**Object Oriented Programming**

When you use ES6 classes these are basically it's called syntactic sugar or a convenient syntax and the reason for that is because you're writing it in a different way using classes which relates more to other languages. in JavaScript it's a weird because almost everything is an object you have a couple data types like strings numbers boolean's these are actually primitives but you can use methods on a string. JavaScript behind the scenes adds a wrapper to it. We also have like Dom objects such as the window so if I say console dot log window that's actually the the parent object of everything.

**Object Literal**

const book1 = {

name: “Book one”,

author: “John Doe”,

year: “2013”,

getSummary: function() {

return `${this.name} was written by ${this.author} in the year ${this.year}`

}

}

console.log(book1.getSummary())

console.log(Object.values(book1)) if we want to see the values

Syntax: Object.values(obj) returns an array whose elements are the enumerable property values found on the object.

console.log(Object.keys(book1)) if we want to see the keys

Syntax: Object.keys(obj) The **Object.keys()** method returns an array of a given object's own enumerable property **names**, in the same order as we get with a normal loop.

console.log(Object.entries(book1))

syntax: Object.entries(obj) returns [key, value] pairs. The **Object.entries()** method returns an array of a given object's own enumerable string-keyed property [key, value] pairs, in the same order as that provided by a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop.

**Object.create()**: The **Object.create()** method creates a new object, using an existing object as the prototype of the newly created object.

const person = {

isHuman: false,

printIntroduction: function () {

console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);

}

};

const me = Object.create(person);

me.name = "Matthew"; // "name" is a property set on "me", but not on "person"

me.isHuman = true; // inherited properties can be overwritten

me.printIntroduction();

**Object.assign():**

Syntax: Object.assign(target, ...sources) = target is the target object, sources is the source object.

The **Object.assign()** method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object. Properties in the target object will be overwritten by properties in the sources if they have the same key. Later sources' properties will similarly overwrite earlier ones.

const target = { a: 1, b: 2 };

const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);

// expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget);

// expected output: Object { a: 1, b: 4, c: 5 }

**Methods:**

**Function.prototype.apply()**

The **apply**() method calls a function with a given this value, and arguments provided as an array (or an [array-like object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Working_with_array-like_objects)).

Syntax: function.apply(thisArg, [argsArray])

Using apply to append an array to another

With apply, you can also use an array literal, for example, func.apply(this, ['eat', 'bananas']), or an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) object, for example, func.apply(this, new Array('eat', 'bananas'))

Since ECMAScript 5th Edition you can also use any kind of object which is array-like

Usages:

1. Using apply to append an array to another: we pass an array to push, it will actually add that array as a single element, instead of adding the elements individually, so we end up with an array inside an array. What if that is not what we want? concat does have the behaviour we want in this case, but it does not actually append to the existing array but creates and returns a new array. But we wanted to append to our existing array... So what now? Write a loop? Surely not? var array = ['a', 'b'];

var elements = [0, 1, 2];

array.push.apply(array, elements);

console.info(array); // ["a", "b", 0, 1, 2]

1. Using apply and built-in functions
2. Using apply to chain constructors: You can use apply to chain [constructors](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) for an object, similar to Java. In the following example we will create a global [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function) method called construct, which will enable you to use an array-like object with a constructor instead of an arguments list.

Function.prototype.construct = function(aArgs) {

var oNew = Object.create(this.prototype);

this.apply(oNew, aArgs);

return oNew;

};

**Function.prototype.bind()**

The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

Syntax: function.bind(thisArg[, arg1[, arg2[, ...]]])

The simplest use of bind() is to make a function that, no matter how it is called, is called with a particular this value. A common mistake for new JavaScript programmers is to extract a method from an object, then to later call that function and expect it to use the original object as its this (e.g. by using that method in callback-based code). Without special care, however, the original object is usually lost. Creating a bound function from the function, using the original object, neatly solves this problem:

this.x = 9; // this refers to global "window" object here in the browser

var module = {

x: 81,

getX: function() { return this.x; }

};

module.getX(); // 81

var retrieveX = module.getX;

retrieveX();

// returns 9 - The function gets invoked at the global scope

// Create a new function with 'this' bound to module

// New programmers might confuse the

// global var x with module's property x

var boundGetX = retrieveX.bind(module);

boundGetX(); // 81

Usages:

**Partially applied functions**

With setTimeout: When working with class methods that require this to refer to class instances, you may explicitly bind this to the callback function, in order to maintain the instance.

**Bound functions used as constructors**

Function.prototype.call()

The **call()** method calls a function with a given this value and arguments provided individually.

Syntax: function.call(thisArg, arg1, arg2, ...)

The call() allows for a function/method belonging to one object to be assigned and called for a different object. call() provides a new value of **this** to the function/method. With call, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

Usages:

Using call to chain constructors for an object

Using call to invoke an anonymous function

Using call to invoke a function and specifying the context for 'this'

Using call to invoke a function and without specifying the first argument

**Object.prototype.toString()**

What is a primitive in Javascript?

In **JavaScript**, a **primitive** (**primitive** value, **primitive** data type) is data that is not an object and has no methods. There are 7 **primitive** data types: string, number, bigint, boolean, null, undefined, and symbol. ... All **primitives** are immutable, i.e., they cannot be altered.

Classes was introduced in es6.

Instantiate: To **instantiate** is to create an instance of an **object** in an **object**-oriented programming

**Prototype Method**: Book.prototype.getSummary()

**Dates:**

const years = new Date().getFullYear()

The four basic concepts of OOP are Inheritance, Abstraction, Polymorphism, and Encapsulation.

1. Polymorphism is the ability of an object to take on many forms.