**Objects in Depth**

**Intro to JavaScript**

Up until this point, we’ve been writing linear codes. From top to bottom.

Using object-oriented programming we will be able to write classes capable of creating countless instances of similarly functioning objects.

We will also learn how delegation and inheritance can be used to build well-architected in performance applications.

**Recap**

* **Array**: an ordered collection of elements, enclosed by square brackets []. Each element of an array is referenced by a numeric key called index (starting from 0). Arrays can store any type of data, including other arrays.
* **Object**: a collection of associated key-value pairs. Objects are created by using curly brackets {}. Keys in an object must be named explicitly. Each distinct key-pair is known as the property of an object.

**Creating Objects**

We can either use literal notation or constructor function to create an object:

// Using literal notation:

const myObject = {};

// Using the Object() constructor function:

const myObject = new Object();

Both methods work, however, it is recommended to use the literal notation, as the constructor function is slower and more verbose.

**Modifying Properties**

Data within objects are mutable:

const cat = {

age: 2,

name: 'Bailey',

meow: function () {

console.log('Meow!');

},

greet: function (name) {

console.log(`Hello ${name}`);

}

};

cat.age ++;

console.log(cat); //3

**Adding Properties**

Properties can be added to objects simply by specifying the property name, then giving it a value.

const printer = {};

printer.on = true;

printer.mode = 'black and white';

printer['remainingSheets'] = 168;

printer.print = function () {

console.log('The printer is printing!');

};

console.log(printer)

We can use either dot notation or square bracket notation to add new properties to the object.

**Removing Properties**

We can also delete properties from objects.

//delete property from object

delete printer.mode;

console.log(printer);

**Passing Arguments**

**Passing a Primitive**

In JavaScript, a primitive data type - string, number, boolean - is immutable. In other words, any changes made to an argument inside a function effectively creates a copy local to that function, and does not affect the primitive outside of that function.

function changeToEight(n) {

n = 8; // whatever n was, it is now 8... but only in this function!

console.log(n);

//8

}

let n = 7;

changeToEight(n);

console.log(n);

// 7

**Passing an Object**

On the other hand, objects in JavaScript are mutable. If we pass an object into a function, JavaScript passes a reference to that object.

let originalObject = {

favoriteColor: 'red'

};

function setToBlue(object) {

object.favoriteColor = 'blue';

}

setToBlue(originalObject);

console.log(originalObject.favoriteColor);

// 'blue'

Since objects in JavaScript are passed by reference, if we make changes to that reference, we are actually directly modifying the original object itself.

The same rule applies when re-assigning an object to a new variable, and then changing that copy. Since objects are passed by reference, the original object is changed as well:

const iceCreamOriginal = {

Andrew: 3,

Richard: 15

};

console.log(iceCreamOriginal)

//{ Andrew: 3, Richard: 15 }

const iceCreamCopy = iceCreamOriginal;

console.log(iceCreamCopy.Richard);

// 15

iceCreamCopy.Richard = 99;

console.log(iceCreamCopy.Richard);

// 99

console.log(iceCreamOriginal.Richard);

// 99

**Comparing an Object with Another Object**

The following objects, parrot and pigeon, have the same methods and properties:

const parrot = {

group: 'bird',

feathers: true,

chirp: function () {

console.log('Chirp chirp!');

}

};

const pigeon = {

group: 'bird',

feathers: true,

chirp: function () {

console.log('Chirp chirp!');

}

};

console.log(parrot === pigeon);

// false

Although both objects look exactly the same, the expression will only return true when comparing two references to exactly the same object:

const myBird = parrot;

console.log(parrot === myBird);

// true

**Function vs Methods**

We can extend functionality to objects by adding methods to them.

We add methods the same way as we add properties: by providing a property name, then giving it a value. This time, the value of the property is a function:

const developer = {

name: 'Andrew'

};

developer.sayHello = function () {

console.log('Hi there!');

};

developer.sayHello();

**Calling Methods**

We can access a method (function) in an object using the property name, plus parenthesis. We can use dot or bracket notation.

developer.sayHello();

// 'Hi there!'

developer['sayHello']();

// 'Hi there!'

**Passing Arguments Into Methods**

Methods may take arguments:

developer.favoriteLanguage = function (language) {

console.log(`My favorite programming language is ${language}`);

};

developer.favoriteLanguage("JavaScript")

**A Method Can Access the Object it was Called On**

Using the ‘***this’*** keyword, a method can access and manipulate a property from the same object. When we say ***this***, what we are really saying is “this object at hand”.

***this*** is a reserved word in JavaScript, and cannot be used as an identifier (variable names, function names, etc).

var chameleon = {

color: "pink",

changeColor: function () {

if (this.color === "green") {

this.color = "pink";

} else if (this.color === "pink") {

this.color = "green";

}

}

};

chameleon.changeColor()

console.log(chameleon.color);

**Things that Belong to Objects**

In the following method we are using the keyword *this* to retrieve a property, while in the function *this* is used to set a property.

function whoThis () {

this.trickyish = true

}

whoThis();

console.log(trickyish);

//true

const chameleon = {

eyes: 2,

lookAround: function () {

console.log(`I see you with my ${this.eyes} eyes!`);

}

};

chameleon.lookAround();

// 'I see you with my 2 eyes!'

When a regular function is invoked, the value of ***this***is the global ***window*** object.

**The *window* Object**

The *window* object is provided by the browser environment and is globally accessible to JavaScript code using the identifier window.

This *window* object has a ton of information about the page itself, for example:

* The page’s URL: window.location

**Global Variables are Properties on *window***

Every variable declaration (using var) that is made on a global level (outside a function) automatically becomes a property on the ***window*** object.

After declaring a new variable, we can see that this variable is a property on the window object.

var currentlyEating = 'ice cream';

window.currentlyEating === currentlyEating

// true

Only variables declared with the var keyword will be added to the window object. Variables declared with let or const keywords won’t be added to window object.

**Global Functions are Methods on *window***

Any global function declaration

**Object.keys() and Object.values()**

At its core, an object is just a collection of key/value pairs.

In case we want to access only the keys from an object, we can make use of Object.keys(), which returns an array with a list of keys (property names):

const dictionary = {

car: 'automobile',

apple: 'healthy snack',

cat: 'cute furry animal',

dog: 'best friend'

};

console.log(Object.keys(dictionary));

//[ 'car', 'apple', 'cat', 'dog' ]

Object.keys() will return an array of strings and will return them in the same order as they would be when using a for loop.

Likewise, if we want to access only the values, we make use of Object.values():

console.log(Object.values(dictionary));

//[ 'automobile', 'healthy snack', 'cute furry animal', 'best friend' ]

Object.values() will return an array of different data types depending on the values. The order of the array returned is the same as using a for loop.