TASK

Object-Oriented Programming

Visit our website

Copyright © 2021 HyperionDev. All rights reserved.

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) has become the dominant programming methodology for building new systems. C++, an object-oriented (OO) version of C, was the first OO language to be used widely in the programming community. This adoption inspired most contemporary programming languages including JavaScript, C#, and Objective-C to be designed as object-oriented/object-based languages. Even languages that were not primarily designed for object-oriented programming, like PHP and Python, have evolved to support fully-fledged object-oriented programming.

So, why is OOP so popular? Firstly, we as humans think in terms of objects, and designers of modern technologies have used this knowledge to their advantage. For instance, we refer to the window that appears when we power up our computer as a "desktop", and use it to keep shortcuts and files we use frequently. But we know this is no real desk! We refer to the folder that our files get transferred to when we delete them as a "recycle bin." But is it really a bin? We use these terms in a metaphorical sense because our minds are already familiar with these objects. Software designers leverage this familiarity to avoid unnecessary complexity by playing to the cognitive biases of our minds.

You know from your experience as a programmer thus far, and from the previous tasks in this course, that solving computation problems can be complex! It is not just the process of developing an algorithm to solve a challenging problem, but the fact that the problem domain is often loaded with delicate intricacies that require expert knowledge to understand. These two problems contribute to the overall complexity of designing and implementing software that solves business problems. OOP uses our mind's natural affinity to think in terms of objects by designing a solution in terms of objects.

This is a paradigm shift away from the procedural paradigm, that is described as OOP.

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

PROCEDURAL PROGRAMMING VS OBJECT-ORIENTED PROGRAMMING

Procedural code executes in a waterfall fashion. We proceed from one statement to another, in sequence. The data are separated from the code that uses them. Functions are defined as standalone modules but they have access to variables we've declared outside the functions. These two properties - sequential execution of code and the separation of data and the code that manipulates the data - are what distinguish procedural code.

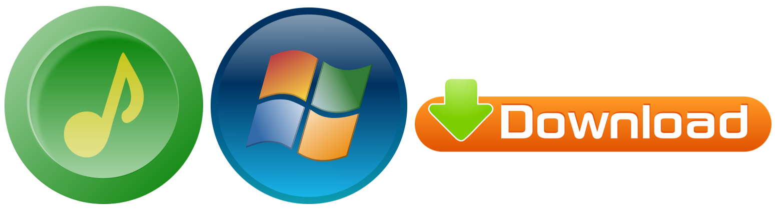
By contrast, with object-oriented programming, a system is designed in terms of objects which communicate with each other to accomplish a given task. Instead of separating data and code that manipulates the data, these two are encapsulated into a single module. Data is passed from one module to the next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

HOW DO WE DESIGN OOP SYSTEMS?

In most OOP languages, an object is created using a **class**. A class is a blueprint from which objects are made. It consists 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 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**. e.g. every button has a *size*, a *background colour*, and a *shape*. It could have a specific *image* on it or it could contain *text*.
2. You expect all buttons to have **methods** that do somethin, e.g., when you click on a button you want 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

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

created using 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 object-based programming language. If you understand objects, then you will understand JavaScript better. 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.

JavaScript is not strictly an object-oriented programming language. It is an object-based scripting language. Like pure object-oriented languages, JavaScript works with objects, but with JavaScript, objects are created using prototypes instead of classes. A prototype is a constructor function.

ES6[1] update: *ES6 allows us to create objects using keywords (like class, super, etc.) that are used in class-based languages, although, under the hood, ES6 still uses functions and prototypal inheritance to implement objects. To see an example of an ES6 class, look at* ***the MDN web docs on the topic of defining classes****.*

Creating classes and inheriting code from existing classes are also important concepts to read more about in OOP, especially when we want to create more efficient and less repetitive code. However, for the purpose of this task, we will focus on understanding what objects are, and how and why we use them.

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:

[1] ES6 stands for ECMAScript 6. ECMAScript was created to standardise JavaScript, and ES6 is the 6th version of ECMAScript.

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

```
let car = {
  // object properties
  brand: "Tesla",
  model: "Model S",
  year: 2023,
  colour: "Gray",
  // object method
  howOld: function () {
    return `The car was made in the year ${this.year}`;
  },
};
```

Let us discuss 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.

- *Object Properties/Attributes:* Objects in JavaScript store a collection of key-value pairs, called properties. The car object, therefore, has four different properties, assigning values for each property shown in the code snippet above as follows:

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

- *Object Methods:* When a function is declared within an object it is known as a **method**. Methods are a series of actions which can be carried out on an object. An object can have a primitive value, a function, or even other objects as its properties. Therefore, an object method is simply a property which 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.

    - You’ll notice “**`this.year`**” was used instead of the “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.

Copyright © 2021 HyperionDev. All rights reserved.

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, colour) {
  // object properties
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.colour = colour;
}
// create object instance
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 3 different car objects: `car1`, `car2` and `car3`.
- The `this` keyword is used to refer to the properties of the object that 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've used dot notation in previous tasks to apply functions like `.length()`.

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

Let us use this approach now 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*:

```
console.log(car1.brand); // Output: Tesla
console.log(car2.model); // Output: e-tron
console.log(car3.colour); // 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*:

```
// the property key is passed as a string enclosed within quotation marks
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.

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, colour) {
  // object properties
  this.brand = brand;
```

Copyright © 2021 HyperionDev. All rights reserved.

```
  this.model = model;
  this.year = year;
  this.colour = colour;
}
// create object instance
let car1 = new carDescription("Tesla", "Model S", 2023, "Gray");

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

console.log(car1.colour); // 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, colour) {
  // object properties
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.colour = colour;
}
// create object instance
let car1 = new carDescription("Tesla", "Model S", 2023, "Gray");

// created the new transmission and engine type properties which do not
exist
car1["transmission"] = "Automatic";
car1.engineType = "Electric";

// Note: in this case either bracket or dot notation can be used to add
these properties to the car object
console.log(car1["transmission"]);
console.log(car1.engineType);
console.log(car1);

/* Output:
carDescription {
  brand: 'Tesla',
```

HyperionDev
Copyright © 2021 HyperionDev. All rights reserved.

```
  model: 'Model S',
  year: 2023,
  colour: '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 the 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.

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

// Created 3 car objects using the Car constructor
let car1 = new Car("Tesla", "Model S", 2023, "Gray");
```

Copyright © 2021 HyperionDev. All rights reserved.

```
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
  );

// Use template and string literals to display the details
 console.log(`The most recently manufactured car is the ${array[0].brand}
 ${array[0].model}, which was made in ${array[0].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 3 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 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 colour matches the value passed to the carColour
    parameter
    if (cars[i].colour === carColour) {
      // Update the colour property to the value passed to the newColour
```

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

```
      parameter
      cars[i]["colour"] = newColour;
      // Return the updated car object
      return cars[i];
    }
  }
}

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

/* Display the updated car details in a table. This is 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 │ colour  │
├─────────┼───────────┼───────────┼──────┼─────────┼─────────┤
│    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.

HyperionDev Copyright © 2021 HyperionDev. All rights reserved.

Let's look at how this works in code.

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

  // getter method to return colour
  get getColour() {
    return this.colour;
  },
  // setter method to change the model
  set newModel(newModel) {
    this.model = newModel;
  },
};

// calling the setter method
car.newModel = 2023; // changes the model to 2023
// 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 the traditional method. The thing to remember is that each method 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.

SPOT CHECK

Let's see what you can remember from this section!

1. What is the difference between procedural programming and object-oriented programming?
2. What is an object?
3. What is the difference between an object literal and a constructor function?

(See if you were right by checking out the answers at the end of the task)

Copyright © 2021 HyperionDev. All rights reserved.

Compulsory 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 4 properties for each of the five shoe instances.
    - A function to order all of 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.

Copyright © 2021 HyperionDev. All rights reserved.

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.

SPOT CHECK ANSWERS

1. Procedural programming follows a waterfall approach where a line will not be executed until all the lines above it have been executed. Object-oriented programming, on the other hand, is divided into modules that execute through the use of methods.
2. An object is a data type containing attributes and methods. For example, you can have a *Button* object that is round and blue (attributes) that will make a song start playing (method).
3. You can create a single object using object literals by defining key-value pairs. When creating many objects, use an object constructor. Inside the constructor function, the `this` keyword is used to refer to the properties of the object. Outside of the constructor function, the `new` keyword is used to create instances of the object.

Copyright © 2021 HyperionDev. All rights reserved.