run the command that you just copied from **packagist**.

```
$ composer require fzaninotto/faker
```


Run the command to install the package in the root directory

Once you press enter, **Composer** will start to auto-magically download the latest stable version of the package that you require. It also downloads all the packages on which the package depends, and generates an autoload file.

```
Yossi@YOSSI-PC /c/wamp/www/faker
$ composer require fzaninotto/faker
Using version ^1.5 for fzaninotto/faker
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing fzaninotto/faker (v1.5.0)
    Loading from cache

Writing lock file
Generating autoload files
```

Composer
downloads the latest
stable version of the
package and
generates autoload
files.

Composer installs the package, its dependencies, and generates an autoload file

At the end of the download process, when you navigate to the `root` directory, you will see a brand new `vendor` directory that holds within it all the packages that **Composer** installed in your website.

index.php

composer.json file

The vendor directory
that holds all the
packages as well as
the autoloader.

The root directory after the package was installed
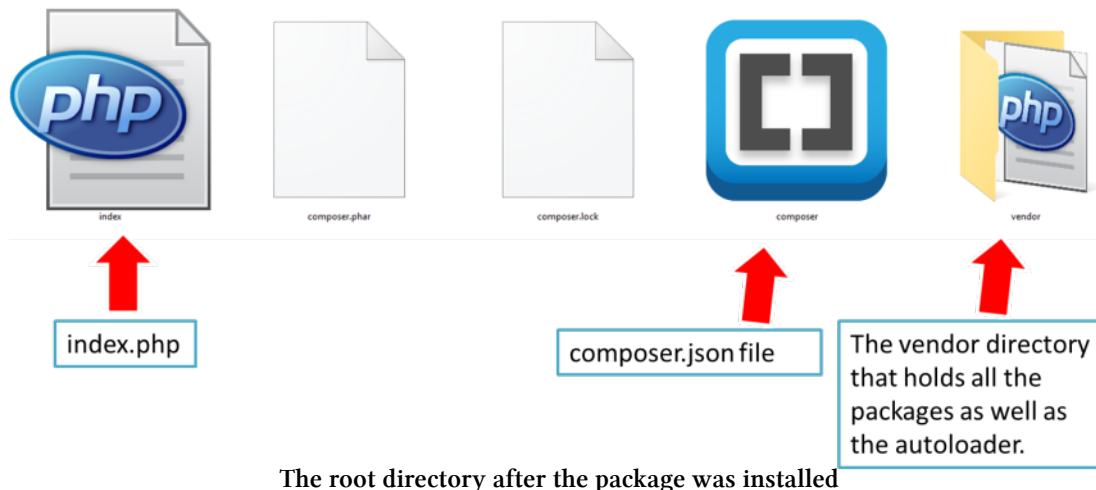
Also in the root directory, you need to create an `index.php` file.

In the `composer.json` file you will find the data that composer needs in order to work with and manage the package that you have just installed. In our example:

```
1  {
2    "require": {
3      "fzaninotto/faker": "^1.5"
4    }
5  }
```

When you open the `vendor` folder, you'll find the **autoloader** file (`autoload.php`) that comes free of charge with the download. We require the autoloader in our scripts in order for the Composer libraries to work.

```php
<?php
require_once __DIR__ . '/vendor/autoload.php';
```

To activate the autoloader, you may need to type the following command into the command line:

```
$ composer dumpautoload -o
```

Let's write in the `index.php` file a code that tests if the package that we have just downloaded really works:

```php
<?php
require_once __DIR__ . '/vendor/autoload.php';

// Use the factory to create a Faker\Generator instance
$faker = Faker\Factory::create();

// Generate fake name
echo $faker->name;

// Generate fake address
echo $faker->address;

// Generate fake text
echo $faker->text;
```

How to add packages?

After we have installed our first package, we may very well want to add a second one. This task, can also be done with much ease. All we need to do, is to require the package, according to its name in packagist, and the package will automatically be downloaded to our website. For example, let's require the `monolog/monolog` package.

```
$ composer require monolog/monolog
```

This will download the new package, with all its dependencies to the `vendor` directory, as well as update the `composer.json` file. Like so:

```
1  {
2    "require": {
3      "fzaninotto/faker": "1.4.0",
4      "monolog/monolog": "~1.13"
5    }
6  }
```

How to update the packages?

It is possible that, over time, we will have to update the packages and, here again; Composer will come to the rescue! All we need to do in order to update all the packages in our project is to write the following command in the command line:

```
$ composer update
```

As a result of this command, **Composer** will automatically download all the updated versions of the packages, including all the dependencies, straight to the project folder.

How to remove a package from Composer?

In order to remove one of the packages, we need to delete the line corresponding to the package from the `composer.json` file and update composer:

```
$ composer update
```

This will remove the package with all of it dependencies from the `vendor` directory.

Conclusion

I am sure that the combination of **Packagist** as a repository of PHP libraries and **Composer** as a dependency manager can help any PHP project, as it has so faithfully been doing for several years now.

In the next chapter, our PHP skills will take a big leap forward as we learn how to Composer autoload the classes, including those that we download from Packagist.

How to autoload PHP classes the Composer way?

In the previous chapter, we demonstrated how to use **Composer** to integrate existing code libraries into any PHP website. We also demonstrated the use of the built-in **Composer autoloader** for the packages that we installed. In the present chapter, we will demonstrate how to autoload our own classes as well as other non-packagist classes with the **Composer autoloader**.

> Autoloading allows us to use PHP classes without the need to `require` or `include` them and is considered a hallmark of modern-day programming.

A short reminder about autoloading Packagist code libraries

In the previous chapter, we saw that all it takes to autoload **Packagist** libraries is to add at the top of our scripts the following line that requires the Composer built-in autoloader:

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

But when working with our own classes (or with non-packagist classes) we may need to be somewhat more savvy...

> **Composer autoloading** can work in one of two main ways: through direct autoloading of classes or through the use of the PSR standards.

How to directly autoload classes with Composer?

The simplest way is to autoload each class separately. For this purpose, all we need to do is define the array of paths to the classes that we want to autoload in the `composer.json` file.

For example:

```
1  {
2    "autoload": {
3      "classmap": [
4        "path/to/FirstClass.php",
5        "path/to/SecondClass.php"
6      ]
7    }
8  }
```

Update the composer autoloader from the command line:

> $ composer dump-autoload -o

Now, we have to include the autoloader at the top of our scripts (e.g., `index.php`):

```php
<?php
require "vendor/autoload.php";
```

In the same way that we autoload classes, we can autoload directories that contain classes also by using the `classmap` key of the autoload:

```
1  {
2    "autoload": {
3      "classmap": [
4        "path/to/FirstClass.php",
5        "path/to/directory"
6      ]
7    }
8  }
```

- In order to autoload directories we need to use namespaces.

As we can see, classmap autoloading is not much different than the long list of requires that we used to use in the older PHP scripts. Yet, the better way to autoload is by using the **PSR-4** standard.

How to autoload the PSR-4 way?

PSR-4 is the newest standard of autoloading in PHP, and it compels us to use namespaces.

We need to take the following steps in order to autoload our classes with **PSR-4**:

a. **Put the classes that we want to autoload in a dedicated directory**. For example, it is customary to convene the classes that we write into a directory called src/, thus, creating the following directory structure:

```
your-website/
    src/
        Db.php
        Page.php
        User.php
```

Directory structure

a. **Give the classes a namespace**. We must give all the classes in the src/ directory the same namespace. For example, let's give the namespce Acme to the classes. This is what the Page class is going to look like:

```php
<?php
namespace Acme;

class Page
{
    public function __construct()
    {
        echo "hello, i am a page.";
    }
}
```

- We give the same namespace Acme to all of the classes in the src/ directory.

c. **Point the namespace to the `src/` directory in the `composer.json` file**. We point the directory that holds the classes to the namespace in the composer.json file. For example, this is how we specify in the composer.json file that we gave the namespace Acme to the classes in the src/ directory:

```
1  {
2    "autoload": {
3      "psr-4": {
4        "Acme\\":"src/"
5      }
6    }
7  }
```

- We use the **psr-4** key.
- The namespace has to end with \\. For example, Acme\\.
- You can replace the generic Acme with the name of your brand or website.

a. **Update the Composer autoloader:**

$ composer dumpautoload -o

a. **Import the namespace to your scripts**. The scripts need to import the namespace as well as the autoloader, e.g., index.php:

```php
<?php
require "vendor/autoload.php";

use Acme\Db;
use Acme\User;
use Acme\Page;

$page1 = new Page();
```

How to autoload if the directory structure is complex?

Up till now, we demonstrated autoloading of classes that are found directly underneath the src/ folder, but how can we autoload a class that is found in a subdirectory? For example, we may want to move the Page class into the Pages directory and, thus, create the following directory structure:

```
your-website/
    src/
        Db.php
        User.php
        Pages/
            Page.php
```

A more complex directory structure

These are the steps that we need to follow:

a. **Redefine the namespace**. We need to give the Page class a namespace in accordance with its new location in the src/Pages directory.

```php
<?php
namespace Acme\Pages;

public class Page
{
    function __construct()
    {
        echo "hello, i am a page.";
    }
}
```

a. **Use the namespaces in the scripts**:

```php
1  <?php
2  require "vendor/autoload.php";
3
4  use Acme\Db;
5  use Acme\User;
6  use Acme\Pages\Page;
7
8
9  $page1 = new Page();
```

Conclusion

As we demonstrated in the last two chapters, **Composer** is a powerful tool that can help us to both manage and autoload our own classes as well as others. Now, that we have such a powerful tool under our belt we're entitled to fully enjoy the best that modern-day PHP has to offer!

MVC and code organization

PHP is a programming language which is easy to learn and implement. However, it is also a language that can easily deteriorate into a "spaghetti code" in which different types of code are tangled together in a messy way, with HTML, CSS, and Java Script all sitting together in the same place with PHP.

It is bad to write a "spaghetti code" because it is much harder to maintain and messier to change, let alone treat its bugs.

To prevent the deterioration of the code into a "spaghetti code", we organize our code into files and folders. I am sure that the separation of the Java Script and the CSS files from the rest of the code is clear by now, but how can we separate the logical part, that PHP is responsible for, from the template part in which HTML is the main component? The simplest way is to put the application's logic (in which we handle the variables and determine their values) into one file or folder, while embedding the variables into the HTML elsewhere in the application. The logical part of the application is known as the **Controller**, while the template part is known as the **View**.

> The logical part of the application is known as the **Controller**, while the template part is known as the **View**.

In the example given below, the file `controller.php` is used as the controller in the application and it receives data from the data source (database, external API, etc.), processes it, and provides the variables that can later be used in the template file.

The data that was retrieved from the data source is grouped in a `$cars` array, that holds the cars' models and prices. We check to see if the price is expensive with the `expensiveOrNot` function, and store the data into a new array with the name of `$carsReviewed`.

```php
// Get the price and return whether it is expensive or not
function expensiveOrNot($price)
{
    if($price > 30000) return "expensive";

    return "not expensive";
}

// The data from the data source is stored in an array
$cars = array (
    "Bmw" => array("model" => "2-Series", "price" => "32000"),
```

```php
    "Audi" => array("model"=> "A3", "price"=>"29900")
);

// Perform some logic on the data to find out whether the cars are expensive
//  and store the returned values into this array
$carsReviewed = array(
    "Bmw" => array(
        "model" => "2-Series",
        "price" => "32000",
        "expensiveOrNot"=>expensiveOrNot(32000)
     ),
    "Audi" => array(
        "model"=> "A3",
        "price"=>"29900",
        "expensiveOrNot"=>expensiveOrNot(29900)
    )
);
```

Now, in the template file `index.php`, we can embed the variables from the controller into the HTML, but first we need to include the controller file:

```php
<?php require "controller.php"; ?>
<!doctype html>
<html>
<head>
</head>
<body>
<p>
BMW <?=$carsReviewed["Bmw"]["model"]?> is
<?=$carsReviewed["Bmw"]["expensiveOrNot"]?>
while Audi<?=$carsReviewed["Audi"]["model"]?> is
<?=$carsReviewed["Audi"]["expensiveOrNot"]?>.
</p>
</body>
</html>
```

Result:

BMW 2-Series is expensive while Audi A3 is not expensive.

Separating logic from presentation can improve efficiency when working in a team because it allows the programmers to work on the logical part of the application and the designers to work on the template, without interfering with each other.

The model

> The part that interacts with the data source is called the **Model** and it has all the code that handles databases.

We have seen how separating the logical part of the application from the template part can be very beneficial, but we have still not handled the part of the application that interacts with the data source.

The part that interacts with the data source is called the **Model** and it has all the code that handles databases (and other data sources). The role of the model is to retrieve the information from the database and maintain, update, or delete that information while keeping everything safe.

The broader context - MVC

> When we separate the code into Model, View and Controller, we use a design pattern with the name of **MVC**

We have, so far, learned about the three layers of the application:

1. The **Model**, which handles the data source,
2. The **View**, which holds the template files, and
3. The **Controller**, which handles the logic and mediates between the View and the Model parts of the application.

When we separate the code into Model, View and Controller, we use a design pattern with the name of **MVC** (an acronym for Model, View, and Controller). A design pattern is a way to construct a code in an acceptable and well tested manner that can solve many problems. We have a few dozen design patterns, the most important for PHP programmers being the MVC pattern.

The data flow in the MVC pattern

In the MVC pattern, the Controller has a special role as a mediator between the Model and View layers. The controller receives the data that the model retrieves from the database, processes it, and then passes it to the View for presentation. It also works the other way around, when the information that is received from the user (through the use of forms, cookies, and sessions) is passed to the Controller for processing and then transferred to the Model layer.

The image below can help in understanding the control flow in MVC.
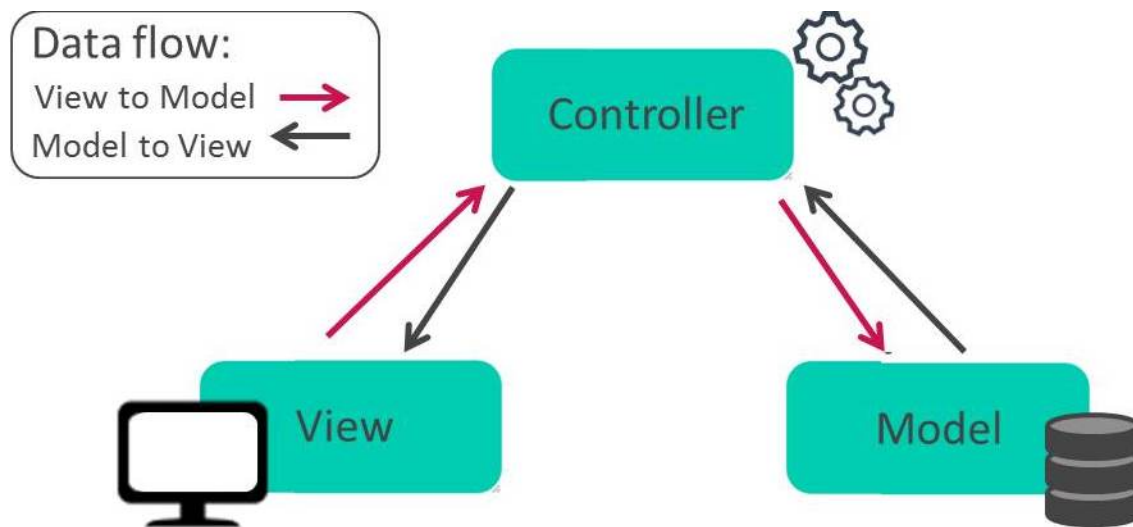
MVC explained

After the application gets the user data (for example, after the user feeds his data into a form and submits it), the data is passed to the controller layer that processes the information and decides to which of the model's classes to transfer the data to, and the model itself filters the data and then stores, updates, or deletes the records in the database.

The data can then make its way back from the model to the controller for further processing (e.g. which page to redirect the user to or which message to present to the user) while the presentation is handled and executed in the View.

The simplest MVC example - retrieving the data from the data source

In the following example, we check if a car's price is expensive the MVC way.

The data is fed to the Model, processed in the Controller, and then passed to the View.

The Model class gets and sets the car data (in a real world application, it will retrieve the data from the database).

```php
class Model
{
    // Store the data
    private $carPrice;

    // Set the data
    public function setPrice($price)
    {
        $this -> carPrice = $price;
    }

    // Get the data
    public function getPrice()
    {
        return $this -> carPrice;
    }
}
```

The Controller class is the mediator. It gets the data about the Car's price from the Model and checks if it is expensive.

```php
class Controller {
    private $model;
    private $limit = 30000;

    // Set the model so we can use its data
    public function __construct(Model $model)
    {
        $this -> model = $model;
    }

    // Set the data in the module
    public function setPrice($price)
    {
        $this->model->setPrice((int)$price);
    }

    // Some logic to check if the price is expensive
    public function expensiveOrNot()
    {
        if($this->model->getPrice() > $this->limit) return "expensive";
```

```php
        return "not expensive";
    }
}
```

The View class gets the data after it has been processed by the Controller, and outputs it to the user:

```php
class View {
    private $controller;

    // Set the controller so we can use it
    public function __construct(Controller $controller)
    {
        $this->controller = $controller;
    }

    // Output the data after processing it by the controller
    public function output()
    {
        return $this->controller->expensiveOrNot();
    }
}
```

Now we can implement the code. We feed the Model with the price, call the controller to perform its logic, and output everything through the View:

```php
// The data can come from the database
$priceToCheck = 31000;

// The data is retrieved by the model
$model1 = new Model();
$model1 -> setPrice($priceToCheck);

// We need the controller in order to process the data
$controller1 = new Controller($model1);

// We need the view in order to output the processed data
$view1 = new View($controller1);
echo $output = $view1 -> output();
```

Result:

expensive

The output can be embedded in the HTML.

The simplest MVC example (part 2) - getting data from the user

The MVC example cannot be complete without a means to get the user data and transfer it to the model. The user data can be fed into the application from a form with a post request. For example:

```php
$priceToCheck = $_POST['price'] = 29900;
```

Now, we need the controller to be able to transfer the data to the model, so we add to the controller a setPrice() method that can set the price within the model.

```php
class Controller {
    private $model;
    private $limit = 30000;

    // Set the model so we can use its data
    public function __construct(Model $model)
    {
        $this -> model = $model;
    }

    // Set the data in the module
    public function setPrice($price)
    {
        $this->model->setPrice((int)$price);
    }

    // Some logic to check if the price is expensive
    public function expensiveOrNot()
    {
        if($this->model->getPrice() > $this->limit) return "expensive";

        return "not expensive";
    }
}
```

Now we can make the ends meet and output the data:

```php
// The data can come from the user
//    e.g. through a post request
$priceToCheck = $_POST['price'] = 29900;

// We need the model to store the data
$model2 = new Model();

// We need the controller in order to get the user data
//   and process it before passing it to the Model
$controller2 = new Controller($model2);
$controller2 -> setPrice($_POST['price']);

// We need the view in order to output the processed data
$view2 = new View($controller2);
echo $output = $view2 -> output();
```

Result:

not expensive

The output can be embedded in the HTML.

Conclusion

The **MVC** pattern is very important to PHP programmers because it forces us to separate the logical part from the presentation part in our applications, and so helps us in maintaining and further developing the application.

Epilogue

What have we learned?

In this book, we have learned the essentials of Object Oriented PHP. We started by understanding what classes and objects are, as well as how to use them to encapsulate code from the rest of the application. Additionally, we learned about inheritance as a means of reducing code duplication. We also learned how to construct our application with the help of interfaces and polymorphism. In the final part, we began scratching the surface of real-world applications with namespaces that will allow the code to work with different code libraries, PDO as the database interface, and MVC as a pattern by which we can construct our applications in a well-tested way.

By this point, I hope that you have a strong grasp of the object-oriented style of programming, which should include the ability to answer the following questions:

1. What are the benefits of encapsulating the code within classes?
2. Why use inheritance?
3. Why are interfaces and polymorphism such great ideas?
4. How can namespaces and the MVC pattern help you build robust applications?

Some additional resources to advance your knowledge

There are still many things to learn in the PHP world, but now that you have a firm foundation, you will be able to grasp the material much more easily.

php.net[5] is the basic manual for the PHP community that contains the most up-to-date and concise knowledge in the field. It should definitely be on your reading list.

stackoverflow[6] is a question and answer website for programmers. On the site, you can find answers to practical and academic questions, ask your own questions, and answer the questions of other users.

You should get used to autoloading the classes that you use in your code, as it allows you to define search paths for classes so you don't require a long list of dependencies.

Packagist[7] is the most important PHP repository in which you can find many libraries of code for use in the applications that you develop. The integration of existing code libraries will shorten the time it takes you to write your applications.

---

[5] http://php.net
[6] http://stackoverflow.com
[7] https://packagist.org

Composer[8] is the tool that lets you use the packages you want to download from the Packagist repository. It provides you with autoloading, as well as dependency management, which means that the packages you download to your application are automatically installed with the packages on which they depend.

You need to know how to do **automated testing** of your code with PHPUnit framework[9] being the most common choice among PHP programmers.

You need to master at least one **PHP framework** with the most popular being Laravel[10], symfony[11], CodeIgniter[12], Yii[13], and CakePHP[14]. The Slim framework[15] is also popular, and has the benefit of being light weight and easy to learn. When you use a framework, you don't need to reinvent the wheel every time that you start a new application; instead, you can use a well-tested code as the foundation.

If you ever wondered how other programmers solve the same programmatic problems that you toil endlessly upon? Or, better yet, if there exist already established solutions that may help you to write your code in a better way, you probably need to learn **design patterns**, which are well tested solutions to common programmatic problems that you may confront on a daily basis. There are fewer than a couple dozen design patterns, some of the most beneficial patterns that you need to learn are the Strategy[16], Factory[17], Singleton[18], Facade[19], and Decorator[20] patterns.

Never stop learning. Learn new skills, gain new expertise, expand your abilities, and flourish in your career.

Wishing you all the best!

---

[8] https://getcomposer.org
[9] https://phpunit.de
[10] http://laravel.com
[11] https://symfony.com
[12] https://www.codeigniter.com
[13] http://www.yiiframework.com
[14] http://cakephp.org
[15] http://slimframework.com
[16] http://phpenthusiast.com/blog/strategy-pattern-the-power-of-interface
[17] http://phpenthusiast.com/blog/the-factory-design-pattern-in-php
[18] http://phpenthusiast.com/blog/the-singleton-design-pattern-in-php
[19] http://phpenthusiast.com/blog/simplify-your-php-code-with-facade-class
[20] http://phpenthusiast.com/blog/the-decorator-design-pattern-in-php-explained

Chapter 1 solutions

1.1 Which of these definitions best explains the term 'class'?

A: A **class** is a collection of variables and functions working with these variables.

1.2 Which of these definitions best explains the term 'object'?

A: An **object** gives us the ability to work with the class, and to have several instances of the same class.

1.3 Which of these definitions best explains the term 'property'?

B: A **property** is a variable within a class.

1.4 Which of these definitions best explains the term 'method'?

A: A **method** is a function within a class.

1.5 Write what you think should be the class name, the names of the properties for the first and last name, and the name of the method that returns hello.

class name: User

class properties: $firstName, $lastName

class method: hello()

1.6 Write the class `User` and add the properties:

```php
class User {
    public $firstName;
    public $lastName;
}
```

1.7 Add the method that says hello:

```php
class User {
    public $firstName;
    public $lastName;

    public function hello()
    {
        return "hello";
    }
}
```

1.8 Create the first instance, and call it $user1. Use the new keyword to create an object from the class.

```php
class User {
    public $firstName;
    public $lastName;

    public function hello()
    {
        return "hello";
    }
}


$user1 = new User();
```

1.9 Set the values for the first and last name to $user1.

```php
class User {
    public $firstName;
    public $lastName;

    public function hello()
    {
        return "hello";
    }
}


$user1 = new User();
$user1->firstName = 'John';
$user1->lastName  = 'Doe';
```

1.10 Get the user first and last name, and print it to the screen (with echo).

```php
class User {
    public $firstName;
    public $lastName;

    public function hello()
    {
        return "hello";
    }
}


$user1 = new User();
$user1 -> firstName = 'John';
$user1 -> lastName  = 'Doe';

echo $user1->firstName . " " . $user1->lastName;
```

Result: John Doe

1.11 Use the `hello()` method with the first and last name variables in order to say hello to the user:

```php
class User {
    public $firstName;
    public $lastName;

    public function hello()
    {
        return "hello";
    }
}


$user1 = new User();
$user1->firstName = 'John';
$user1->lastName  = 'Doe';

$hello = $user1->hello();

echo $hello . ",  " . $user1->firstName . " " . $user1->lastName;
```

Result: hello, John Doe

1.12 Add another object, call it `$user2`, give it a first name of 'Jane' and last name of 'Doe', then say hello to the user.

```php
class User {
    public $firstName;
    public $lastName;

    public function hello()
    {
        return "hello";
    }
}


$user1 = new User();
$user1->firstName = 'John';
$user1->lastName  = 'Doe';


$hello = $user1->hello();


$user2 = new User();
$user2 ->firstName = 'Jane';
$user2 ->lastName  = 'Doe';


echo $hello . ",  " . $user1->firstName . " " . $user1->lastName;
echo "<br />";
echo $hello . ",  " . $user2->firstName . " " . $user2->lastName;
```

Result:

hello, John Doe

hello, Jane Doe

Chapter 2 solutions

2.1 Which keyword would you use in order to approach the class's properties and methods from within the class?

Answer : The `$this` keyword enables us to approach the class's own properties and methods from within the class.

2.2 Add to the `hello()` method the ability to approach the `$firstName` property, so the `hello()` method would be able to return the string "hello, $firstName".

```php
class User {
    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user
    public function hello()
    {
        return "hello, " .  $this -> firstName;
    }
}
```

2.3 Create a new object with the first name of 'Jonnie' and last name of 'Roe'.

```php
class User {
    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user
    public function hello()
    {
        return "hello, " .  $this -> firstName;
    }
}


// Create a new object
```

```php
$user1 = new User();

// Set the user $firstName and $lastName properties
$user1 -> firstName = 'Jonnie';
$user1 -> lastName  = 'Roe';
```

2.4 Echo the `hello()` method for the $user1 object, and see the result.

```php
class User {
    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user
    public function hello()
    {
        return "hello, " .  $this -> firstName;
    }
}


// Create a new object
$user1 = new User();

// Set the user first and last names
$user1 -> firstName = 'Jonnie';
$user1 -> lastName  = 'Roe';

// Echo the hello() method
echo $user1 -> hello();
```

Result: hello, Jonnie

Chapter 3 solutions

3.1 Add a `register()` method to the class that echoes the name of the user plus the verb "registered".

```php
class User {

    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user $firstName
    // The user $firstName property can be approached
    //   with the $this keyword
    public function hello()
    {
        return "hello, " .  $this -> firstName;
    }

    // The register method
    public function register()
    {
        echo $this -> firstName . " " . $this -> lastName . " registered.";
    }
}
```

3.2 Add a `mail()` method to the class that echoes the string "emailed".

```php
class User {
    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user $firstName.
    // The user $firstName property can be approached
    //   with the $this keyword.
    public function hello()
    {
        return "hello, " .  $this -> firstName;
```

```php
    }

    // The register method
    public function register()
    {
        echo $this -> firstName . " " . $this -> lastName . " registered.";
    }

    // The mail method
    public function mail()
    {
        echo " emailed.";
    }
}
```

3.3 Add `return $this` to the `register()` method so that it can be chained to any other method in the class.

```php
class User {
    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user $firstName
    // The user $firstName property can be approached
    //    with the $this keyword
    public function hello()
    {
        return "hello, " .  $this -> firstName;
    }

    // The register method
    public function register()
    {
        echo $this -> firstName . " " . $this -> lastName . " registered.";
        return $this;
    }

    // The mail method.
    public function mail()
    {
        echo " emailed.";
```

```php
    }
}
```

3.4 Create a new $user1 object with the first name of 'Jane' and the last name of 'Roe'. For this object, chain the mail() method to the register() method and see the result.

```php
class User {
    // The class properties
    public $firstName;
    public $lastName;

    // A method that says hello to the user $firstName.
    // The user $firstName property can be approached
    //   with the $this keyword
    public function hello()
    {
        return "hello, " .  $this -> firstName;
    }

    // The register method
    public function register()
    {
        echo $this -> firstName . " " . $this -> lastName . " registered.";
        return $this;
    }

    // The mail method
    public function mail()
    {
        echo " emailed.";
    }
}

$user1 = new User();
$user1 -> firstName = 'Jane';
$user1 -> lastName  = 'Roe';

// Chain the methods mail() to register()
$user1 -> register() -> mail();
```

Result:

Jane Roe registered. emailed.

Chapter 4 solutions

4.1 We use the private access modifier in order to:

D: We use the private access modifier in order to prevent code from outside the class scope from interacting with the class properties and methods. The private access modifier, like the public access modifier, is only relevant to properties and methods, and has no power on classes.

4.2 Create a new class property with the name of `$firstName`, and prevent any code from outside the class from changing the property value by using the appropriate access modifier (private or protected).

```php
class User {
    // We use the private access modifier in order to
    //   prevent code from outside
    //   the class from modifying the property's value
    private $firstName;
}
```

4.3 Create a method to set the `$firstName` property value. Remember to use the right access modifier (`public`/`private`).

```php
class User {
    private $firstName;

    // A public setter method allows us to set the
    //   $firsName property.
    public function setFirstName($str)
    {
        $this -> firstName = $str;
    }
}
```

4.4 Now, create a method to return the `$firstName` value.

```php
class User {
    private $firstName;

    // A public setter method allows us to set the
    //   $firsName property
    public function setFirstName($str)
    {
        $this -> firstName = $str;
    }

    // A public getter method allows us to return the
    //   $firsName property
    public function getFirstName()
    {
        return $this -> firstName;
    }
}
```

Chapter 5 solutions

5.1 Add to the class a constructor method to set a value to the $firstName property as soon as the object is created.

```php
class User {
    private $firstName;
    private $lastName;

    // Constructor function to set a single value
    public function __construct($firstName)
    {
        $this -> firstName = $firstName;
    }
}
```

5.2 Add to the constructor the ability to set the value of the $lastName property (remember that you can pass to a method more than parameter).

```php
class User {
    private $firstName;
    private $lastName;

    // Constructor function to set more than one value
    public function __construct($firstName,$lastName)
    {
        $this -> firstName = $firstName;
        $this -> lastName = $lastName;
    }
}
```

5.3 Add to the class a getFullName() public method that returns the full name.

```php
class User {
    private $firstName;
    private $lastName;

    // Constructor function to set more than one value
    public function __construct($firstName,$lastName)
    {
        $this -> firstName = $firstName;
        $this -> lastName = $lastName;
    }


    public function getFullName()
    {
        return $this -> firstName . ' ' . $this -> lastName;
    }
}
```

5.4 Create a new object, $user1, and pass to the constructor the values of the first and last name. The first name is "John" and the last name is "Doe" (you may choose your preferred combination of first and last name).

```php
class User {
    private $firstName;
    private $lastName;

    // Constructor function to set more than one value
    public function __construct($firstName,$lastName)
    {
        $this -> firstName = $firstName;
        $this -> lastName = $lastName;
    }

    public function getFullName()
    {
        return $this -> firstName . ' ' . $this -> lastName;
    }
}

$user1 = new User("John", "Doe");
```

5.5 Get the full name, and echo it to the screen.

```php
class User {
    private $firstName;
    private $lastName;

    // Constructor function to set more than one value
    public function __construct($firstName,$lastName)
    {
        $this -> firstName = $firstName;
        $this -> lastName = $lastName;
    }

    public function getFullName()
    {
        return $this -> firstName . ' ' . $this -> lastName;
    }
}

$user1 = new User("John", "Doe");

echo $user1 -> getFullName();
```

Result:

John Doe

Chapter 6 solutions

6.1 Which keyword do we use in order to declare that one class inherits from another class?

C: In order to declare that a certain class is the child of another class we use the extends keyword in the child class.

6.2 Which keyword do we use in order to declare that a method or property can only be used within the parent class and its child classes?

B: The `protected` keyword allows access to methods and properties only from within the parent class and its child classes.

6.3 We learned about three access control modifiers (public, private, and protected) that we use to allow/restrict access to the code. In the following table, we will use "V" to mark that the code can be accessed from a certain level, and "X" if it cannot be accessed.

Modifier	Class	Child class	Global scope
Public	✓	✓	✓
Protected	✓	✓	✗
Private	✓	✗	✗

Comparison between access control modifiers

Public methods and properties can be accessed from anywhere in the code.

Protected methods and properties can only be accessed from within the class and its child classes.

Private methods and properties can only be accessed from within the class itself.

6.4 Create a user class.

```php
class User {}
```

6.5 Add to the class a private property with the name of $username.

```php
class User {
    private $username;
}
```

6.6 Create a setter method that can set the value of the $username.

```php
class User {
    private $username;

    public function setUsername($name)
    {
        $this -> username = $name;
    }
}
```

6.7 Create a class Admin that inherits the User class.

```php
class Admin extends User{}
```

6.8 Now, let's add to the Admin class some code. First, add a public method by the name of expressYourRole, and make it return the string: "Admin".

```php
class Admin extends User{
    public function expressYourRole()
    {
        return strtolower(__CLASS__);
    }
}
```

6.9 Add to the Admin class another public method, sayHello, that returns the string "Hello admin, XXX" with the username instead of XXX.

```php
class Admin extends User{
    public function expressYourRole()
    {
        return strtolower(__CLASS__);
    }

    public function sayHello()
    {
        return "Hello admin, " . $this -> username;
    }
}
```

6.10 Create an object $admin1 out of the class Admin, set its name to "Balthazar", and say hello to the user. Do you see any problem?

```php
class User {
    private $username;

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


class Admin extends User {
    public function expressYourRole()
    {
        return strtolower(__CLASS__);
    }

    public function sayHello()
    {
        return "Hello admin, " . $this -> username;
    }
}


$admin1 = new Admin();
$admin1 -> setUsername("Balthazar");
$admin1 -> sayHello();
```

Result:

🐞   Notice: Undefined property: Admin::$username

6.11 What do you think is the cause of the problem?

Answer: The cause of the problem is that we try to access a private method, $username, from outside the class.

6.12 How will you fix this problem?

```php
// We can declare the $usrname property as protected to allow the
//   access for properties and methods that belongs to the child classes
class User {
    // Declare the $username as protected
    protected $username;

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


class Admin extends User {
    public function expressYourRole()
    {
        return strtolower(__CLASS__);
    }

    public function sayHello()
    {
        return "Hello admin, " . $this -> username;
    }
}


$admin1 = new Admin();
$admin1 -> setUsername("Balthazar");
echo $admin1 -> sayHello();
```

Result:

Hello admin, Balthazar

For a more elegant solution, we may use a getter method inside the parent that can be used from the child class.

```php
class User {
    private $username;

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

    public function getUsername()
    {
        return $this -> username;
    }
}


class Admin extends User{
    public function expressYourRole()
    {
        return strtolower(__CLASS__);
    }

    public function sayHello()
    {
        return "Hello admin, " . $this -> getUsername();
    }
}


$admin1 = new Admin();
$admin1 -> setUsername("Balthazar");
echo $admin1 -> sayHello();
```

Chapter 7 solutions

7.1 Which keyword is used to declare a class or method as abstract?

C: abstract

7.2 Which keyword is used to declare a child class that inherits from an abstract class?

D: extends

7.3 What are the main reasons for using an abstract class in the code?

B: The main reason for using an abstract class is to commit the child classes to methods.

7.4 Can we create objects from abstract classes?

Answer: We cannot create objects from abstract classes.

7.5 Create an abstract class with the name of `User`, which has an abstract method with the name of `stateYourRole`.

```
abstract class User {
    abstract public function stateYourRole();
}
```

7.6 Add to the class a protected variable with the name of `$username`, and public setter and getter methods to set and get the `$username`.

```
abstract class User {
    abstract public function stateYourRole();

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

    public function getUsername()
    {
        return $this -> username;
    }
}
```

7.7 Create an `Admin` class that inherits the abstract `User` class.

```php
class Admin extends User {}
```

7.8 Which method should be defined in the class?

Answer: the Admin class should define the stateYourRole() method because it is an abstract method.

7.9 Define the method stateYourRole() in the child class and let it return the string "admin";

```php
class Admin extends User {
    public function stateYourRole()
    {
        return "admin";
    }
}
```

7.10 Create another class, Viewer that inherits the User abstract class. Define the method that should be defined in each child class of the User class.

```php
class Viewer extends User {
    public function stateYourRole()
    {
        return strtolower(__CLASS__);
    }
}
```

7.11 Create an object from the Admin class, set the username to "Balthazar", and make it return the string "admin".

```php
$admin1 = new Admin();
$admin1 -> setUsername("Balthazar");
echo $admin1 -> stateYourRole();
```

Result:

admin

Chapter 8 solutions

8.1 Which keyword should we use in order to implement an interface?

D: We should use the `implements` keyword in order to implement an interface.

8.2 Which of the following code types are allowed in an interface?

D: Interfaces can have methods and constants, but they cannot have properties.

8.3 Which of the following access modifiers is allowed for abstract methods in interfaces?

A: Methods in interfaces can only be public, because the nature of interfaces is to expose their methods to the outside world.

8.4 Which of the following is valid inheritance?

E: A class can inherit from only 1 parent/abstract class, while it can implement more than 1 interface.

8.5 Create a `User` class with a protected `$username` property and methods that can set and get the `$username`.

```
class User {
    protected $username;

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

    public function getUsername()
    {
        return $this -> username;
    }
}
```

8.6 Create an Author interface with the following abstract methods that can give the user an array of authorship privileges. The first method is `setAuthorPrivileges()`, and it gets a parameter of `$array`, and the second method is `getAuthorPrivileges()`.

```php
interface Author {
    public function setAuthorPrivileges($array);

    public function getAuthorPrivileges();
}
```

8.7 Create an `Editor` interface with methods to set and get the editor's privileges.

```php
interface Editor {
    public function setEditorPrivileges($array);

    public function getEditorPrivileges();
}
```

8.8 Create an `AuthorEditor` class that extends the `User` class, and implements both the `Author` and the `Editor` interfaces.

```php
class AuthorEditor extends User implements Author,Editor  {}
```

8.9 Create in the `AuthorEditor` class the methods that it should implement, and the properties that these methods force us to add to the class.

```php
class AuthorEditor extends User implements Author,Editor  {
    private $authorPrivilegesArray = array();

    private $editorPrivilegesArray = array();

    public function setAuthorPrivileges($array)
    {
        $this -> authorPrivilegesArray = $array;
    }

    public function getAuthorPrivileges()
    {
        return $this -> authorPrivilegesArray;
    }

    public function setEditorPrivileges($array)
    {
        $this -> editorPrivilegesArray = $array;
    }
```

```php
    public function getEditorPrivileges()
    {
        return $this -> editorPrivilegesArray;
    }
}
```

8.10 Now, let's create an object with the name of `$user1` from the class `AuthorEditor`, and set its username to "Balthazar".

```php
$user1 = new AuthorEditor();
$user1 -> setUsername("Balthazar");
```

8.11 Set in the `$user1` object an array of authorship privileges, with the following privileges: "write text", "add punctuation".

```php
$user1 = new AuthorEditor();
$user1 -> setUsername("Balthazar");
$user1 -> setAuthorPrivileges(array("write text","add punctuation"));
```

8.12 Set in the `$user1` object an array with the following editorial privileges: "edit text", "edit punctuation".

```php
$user1 = new AuthorEditor();
$user1 -> setUsername("Balthazar");
$user1 -> setAuthorPrivileges(array("write text", "add punctuation"));
$user1 -> setEditorPrivileges(array("edit text", "edit punctuation"));
```

8.13 Use the following code to get the `$user1` name and privileges:

```php
$userName   = $user1 -> getUsername();
$userPrivileges = array_merge($user1 -> getAuthorPrivileges(),
    $user1 -> getEditorPrivileges());

echo $userName . " has the following privileges: ";
foreach($userPrivileges as $privilege)
{
    echo " {$privilege},";
}
echo ".";
```

Result:

Balthazar has the following privileges: write text, add punctuation, edit text, edit punctuation,.

Chapter 9 solutions

9.1 Which of these sentences best describes the **polymorphism principle** in PHP?

A: According to the Polymorphism principle, methods that serve the same functionality in different classes should have the same name.

9.2 Add to the class concrete methods to set and get the number of articles.

```php
abstract class User {
    protected $scores          = 0;
    protected $numberOfArticles = 0;

    public function setNumberOfArticles($int)
    {
        // Cast to integer type
        $numberOfArticles = (int)$int;
        $this -> numberOfArticles = $numberOfArticles;
    }

    public function getNumberOfArticles()
    {
        return $this -> numberOfArticles;
    }
}
```

9.3 Add to the class the abstract method: `calcScores()`, that performs the scores calculations separately for each class.

```php
abstract class User {
    protected $scores          = 0;
    protected $numberOfArticles = 0;

    public function setNumberOfArticles($int)
    {
        // Cast to integer type
        $numberOfArticles = (int)$int;
        $this -> numberOfArticles = $numberOfArticles;
    }
```

```php
    public function getNumberOfArticles()
    {
        return $this -> numberOfArticles;
    }


    // The abstract method.
    abstract public function calcScores();
}
```

9.4 Create an Author class that inherits from the User class. In it create a concrete `calcScores()` method that returns the number of scores from the following calculation:

```php
class Author extends User {
    public function calcScores()
    {
        return $this -> scores =
            $this -> numberOfArticles * 10 + 20;
    }
}
```

9.5 Also create an Editor class that inherits from the User class. In it create a concrete `calcScores()` method that returns the number of scores from the following calculation:

```php
class Editor extends User {
    public function calcScores()
    {
        return $this -> scores =
            $this -> numberOfArticles * 6 + 15;
    }
}
```

9.6 Create an object, $author1, from the Author class, set the number of articles to 8, and echo the scores that it gained.

```php
$author1 = new Author();
$author1 -> setNumberOfArticles(8);
echo $author1 -> calcScores();
```

9.7 Create another object, $editor1, from the Editor class, set the number of articles to 15, and echo the scores that it gained.

```
$editor1 = new Editor();
$editor1 -> setNumberOfArticles(15);
echo $editor1 -> calcScores();
```

Result:

100

105

Chapter 10 solutions

10.1 Type hinting can check if the types of values that are passed to functions belong to the following data types:

C : Complex types (arrays and objects) type hinting is supported in both PHP 5 and 7, while scalar type hinting (int, float, string, bool) is only supported in PHP 7.

Chapter 11 solutions

11.1 Create a `User` interface with set and get methods for both a `$username` property, as well as for a `$gender` property.

```php
interface User {
    public function setUsername($username);
    public function getUsername();

    public function setGender($gender);
    public function getGender();
}
```

11.2 Now, create a `Commentator` class to implement the `User` interface.

```php
class Commentator implements User {
    private $username = '';
    private $gender   = 'none of your business!';

    // Set and get the $username
    public function setUsername($name)
    {
        $this -> username = (is_string($name))? $name : 'N/A';
    }

    public function getUsername()
    {
        return $this -> username;
    }

    // Set and get the gender
    public function setGender($gender)
    {
        $gendersArray = array('female', 'male', 'other');

        if(in_array($gender, $gendersArray))
        {
            $this -> gender = $gender;
```

```php
        }
    }

    public function getGender()
    {
        return $this -> gender;
    }
}
```

11.3 Create function to add "Mr." or "Mrs." to the username. When writing the function make sure to type hint it correctly, so it can only get the types it is expected to.

```php
function addMrOrMrsToUsername(User $user)
{
    $userName   = $user -> getUsername();
    $userGender = $user -> getGender();

    if($userGender === 'female')
    {
        return "Mrs. " . $userName;
    }
    else if($userGender === 'male')
    {
        return "Mr. " . $userName;
    }

    return $userName;
}
```

11.4 Run the code against a user with the name of "Jane" and against another user with the name of "Bob".

```php
$user1 = new Commentator();
$user1 -> setUsername("Jane");
$user1 -> setGender("female");
echo addMrOrMrsToUsername($user1);

$user2 = new Commentator();
$user2 -> setUsername("Bob");
$user2 -> setGender("male");
echo addMrOrMrsToUsername($user2);
```

Result:

Mrs. Jane

Mr. Bob

Chapter 12 solutions

12.1 The use of static methods and variables is appropriate in the following cases...

Answer: **B** : We might consider the use of static methods and variables when we need utility methods, with some common uses being: counters, sanitation, encryption, unit conversion, and any other methods that only give services to our program's main classes.

Chapter 13 solutions

13.1 Which of the following keywords is used to define a trait?

C : the keyword `trait` is used to define a trait.

13.2 Which of the following types can be found within a trait?

D : Within a trait, we can find all of the types such as abstract methods, concrete methods, properties, and constants. Having all of these types inside a trait is very powerful. However, good traits are short and concise, so be sure not to bloat your traits with a code which better resides in your classes.

*13.3** Is it appropriate to use traits in a scenario in which all of the child classes of an interface need the same methods?

In the case that all of the child classes of an interface use the same methods, we put the methods in an interface. We consider the use of traits in those cases in which only some of the child classes (and not all of the child classes) need to use a certain method.

13.4 Write an interface with the name of `User`.

```
interface User {}
```

13.5 Write three classes to implement the `User` interface: `Author`, `Commentator` and `Viewer`.

```
interface User {}

class Author implements User {}

class Commentator implements User {}

class Viewer implements User {}
```

13.6 Add a trait with the name of `Writing` that contains an abstract method with the name of `writeContent()`.

```
trait Writing {
    abstract public function writeContent();
}
```

13.7 Use the trait `Writing` in the `Author` class, and implement its abstract method by making it return the string "Author, please start typing an article...".

```php
class Author implements User {

    use Writing;

    public function writeContent()
    {
        return  "Author, please start typing an article...";
    }
}
```

13.8 Use the trait `Writing` in the `Commentator` class, and implement its abstract method by making it return the string "Commentator, please start typing your comment..".

```php
class Commentator implements User {
    use Writing;

    public function writeContent()
    {
        return  "Commentator, please start typing your comment...";
    }
}
```

13.9 Run the code you have just written and see the result.

```php
$author1 = new Author();
echo $author1 -> writeContent();

$commentator1 = new Commentator();
echo $commentator1 -> writeContent();
```

Result:

Author, please start typing an article...

Commentator, please start typing your comment.

Chapter 14 solutions

14.1 What are the main reasons for using namespaces?

D : The main reasons for the use of namespaces is to group related code and avoid name collisions between libraries.

14.2 Which keyword is used to define a namespace?

B: The keyword `namespace` is used to define a namespace.

14.3 Which keyword is used to import a namespace?

A : The keyword `use` is used to import a namespace.

Chapter 15 solutions

15.1 Add to the `Article` class, a constructor method that gets the parameters of: `$title` and `$author` and sets the class's properties accordingly.

```php
class Article {
    protected $title;
    protected $author;

    public function __construct($title, $author)
    {
        $this -> title = $title;

        // The object is injected to the class
        $this -> author = $author;
    }
}
```

15.2 Write to the class, a getter method that returns the article's title.

```php
class Article {
    protected $title;
    protected $author;

    public function __construct($title, $author)
    {
        $this -> title = $title;

        // The object is injected to the class
        $this -> author = $author;
    }

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

15.3 Add to the class, a getter method that returns the author object.

```php
class Article {
    protected $title;
    protected $author;

    public function __construct($title, $author)
    {
        $this -> title = $title;

        // The object is injected to the class.
        $this -> author = $author;
    }

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

    public function getAuthor()
    {
        return $this -> author;
    }
}
```

15.4 Write another class with the name of `Author`, that has a protected property `$name` and setter and getter methods that handle the property.

```php
class Author {
    protected $name;

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

    public function getName()
    {
        return $this -> name;
    }
}
```

15.5 Now, first create the `Author` object, call it `$author1`, and then set its name to 'Joe'.

```php
$author1 = new Author;

$author1 -> setName("Joe");
```

15.6 Create the article object, call it $article1, and pass to it the title "To PHP and Beyond" and the $author1 object that you have just created.

```php
$title = "To PHP and Beyond";

$article1 = new Article($title,$author1);
```

15.7 Write a code that returns the string: "To PHP and Beyond by Joe".

```php
echo $article1 -> getTitle() . ' by ' .
    $article1 -> getAuthor() -> getName();
```

Result:

To PHP and Beyond by Joe

In this code example, instead of creating the author object inside the Article class, we created it outside the class, and then injected it into the class with the constructor. (The same result can be achieved by using a setter method.)

15.8 It is also highly advisable to typehint the injected objects. For this purpose, we need to first write an Authors interface and then make the Author class implement that interface.

```php
interface Authors {
    public function setName($name);
    public function getName();
}

class Author implements Authors {
    protected $name;

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

    public function getName()
    {
        return $this -> name;
    }
}
```

Now, we need to type hint the $author parameter that is passed to the Article class.

```php
class Article {
    protected $title;
    protected $author;

    public function __construct($title, Author $author)
    {
        $this -> title = $title;

        // The object is injected to the class.
        $this -> author = $author;
    }

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

    public function getAuthor()
    {
        return $this -> author;
    }
}
```

Let's test the code:

```php
$author1 = new Author;
$author1 -> setName("Joe");

$title = "To PHP and Beyond";
$article1 = new Article($title,$author1);

echo $article1 -> getTitle() . ' by ' .
    $article1 -> getAuthor() -> getName();
```

Result:

To PHP and Beyond by Joe

Chapter 16 solutions

16.1 In which of the following cases would you use exception handling?

B : We use exception handling to suppress errors outside the scope of the program in a tested code, and not in order to hide our own bugs or to replace if-then blocks.

16.2 Add to the setName() method a code that throws exception whenever the user's name is shorter than 3 letters. What is a suitable message that can be thrown?

```php
class User {
    private $name;
    private $age;

    public function setName($name)
    {
        $name = trim($name);

        if(strlen($name) < 3)
        {
            throw new Exception("The name should be at least 3 characters long."\
);
        }

        $this -> name = $name;
    }

    public function setAge($age)
    {
        $this -> age = $age;
    }

    public function getName()
    {
        return $this -> name;
    }

    public function getAge()
        {
        return $this -> age;
```

```
    }
}
```

16.3 Add to the setAge() method a code that throws exception whenever the user's age is less than or equal to 0.

```php
class User {
    private $name;
    private $age;

    public function setName($name)
    {
        $name = trim($name);

        if(strlen($name) < 3)
        {
            throw new Exception("The name should be at least 3 characters long."\
);
        }

        $this -> name = $name;
    }

    public function setAge($age)
    {
        $age = (int)$age;

        if($age < 1)
        {
            throw new Exception("The age cannot be zero or less.");
        }

        $this -> age = $age;
    }

    public function getName()
    {
        return $this -> name;
    }

    public function getAge()
        {
```

```php
        return $this -> age;
    }
}
```

16.4 Write a foreach loop (like the one we have already used in this tutorial) that feeds the array to the class, and handles the exceptions by echoing the custom error messages as well as the file path and line number in which the exception was thrown.

```php
$dataForUsers = array(
    array("Ben",4),
    array("Eva",28),
    array("li",29),
    array("Catie","not yet born"),
    array("Sue",1.5)
);

foreach($dataForUsers as $data => $value)
{
    try
    {
        $user = new User();

        $user -> setName($value[0]);
        $user -> setAge($value[1]);

        echo $user -> getName() . " is " . $user -> getAge() . " years old. <br \
/>";
    }
    catch (Exception $e)
    {
        echo "Error: " . $e -> getMessage() . " in the file: " . $e -> getFile()
                . " on line: " . $e -> getLine() . "<hr />";
    }
}
```

16.5 Run the code and see the result.

Result:

Ben is 4 years old.

Eva is 28 years old.

Error: The name should be at least 3 characters long. in the file: C:\wamp\www\index.php on line: 13

Error: The age cannot be zero or less. in the file: C:\wamp\www\index.php on line: 25

Sue is 1 years old.

Chapter 17 solutions

17.1 Write a class with the name of Db that can connect with the database.

```php
// DB credentials.
define('DB_HOST','localhost');
define('DB_USER','root');//database username
define('DB_PASS','');// database passord
define('DB_NAME','testing'); //database name

class Db {
    // The PDO object.
    private $dbh;

    // Establish database connection
    //  or display an error message.
    function __construct()
    {
        try
        {
            $this->dbh = new \PDO("mysql:host=".DB_HOST.";dbname=".DB_NAME,
            DB_USER, DB_PASS,
                array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES 'utf8'"));
        }
        catch (PDOException $e)
        {
            exit("Error: " . $e->getMessage());
        }
    }

    // We need a method to get the database handler
    //  so it can be used outside of this class.
    function get()
    {
        return $this->dbh;
    }

    // Set the PDO object to null to close the connection.
        function close()
```

```
        {
                $this->dbh = null;
        }
}
```

17.2 Write a User class that can get the database connection, and save it into a private property with the name of $dbCon.

```php
class User {
    private $tableName = 'users';
    private $dbCon;

    // First, save the connection in a private property.
    function __construct($dbCon)
    {
        $this->dbCon = $dbCon;
    }
}
```

17.3 Write into the User class the methods that can work with the database.

```php
class User {
    private $tableName = 'users';
    private $dbCon;

    // First, establish the connection with the database.
    function __construct($dbCon)
    {
        $this->dbCon = $dbCon;
    }

    // Insert new user to the database.
    function insert($name,$phone,$city)
    {
        // The insert query.
            $sql = "INSERT INTO `{$this->tableName}`
        (`name`,`phone`, `city`, `date_added`)
        VALUES
        (:name,:phone,:city,:created)";

        // Bind and filter.
        $query = $this->dbCon->prepare($sql);
```

```php
    $query->bindParam(':name',$name,PDO::PARAM_STR);
    $query->bindParam(':phone',$phone,PDO::PARAM_STR);
    $query->bindParam(':city',$city,PDO::PARAM_STR);

        $now = date('Y-m-d');
    $query->bindParam(':created',$now,PDO::PARAM_STR);


    $query -> execute();

    // The id of the newly created row in the table.
    $lastInsertId = $this->dbCon->lastInsertId();

    if($lastInsertId>0)
        return $lastInsertId;
    else
        return false;
}


function getUserById($id)
{
    $sql = "SELECT * FROM `{$this->tableName}` WHERE `id` = :id LIMIT 1";

    $query = $this->dbCon->prepare($sql);

    $query -> bindParam(':id', $id, PDO::PARAM_INT);

    $query -> execute();

    $results = $query -> fetchAll(PDO::FETCH_OBJ);

    if($query -> rowCount() < 1) return false;


    return  $results[0];
}


// Get all the users.
function getAll()
```

```php
    {
        $sql = "SELECT * FROM `{$this->tableName}` WHERE 1";

        $query = $this->dbCon->prepare($sql);

        $query -> bindParam(':id', $id, PDO::PARAM_INT);

        $query -> execute();

        $results = $query -> fetchAll(PDO::FETCH_OBJ);

        if($query -> rowCount() < 1) return false;


        return $results;
    }


    function updateUser($id,$array)
    {
        $sql = "UPDATE `{$this->tableName}` SET ";

        $columns = array();
        foreach($array as $fieldName => $value)
        {
            $columns[] = "`{$fieldName}` = :{$fieldName}";
        }
        $sql .= implode(',',$columns);

        $sql .= ' WHERE `id` = :id';


        $query = $this->dbCon->prepare($sql);


        foreach($array as $fieldName => $value)
        {
            // Use bindValue, not bindParam because
            //  bindParam only gets its value at the time of execution.              \

            $query -> bindValue(":{$fieldName}",$value);
        }
```

```php
        $query -> bindParam(':id', $id, PDO::PARAM_INT);


        $query -> execute();

        if($query -> rowCount()) return false;


        return $id;
    }


    function delete($id)
    {
        $sql = "DELETE FROM `{$this->tableName}` WHERE `id`=:id";

        $query = $this->dbCon->prepare($sql);

        $query -> bindParam(':id', $id, PDO::PARAM_INT);

        $query -> execute();

        if($query -> rowCount() < 1) return false;


        return true;
    }
}
```

17.4 Test your code.

```php
// Create the pdo object.
$db      = new Db;

// Get the connection.
$dbCon   = $db->get();

// When creating the new User class
//  pass the connection.
$userObj = new User($dbCon);

// Test insertion.
```

```php
var_dump($userObj->insert('James Tiberius Kirk','0544308209','Riverside, Iowa'));

// Test selection.
var_dump($userObj->getUserById(1));
var_dump($userObj->getAll());

// Test update.
$array=['name'=>'Captain Kirk','city'=>'Tarsus IV','date_added'=>'2233-03-22'];
var_dump($userObj->updateUser(1,$array));
var_dump($userObj->getUserById(1));

// Test deletion.
var_dump($userObj->delete(1));
```

Introduction How to create classes and objects