TASK

Introduction to OOP - Classes

Visit our website

Introduction

WELCOME TO THE INTRODUCTION TO OOP II TASK!

Object-Oriented Programming (OOP) is a fundamental style of programming for developing larger pieces of software. Up until now, the programs you have written are simple enough to be run from just one file. In the real world of software development, multiple programmers work on large projects that may have hundreds of different files of code that implement the functionality of the project.

Your first step to building more complex programs is understanding OOP. This may be the first truly abstract concept you encounter in programming, but don't worry! Practice will show you that once you get past the terminology, OOP is very simple.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

WHY OOP?

Imagine we want to build a program for a university. This program has a database of students, their information, and their marks. We need to perform computations on this data, such as finding the average grade of a particular student. Here are some observations from the above problem:

- A university will have many students that have the same information stored in the database, for example, age, name, and gender. How can we represent this information in code?
- We need to write code to find the average of a student by simply summing their grades for different subjects, and dividing by the number of subjects taken. How can we only define this code once and reuse it for many students?

OOP is the solution to the above problems, and indeed many real-world implementations of the above systems will use OOP. In fact, up until this point, you have already been using classes and objects even though you may not have been aware of them.

THE COMPONENTS OF OOP

The Class

The concept of a class may be hard to get your head around at first. A class is a specific Javascript construct that can be thought of as a 'blueprint' for a specific data type.

We have discussed different data types in previous tasks, and you can think of a Class as defining your own special data types, with properties you determine.

A class stores properties along with associated functions called 'methods' which run programming logic to modify or return the class properties.

The String class, for example, has the property which is the value of the string, and then also methods in the class that can be used to operate on any string such as **charAt()**, **substr()**, **replace()**, etc.

In the example that we discussed earlier (building a program for a university around a database of students), we would create a class called Student to represent a student. We can create the properties of the Student class to match those stored in the database such as name, age, etc.

Defining a Class in Javascript

Let us assume that the database stores the age, name, and gender of each student. The code to create a blueprint for a Student class is as follows:

```
class Student{
	constructor(age, name, gender){
		this.age = age;
		this.name = name;
		this.gender = gender;
	}
}


philani = new Student(20, "Philani Sithole", "Male");
sarah = new Student(19, "Sarah Jones", "Female");
pieter = new Student(18, "Pieter Viljoen", "Non-binary");

console.log(philani);
console.log(sarah);
console.log(pieter);
```

This may look confusing, so let's break it down:

- **Line 1:**
  This is how you define a class. By convention, classes start with a capital letter to differentiate them from variable names which follow the **UpperCamelCase convention**.

- **Line 2:**
  This is called the constructor of the class. A constructor is a special type of function that basically answers the question "What data does this blueprint need to initialise the Student?". This is why it uses the term 'constructor' which refers to creating - constructing - an object. As you can see, age, name, and gender are passed into the function. The constructor function is called automatically when **instantiating** a new object of a class, and therefore the values of the properties can automatically be assigned for that particular instance of the object by using the constructor function.

- **Line 3-5:**
  We also passed in a parameter called **this**. **this** is a special variable that is hard to define. It is basically a pointer to this Student object that you are creating with your Student class/blueprint. By saying **this.age = age**, you're saying "I'll take the age passed into the constructor, and set the value of the age parameter of this Student object I am creating to have that value". The same logic applies to the name and gender variables.

The above piece of code is more powerful than you may think. All OOP programs you write will use this format to define a class (the blueprint). Once defined, a class gives us the ability

to create thousands of that class - in this case Student objects - which have predefined properties. Let's look at this in more detail.

Creating objects from a class

Now that we have a blueprint for a Student, we can use it to create many Student objects. Objects are basically initialised versions of your blueprint. They each have the properties you have defined in your constructor. Let's look at an example. Say we want to create objects from our object representing three students, **philani**, **sarah**, and **pieter**.

This is what it looks like in Javascript (with the addition of a grades property):

```
class Student{
	constructor(age, name, gender, grades){
		this.age = age;
		this.name = name;
		this.gender = gender;
		this.grades =grades;
	}
}


philani = new Student(20, "Philani Sithole", "Male",[64, 66]);
sarah = new Student(19, "Sarah Jones", "Female", [82, 58]);
pieter = new Student(18, "Pieter Viljoen", "Non-binary", [70, 54]);

console.log(philani);
console.log(sarah);
console.log(pieter);
```

We now have three objects of the class Student called philani, sarah, and pieter.

Pay careful attention to the syntax for creating a new object. As you can see, the age, name, gender, and grades are passed in when defining a new object of type Student.

These three objects are like complex variables. At the moment they can't do much because the class blueprint for Student just stores data, but we can change that - let's add some actions to the Student class with methods!

Creating methods for a class

Methods allow us to define functions that are shared by all objects of a class to carry out some core computations. Recall how we may want to compute the average mark of every student? The code below allows us to do exactly that:

```
class Student{
	constructor(age, name, gender, grades){
		this.age = age;
		this.name = name;
		this.gender = gender;
		this.grades = grades;
	}
	computeAverage(){
		let total = 0;
		for(let i = 0; i < this.grades.length; i++){
			total += this.grades[i];
		}
		let average = total / this.grades.length;
		console.log("The average for "+ this.name + " is "+ average);
	}
}

philani = new Student(20, "Philani Sithole", "Male",[64, 66]);
sarah = new Student(19, "Sarah Jones", "Female", [82, 58]);
pieter = new Student(18, "Pieter Viljoen", "Non-binary", [70, 54]);

philani.computeAverage();
sarah.computeAverage();
pieter.computeAverage();
```

Again, notice that we've added a new property for each student, namely grades, which is a list of ints representing a Student's marks on two subjects. In our example of university students, this can most certainly be retrieved from a database.

Secondly, notice a new **method** called computeAverage has been defined under the Student class/blueprint. This just means that "this method has access to the specific Student object properties which can be accessed through this". Notice this method uses **this.grades** and **this.name** to access the properties for a particular student average calculation.

The program outputs:

```
The average for Philani Sithole is 65
The average for Sarah Jones is 70
The average for Pieter Viljoen is 62
```

This is the output of the method calls on lines 22-24. Note the syntax, especially the **()** for calling this method from one of our objects. Only an object of type Student can call this method, as it is defined only for the Student class/blueprint.

You have now seen how we can call the methods of objects that allow us to carry out calculations. The code for this program is available in your folder in the **student.js** file. Every object we define using this blueprint will be able to run this predefined method,

effectively allowing us to define hundreds of Student objects and efficiently find their averages with only a few lines of code - all thanks to OOP!

Class variables vs instance variables

In the examples covered so far the variables that are used as properties are all examples of what we call instance variables. This means that the values are specific to a particular instance of the class (object). There is another type of property used in classes which is known as **class variables**. These variables have a value that is shared with every instance of that particular class. In order to adhere to DRY (don't repeat yourself) principles, when a specific property's value needs to be shared across all instances of that class, we can define that variable at the class level i.e. *not* in the constructor function.

Let's look at this concept in a bit more detail using some examples. Firstly, let's look at a class that only has class variables:

```
class Wolf{
	classification = "canine";

}

silver = new Wolf();
console.log(silver.classification); // the output will be "canine"
```

In the above example, we can see that we have a class named Wolf. A property concerning wolves that will not change is the fact that it is a "canine". Therefore, any instance of the wolf class will have the property of classification set to "canine" as you will see if you run this example on your computer. We create a wolf object and print out the classification property for that object (note the use of dot notation for accessing the property value **silver.classification**) which will output the value "canine".

Now let's look at how we can use class and instance variables combined so that objects can have shared properties as well as properties that pertain to the specific objects only.

```
class Wolf{
	//	Class variables
	classification = "canine";
	habitat = "forest";

	//	Constructor method with instance variables name and age
	constructor(name, age){
		this.name = name;
		this.age = age;
	}
}
```

```
//     First object, set up instance variables of constructor method
silverTooth = new Wolf("Silver tooth", 5);

//     console output instance variable name
console.log(silverTooth.name);
//     console output class variable habitat
console.log(silverTooth.habitat);

//     Second object
loneWolf = new Wolf("Lone wolf", 8);

//     console output instance variable name
console.log(loneWolf.name);
//     console output class variable classification
console.log(loneWolf.classification);
```

In the above example, both Wolf objects that were created have habitat and classification properties in common, but they each have their own name and age properties that are specific to them. In this way, when creating objects, we eliminate the need to also declare the values of the class variables, thereby eliminating repetition (DRY!).

Changing property values from inside the object

From within an object, it is possible to change the property values when a specific method has been called. The following example shows how we can do this:

```
class Wolf{
	// Class variables
	classification = "canine";
	habitat = "forest";

	// Constructor method with instance name and age variables
	constructor(name, age){
		this.name = name;
		this.age = age;
		this.isSleeping = true; //initialise sleep state
	}

	// method to wake the wolf
	wakeUp(){
		this.isSleeping = false;
	}

	// method to put the wolf to sleep
	sleep(){
		this.isSleeping = true;
```

```
        }

        // method that returns the sleep state of the wolf
        showSleepState(){
                if(this.isSleeping){
                        return this.name + " is sleeping.";
                }else{
                        return this.name + " is awake."
                }
        }
}


// initialising a wolf object and console log the initial sleep
// state which is awake
silverTooth = new Wolf("Silver tooth", 5);
console.log(silverTooth.showSleepState());

// changing sleep state to sleeping and then outputting that state
silverTooth.wakeUp();
console.log(silverTooth.showSleepState());
```

You can run the above code in your IDE and see what output is generated. You are encouraged to try to add your own properties and find creative ways to change the values of those properties.

Instructions

First, read **example.js**, and open it using Visual Studio Code (or another IDE of your choice).

Read through example.js carefully and then run the code and examine the output. When you feel confident that you understand the concept, you can move on to completing the compulsory task below.

Compulsory Task 1

In this task, we're going to be simulating an email message. Some of the logic has been filled out for you in the **email.js** file.

- Open the file called **email.js**.
- Create a class definition for an Email that has four attributes.:

hasBeenRead, ***emailContents***, ***isSpam***, *and* ***fromAddress***.

- The constructor should initialise the sender's email address and the email contents.
- The constructor should also initialise ***hasBeenRead*** and ***isSpam*** to false.
- Create a function in this class called ***markAsRead*** which should change ***hasBeenRead*** to true.
- Create a function in this class called ***markAsSpam*** which should change ***isSpam*** to true.
- Create an empty array called inbox to store all emails (note that you can have an array of objects).

Then create the following functions::

- ***addEmail*** - which takes in the contents and email address from the received email to make a new Email object and push this object into the inbox.
- ***getCount*** - returns the number of messages in the store in the inbox.
- ***getEmail*** - returns the contents of an email in the inbox. For this, allow the user to input an index i.e. *get_email(i)* returns the email stored at position i in the inbox. Once this has been done, *has_been_read* should now be true.
- ***getUnreadEmails*** - should return a list of all the emails that haven't been read.
- ***getSpamEmails*** - should return a list of all the emails that have been marked as spam.

- ***delete*** - deletes an email in the inbox. Allow the user to enter an index on the email that he wants to delete and remove that object from the inbox list.

Now that you have these set up, let's get everything working!

Fill in the rest of the logic for what should happen when the user inputs the number from the menu list on the template. Some of it has been done for you.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.