TASK

Introduction to OOP I - Objects and “this”

Visit our website

Introduction

WELCOME TO THE OOP IN JAVASCRIPT TASK!

In this Task, you will learn about the concept of **object-oriented programming** and how to create JavaScript **objects**. These are extremely important concepts! JavaScript uses objects extensively. You will see by the end of this task, how you have actually been using JavaScript objects all along!

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 career or education needs. Do not hesitate to ask a question or for additional support!

WHAT IS OBJECT-ORIENTED PROGRAMMING?

Object-oriented programming (OOP) has become the dominant programming methodology for building new systems. C++, an object-oriented version of C, was the first OO language to be used widely in the programming community. This adoption inspired most contemporary programming languages like JavaScript, C#, and Objective-C to be designed as object-oriented/object-based languages. Even languages which were not primarily designed for object-oriented programs, 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”. We keep shortcuts and files we use frequently on this “desktop.” 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, 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 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 under the next subheading.

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.

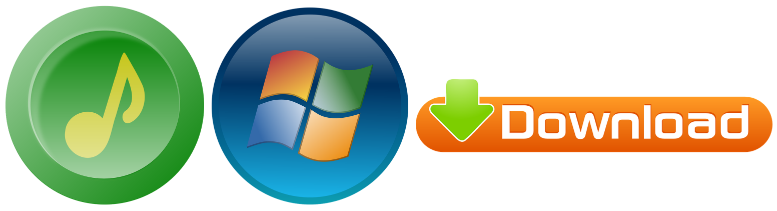
In contrast, with object-oriented programming, a system is designed in terms of objects which communicate with each other to accomplish a given task. Instead of separating data and code that manipulates the data, these two are encapsulated into a single module. Data is passed from one module to the next 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 something. 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 created using that class contains. Each object is called an *instance* of a class. 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 serves to allow 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. You will learn more about this distinction later.

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 **here**.*

We've covered the basic theory of what objects are and why we use them - now let's get coding and learn to create JavaScript objects!

Objects in JavaScript store key-value pairs. Each key-value pair is known as a property. When a function is declared within an object it is known as a method.

CREATING DEFINED OBJECTS

There is more than one way of creating objects. We will consider 2 key ways in this Task.

Method 1: Using Object Literals (easiest)

Let's consider the following object declaration:

```
let car = {
        brand: "Porsche",
        model: "GT3",
        year: 2004,
        colour: "White",
        howOld: function(){
            return `The car was made in the year ${this.year}`;
        }
};
```

Here an object is declared with four different properties, which are then stored in an object instance called *car*. This object also contains a method called "howOld" that will return the year age of the car passed as a string.

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

- *Object Properties/Attributes:* An object comprises a collection of named values, called properties. To demonstrate this concept, we'll consider the car object, assigning values for each property (as shown in the code snippet above) as follows:

Property	Value
Brand	Porsche
Model	GT3
Year	2004
Colour	White

- *Object Methods:* Methods (functions), 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 above contains a method called "howOld".
    - 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.

This first method for creating objects that we have considered is used to create a single object at a time. What if you want to create several objects that all have the same properties and methods but different values? The second method, which you will consider next, provides an effective way of creating many objects using a single *object constructor*.

Method 2: Using Object Constructors

The second way of creating an object is using an object constructor. A *constructor* is a special type of function that is used to make or construct several different objects. A constructor function in JavaScript is also known as a *factory function*. As implied by the term "factory", *factory functions* allow us to create multiple instances of a class or object quickly.

Consider the example below. The function, 'carDescription' is the constructor. It is used to create 3 different car objects: car, car2 and car3.

```
function carDescription(brand, model, year, colour) {
   this.brand = brand;
   this.model = model;
   this.year = year;
   this.colour = colour;
};

let car = new carDescription("Porsche", "GT3", 2012, "White");
let car2 = new carDescription("Ford", "Fiesta", 2015, "Red");
let car3 = new carDescription("Opel", "Corsa", 2014, "White");
```

The *this* keyword is used to refer to the properties of the object in this case. It is important to note that the *this* keyword is used to eliminate confusion between class attributes and parameters with the same name.

For more information on this please have a look at the MDN documentation **here**.

N.B. Remember to always reference your object properties using the *this* keyword

Accessing and Assigning Object Properties

There are two ways to access the properties of an object. The dot and bracket notations provide two ways to modify any value in an object.

The first way we'll look at is using the dot notation. You've used the dot notation in previous tasks to apply functions like *.length*.

Example using the **dot notation**:

```
let monster = {
    numberOfTeeth: 52,
    monsterColor: "yellow",
    monsterHome: "Jupiter"
};

let colour = monster.monsterColor;
let home = monster.monsterHome;

console.log(colour) // Return and output the monster's colour, yellow
console.log(home)   // Return and output the monster's home, Jupiter
```

Now let's look at the second way to access the properties of an object, using the bracket notation **[ ]**. When using the 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 the bracket notation.

Example of the **bracket notation**:

```
monster['numberOfTeeth']; // the key is passed a string enclosed within quotes
```

Another aspect to note is that 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.
The first way using bracket notation:

```
let monster = {
    numberOfTeeth: 52,
    colour: "yellow",
    monsterHome: "Jupiter",
    catchPhrase: "get a load of this guy!"
};

console.log(monster['numberOfTeeth']); // Return the number of teeth
console.log(monster['catchPhrase']);   // Returns the monsters' catchphrase
```

Both the dot notation and bracket 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:

```
// this will change the value of "yellow" to "blue" in the object
monster.color = "blue";
```

2) If there was no property with the specific name you have passed, Javascript will create a new property and pass it to the object with the new value.

```
// this will add the height property to the monster, as it doesn't exist yet
monster['height'] = '50m';     // using bracket notation
monster.legs = 45;    // using dot notation

/* in the output statements below either notation could be used - it doesn't
have to be bracket for height because it was set with bracket notation, and the
same for dot notation for legs */
console.log(monster[height]); // could also use dot notation
console.log(monster.legs);
```

We may also encounter scenarios where we want to remove a property from an object This is where the *delete* operator comes in. To delete properties of an object, you use the delete keyword as follows:

```
delete monster.color; //removes the color property from the monster
```

JAVASCRIPT GETTERS AND SETTERS

Javascript getter and setter methods act as interceptors, because they offer a way to intercept property access and assignment. We use the getter method to get information about our object, and we use the setter method to make changes to properties within our object.

- **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.

```
let monster = {
    numberOfTeeth: 52,
    color: "yellow",
    monsterHome: "Jupiter",
    catchPhrase: "get a load of this guy!",

    // getter method to return catchPhrase
    get getCatchPhrase() {
        return this.catchPhrase;
    }
};

// calling the getter method and display to console
console.log(monster.getCatchPhrase);
```

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

```
let monster = {
    numberOfTeeth: 52,
    colour: "yellow",
    monsterHome: "Jupiter",
    catchPhrase: "get a load of this guy!",

    // getter method to return catchPhrase
    get getCatchPhrase() {
        return this.catchPhrase;
    }

    // setter method to change the monsterHome
    set newMonsterHome(newHome){
        this.monsterHome = newHome;
    }
}

// calling the setter method
monster.newMonsterHome = "Mars";

console.log(monster.monsterHome);
```

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 both methods have their 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.

FUNCTION MEETS OBJECT

Now that you have a good understanding of functions and objects, let's consider the declaration of an object with implementation through a function (after all, you do want the object to serve a purpose):

```
let loaded = {}; // blank object without properties created
loaded.testing = function(signal) {
       alert("Hello World! " + signal);
       loaded.signal = signal;
}
loaded.testing("This page has loaded!");
```

Here a blank object is created. An object method is created to display a message to the user when the page has loaded. The function is called with a particular value which is then set as the value of a loaded.signal property; the property is created by this statement as the object was initially blank.

SPOT CHECK 1

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 a class?

Compulsory Task

Follow these steps:

- Create a new file named **inventory.js**. Within this file create a class named **Shoes**.
    - Within this class create the following attributes :
        - Name
        - Product Code
        - Quantity
        - Value per item
- Then, create 5 instances of the Shoes class and push all to an array.
- Your code should have the following functions that 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 any aspect of an instance of the shoes class.
    - A function to order all of the objects in ascending order.
- Keep in mind that all of your outputs when running these functions should be displayed in an easy to read, presentable manner.

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

Things to look out for

1. Make sure that you have installed and set up all programs correctly. You have set up **Dropbox** correctly if you are reading this, but **Visual Studio Code** may not be installed correctly.

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 1 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. A class is a blueprint of an object to show what an object needs in order to be created. For example, the *Button* class might specify that a button needs colour and a shape (e.g. blue and round) and must be able to make a song start, stop, or skip.