Hyperiondev

TASK

Object-Oriented Programming

Visit our website

Introduction

WELCOME TO THE OBJECT-ORIENTED PROGRAMMING TASK!

In this task, you will learn about the concept of object-oriented programming and how to create JavaScript objects. JavaScript uses objects extensively in order to store data in a more structured manner. You will see, by the end of this task, how you have actually been using JavaScript objects all along!

WHAT IS OBJECT-ORIENTED PROGRAMMING?

Object-oriented programming (OOP) became popular because it mirrors how people perceive the world—through objects. Early languages like C++ set the stage for OOP's rise, influencing modern languages like JavaScript and C#. Even languages like PHP and Python adapted to support OOP.

OOP organises code around real-world entities, simplifying complex problems and making systems intuitive. Concepts like "windows" or "folders" in software mirror their physical counterparts, reducing cognitive load for users and developers.

By promoting modularity and reusability, OOP allows scalable, maintainable systems. This marks a major shift from procedural programming's focus on linear instruction sequences.

PROCEDURAL PROGRAMMING VERSUS OBJECT-ORIENTED PROGRAMMING

In procedural programming, instructions are executed sequentially, one after another. Functions are independent but can access global variables. The data is stored separately from the code that processes it, meaning functions interact with data through parameters and return values rather than directly modifying data structures. The key features are sequential execution and the separation of data from the functions that operate on it.

By contrast, with OOP, a system is designed in terms of objects that communicate with each other to accomplish a given task. Instead of separating data and code that manipulates the data, these two are encapsulated into a single module. Data is passed from one module to the next using methods.

HOW DO WE DESIGN OBJECT-ORIENTED PROGRAMMING SYSTEMS?

In most OOP languages, an object is created using a **class**. A **class** is a blueprint from which objects are made, consisting of both data and the code that manipulates the data.

To illustrate this, let's consider an object you encounter every time you use software: a button. As shown in the images further below, although there are many different kinds of buttons you might interact with, all these objects have certain things in common:

1. They are described by certain **attributes**. For instance, every button has a *size*, a *background color*, and a *shape*. It could have a specific *image* on it or it could just contain *text*.

2. You expect all buttons to have **methods** that do something, for example, when you click on a button you expect something to happen – you may want a file to download or an app to launch. The code that tells your computer what to do when you click on the button is written in a **method**. A **method** is just a function that is related to an object.

Although every object is different, you can create a single class that describes all objects of that kind. The class defines what **attributes** and **methods** each object creates using what that class contains. Each object created from a particular class is called an **instance** of a class. For example, each of the buttons shown in the images above is an instance of the class “Button”.

JAVASCRIPT OBJECTS

JavaScript is an OOP language. If you understand objects, then you will understand JavaScript a lot more. Almost everything in JavaScript is an object!

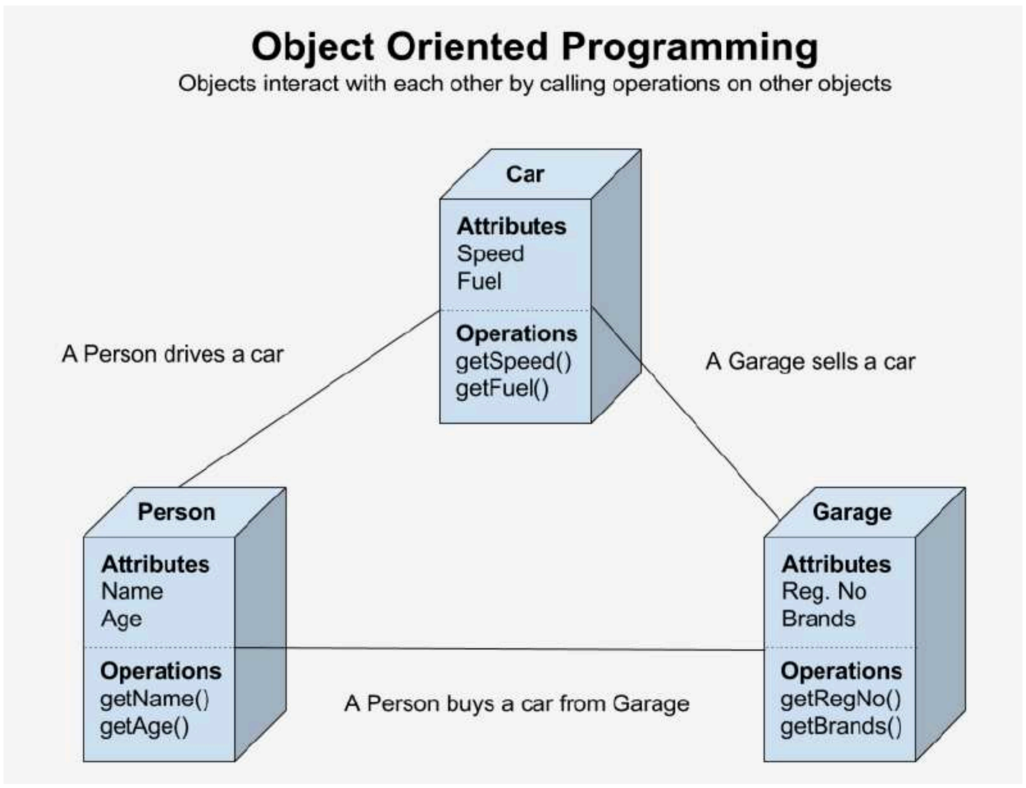
OOP allows the components of your code to be as modular as possible. The benefits of this approach are a shorter development time and easier debugging, because you're reusing program code that has already been proven to work. However, JavaScript is not strictly an OOP language: It is an object-based scripting language. JavaScript works with objects, but instead of using classes like in many object-oriented languages, it creates objects using prototypes. A prototype is a special type of function used to set up a blueprint for new objects, allowing them to inherit properties and methods from the prototype.

Take note

ES6 update: ES6, which stands for ECMAScript 6, allows us to create objects using keywords (like `class`, `super`, etc.) that are used in class-based languages. However, under the hood, ES6 still uses functions and prototypal inheritance to implement objects. To see an example of an ES6 class, look at this **webpage on classes**.

Understanding classes and inheritance is key in OOP for efficient, non-repetitive code. For now, we'll focus on understanding objects, their usage, and their importance.

The image below illustrates the relationships between different objects in OOP. It shows that a `Person` interacts with a `Car` by driving it, and a `Person` buys a `Car` from a `Garage`, which sells cars. Each of these entities – `Person`, `Car`, and `Garage` – has specific characteristics (like a person's name or a car's speed) and actions (like driving or selling). The image demonstrates how these objects are connected and how they work together by exchanging actions and information.



The relationships between different objects in OOP (Chigozie, 2021)

Now, let's get coding and learn to create JavaScript objects!

CREATING DEFINED OBJECTS

There is more than one way of creating objects. We will consider two key ways in this task.

Method 1: Using object literals

Let's consider the following object declaration:

```
let car = {
  // Object properties
  brand: "Tesla",
  model: "Model S",
  year: 2023,
  color: "Gray",

  // Object methods
  howOld() {
    return `The car was made in the year ${this.year}`;
  },
};
```

Let's break down the code above:

- The object we have created is a `car`. (It is best to use object literals if you are creating a single object.)

- Objects in JavaScript store a collection of key–value pairs called properties. The `car` object, therefore, has four different properties, and values are assigned for each property in the code snippet above as follows:

Property	Value
`Brand`	`Tesla`
`Model`	`Model S`
`Year`	`2023`
`Colour`	`Gray`

- When a function is declared within an object, it is known as a **method**. Methods are a series of actions that can be carried out on an object. An object can have properties that store different types of data, such as numbers, strings, functions, or even other objects. Therefore, an object method is simply a property that has a function as its value.

Notice that the `car` object contains a method called `howOld` that will return the year of the car passed as a string.

For the `howOld` method, the `function` keyword has been omitted. This is due to the **object method shorthand syntax**, which allows us to define methods within objects without explicitly using the `function` keyword.

- Within the method, you'll notice `this.year` was used instead of `car.year`. "`this`" is a keyword that you can use within a method to refer to the *current object*. "`this`" is used because it's a property within the same object, thus it simply looks for a "`year`" property within the object and uses that value.

You now understand how to use object literals to create a single object at a time, but what if you want to create several objects that all have the same properties and methods but different values?

Method 2: Using object constructors

The second method we will now consider provides an effective way of creating many objects using an object constructor. A *constructor* is a special type of function that is used to make or construct several different objects.

Consider the example below:

```
function carDescription(brand, model, year, color) {
  // Object properties
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.color = color;
}

// Creating three objects of the carDescription class
let car1 = new carDescription("Tesla", "Model S", 2023, "Gray");
let car2 = new carDescription("Audi", "e-tron", 2022, "Red");
let car3 = new carDescription("Porsche", "Taycan", 2021, "Blue");
```

Here are some important points to understand about the code above:

- The function, `carDescription`, is the constructor. It is used to create the three different car objects: `car1`, `car2`, and `car3`.

- The `this` keyword is used to refer to the properties of the object, which helps to eliminate confusion between parameters with the same name.
- The `new` keyword is used to create an instance of the `car` object using the `carDescription` constructor function.

ACCESSING OBJECT PROPERTIES

There are two ways to access the properties of an object to modify the values:

Dot notation

The first way is using dot notation. You may have used dot notation in previous tasks to apply functions like `.length()`.

Let us use this approach to access object properties in the example below. When using dot notation, we need to specify the name of the object followed by a dot and the key of the property you would like to access.

Example of dot notation:

```
function carDescription(brand, model, year, color) {
  // Object properties
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.color = color;
}

// Creating three objects of the carDescription class
let car1 = new carDescription("Tesla", "Model S", 2023, "Gray");
let car2 = new carDescription("Audi", "e-tron", 2022, "Red");
let car3 = new carDescription("Porsche", "Taycan", 2021, "Blue");

// Accessing the object properties using dot notation
console.log(car1.brand); // Output: Tesla
console.log(car2.model); // Output: e-tron
console.log(car3.color); // Output: Blue
```

Bracket notation

Now let's look at the second way to access the properties of an object using bracket notation `[]`. When using bracket notation, we have to make sure that the property name is passed in as a string within the brackets. This is only applicable to bracket notation.

Example of bracket notation:

```
function carDescription(brand, model, year, color) {
  // Object properties
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.color = color;
}

// Creating three objects of the carDescription class
let car1 = new carDescription("Tesla", "Model S", 2023, "Gray");
let car2 = new carDescription("Audi", "e-tron", 2022, "Red");
let car3 = new carDescription("Porsche", "Taycan", 2021, "Blue");

// Accessing the object properties using square brackets notation.
// The property name is passed as a string inside the square brackets.
console.log(car1["brand"]); // Output: Tesla
console.log(car2["model"]); // Output: e-tron
console.log(car3["year"]); // Output: 2021
```

Note: We must use bracket notation when accessing keys that have **numbers**, **spaces**, or **special characters** in them. If we try to access keys that fall under those rules without bracket notation, then our code will throw an error. This happens because JavaScript treats keys accessed with dot notation as valid JavaScript identifiers. If a key contains invalid characters for identifiers (like spaces or special characters), JavaScript will misinterpret the key as separate variables or expressions, causing a syntax error and potentially crashing the code.

ASSIGNING OBJECT PROPERTIES

Both dot and bracket notation allow us to edit or grab specific properties of an object. You may be thinking that once we've defined an object we're stuck with the properties we've defined; this is not true, because objects in JavaScript are **mutable**, meaning we can update or change properties after we've created them.

There are two main things to note when assigning properties:

1. If the property already exists within the object, then whatever value it holds will be updated (overwritten) by the new value we are passing in:

```
function carDescription(brand, model, year, color) {
  // Object properties
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.color = color;
}

// Create a car object with Gray as the color.
let car1 = new carDescription("Tesla", "Model S", 2023, "Gray");

// The property value for the color of car1 can be changed from "Gray" to
"Green"
car1.color = "Green";

console.log(car1.color); // Output: Green
```

2. If you have created key-value pairs with properties and values that do not exist, JavaScript will add these new properties to the object:

```
function carDescription(brand, model, year, color) {
  // Object properties
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.color = color;
}

// Create object instance
let car1 = new carDescription("Tesla", "Model S", 2023, "Gray");

// Create the new transmission and engine type properties which did not exist.
// This has been done using both bracket and dot notation.
car1["transmission"] = "Automatic";
car1.engineType = "Electric";

// Displaying the new properties and the object itself.
console.log(car1["transmission"]);
console.log(car1.engineType);
console.log(car1);

/* Output:
Automatic
Electric
carDescription {
  brand: 'Tesla',
  model: 'Model S',
  year: '2023',
  color: 'Gray',
  transmission: 'Automatic',
  engineType: 'Electric'
} */
```

WORKING WITH OBJECTS

Now that you have a good understanding of functions and objects, we can look at the declaration of an object with implementation through a function (after all, you do want the object to serve a purpose). Let's explore a few examples of how we can create functions to access and manipulate an array of `car` objects to perform different operations using some built-in JavaScript methods.

The example below illustrates how we can define a function and create the operations to return the newly manufactured car based on the `year` property. We can manipulate the array by rearranging the cars in descending order using the `sort()` method. Notice the dot notation used to access certain properties. We have also used the **ternary operator** to evaluate whether the condition is true or false. The ternary operator takes a condition followed by a question mark (`?`) and then what to do if the condition is true, a colon, and what to do if the condition is false. It looks like this:

```
firstCar.year > secondCar.year ? -1 : 1
```

You can read this as "Evaluate the expression to check whether the year of the first car is greater than the year of the second car. If it's true, return -1, and if false, return 1."

Now let's look at the full example:

```
// Define a constructor function to create car objects
function Car(brand, model, year, color) {
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.color = color;
}

// Create 3 car objects using the Car constructor
let car1 = new Car("Tesla", "Model S", 2023, "Gray");
let car2 = new Car("Audi", "e-tron", 2022, "Red");
let car3 = new Car("Porsche", "Taycan", 2021, "Blue");

// Create an array to store the car objects
let cars = [car1, car2, car3];

// Define a function to find the most recently manufactured car
function findNewestCar(array) {
  // Sort the array of cars based on the manufacturing year in descending order
  array.sort((firstCar, secondCar) =>
    firstCar.year > secondCar.year ? -1 : 1
  );
  // Retrieve the most recently manufactured car,
  // which is the first element in the sorted array.
  let newestCar = array[0];

  // Use template and string literals to display the details
  console.log(`
    The most recently manufactured car is the ${newestCar.brand}
${newestCar.model},
    which was made in ${newestCar.year}
  `);
}

// Call the findNewestCar function
findNewestCar(cars);
```

Now, what if we want to update one of the properties for all the cars? We can define a function that takes three parameters, each of which will act as placeholders to pass the values of the car in the array, as well as the current and new colour of the car.

We can use dot or bracket notation to access the colour properties of the `car` objects and compare the existing values in order to make the updates. It is important to also display the details of the results in an easy-to-read and presentable layout to positively influence user experience.

This can be achieved through the use of the **`console.table()`** method, template and string literals (as shown above), as well as newline characters:

```
// Define a constructor function to create car objects
function Car(brand, model, year, color) {
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.color = color;
}

// Define an edit function that takes 3 parameters
function editColour(cars, carColour, newColour) {
  // Loop through each car object in the array
  for (let i = 0; i < cars.length; i++) {
    // If current car color matches the value passed to the carColour
    if (cars[i].color === carColour) {
      // Update the color property to the value passed to the newColour
      cars[i]["color"] = newColour;
      // Return the updated car object
      return cars[i];
    }
  }
}

// Created 3 car objects using the Car constructor
let car1 = new Car("Tesla", "Model S", 2023, "Gray");
let car2 = new Car("Audi", "e-tron", 2022, "Red");
let car3 = new Car("Porsche", "Taycan", 2021, "Blue");

// Create an array to store the car objects
let cars = [car1, car2, car3];

// Call the function to change the color of a car from "Gray" to "White".
editColour(cars, "Gray", "White");

/*
Display the updated car details in a table.
This may be the first time you're seeing the .table built-in function in action!
*/
console.log("\nThe updated table:");
console.table(cars);

/*
The updated table:
(index)   brand     model     year    color
  0       Tesla     Model S   2023     White
  1       Audi      e-tron    2022    Red
  2       Porsche  Taycan     2021    Blue

*/
```

JAVASCRIPT GETTERS AND SETTERS

JavaScript getter and setter methods act as *interceptors* because they offer a way to intercept property access and assignment.

- **Getters:** These are methods that **get and return** the properties of an object. When calling getter methods, we generally do not need to pass parentheses as they are seen as accessing properties.

- **Setters:** These are methods that **change the values** of existing properties within an object. Setter methods do not need to be called with brackets as we are changing the value of a property.

Let's look at how this works in code:

```
let car = {
  brand: "Tesla",
  model: "Model S",
  year: 2022,
  color: "Gray",

  // Getter method to return color
  get getColour() {
    return this.color;
  },

  // Setter method to change the model
  set newModel(newModel) {
    this.model = newModel;
  },
};

// Displaying the model before using the setter method.
console.log(car.model);

// Calling the setter method
car.newModel = "Model Z"; // changes the model to Model Z

// Calling the getter method and displaying to console
console.log(car.getColour);

// The properties can still be accessed the simple way without a getter
console.log(car.model);
```

An aspect to note here is that we can still reassign the property of an object using dot notation. Each approach, whether using dot notation or getter and setter methods, has its advantages and disadvantages. It is always good to take a step back and analyse which method works best for the task you're working on.

GETTERS AND SETTERS IN OBJECT CONSTRUCTOR FUNCTIONS

In addition to using getters and setters with individual objects, you can also define them in object constructor functions. This approach is useful for creating multiple instances of objects with shared behaviour.

Here's an example using a constructor function:

```
function Car(brand, model, year, color) {
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.color = color;
}

// Getter method for color
Object.defineProperty(Car.prototype, 'getColor', {
  get: function() {
    return this.color;
  }
});

// Setter method for model
Object.defineProperty(Car.prototype, 'setModel', {
  set: function(newModel) {
    this.model = newModel;
  }
});

// Creating a new Car instance
let myCar = new Car("Tesla", "Model S", 2022, "Gray");

// Using the explicit getter and setter methods
console.log(myCar.getColor); // displays 'Gray'
myCar.setModel = "Model Z"; // changes the model to 'Model Z'

// Accessing properties directly
console.log(myCar.model); // displays 'Model Z'
```

`Object` is a built-in JavaScript object that provides methods for creating, modifying, and interacting with other objects. In this example, `Object.defineProperty()` is used to add a getter for `color` and a setter for `model` to the `Car.prototype`. This allows all instances of `Car` to inherit these methods and make use of the behaviour consistently across multiple objects. Even though getter and setter methods are defined using `Object.defineProperty()`, you can still directly access and modify properties using dot notation, such as `myCar.model`, without invoking the setter.

Practical task

Follow these steps:

- Create a new file named **inventory.js**.
- Within this file, create an object constructor function named `Shoes`.
    - Within this object, create the following properties:
        - Name
        - Product code
        - Quantity
        - Value per item
- Then, create five instances of the `Shoes` object and push all of them to an array.

- Create the following functions that can interact with the array:
    - A function to search for any shoe within the array.
    - A function to find the shoe with the lowest value per item.
    - A function to find the shoe with the highest value per item.
    - A function to edit all four properties for each of the five shoe instances.
    - A function to order all the objects in ascending order based on the “Value per item” property.
- Keep in mind that when running these functions, all of your outputs should display all the results in an easy-to-read manner in the console. Remember that you can use the `console.table()` method, newline characters, strings, and template literals to lay out your output in a well-presented manner.

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

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?

Share your thoughts anonymously using this **form**.

Reference list

Chigozie, O. (2021, December 29). *Using object-oriented programming to solve problems in JavaScript*. Hashnode. **https://hashnode.com/post/using-object-oriented-programming-to-solve-problems-in-javascript-ckxrh4rc300g20ns1enzj23fm**