Introducing Object Oriented Programming

Chapter 15 of

A Web-Based Introduction Programming, Fourth Edition

(to be published Summer/Fall 2017)

Intended Learning Outcomes

After completing this chapter, you should be able to:

- Explain the significance of Object-Oriented Programming (OOP)
- Summarize the general structure and purpose of an object.
- Interpret a class definition for a simple object that includes private class variables and public get and set methods.
- Create instances of a class using the new operator with the class constructor.
- Use an object's public methods to work with class instances.
- Create a class definition for a simple object that includes private class variables, and public get and set methods.
- Use the $this variable to refer to class variables inside methods, to distinguish these from local variables.
- Add a custom constructor method to a class.
- Distinguish between the private, protected, and public access modifiers.
- Explain the meaning and value of class inheritance.
- Create a child class that extends (inherits from) a parent class.
- Explain the meaning and purpose of the abstract keyword.
- Label a class as abstract to ensure that it cannot be instantiated.
- Define abstract methods to ensure that these methods appear in child classes.
- Explain the meaning and purpose of encapsulation, instantiation, inheritance, method overloading, method over-riding, and polymorphism.
- Recognize the important relationships between object oriented design, user interfaces, and data sources.

Introduction

As we have seen, programming is essentially concerned with performing useful work on data. We have also learned from our work with data files and databases that our applications usually work with groups of related data values, for example: data related to an **employee** might consist of the employee's ID, social security number, first name, last name, job title, starting date, and hourly wage; a **game character** might consist of a name, current score, health rating, and various skill ratings; a **bank account** might contain an account ID and a current balance; a **weather station reading** might consist of a temperature, wind speed, wind direction, precipitation amount, humidity level, and cloud conditions; an **item for sale** might consist of a stock number, a sale price, a description, and other details such as size, color, and weight; an **email message** might consist of message headers (to, from, cc, and bcc fields), a subject, and the email message; even a **rectangle** on the computer screen must be described by a group of individual data items such as height, width, x and y coordinates (position), and fill color.

It is helpful to consider any group of related data values as a single entity. Consider an employee: instead of working with many separate variables such as the employee's ID, first and last names, job title, hourly wage, etc., it would be far more efficient if we could simply work with an employee as a complete item, referenced by a single variable. This employee entity could then “contain” all of the different data values, or **attributes**, that describe the employee, and the attributes could also be associated with a set of functions that process the employee data in useful ways. For example we could call an employee-related function to change the last name of an employee, or call another employee-related function to calculate an employee's weekly pay.

This approach, combining a group of related data with the functions that operate on the data, is the basis of **Object Oriented Programming (OOP). OOP** provides a powerful modular approach to the way that we work with groups of related data. In the language of OOP, an entity such as an employee is termed an **object**. An object consists of a set of data values (attributes) combined with a set of operations (methods) that allow us to work with the object's data in any way needed to meet our requirements.

OOP brings enormous benefits. We have already discussed the value of code modularity in modern software design. Objects can be designed and developed independently of any application, and application programmers can simply make use of these pre-defined objects in their code without reinventing the wheel. An employee object can be passed between modules very easily , as a single item. And the attributes and methods that make up an object can be maintained, modified and tested independently of the applications that use the object.

It would take far too much space to examine OOP in great detail in this introductory textbook, and PHP is not the most suitable language to develop object oriented code (Java and Python are examples of fully object oriented languages). But every programmer should be aware of the general concepts and application of OOP. In this chapter we will introduce you to the design and use of objects, explain some common OOP terminology, and provide some simple working examples in PHP that will get you started and prepare you for further study in subsequent courses or personal research.

What is an Object?

Simply stated, an **object** combines a group of related data values (usually known as the object's **attributes**, **fields** or **class variables**) with a set of functions (usually known as the object's **methods**) that perform useful operations on these fields. The code structure that defines an object is known as a **class**. For example a simplified **Employee class** might contain the following class variables that, taken together, define an employee:

```
empID                          // the employee's ID
firstName                      // the employee's first name
lastName                       // the employee's last name
jobTitle                       // the employee's job title
hourlyWage                     // the employee's hourly wage
```

The Employee class might also provide the following methods that would allow a programmer to perform useful operations on these five variables:

```
getID()                      // retrieve the employee's ID
setID($id)                   // modify the employee's ID
getFirstName()               // retrieve the employee's first name
setFirstName($firstName)     // modify the employee's first name
getLastName()                // retrieve the employee's last name
setLastName($lastName)       // modify the employee's last name
getJobTitle()                // retrieve the employee's job title
setJobTitle($jobTitle)       // modify the employee's job title
getHourlyWage()              // retrieve the employee's hourly wage
setHourlyWage($hourlyWage)   // modify the employee's hourly wage
getWeeklyPay ()              // calculate the employee's weekly pay
findEmployee($id)            // search the employee data source
                             // (file or database) for a record
                             // that matches the ID, and assign
                             // this employee's data to the
                             // class variables
addEmployee()                // add the employee to the employee
                             // data system (file or database)
```

As you work through this chapter you will learn how to code this class, and to develop applications that make use of the class. Of course a real Employee class would have many more data values and allow many more operations. Once an Employee class has been defined, your applications can generate copies of the class as needed, one copy for each employee that the application needs to work with. In Object Oriented jargon each individual copy (each employee) is called an **instance** of the Employee class, and each instance has its own class variables, separate from any other instances. For example a copy of the Employee class used for an employee named "Chris Smith" would be one Employee instance, and a copy of the Employee object used for an employee named "Mary King" would be another, separate, Employee instance. Once an instance has been created, your application can use the object's methods to: assign values to the class fields for that instance; retrieve these values; and perform other useful operations. For example, the application might use the setFirstName() method to set an instance's first name field to "Chris", and the setLastName() method to set the same instance's last name field to "Smith".

Creating and Using Instances of a Class

In order to work with an instance of an Employee in your application, you must first create, or **instantiate**, the instance. Every object class includes a special method called a **constructor method**, which always has the same name as the class name, and this is used with a special operator, the **new** operator to create a new instance. The instance can then be assigned to a variable for use in your application. The constructor method of the **Employee** class will be automatically named **Employee()**; here's how to create an instance of Employee and assign this instance to a variable named **$emp1**:

```
$emp1 = new Employee();
```

And here's how to create a second instance and assign it to a variable named **$emp2**:

```
$emp2 = new Employee();
```

We now have two instances (or copies) of the Employee class, and the methods that are a part of the Employee class can now be used to work with each instance's class variables as needed. Here's how you can use the Employee methods that we listed above to assign values to each of the class variables of the $emp1 instance:

```
$emp1->setID("012345");
$emp1->setFirstName("Chris");
$emp1->setLastName("Smith");
$emp1->setJobTitle("Manager");
$emp1->setHourlyWage(36.50);
```

As you can see, there is a special syntax that must be used when we want to work with an instance of a class; we use the name of the variable that refers to the instance (in this case **$emp1**), followed by the **->** characters (no spaces between these), followed by the name of the class method that we wish to use, for example **setID()**. You can use the same class methods to assign values to the **$emp2** instance:

```
$emp2->setID("345678");
$emp2->setFirstName("Mary");
$emp2->setLastName("King");
$emp2->setJobTitle("Accountant");
$emp2->setHourlyWage(35.75);
```

If you wanted to print the ID and job title of the employee stored in the $emp1 instance, you can do this using the getID() and getJobTitle() methods of the Employee class:

```
print ("<p>The job title of employee #".$emp1->getID().
        " is ".$emp1->getJobTitle().".</p>");
```

The list of Employee class methods also included a method called getWeeklyPay() that will calculate the pay. You could obtain the weekly pay of $emp1 and store it in a variable named **$emp1Pay** as follows:

```
$emp1Pay = $emp1->getWeeklyPay();
```

And to obtain the weekly pay for $emp2 and store it in a variable named **$emp2Pay** :

```
$emp2Pay = $emp2->getWeeklyPay();
```

Note that, as an application programmer, you no longer need to code the weekly pay calculations, or even know how these calculations are performed. You only need to call the object's getWeeklyPay() method which will return the pay to your application. To do this you just need to know what class methods and variables are provided by the Employee class.

Using Employee Objects in an Application

Review the list of methods that are contained in the Employee class, and the purpose of each method. Let's look at two simple applications that each use instances of Employee to work with employee data in different ways. First, consider the following requirement:

*Requirement: develop a Web application named **new-employee** that provides the user with a form to obtain the ID, first name, last name, job title, and hourly wage of a new employee. Use an Employee instance to add this data to the company's data source. The code that defines the Employee class is provided for you in a file named inc-employee-object.php so this file must be included in your code. Note that you don't need to know anything about the data source (which may be a file or database) since this is handled by the addEmployee() method which is part of the Employee class.*

The Web form is simple, and can be coded as follows:

```
<h1>Add New Employee Form</h1>
<form action="new-employee.php" method="post">
      <table>
      <tr><td>ID:</td><td><input type="text" name="id"></td></tr>
      <tr><td>First Name:</td>
                  <td><input type="text" name="firstName"></td></tr>
      <tr><td>Last Name:</td><td><input type="text" name="lastName"></td></tr>
      <tr><td>Job Title:</td><td><input type="text" name="jobTitle"></td></tr>
      <tr><td>Hourly Wage:</td><td><input type="text" name="wage"></td></tr>
      </table>

      <p><input type = "submit" value = "Add Employee">
</form>
```

Code Example: new-employee.html

The PHP code that processes this form must:

Include the code that provides the Employee class definition
Retrieve the employee's attributes from the form data
Create an Employee instance
Call appropriate Employee methods to assign the employee's attributes to the instance
Call the appropriate Employee method to add the employee to the data source

Here is the code for new-employee.php:

```
      <?php
      include("inc-employee-object.php");

      $id = $_POST["id"];
      $firstName = $_POST["firstName"];
      $lastName = $_POST["lastName"];
      $title = $_POST["jobTitle"];
      $hourlyWage = $_POST["wage"];

      $emp = new Employee();
```

```
$emp->setID($id);
$emp->setFirstName($firstName);
$emp->setLastName($lastName);
$emp->setJobTitle($title);
$emp->setHourlyWage($hourlyWage);

$emp->addEmployee();

print("<p>The new employee ($id) has been added to the employee
file.</p>");
?>
```

Code Example: new-employee.php

This code first includes the code (**inc-employee-object.php**) that defines the Employee class (we will examine this code later in the chapter), then retrieves the five values from the form submitted by the user (the new employee's ID, first name, last name, job title and hourly wage). The next statement creates a new Employee instance and assigns this to a variable named **$emp**. This is followed by five statements that call the setID(), setFirstName(), setLastName(), setJobTitle(), and setHourlyWage() methods of the Employee class to assign the values from the form to the five fields in the **$emp** instance. These values are now stored in the instance but they have not yet been added to the data system. To do this we must use the addEmployee() method which has been coded to add the values of the class fields as a new record in the employee data source, which will be described later in the chapter. Note that the application programmer doesn't need to know the names of the class fields in the Employee class, or the connection to, or structure of, the data sources that contains the employee data, or even whether the data is being stored in a file or database; he or she only needs to know the names and purpose of each of the methods that will perform the different tasks.

Now let's look at another application that calculates the weekly pay for two employees as follows:

Requirement: develop a Web application named employee-pay that provides the user with a form to obtain the ID for an employee. Use an Employee instance to find the employee in the data source, then displays the employee's first name, last name and weekly pay. The code that defines the Employee class is provided for you in a file named inc-employee-object.php so this file must be included in your code.

In this case the Web form only needs to obtain the employee's ID and hours worked from the user:

```
<h1>Weekly Pay Calculator</h1>
<form action="employee-pay.php" method="post">
<table>
<tr><td>Employee ID</td><td><input type="text" name="id"></td></tr>
</table>
<p><input type = "submit" value = "Calculate Weekly Pay">
</form>
```

Code Example: employee-pay.html

The PHP code that process this form must:

Include the code that provides the Employee class definition
Retrieve the employee's ID from the form data
Create an Employee instance
Call the appropriate Employee method to find the employee with the requested ID
Call the appropriate Employee methods to print the first names, last names, and weekly pay of the employee.

Here is the code for employee-pay.php:

```
<?php
include("inc-employee-object.php");

$id = $_POST["id"];
$emp1 = new Employee();
$emp1->findEmployee($id);

print ("<p>Weekly Pay for ".$emp1->getFirstName().
            " ". $emp1->getLastName().": $".
            $emp1->getWeeklyPay()."</p>");
?>
```

Code Example: employee-pay.php

This code first includes the code that defines the Employee object (we will examine this code later in the chapter), then retrieves the form data submitted by the user (the employee's ID). The next statement creates an Employee instance. The next statement uses the object's **findEmployee()** method to retrieve the data for the employee from the data system and store this data in the instance's class variables. The application then calls the **getFirstName()**, **getLastName()**, and **getWeeklyPay()** methods to display the employee's first name, last name, and weekly pay.

Once again note how elegant our application code has become; the application programmer only needs to know the names and purpose of the object's methods to work with an Employee instance. Do you see from these examples how greatly this simplifies the work of the application programmer, and how the Employee objects can be used in different ways to meet the requirements of different applications? Can you think of other ways that Employee objects might be used by an application?

Defining an Object

We have seen two examples showing how our application might **use** an Employee object. But we still don't know how the Employee class is actually coded, what it actually looks like. In fact, many application programmers use object classes without actually seeing the class code. They simply work from the class documentation to create instances and use the class methods for the purposes of their application. Other programmers design, code, and documents object classes for use by application programmers. Let's look at the code of the Employee class so we can get a better understanding of how object classes are designed and coded. The class fields of our simplified Employee class are as follows:

```
empID                        // the employee's ID
firstName                    // the employee's first name
lastName                     // the employee's last name
jobTitle                     // the employee's job title
hourlyWage                   // the employee's hourly wage
```

The functions (methods) that will allow a programmer to work with these fields are as follows:

```
findEmployee(id)             // read an employee from the data source
addEmployee()                // add the employee to the data source
getID()                      // to obtain the employee's ID
setID(id)                    // to modify the employee's ID
getFirstName()               // to obtain the employee's first name
setFirstName(firstName)      // to modify the employee's first name
getLastName()                // to obtain the employee's last name
setLastName(lastName)        // to modify the employee's last name
getJobTitle()                // to obtain the employee's job title
setJobTitle(jobTitle)        // to modify the employee's job title
getHourlyWage()              // to obtain the employee's hourly wage
setHourlyWage(hourlyWage)    // to modify the employee's hourly wage
getWeeklyPay ()              // to calculate the employee's weekly pay
```

Each of these methods perform a very specific and often quite simple task, so that programmers who need to work with objects of this class can easily perform any operation that is required. The idea is that a program that needs to work with employee data for one or more employees can create an **instance** of Employee for each employee and then use the object's methods to work with any employee's data (class variables) as needed. Of course this is a simplified example. In a real world application we would include all kinds of additional data, as well as functions designed to handle tasks related to health insurance, retirement contributions, tax exemptions and tax calculations.

Coding the Object Class

Here is how an Employee object class might actually be defined in PHP (this definition is provided in the **inc-employee-object.php** file in your **samples** folder). You will see that object class definitions incorporate some operators and other features that we have not seen before:

```
<?php
class Employee
{
      private $empID;
      private $firstName;
      private $lastName;
      private $jobTitle;
private $hourlyWage;

public function getID()
{
	return $this->empID;
}

public function setID($empID)
{
	$this->empID = $empID;
}

public function getFirstName()
{
	return $this->firstName;
}

public function setFirstName($fName)
{
	$this->firstName = $fName;
}

public function getLastName()
{
	return $this->lastName;
}

public function setLastName($lName)
{
	$this->lastName = $lName;
}

public function getJobTitle()
{
	return $this->jobTitle;
}

public function setJobTitle($title)
{
	$this->jobTitle = $title;
}

public function getHourlyWage()
{
	return $this->hourlyWage;
}

public function setHourlyWage($hourlyWage)
{
	$this->hourlyWage = $hourlyWage;
}

public function getWeeklyPay()
{
```

```
            return number_format ($this->hourlyWage * 40, 2);
        }

        public function addEmployee()
        {
            $empRecord = $this->empID.", ".$this->firstName.
                  ", ".$this->lastName.", ".$this->jobTitle.
                  ", ".$this->hourlyWage."\n";

            $empFile = fopen("employees.txt", "a");
            fputs($empFile, $empRecord);
            fclose($empFile);

        } // end of addEmployee method

        public function findEmployee($id)
        {
            $empFile = fopen("employees.txt", "r");
            $empRecord = fgets($empFile);
            $notFound = true;
            while (!feof($empFile) or $notFound)
            {
                  list ($empID, $fName, $lName, $title, $wage) =
                                    explode(",", $empRecord);
                  if ($id == $empID)
                  {
                        $this->empID = $empID;
                        $this->firstName = $fName;
                        $this->lastName = $lName;
                        $this->jobTitle = $title;
                        $this->hourlyWage = $wage;
                        $notFound = false;
                  }
                  $empRecord = fgets($empFile);
            }
            fclose($empFile);
        } // end of findEmployee method

  } // end of class definition
  ?>
```

Code Example: inc-employee-object.php

The entire class definition for the Employee object begins with a class heading that consists of the keyword **class** followed by the name of the class. The entire body of the class definition is enclosed in **{ }** braces. The variables that will store the data values of the object are known as **class variables** (also known as class fields or attributes). The class variables are listed separately from (and usually above) the class methods. The class variables are used to store the object's data values. Note that the definitions of each of the class variables of the Employee class are preceded by the keyword **private**, which is an **access modifier**. This means that these variables can only be accessed or modified by the methods that are part of this class; they cannot be accessed or modified directly by other classes or by applications that use the class. Defining these variables as private ensures that the only way to work with them is by using the class methods that are provided. We say that the private class variables are **encapsulated** by the class methods. If for some reason you wanted to allow applications to directly access or modify any of these variables you could declare them as **public**, but usually we want to avoid this: encapsulation is a key feature of object oriented programming: by forcing applications to use the methods that have been provided, we reduce the chance of badly written application code using the data incorrectly. There are actually three different access modifiers, the third is **protected**. The **protected** access modifier is similar to **private**, but **protected** class variables may also be accessed and modified by methods in other classes that **inherit** this class; we will discuss inheritance and the use of the **protected** modifier later in the chapter.

The **class methods** are listed below the class variables. As you can see the methods are defined as functions, just like any other functions in PHP. The methods are generally defined as **public**, which means they can be used by other classes or applications. The public class methods therefore provide a **programming interface** between the class variables and any application that uses instances of the class; the only way to work with private class variables is to use the public class methods. If for any reason you wanted to restrict access to some methods so that they can only be called by other methods within this class, you could define them as **private** (or **protected**), just like the class variables.

Most class methods are designed to perform a simple operation of some kind that usually involves accessing or modifying one or more class variables. Some methods include parameters since they must be sent values when they are called, other do not need parameters. For example the **getHourlyWage()** method simply **returns** the value stored in the **$hourlyWage** class variable, so this method does not require any parameters. On the other hand, the **setHourlyWage()** method receives a value from an application and **stores** this in the **$hourlyWage** class variable, so this method includes a parameter to receive the hourly wage.

It is standard practice to use the words **get** or **set** followed by the name of a class variable when naming methods that return or update the values of class variables. This approach ensures that the method names clearly indicate their purpose as well as which class variable they are working with. These types of method are often referred to as **getter** and **setter** methods.

You will notice that the code inside the class methods uses an unfamiliar syntax to refer to the class variables. For example, rather than referring to **$hourlyWage**, the **getHourlyWage()** method refers to the variable as **$this->hourlyWage**. Remember that a function can be considered a small independent program. Without the use of the special identifier **$this**, the method would assume that **$hourlyWage** is a variable for use only within the function. The use of **$this->hourlyWage** refers to **this** instance's **class** variable **$hourlyWage**; this ensures that the class-level variables are not confused with any variables defined inside the functions that may have the same name.

You can see the importance of this if you look at the **setHourlyWage()** method. This method contains a parameter named **$hourlyWage**. The method is designed to assign the value received in the $hourlyWage parameter to the class variable **$hourlyWage**. But the parameter has the same name as the class variable. The statement **$this->hourlyWage = $hourlyWage** avoids any ambiguity by using **$this->hourlyWage** to specify that the value received in the method's **parameter** variable **$hourlyWage** is to be assigned to the **class** variable **$hourlyWage**. The use of the **$this** identifer ensures that these two variables are not confused although both have the same name.

As you can see, get and set methods are provided for each class variable. This ensures that any application that works with Employee instances can access and modify any of these variables as needed. There are also three other methods in the Employee class. The **getWeeklyPay()** method is designed to multiply the employee's hourly wage by 40, and return this as the weekly pay, formatted to two places (of course in a real world application the calculation would depend on the actual hours worked, but here we'll keep it simple). The **findEmployee()** method receives an employee ID and uses this to search through a data source to locate that employee's attributes (first name, last name, job title, and hourly wage). If it finds the employee, the method then assigns these values to the instance's class variables so the instance now contains information concerning a specific employee. Note that the method is actually coded to search through a text file named **employees.txt** in which individual data values are separated by commas, but this could be changed to search through a different data source (usually of course, employee data would be stored in a database table and not a text file; a text file is used here simply to keep the focus on object design, and to serve readers who might have skipped the MySQL chapter). The application programmer who uses the findEmployee() method doesn't need to know anything about where the employee data is stored; he or she just calls this method to obtain an employee's data. The other method **addEmployee()** simply takes the values stored in the instance's class variables and adds these as a new record in the data source. In this case, they are appended to the employees.txt file, but, once again, this code could be changed if the employee data was relocated to a different file or database.

Rather than thinking about the requirements for a specific application, the designer of an object class such as the Employee class considers all the different ways that programmers might need to work with the Employee data for different applications, and develops methods to allow for all these different uses. Often class variables and methods are added or modified as new requirements are identified.

Creating and Using Instances of an Object Class

In order to use a class, a programmer needs to know the name of the class and also what methods are available to work with the data that the class provides. Now that the Employee class has been defined, we can write applications that create and use **instances** of this class as needed, one instance for each employee that our application needs to process. Each instance will contain its own set of class variables, and the class methods can be used as needed to work with these values. We have already looked at two examples of applications that work with instances of the Employee class.

The Class Constructor Method

We have seen that each new instance of an Employee object is created by using the **new** operator to call a special **constructor** method, which has the same name as the class. In this case the constructor method is named **Employee()**. The constructor method automatically creates a new **copy** (instance) of the object for use by the program. The constructor method is only used to create new instances. Once the instance has been created, the other functions (methods) in the Employee class can be used to work with the class variables.

A default version of the class constructor method is provided automatically as a part of the class definition, so you don't have to create it in your code. However if you want to ensure that certain tasks are always performed whenever a new instance of a class is created, you can add your own constructor method to the class and add any code that you want. If you do this, your customized constructor will replace the default constructor, and will be used whenever a new instance of the class is created. You can also include parameters if you want arguments to be passed to the constructor method. Constructor methods in PHP must use the following format (note the two underscores):

```
public function __construct()
{
      // code that you want execute whenever a new instance is created
}
```

Note that there are **two** underlines in the standard method name for constrcutors: **__construct()**. When would you want to write your own constructor method? One example might be if you always wanted to obtain values from a file or database in order to initialize the class variables whenever a new instance is created. This would ensure that your instance contains values before any other methods are used. In this case you could add a parameter to receive an employee ID, and include a call to the findEmployee() method in the body of your constructor to look up the ID in a file or database and assign the employee's data values to the instance's data fields:

```
public function __construct($id)
{
      $this->findEmployee($id);
}
```

New instances of Employee will now be created using your customized constructor, so any application that needs to create a new Employee instance will need to send an ID to the constructor, for example:

```
$emp = new Employee("012345");
```

Alternatively, you could specify **five** parameters for the constructor (in the case of the Employee class) and code statements in the constructor to assign these to the instance's five class variables:

```
public function __construct($id, $firstName, $lastName, $title, $wage)
{
      $this->empID = $id;
      $this->firstName = $firstName;
      $this->lastName = $lastName;
      $this->jobTitle = $title;
      $this->hourlyWage = $wage;
```

```
    }
```

Now any programmer that needs to create a new Employee instance will always need to send five values to the constructor, for example:

```
$emp = new Employee("555666", "Jane", "Doe", "Sales", 17.50);
```

Method Overloading

You might be thinking that all three of the constructors listed above might be useful at different times. Sometimes you might want to use the constructor with no arguments, to create an instance without assigning any values to the class variables. Sometimes you might want to use a constructor that receives an employee ID as an argument, and uses this to retrieve the employee data from a file or database. And sometimes you might want to use a constructor that you can send five arguments (an employee's ID, first name, last name, job title and hourly wage), so the constructor can assign these values to the class variables when the instance is created. If you were coding this class using a fully object oriented language like Java, you can provide multiple constructors in this way, each with a different parameter list. The compiler will determine which constructor method is intended based on the number and type of arguments sent to the constructor. The term for using multiple methods with the same name but different parameters is **method overloading**. Unfortunately, PHP does not allow method overloading since it is not a compiled language. As a result only one constructor is allowed, and although there are workarounds these can be quite complicated.

Why do Objects Matter?

There are many reasons to follow the Object Oriented approach. First, just consider the number of variables that the application would need just to work with a single employee. In our example we just use five data values for each employee but in the real world, there would be many more, for example: employee ID, first name, middle name, last name, job title, department, office phone, office location, email address, hourly wage or salary, tax exemptions, accumulated vacation, accumulated sick leave, type of health insurance, etc.. Imagine creating and dealing with all these variables, not just for one employee but for many employees! It is much easier to just assign an Employee object to a single variable such as **$emp**, knowing that **all** of these data values are now associated with this single variable, and that all the class methods associated with these data values are available for your use.

The Employee object can be used by any number of applications. If additional fields are needed they can be added to the object so and these applications can be updated to use the latest version. Also the methods in the Employee object can be updated without having to change the code in the applications that use the object; for example the getWeeklyPay() might be updated to meet new policy requirements. And new methods can be added if needed.

Furthermore, the use of a standard Employee object ensures that all application programmers are working with the same methods and procedures and not writing their own custom code. The Employee class might even be shared between different institutions or offices, making it easier for Employee objects to be shared more easily between organizations and allowing operations in Human Resources offices to be more streamlined as applications become standardized and more easily maintained.

The object oriented approach is fundamental to most modern application design and development; application programmers are often required to familiarize themselves with object classes that have been provided for their use: they must read the class documentation and learn the names and characteristics of each class's attributes and methods. For example graphics programmers frequently need to work with all kinds of standard graphics objects, such as points, lines, rectangles, and circles. Rather than code these themselves, these programmers simply learn to make use of pre-written classes of common graphics objects. Here's how a graphic artist might display a moon, first creating a $moon instance of a Circle class, and then applying various Circle methods:

```
$moon = new Circle();
$moon->setRadius (40);
$moon->fillColor (white);
$moon->setPosition (20, 40);
$moon->draw();
```

And here's another example, where a game programmer makes use of a GameCharacter class to develop a game character named "Hercules":

```
$hercules = new GameCharacter();
$hercules->setName ("Hercules");
$hercules->setPosition (20, 40);
$hercules->setPoints (0);
$hercules->setHealth (10);
$hercules->fight();
```

These are both simplified and made-up examples to demonstrate how different kinds of objects are used for a wide range of purposes. In both examples the application programmer makes use of pre-written classes to develop his or her application. The OOP approach allows a high degree of standardization and reusability that greatly simplifies and speeds up application development. It allows data to be exchanged easily between applications and organizations. As a result, OOP has quickly become the standard approach for most modern application development, including development of mobile apps, data driven Web sites, scientific and health-related applications, games, and much more.

Object Design and Inheritance

Another key concept of OOP is an **object hierarchy**, where the class variables and functions of one object class are **inherited** by another object class, which then **extends** these with additional variables and methods for a more specific purpose. Inheritance plays a critical role in object oriented design, since it ensures that different but related classes can be developed without duplication, and also that new classes follow certain design requirements. To understand the general principle of inheritance, let's take another look at our Employee class. Our earlier version of this class used five class variables: $empID, $firstName, $lastName, $jobTitle, and $hourlyWage. Four of these ($empID, $firstName, $lastName, and $jobTitle), are valid for **all** employees, but **$hourlyWage** might not always be appropriate. For example some employees might receive an an annual **salary** instead of an hourly wage.

One solution to this problem would be to forget about the Employee class altogether and just create two separate classes: a SalariedEmployee class and an HourlyEmployee class. These two classes would each contain the same class variables and methods that we used in the Employee class, except that the SalariedEmployee class would contain a $salary class variable, and getSalary() and setSalary() methods, while the HourlyWage class would contain an $hourlyWage class variable, along with getHourlyWage() and setHourlyWage() methods. In addition the two classes would need different code inside their getWeeklyPay() methods: the getWeeklyPay() method in the SalariedEmployee class would divide the annual salary by 52 to calculate the weekly pay, whereas the getWeeklyPay() method in the HourlyEmployee class would multiply the hourly wage by 40 to calculate the weekly pay. They would also need slightly different addEmployee() and findEmployee() methods since these would would need to refer either the $salary or $hourlyWage variables. Once we have created these two classes, we can just create instances of each class as needed, depending on whether an employee is salaried or hourly.

The problem with this approach is that much of the code in these two classes is duplicated. Four of the class variables are the same, as well as the get and set methods related to these four variables. This can create serious problems in terms of maintenance and updates to the code. Whenever changes are needed to the class variables and methods that are common to both classes, these must now be updated in **both** classes. This not only means more work but increases the possibility of an error: a programmer updating one class may not realize that the same code must also be updated in the other class.

The solution is to **keep** our original Employee class, but modify it so that it only contains the class variables and methods that are common to both salaried **and** hourly employees. Then create new SalariedEmployee and HourlyEmployee classes that **inherit** the variables and methods of the Employee class, and also add the class variables and methods that are specific to their own type of employee. We say that these classes **extend** the Employee class. In other words the Employee class will be the same as before except that it will no longer include the $hourlyWage class variable or the getHourlyWage() and setHourlyWage() methods. The SalariedEmployee class will **extend** the Employee class by providing the $salary class variable and the getSalary() and setSalary() methods, while the HourlyEmployee class will **extend** the Employee class by providing the $hourlyWage class variable, and getHourlyWage() and setHourlyWage() methods. And each of these classes will also provide their own getWeeklyPay(), addEmployee(), and findEmployee() methods, since these methods will require different code depending on whether the employee is salaried or hourly. When one class extends another, the extending class inherits all of the class variables and methods of the class that is being extended. We often say that an extending class is a **child** class of the extended class, and the extended class is the **parent** class of the extending class.

It is very simple to create a class that extends another: for example, to create a SalariedEmployee class that extends the Employee class we simply add **extends Employee** to the header of the SalariedEmployee class:

```
	class SalariedEmployee extends Employee
	{
		// class variables and methods here
	}
```

Similarly, to create an HourlyEmployee class that extends the Employee class:

```
class HourlyEmployee extends Employee
	{
		// class variables and methods here
	}
```

Since the methods of the two extending classes must be able to work with the class variables of the Employee class ($empID, $firstName, $lastName, and $jobTitle), the access modifier that was specified in the Employee class for these variables must be changed from **private** to **protected**. That's because the **private** modifier only allows methods in the **same** class to access these variables, whereas the **protected** modifier allows methods in the same class **and also** classes that **extend** the class to access the variables.

In order to separate this inheritance version from the previous Employee example, your **samples** folder includes a sub-folder named **oo-inheritance**. The **inc-employee-object.php** file in this sub-folder contains the complete code for **all three** classes, the revised Employee class as well as the SalariedEmployee and HourlyEmployee classes (these classes could also have been stored in three separate include files).

Here is the code for the revised Employee class, followed by the code for the SalariedEmployee and HourlyEmployee classes. For the moment don't worry about the use of a new keyword **abstract** that appears in the heading and three method headings in the Employee class; this will be explained later in the chapter.

```
<?php
abstract class Employee
{
	protected $empID;
	protected $firstName;
	protected $lastName;
	protected $jobTitle;

	public function getID()
	{
		return $this->empID;
	}

	public function setID($empID)
	{
		$this->empID = $empID;
```

```
      }

      public function getFirstName()
      {
            return $this->firstName;
      }

      public function setFirstName($fName)
      {
            $this->firstName = $fName;
      }

      public function getLastName()
      {
            return $this->lastName;
      }

      public function setLastName($lName)
      {
            $this->lastName = $lName;
      }

      public function getJobTitle()
      {
            return $this->jobTitle;
      }

      public function setJobTitle($title)
      {
            $this->jobTitle = $title;
      }

      abstract public function getWeeklyPay();
      abstract public function addEmployee();
      abstract public function findEmployee($id);
} // end of Employee class definition

class SalariedEmployee extends Employee
{
      private $salary;

      public function addEmployee()
      {
            $empRecord = $this->empID.", ".$this->firstName.",
                  ".$this->lastName.", ".$this->jobTitle.",
                  ".$this->salary."\n";
            $empFile = fopen("employees.txt", "a");
            fputs($empFile, $empRecord);
            fclose($empFile);
      }

      public function findEmployee($id)
      {
            $empFile = fopen("employees.txt", "r");
            $empRecord = fgets($empFile);
            $notFound = true;
            while (!feof($empFile) or $notFound)
            {
                  list ($empID, $fName, $lName, $title,
                        $salary) = explode(",", $empRecord);
                  if ($id == $empID)
                  {
                        $this->empID = $empID;
                        $this->firstName = $fName;
                        $this->lastName = $lName;
                        $this->jobTitle = $title;
                        $this->salary = $salary;
                        $notFound = false;
                  }
                  $empRecord = fgets($empFile);
            }
            fclose($empFile);

      }

      public function getSalary()
      {
            return $this->salary;
      }

      public function setSalary($salary)
      {
            $this->salary = $salary;
      }

      public function getWeeklyPay()
      {
            return number_format ($this->salary/52, 2);
      }

} // end of SalariedEmployee class definition

class HourlyEmployee extends Employee
{
      private $hourlyWage;

      public function addEmployee()
      {
            $empRecord = $this->empID.", ".$this->firstName.",
                  ".$this->lastName.", ".$this->jobTitle.",
                  ".$this->hourlyWage."\n";
            $empFile = fopen("employees.txt", "a");
            fputs($empFile, $empRecord);
            fclose($empFile);
      }

      public function findEmployee($id)
      {
            $empFile = fopen("employees.txt", "r");
            $empRecord = fgets($empFile);
            $notFound = true;
            while (!feof($empFile) or $notFound)
            {
                  list ($empID, $fName, $lName, $title, $wage) =
                              explode(",", $empRecord);
                  if ($id == $empID)
                  {
                        $this->empID = $empID;
                        $this->firstName = $fName;
                        $this->lastName = $lName;
                        $this->jobTitle = $title;
                        $this->hourlyWage = $wage;
                        $notFound = false;
                  }
                  $empRecord = fgets($empFile);
            }
            fclose($empFile);

      }

      public function getHourlyWage()
      {
            return $this->hourlyWage;
      }

      public function setHourlyWage($hourlyWage)
      {
            $this->hourlyWage = $hourlyWage;
      }

      public function getWeeklyPay()
      {
            return number_format ($this->hourlyWage * 40, 2);
      }
} // end of HourlyEmployee class definition

?>
```

Code Example: inheritance version of inc-employee-object.php

Review the code of each class carefully. Notice how the class variables and methods are distributed between the three classes: the Employee class provides the class variables that are common to all employees (**$empID**, **$firstName**, **$lastName**, and **$jobTitle**) along with the get and set methods used with these variables, and these class variables are now declared to be **protected** so that they are accessible to methods in any class that extends Employee; the SalariedEmployee class **extends** Employee (so it has access to the class variables and methods of the Employee class), and adds just one class variable, **$salary**, along with **getSalary()** and **setSalary()** methods; the HourlyEmployee class also extends Employee and adds one class variable **$hourlyWage**, along with **getHourlyWage()** and **setHourlyWage()** methods. SalariedEmployee and HourlyEmployee each contains its own version of the **getWeeklyPay()**,

addEmployee(), and **findEmployee()** methods; for example, the getWeeklyPay() method in SalariedEmployee calculates the pay by dividing the salary by 52, while the getWeeklyPay() method in HourlyEmployee calculates the pay by multiplying the hourly wage by 40.

The **oo-inheritance** sub-folder also includes two applications that demonstrate how these new classes might be used: a modified version of the **employee-pay** application asks the user for an employee ID and employee type (salaried or hourly), then creates an instance of either SalariedEmployee or HourlyEmployee, as appropriate, and displays the employee's weekly pay; a modified version of the **new-employee** application, obtains the data values for a new employee, as well as the employee type (salaried or hourly) then creates an instance of either SalariedEmployee or HourlyEmployee, as appropriate, assigns the data values to the instance, then calls the addEmployee() method to update the employee file.

Here is the revised form for **employee-pay.html**:

```
<body>
      <h1>Weekly Pay Calculator</h1>
      <form action="employee-pay.php" method="post">
            <table>
                  <tr>  <td>Employee ID</td>
                        <td><input type="text" name="id"></td></tr>
                  <tr><td>Employee Type</td>
                        <td><select name = "empType">
                        <option>Salaried</option>
                        <option>Hourly</option>
                        </select></td></tr>
            </table>
            <p><input type = "submit" value = "Calculate Weekly Pay">
      </form>
</body>
```

Code Example: inheritance version of employee-pay.html

And here is the revised code for employee-pay.php, that works with an instance of either SalariedEmployee or HourlyEmployee depending on the form input:

```
<body>
<?php
      include("inc-employee-object.php");

      $id = $_POST["id"];
      $empType = $_POST["empType"];

      if ($empType=="Salaried")
                  $emp1 = new SalariedEmployee();
      else
                  $emp1 = new HourlyEmployee();

      $emp1->findEmployee($id);

      print ("<p>Weekly Pay for ".$emp1->getFirstName().
            " ". $emp1->getLastName().": $".
            $emp1->getWeeklyPay()."</p>");
?>
</body>
```

Code Example: inheritance version of employee-pay.php

A selection structure determines whether the employee instance **$emp1** is declared as a SalariedEmployee or HourlyEmployee instance. The $emp1 instance then calls the findEmployee(), getFirstName(), getLastName(), and getWeeklyPay() methods. Since $emp1 can be an instance of either of two different classes, these calls are only possible if the methods in both of these classes have been given the same names. An important feature of OOP allows us to require that **all** classes that extend another class **must** include certain methods with a pre-defined name and parameter list. We do this using the **abstract** keyword.

Abstract classes and methods

A review of the revised Employee class reveals that the class heading is preceded by the keyword **abstract**:

```
abstract class Employee
{
      // class variables and methods here
}
```

When a class is defined to be abstract, it is not possible for any application to create instances of the class. This is important: we want programmers to be able to create instances of SalariedEmployee and HourlyEmployee, which both inherit the Employee class, but we don't want programmers to be able to create instances of the Employee class because the Employee class does not include any wage or salary class variables or methods. The Employee class is only being used to provide shared class variables and methods to the classes that extend it, and so we define this class as abstract.

You will also notice that, although the Employee class includes the **headings** for methods named getWeeklyPay(), addEmployee(), and findEmployee(), each of these methods is also declared to be abstract and does not contain a body with the method's code; instead each method's heading simply ends with a semi-colon:

```
abstract public function getWeeklyPay();
abstract public function addEmployee();
abstract public function findEmployee($id);
```

You may wonder why these methods are included in the Employee class at all. After all they are not used in the Employee class; different versions of these methods are fully defined in the SalariedEmployee class and HourlyEmployee class. It is quite true that these three statements could be removed from the Employee class and the two child classes would function just as well.

However the inclusion of an **abstract method definition** in the Employee class ensures that all classes that extend this class **must** provide a method with the same name and parameter list as the abstract method defined in the parent class. This valuable OOP feature enforces a **design requirement** on any programmer who wants to extend the Employee class. The getWeeklyPay(), addEmployee(), and findEmployee() methods in any extending class can be coded in any manner that is appropriate for a particular type of employee, but no matter how they are coded, these three methods **must** be provided. This requirement ensures that application programmers who use instances of any class that extends Employee can do so knowing that these methods are available to them.

Any class that defines at least one abstract method **must** also be defined to be an abstract class. This makes sense when you consider that a class that contains at least one abstract method cannot be instantiated since the abstract method contains no code and cannot be used.

There is no restriction on the number of levels that classes can inherit from one another: classes that extend other classes and methods can also be extended. Similarly classes that extend abstract classes can also be abstract and can include, or pass on, abstract methods.

Method over-riding

Sometimes a class that extends (inherits) another class needs to replace a method from the parent class with a new method of the same name. This is done simply by coding the new method in the extending class. If the method has the same name and parameter list as a method in the parent class, it will **over-ride** the method of the parent class. Including a method in a child class that replaces an abstract method in the parent class is an example of method over-riding.

Polymorphism

Another important feature of OOP is **polymorphism**, which is beyond the scope of this book. In general usage, the term polymorphic means "many shaped"; in OOP, polymorphism allows applications to determine the specific type of object within an object hierarchy that it is to be processed at the time the application is running. For example an application might not "know" until it is actually executing whether to treat an employee as a salaried or hourly employee. This type of dynamic selection provides great flexibility for software designers, and once again reduces code duplication.

OOP and Databases

Our Employee class includes two methods that connect our object to an external data source, in this case a text file containing employee records. Of course in a real world application, employee data would almost certainly be stored in a database, not a text file; these methods connect to a text file only to allow flexibility for readers who wish to skip the database chapter. If you have studied Chapter 14, you will see that it would be a simple matter to modify this code to work with a database.

OOP and databases are very good neighbors: object classes are frequently developed to give application programmers a programming interface to databases. This is of such critical importance to modern application design that it's worth discussing the reasons here:

Ease of use: with an object interface between application and data source, the application developer does not need to design and write custom code to access the database directly, and does not need to know the syntax of the database language. Instead he or she simply instantiates an object that provides all the database operations he or she needs, and then uses the object's methods as needed. The object therefore provides an application programming interface (API) to the database; once developed, this interface can be used by any number of applications, which dramatically reduces the time for coding, testing and maintenance.

Protecting the data source: since application programmers have no direct access to the data source but only to the classes that provide a programming interface, they can only modify the data source in the ways permitted by the class methods. This protects the data source from accidental or intentional misuse, and from possible security breaches that may occur when connection information must be widely shared.

Separation of the user interface from the data source: the use of an object to interface between the application and its data sources enables a complete separation between these two components. This is extremely important. As a result of this separation, the data source can be changed in any way without requiring changes to the application: field names can be changed, tables can be redesigned and modified, the location and connection may change, the data source itself may transition from one database product to another. Since the application is not interacting directly with the data source, but instead calling pre-written methods in an object, the application code does not to be changed; only the code in the object's methods needs to be updated. The findEmployee() and addEmployee() methods in our Employee class provide a simple example of this. In their current form, these two methods connect to a text file containing a list of records comprised of five data values each, separated by commas. This file might need to be updated: additional data values might be added to each record, or the data values might be re-ordered, or the separators might be changed from a comma to a colon. Or the text file might be replaced by a MySQL or some other database, and then the database might itself be modified over time. Consider the nightmare if the code inside every application that worked with the employee data had to be changed every time the data source changed! Instead the applications use instances of an Employee class to call Employee methods that contain all the code to interact with the data source; when the data source changes, only the code in the Employee methods needs to be changed; the application code will continue to work in its original form.

OOP Development

After reading the database and OOP chapters, it is easy to see that there are quite different components of software design and development, and these are often quite independent of one another. Application (interface) programmers develop the “front end” interfaces between users and the data that they need to work with. Database programmers design, develop, protect, and maintain data sources that may serve any number of applications, and different types of user. Object programmers, design and develop objects that serve the needs and reduce the coding requirements of application developers, and often provide an interface between applications and their data sources. Object programmers think beyond the needs of specific applications and instead focus on encapsulating data within an object which provides methods to enable and control access to the data.

OOP Languages

Fully Object-Oriented languages, such as **Java**, are designed so that **all** application development is derived from objects. OOP languages provide large numbers of standard object classes that programmers can use to develop applications quickly and easily. For example, Java provides a rich set of standard object classes for developing graphical user interfaces and graphics quickly and easily. Java is an excellent language to develop a complete understanding of object oriented design and programming.

Summary

Object-Oriented Programming (OOP) is a very important approach to code modularity. The combination of a specific set of data with all of the functions needed to operate on this data constitutes an **object**, and an object is defined as a **class.** In OOP terminology, the variables that contain the object's data values are generally referred to as **fields** or **class variables** or **attributes**, while the object's functions are usually referred to as the class **methods**. The class variables and methods together constitute the **members** of the class. In object oriented design, programmers who wish to work with the data associated with an object are expected to use the methods that are provided for this purpose. This is usually enforced by making the class variables **private** (or **protected**) and the methods **public**. The methods therefore provide a **programming interface** to the data, and we say that the data is **encapsulated**.

Once a class has been developed, any number of **instances** of the class can be created for use by any application that has access to the class code. Each class contains a special **constructor** method that has the same name as the class and is executed only when an instance is created. The **new** operator is used to create an instance and call the constructor method.

The class variables and functions of one class can be **inherited** by another class, which then **extends** these with additional variables and functions for a more specific purpose. This constitutes an **object hierarchy**; the extended class is often referred to as the **parent** class, while an extending class is referred to as a **child** class. For example, an **Employee** class may consist of class variables and functions common to **all** employees, while a **SalariedEmployee** class might inherit these variables and methods and provide additional variables and methods that are appropriate only for **salaried** employees. Similarly, an **HourlyEmployee** class might **also** inherit the data and methods of the **Employee** class but would add data and methods that are appropriate only for **hourly** employees. Inheritance provides a powerful design structure for OOP and minimizes code duplication. For example by inheriting the data and methods of **Employee** class, the **SalariedEmployee** and **HourlyEmployee** classes do not need to duplicate the class variables and methods that are common to all employees; they share these while adding other variables and methods that are unique to themselves.

Classes that are defined to be **abstract** can be extended but cannot be instantiated. Methods that are defined to be abstract in one class **must** be included in any class that extends that class. The presence of an abstract method therefore creates a design requirement for programmers who are coding child classes. A class that contains at least one abstract method must also be defined to be abstract.

Another important feature of OOP is **polymorphism**, which allows applications to dynamically determine the specific type of object within an object hierarchy that it is to be processed at the time the application is running.

OOP allows different methods to share the same name if they have different parameter lists, as long as the compiler can determine which method is intended based on the arguments that are sent. This feature is termed **method overloading**. Constructor methods can be overloaded, allowing new instances of a class to be instantiated in different ways by calling different constructors. Method overloading is not available in PHP.

Sometimes an extending class needs to replace a method in the parent class with a different method. This is achieved by coding a new method with the same name and parameter list in the extending class. This is termed **method over-riding.**

Object Oriented Programming is strongly associated with access to databases by user applications. Objects are developed to provide an interface between applications and data sources. This interface: eliminates the need for application programmers to write code to directly query the data source; protects the data source from unintentional or intentional misuse; provides a complete separation between the user application (interface) and the data source, so that changes to the data source do not require changes to the application, but only to the object.

Java is a fully object oriented programming language.

Chapter 15 Review Questions

1. Which term refers to a structure that defines an object's attributes and methods?
   a. polymorphism
   b. class variables
   c. class
   d. instance
   e. hierarchy

2. What keyword is used to create new instances of a class?
   a. new
   b. instantiate
   c. $this
   d. public

e. class

3. What is the purpose of the constructor method?
   a. create a class variable
   b. create an instance of a class
   c. create a class method
   d. extend a class
   e. overload a class method

4. Consider the following line of code:

   $catWoman = new GameCharacter();

Which one of the following is NOT true?
   a. GameCharacter() is a constructor of the GameCharacter class
   b. $catWoman is a constructor of the GameCharacter class
   c. $catWoman contains an instance of the GameCharacter class
   d. GameCharacter is a class
   e. The new operator is used to create new instances of a class

5. Assume you are working with a class named Account. Which of the following statements correctly creates an instance of the Account class?
   a. $acct = Account;
   b. $acct = new Account;
   c. $acct = Account();
   d. $acct = new Account();
   e. $acct = $this->Account

6. Assume that an instance of the Account class has been assigned to a variable named $acct. Assume also that the Account class includes a method named getBalance(). Which of the following statements correctly uses $acct to obtain the account balance for that instance?

   a. $acct = Account($balance)
   b. $acct = $this->getBalance()
   c. $acct->getBalance()
   d. getBalance($acct)
   e. $acct.getBalance()

7. Assume that you are developing an Account class and must define a method named getBalance() that returns the value of a class variable named $balance. Which of the following statements should you use in the body of the getBalance() method?

   a. return $balance;
   b. return this->$balance;
   c. return $this->balance;
   d. return new Balance;

e. return getBalance();

8. What term is used to describe the use of multiple methods with the same name but different parameter lists?

a. Overriding
b. Overloading
c. Instantiation
d. Polymorphism
e. Extension

9. What keyword is used to indicate that a class inherits the class variables and methods of another class?

a. overrides
b. overloads
c. instantiates
d. extends
e. attributes

10. What is the purpose of the "private" modifier?

a. Prevents a class variable or method from being changed
b. Ensures that a class variable or method can only be accessed by methods that are part of the same class
c. Prevents a class from being extended
d. Ensures that new instances of a class cannot be created
e. Prevents a class from being included in another application

11. Which modifier allows a class variable to be accessed by methods in a class that extends the class that contains the variable?

a. public
b. private
c. protected

12. If a class named Queen extends a class named Chesspiece, which statement is true?

a. Queen is the parent class of Chesspiece
b. Chesspiece is the parent class of Queen

13. What does it mean if a class is defined to be abstract?

a. The class does not have a constructor method
b. The class can be extended but cannot be instantiated
c. The class can be instantiated but cannot be extended
d. The class is the child of another class
e. The class is the parent of another class

14. What does it mean if a method is defined to be abstract?

a. A method with the same name and parameters must be defined in any class that extends the class that contains the abstract method

b. The method must be defined as protected
c. The method must return a value
d. The method must modify a class variable
e. The method is a constructor method

15. Which of the following is true?
a. You must always include a customized constructor method in your class definition.
b. A default constructor method is automatically provided and cannot be replaced with a customized constructor.
c. A default constructor method is automatically provided but you can replace this with a customized constructor.

Chapter 15 Code Exercises

Your Chapter 15 code exercises can be found in your **chapter15** folder. This folder is included in your customized XAMPP installation at the following location:

xampp\htdocs\webtech\coursework\ chapter15

Type your name and the date in the **Author** and **Date** sections of each file as you work on each exercise.

Debugging Exercises

Your **chapter15** folder should contain a number of "fixit" files. Each of these files contains PHP code that has an error of some kind. The type of error is indicated in the comment section of each file. You will need to run each program in order to see the errors, and to debug and test the code to see if it works correctly. For example to run **fixit1.php**, first run the Web server, then use the URL:

http://localhost/webtech/coursework/chapter15/fixit1.php

Code Modification Exercises

Your **chapter15** folder contains a number of "modify" files. Each pair of files contains HTML and PHP code that needs to be modified to meet a requirement. The requirements are included in the comment section of each file. Modify the algorithms, being careful to make changes to the .html and .php files as directed. You will need to run each program in order to test your changes. For example to run **modify1.html**, first run the Web server, then use the URL:

http://localhost/webtech/coursework/chapter13/modify1.html

Code Completion Exercises

1. Read this exercise carefully and take your time to work out the logic. Your **chapter15** folder contains versions of **paint-estimate.html** and **paint-estimate.php** as well as a file named **inc-rectangle-object.php** which contains code that defines a class named Rectangle. Review the Rectangle class carefully: it contains two class variables, **$x** and **$y**, that store the height and width of a rectangle, and five methods, to get and set the height and width, and get the area of the rectangle.

The form provided in **paint-estimate.html** asks the user to submit the height, length, and width of a room. The **paint-estimate.php** program already includes the code to receive the form inputs from **paint-estimate.html**. Add code to; include the inc-rectangle-object.php file; create two Rectangles instances named $longWall and $shortWall; call the Rectangle class's setX() and setY() methods to assign values for each of the two instances (for example, if the height, width and length of the room are 90 and 180, and 120 then the x and y values of the $longWall rectangle will be 90 and 180, and the x and y values of the $shortWall rectangle will be 90 and 120); call the getArea() method of the Rectangle class for each of the two instances; calculate the total area for all four walls by adding the areas together (remember there are two long walls and two short walls); display the total area.

2. Read this exercise carefully and take your time to work out the logic. Your **chapter15** folder contains versions of **software-order.html** and **software-order.php** as well as a file named **inc-order-object.php**, which contains the code that defines an Order class. Review the Order class carefully: it contains two class variables, $itemCost and $numItems, that stores the cost of an item, and the number of items ordered, and eight methods, four of which are used to get and set the item cost and the number of items, the remaining methods work as follows:

The **getSubtotal()** method returns the item cost multiplies by the number of items.
The **getSalesTax()** method returns the sales tax.
The **getShippingHandling()** method returns the shipping and handling cost.
The **getTotal()** method returns the total cost.

Your software-order.html file already contains a form to request the item cost and number of items being ordered. Your software-order.php already contains the code to receive the form input. Add code to: include the **inc-order-object.php** file; create an Order instance named **$order**; call the Order class's **setItemCost()** and **setNumItems()** methods to assign the values received from the form to the class variables of the $order instance; call each of the four methods described above and use the values that are returned to display the sub-total, sales tax, shipping and handling, and total of the order.

3. Create a class of your own named **GameCharacter** that contains two class variables, $playerName and $score. Add get and set methods to work with each class variable. Save this in your Chapter15 folder in a file named **inc-game-character-object.php**. Now create a PHP application that includes the class, creates two instances (two different game characters), assigns names and scores to each instance, and displays these. Add code to the application (not the class) that compares the two scores and displays the winning character.

4. Your **chapter15** folder contains a file named **inc-rectangle-object.php** which contains code that defines a class named Rectangle. Review the Rectangle class carefully: it contains two class variables, **$x** and **$y**, that store the height and width of a rectangle, and five methods, to get and set the height and width, and get the area of the rectangle. Now create a class named **Cube** that **extends** the Rectangle class and save this in your Chapter15 folder in a file named **inc-cube-object.php**. Add a class variable named **$z** to the Cube class which will be used to store the height of a cube. Add get and set methods to work with the $z class variable. Note that the Cube class inherits the class variables and methods of the Rectangle class, so it will contain the attributes and methods to work with all three dimensions ($x, $y, and $z) of a cube. Now add a

method to the Cube class named getArea() that calculates and returns the total area of the six sides of the cube. The total area will be 2 multiplied by the value of $x multiplied by the value of $y plus 2 multiplied by the value of $x multipled by the value of $z plus 2 multiplied by the value of $y multipled by the value of $z. Add another method to the Cube class named getVolume() that calculates and returns the **volume** of the cube, which will be the value of $x multiplied by the value of $y multiplied by the valiue of $z.

Important: since the getArea() and getVolume() methods in the Cube class need to accces the $x and $y class variables in the Rectangle class, you will need to change the access modifiers that are used with these two variables in the Rectangle class.

An application named **test-cube.php** has been provided in your Chapter15 folder which tests your Cube class using three Cube instances. You don't have to change this application, just run it. If you have coded your Cube class correctly and modified the access modifiers in the Rectangle class, the three areas should be displayed as 104, 150, and 136, and the three volumes should be 60, 125, and 80.

5. Create a simple class of your own using what you have learned in this chapter, and develop two simple applications that use the class in different ways. Keep it simple, you don't have to extend the class; the main purpose here is to get some practice developing and using objects. If you're stuck for ideas, consider a simplified version of one of the examples in the introduction to this chapter.