Contents

Chapter 1

Introduction

I am sure that by now you must have heard that Drupal 8 is using Symfony components and is based on object-oriented programming (OOP) in PHP. If you are a Drupal 7 developer, then you may not know what is object-oriented programming or fail to understand the benefits it offers. In this chapter, you will learn:

- Basics of object-oriented PHP programming
- What is a class and how to define one?
- What is an object and how to create one?
- What is the difference between a class and an object?
- What is a constructor and how is it helpful?

What are objects and classes?

The central concept in the understanding of object-oriented programming is the concept of object. An "object" is a group of data (called properties) and functions (called methods) that go together. Let's consider a real-life example. A vehicle is an object. It can be defined to have three properties: `$color`, `$milesDriven` and `$warranty` (number of miles). It can be defined to have one method: `getWarrantyLeft()`. This method returns the number of miles of warranty left on the vehicle. Let's create a template for such a `Vehicle` object. Open a file `Vehicle.php` and write the following:

Vehicle.php

```php
<?php

/**
 * Vehicle class.
 */
class Vehicle {

  // Color of the vehicle.
  var $color = 'red'

  // Miles driven.
  var $milesDriven = 0;

  // Warranty available.
  var $warranty = 100000;

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return ($this->warranty - $this->milesDriven);
    }

    return 0;
  }
}

?>
```

Ignore the declaration `$this` in the code above for now. We'll explain it later. What we have defined above is a template for a vehicle. Such a template is called "class". It's not an object. The difference between class and object is akin to the difference between a person and you. Person is a common noun and there are billions of people roaming around in the world. On the other hand, there is only one of you. You have all the features of a person, but you are unique from every other person out there. If God were real and were to create another person, he'll use the same Person template and create one more. That new person will still be different from you. In the same way, there will be multiple objects created from a class template. Your car and my truck, both are vehicles but they are different from each other. To create (or instantiate) an object from a class, you use the `new` operator.

```php
$yourCar = new Vehicle();
$myTruck = new Vehicle();

// This will return FALSE.
echo ($yourCar === $myTruck);
```

If you compare the variables `$yourCar` and `$myTruck`, you will get `FALSE`. Notice the operator `===`. If you use `==` operator, you will see `TRUE` since `==` checks whether both the objects belong to the same class (which they do), but `===` checks whether the objects are identical or not.

Now let's check what the method `getWarrantyLeft()` returns. When you are writing procedural code, you can write the following code at the end of `Vehicle.php` and it will work:

```php
echo getWarrantyLeft();
```

But if you add the above code at the end of `Vehicle.php` and execute the file, you will get an error saying that `getWarrantyLeft()` is not defined. But you have defined this method. Then why is PHP complaining?

The reason is that you have defined it inside the class `Vehicle`. For it to run, it needs to know which object it needs to run on because the number of miles of warranty left on my truck can be different from the number of miles of warranty left on your car. To get the number for each vehicle, we need to invoke it with the following syntax:

```php
echo "Warranty left on your car: " . $yourCar->getWarrantyLeft() . "\n";
echo "Warranty left on my truck: " . $myTruck->getWarrantyLeft() . "\n";
```

Copy the above code and paste it at the end of `Vehicle.php` file. At this point, the file `Vehicle.php` looks as follows:

Vehicle.php

```php
<?php

/**
 * Vehicle class.
 */
class Vehicle {

  // Color of the vehicle.
  var $color = 'red'

  // Miles driven.
  var $milesDriven = 0;

  // Warranty available.
  var $warranty = 100000;

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return ($this->warranty - $this->milesDriven);
    }

    return 0;
  }
}

$yourCar = new Vehicle();
$myTruck = new Vehicle();

echo 'Warranty left on your car: ' . $yourCar->getWarrantyLeft() . "\n";
echo 'Warranty left on my truck: ' . $myTruck->getWarrantyLeft() . "\n";

```

```
?>
```

Now run this file using the command: `php Vehicle.php`

You will see the output:

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 100000
```

You are seeing the value 100000 because we have defined `$warranty` to be 100000 in the declaration on line 15 of `Vehicle.php` and `$milesDriven` to be 0 on line 12. As a result, 100000 miles of warranty is still left on both the vehicles.

Constructor

In the above example, what if my truck has manufacturer's warranty of 200000? How will I tell PHP about it? That's where the constructor comes into play. It's a function which gets executed when you create a new object from a class using the `new` operator. Constructor can take any number of arguments and that's where you define the initial properties of your object. Let's modify the `Vehicle` class to add a constructor.

Vehicle.php

```
/**
 * Class Vehicle.
 */
class Vehicle {

  // Color of the vehicle.
  var $color = 'red'

  // Miles driven.
  var $milesDriven = 0;

  // Warranty available.
  var $warranty;

  /**
   * Default constructor.
   *
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($warrantyProvided = 100000) {
    $this->warranty = $warrantyProvided;
  }

  /**
   * Returns miles of warranty left on the vehicle.
27     *
28     * @return int
29     *   Miles of warranty left on the vehicle.
30     */
31    function getWarrantyLeft() {
32      if ($this->warranty - $this->milesDriven > 0) {
33        return ($this->warranty - $this->milesDriven);
34      }
35
36      return 0;
37    }
38  }
```

You might have noticed that I used a variable `$this->warranty`, although I never defined `$this` anywhere. The `$this` pseudo-variable is automatically defined by PHP inside a method in a class and it refers to the object being called. If you want to refer to any object property from within a method, you will need to refer to it as `$this->{property_name}`. If you just write `${property_name}`, PHP will assume it's a new variable that's distinct from the property of the object. That's the reason we are using `$this->warranty` and `$this->milesDriven` even in `getWarrantyLeft()` function. If you had used `$warranty` and `$milesDriven` within `getWarrantyLeft()` function, then their values would not have been defined and the function would have returned `0`. Similarly to call any method of the class from within the object, you will need to call `$this->{method_name}()`. This concept may take a while to understand so read this paragraph again.

You might have also noticed that since I am setting the value of `$this->warranty` variable inside the constructor on line 22, I have removed its initial value from line 13. This is perfectly acceptable. Before you can call any method, such as `getWarrantyLeft()`, you need to initialize the object using the `new` operator. Initializing the object will automatically call the constructor. The constructor sets the value of `$this->warranty` variable so any method called on this object will have the value of the variable defined.

Since `$this->warranty` and `$warranty` are different variables inside the method of a class, we can rename `$warrantyProvided` variable to `$warranty` and the code will run perfectly fine.

```
Vehicle.php
```

```
1   /**
2    * Class Vehicle.
3    */
4   class Vehicle {
5
6     // Color of the vehicle.
7     var $color = 'red'
8
9     // Miles driven.
10    var $milesDriven = 0;
11
12    // Warranty available.
13    var $warranty;
14
15    /**
16     * Default constructor.
```

```php
   *
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($warranty = 100000) {
    $this->warranty = $warranty;
  }

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return ($this->warranty - $this->milesDriven);
    }

    return 0;
  }
}
```

In the `__construct()` method, we are initializing the object's `$warranty` variable to the one provided when initializing the object. As an example, if I were to initialize my truck as:

```php
$myTruck = new Vehicle(200000);
```

then `$warranty` property inside `$myTruck` object will be set to 200000. After making the above change, if you execute `Vehicle.php` file, then you will see the output:

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 200000
```

You can see that the warranty left on my truck is 200000 miles while that left on your car is 100000 miles. This data is stored in the object itself so in future, if you do any calculation based on warranty left on the vehicle, my truck will provide a different answer than your car. The constructor can take multiple arguments as inputs so we can initialize the `Vehicle` with different color and miles driven as well. Here is the completed code with those changes:

`Vehicle.php`

```php
<?php

/**
 * Vehicle class.
 */
class Vehicle {

  // Color of the vehicle.
  var $color;

  // Miles driven.
  var $milesDriven;

  // Warranty available.
  var $warranty;

  /**
   * Default constructor.
   *
   * @param string $color
   *  Color of the vehicle.
   * @param int $milesDriven
   *  Number of miles driven already.
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($color, $milesDriven = 0, $warranty = 100000) {
    $this->color = $color;
    $this->milesDriven = $milesDriven;
    $this->warranty = $warranty;
  }

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return ($this->warranty - $this->milesDriven);
    }

    return 0;
  }
}

$yourCar = new Vehicle('white');
$myTruck = new Vehicle('red', 25000, 200000);

echo 'Warranty left on your car: ' . $yourCar->getWarrantyLeft() . "\n";
echo 'Warranty left on my truck: ' . $myTruck->getWarrantyLeft() . "\n";

?>
```

We are defining your car to be white in color. Since we are not providing the miles driven by your car or the warranty, it will take the default values of 0 and 100000 respectively. On the other hand, my truck is defined to be red in color with 200000 miles of warranty out of which I have already driven 25000 miles. As a result, warranty left on your car should be full 100000 miles while that left on my truck should be 175000 miles. That's what we get when we execute the file above using the command: `php Vehicle.php`

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 175000
```

Congratulations!! If you understood this chapter, then you understand the basics of Object Oriented Programming, not just in PHP but in any language including C++, Java, C#, etc. You now know what are classes and objects and you understand the difference between them. You know how to define a class and instantiate objects of that class. You are knowledgeable about creating a constructor function within a class so that multiple objects can be initialized from the same class using different initial values of its properties.

In the next chapter, you'll learn about inheritance in PHP using which you can nest classes one below the other. This technique is very handy for code re-use as well making your code more secure.

Chapter 2

Inheritance

In Object Oriented PHP Programming chapter, you learned how to create classes and objects and how to use them in your code. In this post, we'll dig a little deeper and introduce the concept of inheritance. By end of this chapter, you'll learn the following:

- What is inheritance?
- Method Overriding
- Visibility of properties and methods

Concept

In Object-Oriented PHP post, we created a `Vehicle` class and initialized two objects: `$myTruck` and `$yourCar`. For this example, let's assume that your car is Honda Accord and my truck is Chevy Ram. Honda Accord has a trunk while Chevy Ram doesn't. To use the trunk, Honda Accord needs two methods: `putInTruck()` and `getFromTrunk()`. For simplicity, let's assume that trunk has space for only one thing and you can not put multiple items in the trunk. If we add these trunk related methods to `Vehicle.php`, they will be available for my truck as well, which doesn't have a trunk. We need to make sure that only your car supports the above two methods and not my truck. One easy way to deal with this is to delete the Vehicle class and use two separate classes: `HondaAccord` and `ChevyRam`.

```
This is caption
```

```php
<?php

/**
 * HondaAccord class.
 */
class HondaAccord {

  // Color of the vehicle.
  var $color;

```

```
// Miles driven.
var $milesDriven;

// Warranty available.
var $warranty;

// Stuff in the trunk.
var $stuff;

/**
 * Default constructor.
 *
 * @param string $color
 *  Color of the vehicle.
 * @param int $milesDriven
 *  Number of miles driven already.
 * @param int $warrantyProvided
 *   Number of miles of warranty provided with the vehicle.
 */
function __construct($color, $milesDriven = 0, $warranty = 100000) {
  $this->color = $color;
  $this->milesDriven = $milesDriven;
  $this->warranty = $warranty;
}

/**
 * Returns miles of warranty left on the vehicle.
 *
 * @return int
 *   Miles of warranty left on the vehicle.
 */
function getWarrantyLeft() {
  if ($this->warranty - $this->milesDriven > 0) {
    return ($this->warranty - $this->milesDriven);
  }

  return 0;
}

/**
 * Put stuff in trunk.
 *
 * @param mixed $stuff
 *   Stuff to be put in the trunk.
 */
function putInTrunk($stuff) {
  $this->stuff = $stuff;
}
```

```php

  /**
   * Take stuff out from the trunk.
   *
   * @return mixed
   *   Stuff returned from the trunk.
   */
  function takeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}

/**
 * ChevyRam class.
 */
class ChevyRam {

  // Color of the vehicle.
  var $color;

  // Miles driven.
  var $milesDriven;

  // Warranty available.
  var $warranty;

  /**
   * Default constructor.
   *
   * @param string $color
   *  Color of the vehicle.
   * @param int $milesDriven
   *  Number of miles driven already.
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($color, $milesDriven = 0, $warranty = 100000) {
    $this->color = $color;
    $this->milesDriven = $milesDriven;
    $this->warranty = $warranty;
  }

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return ($this->warranty - $this->milesDriven);
    }

    return 0;
  }
}

$yourCar = new HondaAccord('white');
$myTruck = new ChevyRam('red', 25000, 200000);

echo 'Warranty left on your car: ' . $yourCar->getWarrantyLeft() . "\n";
echo 'Warranty left on my truck: ' . $myTruck->getWarrantyLeft() . "\n";

$stuff = 'My Stuff';
$yourCar->putInTrunk($stuff);
echo "What is in Honda Accord's trunk: " . $yourCar->takeFromTrunk() . "\n";

?>
```

As you can see above, `ChevyRam` class is identical to the old `Vehicle` class. `HondaAccord` class is similar to the `Vehicle` class except that we have added a property `$stuff` and two methods, `putInTrunk()` and `takeFromTrunk()`, in it. This property and the two methods do not exist in `ChevyRam` class since it does not have a trunk. If you execute the above file, you will get the following result:

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 175000
What is in Honda Accord's trunk: My Stuff
```

The above code will work absolutely fine. The problem comes in maintaining it. Suppose you want to change how miles of warranty left on the vehicle is being calculated. You will have to update the method `getWarrantyLeft()` in both the classes, `HondaAccord` and `ChevyRam`. In future, you may want to add another method that applies to all the vehicles such as `getNumberOfTires()`, which returns the number of tires the vehicle runs on. Again you will need to add this method in both the classes: `HondaAccord` and `ChevyRam`. The problem is compounded when you have not just two but five different vehicles of different types. In order to reduce the complications in managing multiple classes, which are similar but not exactly the same, we can use inheritance.

At a very basic level, inheritance is the definition of parent-child relationship between two classes. In our example, both Honda Accord and Chevy Ram are vehicles. So we can define `HondaAccord` and `ChevyRam` classes as children of the `Vehicle` class. We use `extends` keyword to define a child class from a parent class:

`Vehicle.php`

```
<?php

```

```php
/**
 * Vehicle class.
 */
class Vehicle {

  // Color of the vehicle.
  var $color;

  // Miles driven.
  var $milesDriven;

  // Warranty available.
  var $warranty;

  /**
   * Default constructor.
   *
   * @param string $color
   *  Color of the vehicle.
   * @param int $milesDriven
   *  Number of miles driven already.
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($color, $milesDriven = 0, $warranty = 100000) {
    $this->color = $color;
    $this->milesDriven = $milesDriven;
    $this->warranty = $warranty;
  }

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return ($this->warranty - $this->milesDriven);
    }

    return 0;
  }
}

/**
 * HondaAccord class.
 */
```

```php
class HondaAccord extends Vehicle {

  // Stuff in the trunk.
  var $stuff;

  /**
   * Put stuff in trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  function putInTrunk($stuff) {
    $this->stuff = $stuff;
  }

  /**
   * Take stuff out from the trunk.
   *
   * @return mixed
   *   Stuff returned from the trunk.
   */
  function takeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}

/**
 * ChevyRam class.
 */
class ChevyRam extends Vehicle {

}

$yourCar = new HondaAccord('white');
$myTruck = new ChevyRam('red', 25000, 200000);

echo 'Warranty left on your car: ' . $yourCar->getWarrantyLeft() . "\n";
echo 'Warranty left on my truck: ' . $myTruck->getWarrantyLeft() . "\n";

$stuff = 'My Stuff';
$yourCar->putInTrunk($stuff);
echo "What is in Honda Accord's trunk: " . $yourCar->takeFromTrunk() . "\n";

?>
```

On executing the `Vehicle.php` file, we get the following output:

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 175000
What is in Honda Accord's trunk: My Stuff
```

This is identical to the output obtained earlier. You might be wondering that we haven't defined `getWarrantyLeft()` method in either `HondaAccord` or `ChevyRam` classes, then how is PHP returning the

```
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($color, $milesDriven = 0, $warranty = 100000) {
    $this->color = $color;
    $this->milesDriven = $milesDriven;
    $this->warranty = $warranty;
  }

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return ($this->warranty - $this->milesDriven);
    }

    return 0;
  }
}

/**
 * HondaAccord class.
 */
class HondaAccord extends Vehicle {

  // Stuff in the trunk.
  var $stuff;

  /**
   * Put stuff in trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  function putInTrunk($stuff) {
    $this->stuff = $stuff;
  }

  /**
   * Take stuff out from the trunk.
   *
   * @return mixed
   *   Stuff returned from the trunk.
   */
  function takeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}

/**
 * ChevyRam class.
 */
class ChevyRam extends Vehicle {

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return (2 * ($this->warranty - $this->milesDriven));
    }

    return 0;
  }

}

$yourCar = new HondaAccord('white');
$myTruck = new ChevyRam('red', 25000, 200000);

echo 'Warranty left on your car: ' . $yourCar->getWarrantyLeft() . "\n";
echo 'Warranty left on my truck: ' . $myTruck->getWarrantyLeft() . "\n";

$stuff = 'My Stuff';
$yourCar->putInTrunk($stuff);
echo "What is in Honda Accord's trunk: " . $yourCar->takeFromTrunk() . "\n";

?>
```

On executing the file, you will see the output:

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 350000
What is in Honda Accord's trunk: My Stuff
```

Notice that the miles of warranty left on Chevy Ram reflects the new calculation method but that on Honda Accord reflects the old calculation method. That's because `ChevyRam` class redefines (overrides) Vehicle class' method `getWarrantyLeft()` but `HondaAccord` class doesn't. That's why when you call `getWarrantyLeft()` method on `ChevyRam` object, the method defined in `ChevyRam` class will get executed and when you call the same method on `HondaAccord` object, the method defined in `Vehicle` class will get executed.

Visibility

One thing that we have completely omitted till now from the discussion of object oriented programming in PHP is the concept of visibility. Every property and method in a class can be defined to have one of three different types of visibilities:

- public
- protected
- private

If visibility is set to public, then that property or method can be accessed from anywhere outside the class. If visibility is set to protected, then that property or method can be accessed from the class itself or any of its child classes only. If visibility is set to private, then that property or method can be accessed from the class itself and nowhere else. If you haven't defined visibility explicitly for any property or method, then it's assumed to be public.

Benefits

You may ask why do we need visibility? Why not make all properties and methods public? For better separation of concern and security. Let's take an example. If you live in the US, you must be aware that it is illegal to tamper with the vehicle's odometer, which records the number of miles that the vehicle has driven. This means that `$milesDriven` should never go down and can only increase. But if visibility of `$milesDriven` is set to public, then any code outside the `Vehicle` class can easily change it back to 0. In other words, I could write the following code at the end of `Vehicle.php` file and set `$milesDriven` to 0 for my truck. This in turn increases the number of miles of warranty left on it.

```
$myTruck->milesDriven = 0;
echo 'Warranty left on my truck after setting miles driven to 0: ' .
    $myTruck->getWarrantyLeft() . "\n";
```

If you execute `Vehicle.php` file, you will get the following output:

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 350000
What is in Honda Accord's trunk: My Stuff
Warranty left on my truck after setting miles driven to 0: 400000
```

You can see that warranty left on my truck increased to 400000 miles. Setting `$milesDriven` to 0 shouldn't be allowed. To prevent such an operation, let's set visibility of `$milesDriven` to be protected. Just to make the code more readable, let's explicitly define visibility of all other properties and methods to be public.

Vehicle.php

```php
<?php

/**
 * Vehicle class.
 */
class Vehicle {

  // Color of the vehicle.
  public $color;

  // Miles driven.
  protected $milesDriven;

  // Warranty available.
  public $warranty;

  /**
   * Default constructor.
   *
   * @param string $color
   *  Color of the vehicle.
   * @param int $milesDriven
   *  Number of miles driven already.
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($color, $milesDriven = 0, $warranty = 100000) {
    $this->color = $color;
    $this->milesDriven = $milesDriven;
    $this->warranty = $warranty;
  }

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  public function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return ($this->warranty - $this->milesDriven);
    }

    return 0;
  }
}

```

```
/**
 * HondaAccord class.
 */
class HondaAccord extends Vehicle {

  // Stuff in the trunk.
  public $stuff;

  /**
   * Put stuff in trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  public function putInTrunk($stuff) {
    $this->stuff = $stuff;
  }

  /**
   * Take stuff out from the trunk.
   *
   * @return mixed
   *   Stuff returned from the trunk.
   */
  public function takeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}

/**
 * ChevyRam class.
 */
class ChevyRam extends Vehicle {

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  public function getWarrantyLeft() {
    if ($this->warranty - $this->milesDriven > 0) {
      return (2 * ($this->warranty - $this->milesDriven));
    }

    return 0;
  }

}

$yourCar = new HondaAccord('white');
$myTruck = new ChevyRam('red', 25000, 200000);

echo 'Warranty left on your car: ' . $yourCar->getWarrantyLeft() . "\n";
echo 'Warranty left on my truck: ' . $myTruck->getWarrantyLeft() . "\n";

$stuff = 'My Stuff';
$yourCar->putInTrunk($stuff);
echo "What is in Honda Accord's trunk: " . $yourCar->takeFromTrunk() . "\n";

$myTruck->milesDriven = 0;
echo 'Warranty left on my truck after setting miles driven to 0: ' .
    $myTruck->getWarrantyLeft() . "\n";

?>
```

Now `$milesDriven` can only be accessed by the class `Vehicle` and its child classes: `ChevyRam` and `HondaAccord`. On executing the `Vehicle.php` file, you will see the following error:

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 350000
What is in Honda Accord's trunk: My Stuff
PHP Fatal error:  Cannot access protected property ChevyRam::$milesDriven in Vehicle.php
    on line 110
```

PHP is giving an error because we are accessing the property `$milesDriven` by code that is outside the `Vehicle` class or any of its child classes. This is a good thing because now no other code can decrease the value of `$milesDriven` variable even by mistake. Next we need to add a method to increase `$milesDriven` variable. Let's call that method `drive()`.

```
/**
 * Drive the vehicle. This will add to miles driven.
 *
 * @param int $miles
 *   Number of miles driven in the current trip.
 */
public function drive($miles) {
  if ($miles > 0) {
    $this->milesDriven += $miles;
  }
}
```

In `drive()` method, we are first checking if the input is greater than 0 and if it is, then we add the input to `$milesDriven` variable. This way we ensure that `$milesDriven` will never decrease. There is only one problem now. What if in future, you add one more child of `Vehicle` class and then decrease

`$milesDriven` from that child class? PHP will not prevent it because visibility of `$milesDriven` is set to protected. This means that any child class will be able to access that property and change its value. To make the code even more idiot-proof, let's set visibility of `$milesDriven` to private and add a new public method `getMilesDriven()` that child classes can use to calcuate the miles of warranty left. Now even child classes will not be able to decrease the value of the property `$milesDriven` but they can call the method `getMilesDriven()` to get its value. Here is how the updated `Vehicle.php` file looks:

Vehicle.php

```php
<?php

/**
 * Vehicle class.
 */
class Vehicle {

  // Color of the vehicle.
  public $color;

  // Miles driven.
  private $milesDriven;

  // Warranty available.
  public $warranty;

  /**
   * Default constructor.
   *
   * @param string $color
   *  Color of the vehicle.
   * @param int $milesDriven
   *  Number of miles driven already.
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($color, $milesDriven = 0, $warranty = 100000) {
    $this->color = $color;
    $this->milesDriven = $milesDriven;
    $this->warranty = $warranty;
  }

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  public function getWarrantyLeft() {
    if ($this->warranty - $this->getMilesDriven() > 0) {
      return ($this->warranty - $this->getMilesDriven());
    }

    return 0;
  }

  /**
   * Drive the vehicle. This will add to miles driven.
   *
   * @param int $miles
   *   Number of miles driven in the current trip.
   */
  public function drive($miles) {
    if ($miles > 0) {
      $this->milesDriven += $miles;
    }
  }

  /**
   * Returns the number of miles driven in total.
   *
   * @param int
   *   Total number of miles driven.
   */
  public function getMilesDriven() {
    return $this->milesDriven;
  }
}

/**
 * HondaAccord class.
 */
class HondaAccord extends Vehicle {

  // Stuff in the trunk.
  public $stuff;

  /**
   * Put stuff in trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  public function putInTrunk($stuff) {
    $this->stuff = $stuff;
  }

  /**
```

```php
   * Take stuff out from the trunk.
   *
   * @return mixed
   *   Stuff returned from the trunk.
   */
  public function takeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}

/**
 * ChevyRam class.
 */
class ChevyRam extends Vehicle {

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  public function getWarrantyLeft() {
    if ($this->warranty - $this->getMilesDriven() > 0) {
      return (2 * ($this->warranty - $this->getMilesDriven()));
    }

    return 0;
  }

}

$yourCar = new HondaAccord('white');
$myTruck = new ChevyRam('red', 25000, 200000);

echo 'Warranty left on your car: ' . $yourCar->getWarrantyLeft() . "\n";
echo 'Warranty left on my truck: ' . $myTruck->getWarrantyLeft() . "\n";

$stuff = 'My Stuff';
$yourCar->putInTrunk($stuff);
echo "What is in Honda Accord's trunk: " . $yourCar->takeFromTrunk() . "\n";

// Drive the truck for 25000 more miles.
$myTruck->drive(25000);
echo 'Warranty left on my truck after for 25000 more miles: ' .
    $myTruck->getWarrantyLeft() . "\n";

```

```
?>
```

On executing the file `Vehicle.php`, you will see the output:

```
$ php Vehicle.php
Warranty left on your car: 100000
Warranty left on my truck: 350000
What is in Honda Accord's trunk: My Stuff
Warranty left on my truck after driving for 25000 more miles: 300000
```

As expected, miles of warranty left on Chevy Ram reduced to 300000 miles.

Awesome!! You now know the concepts of inheritance, method overriding and visibility. In the next chapter, we'll introduce the concept of dependency injection.

Chapter 3

Dependency Injection

In this chapter, you'll learn:

- What is dependency injection?
- Different types of dependency injections
- Benefits of dependency injection
    1. Decoupling code
    2. Unit testing

What is dependency injection?

Let's start with the example where we left off in the previous chapter of inheritance in PHP. We have two child classes: `HondaAccord` and `ChevyRam`, both extending the parent class `Vehicle`. `Vehicle` has `getWarrantyLeft()`, `drive()` and `getMilesDriven()` methods. `getWarrantyLeft()` method is being overridden by `ChevyRam`. `HondaAccord` has a trunk and implements `putInTrunk()` and `takeFromTrunk()` methods. `Vehicle` also has `$color`, `$warranty` and `$milesDriven` properties.

Let's extend the example. Assume that every person owns one Honda Accord. Create such a `Person` class in `Person.php` file. Put this `Person.php` file in the same directory as `Vehicle.php` file.

`Person.php`

```php
<?php

require_once 'Vehicle.php';

/**
 * Person class.
 */
class Person {

  // Vehicle
  public $vehicle;

  /**
   * Default constructor.
   */
  function __construct() {
    $this->vehicle = new HondaAccord('red');
  }

  /**
   * Let the person travel in his vehicle.
   *
   * @param int $miles
   *   Number of miles that the person travels.
   */
  public function travel($miles) {
    $this->vehicle->drive($miles);
  }

  /**
   * Returns the number of miles that the person has traveled.
   *
   * @return int
   *   Number of miles that the person has traveled.
   */
  public function getDistanceTraveled() {
    return $this->vehicle->getMilesDriven();
  }
}

$you = new Person();
$you->travel(2000);
echo 'Distance traveled: ' . $you->getDistanceTraveled() . "\n";

?>
```

Following is the output when executing `Person.php` file:

```
$ php Person.php
Distance traveled: 2000
```

Notice that in the constructor of `Person` class, we are initializing the vehicle he owns to be a red Honda Accord. This code works fine as long as we expect all the people to own a red Honda Accord. But obviously this is not the case in real life. In Chapter 2 on Inheritance, we had a case where I owned a red Chevy Ram truck while you owned a white Honda Accord. So how do we initialize two people, you and me, such that we both own different vehicles? The easiest way is to remove the initialization of the vehicle from the `Person` class. Instead initialize it outside and pass the initialized `Vehicle` object to the `Person` class in the constructor. This is how the file `Person.php` will look after this change:

```
Person.php
```

```php
<?php

require_once 'Vehicle.php';

/**
 * Person class.
 */
class Person {

  // Vehicle
  public $vehicle;

  /**
   * Default constructor.
   *
   * @param Vehicle
   *   Vehicle object.
   */
  function __construct($vehicle) {
    $this->vehicle = $vehicle;
  }

  /**
   * Let the person travel in his vehicle.
   *
   * @param int $miles
   *   Number of miles that the person travels.
   */
  public function travel($miles) {
    $this->vehicle->drive($miles);
  }

  /**
   * Returns the number of miles that the person has traveled.
   *
   * @return int
   *   Number of miles that the person has traveled.
   */
  public function getDistanceTraveled() {
    return $this->vehicle->getMilesDriven();
  }
}

$hondaAccord = new HondaAccord('white');
$you = new Person($hondaAccord);
$you->travel(2000);
echo "Distance traveled by you: " . $you->getDistanceTraveled() . "\n";

```

```
$chevyRam = new ChevyRam('red');
$me = new Person($chevyRam);
$me->travel(5000);
echo "Distance traveled by me: " . $me->getDistanceTraveled() . "\n";

?>
```

Following is the output when executing `Person.php` file:

```
$ php Person.php
Distance traveled by you: 2000
Distance traveled by me: 5000
```

In lines 44-52 above, we initialized the `Vehicle` object outside the `Person` class and passed the `Vehicle` object to the `Person` class when initializing a person. Earlier since `Vehicle` object was initialized within the `Person` class, `Person` class had a hard dependency on the `Vehicle` object. Now that we are passing an initialized `Vehicle` object to the `Person` class, the dependency has been reduced to an extent. We can initialize any type of vehicle and pass it to the person being initialized. The `Person` class doesn't really care what type of `Vehicle` object is being passed as long as that object implements `drive()` and `getMilesDriven()` methods. This is dependency injection. Earlier `Person` class was dependent on the `Vehicle` class but now we are initializing the `Vehicle` object elsewhere and injecting it into the `Person` object. If you understood what we did here, you understand what dependency injection is.

Types of Dependency Injections

There are multiple types of dependency injections, depending on how `Vehicle` object or any of its children is passed to the `Person` object.

Constructor Injection

The code above uses constructor injection. That's because `HondaAccord` or `ChevyRam` object is being passed to the `Person` class during initialization and the mapping of this passed object to the class' internal property is being handled by the constructor.

Setter Injection

In our example demonstrating constructor injection, we have assumed that every person has a vehicle. But that's not true in real life, is it? In general, people are not born with vehicles and at some point, they buy a vehicle, if at all. Let's modify the `Person` class to reflect this.

`Person.php`

```
<?php

require_once 'Vehicle.php';

/**
```

```
 * Person class.
 */
class Person {

  // Vehicle
  private $vehicle;

  /**
   * Buy a vehicle.
   *
   * @param Vehicle $vehicle
   *   Vehicle to buy.
   */
  public function buy($vehicle) {
    $this->vehicle = $vehicle;
  }

  /**
   * Let the person travel in his vehicle.
   *
   * @param int $miles
   *   Number of miles that the person travels.
   */
  public function travel($miles) {
    $this->vehicle->drive($miles);
  }

  /**
   * Returns the number of miles that the person has traveled.
   *
   * @return int
   *   Number of miles that the person has traveled.
   */
  public function getDistanceTraveled() {
    return $this->vehicle->getMilesDriven();
  }
}

$you = new Person();
$yourCar = new HondaAccord('white');
$you->buy($yourCar);
$you->travel(2000);
echo "You have traveled for " . $you->getDistanceTraveled() . " miles in your Honda
    Accord.\n";

$me = new Person();
$myTruck = new ChevyRam('red');
$you->buy($myTruck);
```

```php
$you->travel(2000);
echo "I have traveled for " . $you->getDistanceTraveled() . " miles in my Chevy Ram.\n";

?>
```

Instead of passing `HondaAccord` or `ChevyRam` object to the `Person` class during initialization, we are first initializing a `Person` object and then making him buy the `Vehicle` object. Method `buy()` sets `$this->vehicle` property to the object that is passed. Hence this is called setter injection.

Interface Injection

In interface injection, instead of passing an initialized object such as `HondaAccord` or `ChevyRam` to the `Person` object, an interface such as `VehicleInterface` is passed. We'll cover this in Chapter 4 on Interface. This is the type of injection that is being used most often in Drupal 8.

Benefits

Decoupled code, which leads to more flexible architecture

The reason we used dependency injection in the above example is to increase the code flexibility. Before using dependency injection, every person was initialized to own a red Honda Accord. After using dependency injection, every person will own any vehicle that you pass to him, and not necessary a red Honda Accord. In our examples, `$me` bought a red Chevy Ram and `$you` bought a white Honda Accord. Interfaces, explained in Chapter 4, will take decoupling one step further.

Easier Unit Testing

The second advantage of dependency injection is the ability to unit test the classes `Vehicle` and `Person` independently of each other. Consider the initial code when `Person` was initializing the `Vehicle` object in its constructor. This is how a PHPUnit test for `Person` class will look:

PersonTest.php

```php
<?php

/**
 * PersonTest class.
 */
class PersonTestCase extends \PHPUnit_Framework_TestCase {

  /**
   * Make sure that distance traveled is correct.
   */
  public function testDistanceTraveled() {
    $person = new Person();
    $person->travel(1000);
    $this->assertEquals(1000, $person->getDistanceTraveled(), 'Distance traveled does
        not match.');
  }
}

?>
```

The above test will pass. Let's assume that in future, there is a mistake in the implementation of `getMilesDriven()` in the `Vehicle` class. `getMilesDriven()` function in `Vehicle` class starts returning 0 irrespective of the miles actually driven. Now the above test will fail since `getDistanceTraveled()` method, which in turn calls `Vehicle`'s `getMilesDriven()` method, returns 0 and does not match 1000. This is undesirable. Why should a test testing `Person` class fail if there is a mistake in the `Vehicle` class? Agreed that in the above code, it's very easy to debug what's going on but when the project grows, just figuring the root cause of a failing test can become time-consuming.

The expectation is that a unit test testing the `Person` class should fail if and only if there is a problem in the `Person` class and not anywhere else. This is where dependency injection is useful. Now that we are using setter injection, we can use mocks in PHPUnit to isolate the tests for `Person` class from bugs in any other class. Here is how the PHPUnit test for `Person` class will look:

```
PersonTest.php
```

```
<?php

/**
 * PersonTest class.
 */
class PersonTestCase extends \PHPUnit_Framework_TestCase {

  /**
   * Make sure that distance traveled is correct.
   */
  public function testDistanceTraveled() {
    // Create a stub for VehicleInterface.
    $stub = $this->getMockBuilder('Vehicle')->getMock();

    // Configure the stub to return 1000 whenever getMilesDriven() method is called.
    $stub->method('getMilesDriven')->willReturn(1000);

    $person = new Person();
    $person->buy($stub);
    $person->travel(1000);
    $this->assertEquals(1000, $this->getDistanceTraveled(), 'Distance traveled does not
        match.');
  }
}

?>
```

The test code above is slightly more complicated than the one earlier. First we are creating a mock object for

Vehicle and instructing the mock to return 1000 whenever its method getMilesDriven() is called. Next we pass this mock object to the Person object using buy() method. As a result, in the assertEquals() statement, $this->getDistanceTraveled() will invoke getMilesDriven() method of the mock object and will return 1000 as programmed. You will notice that in the unit test, we have eliminated the dependency on any external class. If the implementation of getMilesDriven() method in the Vehicle or any of its child classes has bugs, the above test will pass. So in future, if the above test starts failing, then we know for sure that there is a problem in the Person class and nowhere else. It becomes much easier to debug.

In this chapter, you learned about dependency injection and its benefits. In the next chapter, you'll learn about interface and how it works with dependency injection to make the architecture even more loosely coupled.

Chapter 4

Interface

In this chapter, you'll learn:

- What is an interface?
- Benefits of interface
    1. Make architecture flexible and modular
    2. Circumvent the lack of multiple inheritance in PHP

What is an interface?

Interface defines the methods that a class implementing the interface must implement. Interfaces don't define the implementation, which is left to the implementing class. Any class that implements an interface needs to define at least the methods prescribed by the interface, although it can implement more methods of its own as well.

If you got lost reading the above paragraph, don't worry. You are not alone. It's much easier to understand interfaces based on the benefits it provides. So let's start with that.

Benefits of using an interface

Make architecture flexible and modular

Let's start with the example where we left off in the previous chapter of dependency injection. We have two child classes: `HondaAccord` and `ChevyRam` extending the parent class `Vehicle`. We also defined a `Person` class. It is being passed the vehicle he or she owns using setter injection through the `buy()` method.

Look through the code of the `Person` class in Chapter 3 on dependency injection. Here is how it looks:

```
Person.php
```

```
<?php
```

```php

require_once 'Vehicle.php';

/**
 * Person class.
 */
class Person {

  // Vehicle
  public $vehicle;

  /**
   * Default constructor.
   *
   * @param Vehicle
   *   Vehicle object.
   */
  function __construct($vehicle) {
    $this->vehicle = $vehicle;
  }

  /**
   * Let the person travel in his vehicle.
   *
   * param int $miles
   *   Number of miles that the person travels.
   */
  public function travel($miles) {
    $this->vehicle->drive($miles);
  }

  /**
   * Returns the number of miles that the person has traveled.
   *
   * @return int
   *   Number of miles that the person has traveled.
   */
  public function getDistanceTraveled() {
    return $this->vehicle->getMilesDriven();
  }
}

$hondaAccord = new HondaAccord('white');
$you = new Person($hondaAccord);
$you->travel(2000);
echo "Distance traveled by you: " . $you->getDistanceTraveled() . "\n";

$chevyRam = new ChevyRam('red');
```

```php
$me = new Person($chevyRam);
$me->travel(5000);
echo "Distance traveled by me: " . $me->getDistanceTraveled() . "\n";

?>
```

You will notice that in the `Person` class, the only methods of the `Vehicle` class that are being used are `drive()` and `getMilesDriven()`. So if we replace the `Vehicle` object with any other object which has these two methods, and then let the person buy that object, the code will work without any problem. This concept is important to understand so read this paragraph again.

An example could be a bullock cart. Based on how we have described our `Vehicle` class, with warranty and color, bullock cart is definitely not a vehicle. But a person can still buy it, travel in it and then get the number of miles traveled in it. So for the `Person` class, bullock cart is a practical replacement for a vehicle. How does the Person class specify that it can buy and travel in any object that defines two methods: `drive()` and `getMilesDriven()`? That's where an interface comes in handy.

Using an interface, we can define the methods that any class implementing it should have. The implementing class can have more methods but it should at least have the methods that the interface defines otherwise PHP will throw an error. In our case, we can create `VehicleInterface` which has only two methods: `drive()` and `getMilesDriven()`. Both `Vehicle` and `BullockCart` classes will implement this interface, and `Person` class can buy and drive the class that implements it. Here is how the `VehicleInterface` looks in `VehicleInterface.php` file (again put this file in the same directory as `Vehicle.php` file).

```
VehicleInterface.php
```

```php
<?php

/**
 * Interface VehicleInterface
 */
interface VehicleInterface {

  /**
   * Drive the vehicle. This will add to the miles driven.
   *
   * @param int $miles
   *   Number of miles driven.
   */
  public function drive($miles);

  /**
   * Return the number of miles driven.
   *
   * @return int
   *   Number of miles driven.
   */
  public function getMilesDriven();
}

```

```
?>
```

Class `Vehicle.php` now implements `VehicleInterface`.

```
Vehicle.php
```

```php
<?php

require_once 'VehicleInterface.php';

/**
 * Vehicle class.
 */
class Vehicle implements VehicleInterface {

  // Color of the vehicle.
  public $color;

  // Miles driven.
  private $milesDriven;

  // Warranty available.
  public $warranty;

  /**
   * Default constructor.
   *
   * @param string $color
   *  Color of the vehicle.
   * @param int $milesDriven
   *  Number of miles driven already.
   * @param int $warrantyProvided
   *   Number of miles of warranty provided with the vehicle.
   */
  function __construct($color, $milesDriven = 0, $warranty = 100000) {
    $this->color = $color;
    $this->milesDriven = $milesDriven;
    $this->warranty = $warranty;
  }

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  public function getWarrantyLeft() {
    if ($this->warranty - $this->getMilesDriven() > 0) {
      return ($this->warranty - $this->getMilesDriven());
    }

    return 0;
  }

  /**
   * Drive the vehicle. This will add to miles driven.
   *
   * @param int $miles
   *   Number of miles driven in the current trip.
   */
  public function drive($miles) {
    if ($miles > 0) {
      $this->milesDriven += $miles;
    }
  }

  /**
   * Returns the number of miles driven in total.
   *
   * @param int
   *   Total number of miles driven.
   */
  public function getMilesDriven() {
    return $this->milesDriven;
  }
}

/**
 * HondaAccord class.
 */
class HondaAccord extends Vehicle {

  // Stuff in the trunk.
  public $stuff;

  /**
   * Put stuff in trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  public function putInTrunk($stuff) {
    $this->stuff = $stuff;
  }

  /**
   * Take stuff out from the trunk.
```

```php
   *
   * @return mixed
   *   Stuff returned from the trunk.
   */
  public function TakeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}

/**
 * ChevyRam class.
 */
class ChevyRam extends Vehicle {

  /**
   * Returns miles of warranty left on the vehicle.
   *
   * @return int
   *   Miles of warranty left on the vehicle.
   */
  public function getWarrantyLeft() {
    if ($this->warranty - $this->getMilesDriven() > 0) {
      return (2 * ($this->warranty - $this->getMilesDriven()));
    }

    return 0;
  }
}

?>
```

We also have the new `BullockCart` class that implements the `VehicleInterface`.

BullockCart.php

```php
<?php

require_once 'VehicleInterface.php';

/**
 * BullockCart class.
 */
class BullockCart implements VehicleInterface {

  // Miles driven.
  private $miles;

  /**
   * Default constructor.
   */
  function __construct() {
    $this->miles = 0;
  }

  /**
   * Drive the bullock cart.
   *
   * @param int $miles
   *   Miles driven.
   */
  public function drive($miles) {
    $this->miles += $miles;
  }

  /**
   * Returns the total miles driven.
   *
   * @return int
   *   Total miles driven.
   */
  public function getMilesDriven() {
    return $this->miles;
  }
}

?>
```

Finally we have the `Person` class that uses `VehicleInterface` instead of `Vehicle`. As a result, any object that implements `VehicleInterface` can be used with `Person` class without breaking the code.

Person.php

```
<?php

require_once 'Vehicle.php';
require_once 'BullockCart.php';

/**
 * Person class.
 */
class Person {

  // VehicleInterface
  public $vehicle;

  /**
15    * Buy.
16    *
17    * @param Vehicle $vehicle
18    *   Vehicle to buy.
19    */
20   public function buy(VehicleInterface $vehicle) {
21     $this->vehicle = $vehicle;
22   }
23 
24   /**
25    * Let the person travel in his vehicle.
26    *
27    * param int $miles
28    *   Number of miles that the person travels.
29    */
30   public function travel($miles) {
31     $this->vehicle->drive($miles);
32   }
33 
34   /**
35    * Returns the number of miles that the person has traveled.
36    *
37    * @return int
38    *   Number of miles that the person has traveled.
39    */
40   public function getDistanceTraveled() {
41     return $this->vehicle->getMilesDriven();
42   }
43 }
44 
45 $you = new Person();
46 $yourCar = new HondaAccord('white');
47 $you->buy($yourCar);
48 $you->travel(2000);
49 echo "You have traveled for " . $you->getDistanceTraveled() . " miles in your Honda
       Accord.\n";
50 
51 $me = new Person();
52 $bullockCart = new BullockCart();
53 $me->buy($bullockCart);
54 $me->travel(5000);
55 echo "I have traveled for " . $me->getDistanceTraveled() . " miles in my bullock
       cart.\n";
```

You can see in lines 45-55 above that I bought a bullock cart and traveled in it for 5000 miles (poor bullock!) using almost the same code as you bought a Honda Accord and traveled in it for 2000 miles. `Person` class, as a result, is no longer dependent on the `Vehicle` class. In fact, it can work with any class that implements the `VehicleInterface`. This adds a lot of flexibility to our code and makes the architecture modular.

On a side note, the argument to the `buy()` method changed from `Vehicle` class to `VehicleInterface`. This type of dependency injection is called Interface Injection.

Interfaces, combined with dependency injection which we learned in the previous chapter, make it very easy to replace one class by another without making much changes in the code. In real life, a simple example is interacting with the database. `MySQL` class, `Postgres` class, `Oracle` class can all implement the `DatabaseInterface` which provides a few methods such as `select()`, `insert()`, etc. Now you can write your application without worrying about what database it is using as long as your application code uses the methods specified in the interface. If in future, MySQL in your application gets replaced by Oracle, it's as easy as initializing Oracle class instead of MySQL and using it without any further changes in your code.

Circumvent the lack of multiple inheritance in PHP

As you already know, `HondaAccord` has two methods: `putInTrunk()` and `takeFromTrunk()` that `ChevyRam` does not. In real life, there are other objects which are not vehicles but have a trunk and you can put something in their trunk and take something out. A simple example is any container, such as a suitcase. You can put something in it and later take something out. So theoretically we can define a `Container` class with a property `$stuff` and two methods: `putInTrunk()` and `takeFromTrunk()`. Then we can make `HondaAccord` a child of `Container` class and it will inherit these methods. But there's a rub. `HondaAccord` already extends the `Vehicle` class. Unfortunately PHP does not allow a child class to extend two parent classes (multiple inheritance). This means that `HondaAccord` class can't extend `Container` class as well. Instead we use an interface.

Define a `ContainerInterface` with two methods: `putInTrunk()` and `takeFromTrunk()`. `Container` class will implement these methods and so will `HondaAccord` class. Once `HondaAccord` implements `ContainerInterface`, we can use it anywhere `ContainerInterface` can be used. Here is how `ContainerInterface`, `Container` class and `HondaAccord` class will look:

`ContainerInterface.php`

```php
<?php

/**
 * Interface ContainerInterface
 */
interface ContainerInterface {

  /**
   * Put something trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  public function putInTrunk($stuff);

  /**
   * Take something out from the trunk.
   *
   * @return mixed
   *   Stuff to be taken out from the trunk.
   */
  public function takeFromTrunk();
}

?>
```

```
Container.php
```

```
<?php

require_once 'ContainerInterface.php';

/**
 * Class Container
 */
class Container implements ContainerInterface {

  // Stuff
  private $stuff;

  /**
   * Put stuff in the trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  public function putInTrunk($stuff) {
    $this->stuff = $stuff;
  }

  /**
   * Take stuff out from the trunk.
   *
   * @return mixed
   *   Stuff to be taken out from the trunk.
   */
  public function takeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}

?>
```

```
require_once 'ContainerInterface.php';

/**
```

```
 * Class HondaAccord
 */
class HondaAccord extends Vehicle implements ContainerInterface {

  // Stuff in the trunk.
  public $stuff;

  /**
   * Put stuff in trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  public function putInTrunk($stuff) {
    $this->stuff = $stuff;
  }

  /**
   * Take stuff out from the trunk.
   *
   * @return mixed
   *   Stuff returned from the trunk.
   */
  public function takeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}
```

Note that although in PHP, a child class can extend only one parent class, there is no such limitation on the number of interfaces a class can implement. A class can implement 1, 2 or even 100 interfaces. Once you make a class implement an interface, you can use it anywhere that interface is being used. The downside is that the class implementing the interface still needs to define the implementation itself. It can not get it from its interface, as can be done in case of inheritance. In my opinion, that makes the code ugly and prevents code reuse. That's where traits come in, which we'll discuss in the next chapter.

Chapter 5

Traits

In this chapter, you'll learn:

- What is a trait?
- When to use traits?

What is a trait?

In Chapter 4 on interfaces, we made `HondaAccord` class implement the `ContainerInterface` because it has a trunk. As a result, we could use `HondaAccord` class wherever `ContainerInterface` is used. But the code looks ugly! Since interfaces can't provide implementation, the methods `putItTrunk` and `takeFromTrunk` needed to be implemented in both the classes. This prevents code reuse. That's where traits come to the rescue.

Traits help you embed a set of properties and methods in several independent classes that could be living in different hierarchies. Here is the `ContainerTrait` trait created in `ContainerTrait.php` file.

```
/**
 * ContainerTrait trait.
 */
trait ContainerTrait {
  // Stuff in the trunk.
  public $stuff;

  /**
   * Put stuff in trunk.
   *
   * @param mixed $stuff
   *   Stuff to be put in the trunk.
   */
  public function putInTrunk($stuff) {
    $this->stuff = $stuff;
  }

  /**
   * Take stuff out from the trunk.
   *
   * @return mixed
   *   Stuff returned from the trunk.
   */
  public function takeFromTrunk() {
    $stuff = $this->stuff;
    unset($this->stuff);
    return $stuff;
  }
}
```

Now we could modify `Container` and `HondaAccord` classes as follows:

```
1 require_once 'ContainerInterface.php';
2 require_once 'ContainerTrait.php';
3
4 /**
5  * Class Container
6  */
7 class Container implements ContainerInterface {
8   use ContainerTrait;
9 }
```

```
1 require_once 'ContainerInterface.php';
2 require_once 'ContainerTrait.php';
3
4 /**
5  * Class HondaAccord
6  */
7 class HondaAccord extends Vehicle implements ContainerInterface {
8   use ContainerTrait;
9 }
```

Notice that on line 8 of both the files, we are using `use ContainerTrait;` to include `ContainerTrait` in each of the classes. Just by using this statement, both `HondaAccord` and `Container` class can now use the properties defined in the `ContainerTrunk` as its own. As an example, add the following code to the end of `HondaAccord.php`.

```
$yourCar = new HondaAccord('white');
$yourCar->putInTrunk('umbrella');
echo "Things taken out from the trunk: " . $yourCar->takeFromTrunk() . "\n";
```

On executing the `HondaAccord.php` file, you'll see the following output:

```
$ php HondaAccord.php
Things taken out from the trunk: umbrella
```

Note that we didn't have to define the methods `putInTrunk()` and `takeFromTrunk()`, and the property `$stuff` in `HondaAccord` class but we could still use these since `HondaAccord` class used `ContainerTrait`. These methods and properties will be available in whichever class we use this trait. This makes for very efficient code reuse.

If you followed everything till this point, then you understand traits well enough to be able to follow Drupal 8 code. PHP spec and rules about traits are quite extensive but those are outside the scope of this book. If you want to read them, go to http://php.net/manual/en/language.oop5.traits.php.

When to use traits?

Use traits when you want a lot of independent classes to have the same method or property without duplicating code in all these classes. In this chapter, we saw how using `ContainerTrait` helped us reduce code in `Container` and `HondaAccord` classes. Another example is `ColorTrait`. You could define the property `$color` and methods `getColor()` and `setColor()`. This trait could be used in any class which tries to mimic a real object having color.