#_Object-Oriented Programming (OOP) in JavaScript

Section 1: Introduction

1.1 What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects". Objects are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated.

The OOP paradigm provides several benefits such as:

- Code reusability **through** inheritance.
- Simplification **of** complex problems **through** abstraction.
- Organized code **through** encapsulation.
- The ability **to** change behavior **at** runtime **through** polymorphism.

1.2 Why OOP in JavaScript?

JavaScript is a multi-paradigm language, meaning that it supports programming in many different styles, including procedural, object-oriented, and functional programming. OOP in JavaScript can be a bit different from other languages because JavaScript is a prototype-based language, not a class-based language.

However, as of ES6 (ECMAScript 2015), JavaScript introduced a `class` syntax that allows developers to write classes and use OOP concepts in a way that's closer to other OOP languages such as Java or C++.

OOP in JavaScript is useful because it:

- Organizes your **code**, which is especially helpful **in** large applications.
- Makes your **code** more reusable and easier to maintain.

By: Waleed Mousa

- Encourages the DRY principle, which stands **for** "Don't Repeat Yourself."

Despite the `class` syntax, JavaScript still uses prototypical inheritance under the hood, which we'll explore in later sections.

Section 2: JavaScript Basics

Before we get into OOP with JavaScript, let's review some JavaScript fundamentals. If you're already comfortable with these concepts, feel free to move to the next section.

2.1 Syntax

In JavaScript, statements are terminated by semicolons, and blocks are denoted by curly braces `{}`. Variables can be declared using `var`, `let`, or `const`, each with different scoping rules.

```
let x = 10;
const y = 20;
```

2.2 Data Types

JavaScript has six primitive data types:

- `String`: represents a sequence of characters.
- `Number`: represents numeric values.
- `Boolean`: represents either `true` or `false`.
- `undefined`: represents an uninitialized variable.
- `null`: represents the intentional absence of value.
- `Symbol`: a unique and immutable primitive value.

In addition to these, JavaScript has the `Object` data type which can store collections of data.

2.3 Functions

Functions in JavaScript are blocks of code designed to perform a particular task, defined using the `function` keyword:

```
function sayHello() {
    console.log("Hello, world!");
}
```

2.4 Control Structures

JavaScript includes typical control structures including `if-else` statements, `switch` statements, `for` loops, `while` loops, and `do-while` loops.

Here's an example of a simple `if-else` statement:

```
if (x > y) {
    console.log("x is greater than y");
} else {
    console.log("x is not greater than y");
}
```

Now that we have some basic JavaScript understanding, let's start exploring objects in JavaScript.

Section 3: Objects in JavaScript

In JavaScript, objects are key-value pairs, also known as properties. A property's value can be a function, in which case the property is known as a method.

3.1 Object Creation

There are several ways to create objects in JavaScript.

Using object literal syntax:

```
let dog = {
    name: "Spot",
    breed: "Dalmatian",
    age: 3,
    bark: function() {
        console.log("Woof!");
    }
};
```

Using the new Object() syntax:

```
let dog = new Object();
dog.name = "Spot";
dog.breed = "Dalmatian";
dog.age = 3;
dog.bark = function() {
    console.log("Woof!");
};
```

Using Object.create():

```
let dog = Object.create(Object.prototype, {
    name: {
        value: "Spot",
        enumerable: true,
        writable: true,
        configurable: true
    },
    breed: {
        value: "Dalmatian",
```

```
        enumerable: true,
        writable: true,
        configurable: true
    },
    age: {
        value: 3,
        enumerable: true,
        writable: true,
        configurable: true
    }
});

dog.bark = function() {
    console.log("Woof!");
};
```

`Object.create()` is a method in JavaScript that is used to create a new object with the specified prototype object and properties.

Here's a breakdown of the `Object.create()` syntax:

```
Object.create(proto, [propertiesObject])
```

proto : This is a required argument, and it should be an object or `null`. This will be set as the [[Prototype]] of the newly created object.

propertiesObject (Optional) : This is an object where the keys represent the names of the properties to be added to the new object, and the values associated with those keys are property descriptors for those properties.

Each property descriptor is an object with the following possible keys (all optional):

- `value`: The value associated with the property (data descriptors only). Defaults to `undefined`.
- `writable`: `true` if and only if the value associated with the property may be changed (data descriptors only). Defaults to `false`.

- `get`: A function which serves as a getter for the property, or `undefined` if there is no getter (accessor descriptors only).
- `set`: A function which serves as a setter for the property, or `undefined` if there is no setter (accessor descriptors only).
- `configurable`: `true` if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object. Defaults to `false`.
- `enumerable`: `true` if and only if this property shows up during enumeration of the properties on the corresponding object. Defaults to `false`.

3.2 Accessing Properties and Methods

You can access properties and methods on an object using dot notation or bracket notation:

```
console.log(dog.name);  // outputs: "Spot"
dog.bark();  // outputs: "Woof!"

// Or using bracket notation
console.log(dog["name"]);  // outputs: "Spot"
dog["bark"]();  // outputs: "Woof!"
```

3.3 Adding and Deleting Properties

You can add new properties to an object simply by assigning a value to them:

```
dog.color = "white with black spots";
console.log(dog.color);  // outputs: "white with black spots"
```

And you can delete properties from an object using the `delete` keyword:

```
delete dog.age;
console.log(dog.age);  // outputs: undefined
```

That's the basic idea of objects in JavaScript! In the next section, we'll explore more complex topics, like constructor functions and prototypes.

By: Waleed Mousa

Section 4: JavaScript Constructor Functions

In JavaScript, a constructor function is a special kind of function that is used to create objects of a particular type. The name of a constructor function usually starts with a capital letter to distinguish it from regular functions.

4.1 What is a constructor?

A constructor is a function that initializes an object. In JavaScript, the constructor method is called when an object is created.

4.2 Creating objects with constructors

Let's create a constructor function for a `Dog` object:

```
function Dog(name, breed, age) {
    this.name = name;
    this.breed = breed;
    this.age = age;
    this.bark = function() {
        console.log(this.name + " says woof!");
    };
}
```

You can then create a new `Dog` object using the `new` keyword:

```
let myDog = new Dog("Spot", "Dalmatian", 3);
myDog.bark();  // Outputs: "Spot says woof!"
```

4.3 Adding properties and methods to objects

You can add a property or method to an object after it's been constructed. Here's how you could add a `color` property and a `birthday` method to the `myDog` object:

```
myDog.color = "Black and White";
myDog.getBirthday = function() {
    return new Date().getFullYear() - this.age;
};
console.log(myDog.color);  // Outputs: "Black and White"
console.log(myDog.getBirthday());  // Outputs the birth year of the dog
```

Section 5: Prototypal Inheritance in JavaScript

Inheritance is a core concept in object-oriented programming. It allows classes or objects to inherit properties and methods from another, promoting code reuse and modularity.

In JavaScript, inheritance is prototype-based. When an object is created, it inherits the properties and methods from its prototype. Here's a quick example:

```
let animal = {
    species: "animal",
    describe: function() {
        return `This is a ${this.species}.`;
    }
};

let dog = Object.create(animal);
dog.species = "dog";

console.log(dog.describe());  // Outputs: "This is a dog."
```

In this example, `dog` is created with `animal` as its prototype, so it inherits the `describe` method from `animal`.

5.1 Inheritance with Constructor Functions

We can also use constructor functions to create a form of inheritance. To do this, we'll use the `call()` function, which allows us to call a function with a given `this` value and arguments provided individually.

```
function Animal(species) {
    this.species = species;
}

Animal.prototype.describe = function() {
    return `This is a ${this.species}.`;
};
```

```
function Dog(name) {
    Animal.call(this, "dog");
    this.name = name;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {
    return `${this.name} says woof!`;
};

let myDog = new Dog("Spot");

console.log(myDog.describe());  // Outputs: "This is a dog."
console.log(myDog.bark());  // Outputs: "Spot says woof!"
```

In this example, `Dog` is a "subclass" of `Animal` and inherits its `describe` method. The `Dog` constructor calls the `Animal` constructor to initialize the `species` property, and the `Dog` prototype is set to a new object created from the `Animal` prototype. This allows `Dog` instances to inherit `Animal` methods.

Then we add a `bark` method to the `Dog` prototype. This method is specific to `Dog` instances and is not shared by `Animal` instances.

The line `Dog.prototype.constructor = Dog;` is needed because setting `Dog.prototype` to `Object.create(Animal.prototype)` makes `Dog.prototype.constructor` point to `Animal`. We want `Dog.prototype.constructor` to correctly point to `Dog`, so we set it explicitly.

Section 6: ES6 Classes

While JavaScript is prototype-based, ECMAScript 6 (ES6) introduced a `class` syntax that allows you to write code that looks more like traditional class-based languages. Under the hood, this syntax is just syntactic sugar over JavaScript's existing prototype-based inheritance. It does not change the core of how object-oriented programming works in JavaScript. It just provides a cleaner, more elegant syntax for creating objects and dealing with inheritance.

6.1 Defining classes

Here is a basic example of how to define a class in JavaScript:

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    area() {
        return this.height * this.width;
    }
}

const myRectangle = new Rectangle(5, 10);
console.log(myRectangle.area());  // Outputs: 50
```

In this example, `Rectangle` is defined as a class with a constructor and a method named `area`. The constructor is a special method that is automatically called when a new instance of the class is created.

6.2 Inheritance with classes

Inheritance can be implemented in ES6 classes using the `extends` keyword. Here's an example:

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(this.name + ' makes a noise.');
    }
}

class Dog extends Animal {
    speak() {
        console.log(this.name + ' barks.');
    }
}

let dog = new Dog('Rover');
dog.speak();  // Outputs: "Rover barks."
```

In this example, `Dog` is a subclass of `Animal` and inherits its constructor. The `speak` method is overridden in the `Dog` class, so `Dog` instances will use this version of the method.

6.3 Using super

The `super` keyword can be used in the constructor of a subclass to call the constructor of the superclass. Here's an example:

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(this.name + ' makes a noise.');
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name);
        this.breed = breed;
    }
```

```
    speak() {
        super.speak();
        console.log(this.name + ' barks.');
    }
}

let dog = new Dog('Rover', 'Retriever');
dog.speak();  // Outputs: "Rover makes a noise." and then "Rover barks."
```

In this example, the `Dog` constructor calls `super(name)` to run the `Animal` constructor, then adds a `breed` property. The `speak` method also calls `super.speak()` to run the `Animal` version of the method before adding its own additional behavior.

Section 7: Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming. It refers to the bundling of data, and the methods that act on that data, into a single unit, and restricting direct access to it. In JavaScript, encapsulation can be achieved using closures, ES6 classes with constructor function and getter/setter methods.

7.1 Encapsulation using Closures

Here is an example:

```
function createCar(make, model) {
    let odometer = 0;
    return {
        make,
        model,
        drive(distance) {
            odometer += distance;
        },
        readOdometer() {
            return odometer;
        }
    };
```

```
}

const myCar = createCar('Toyota', 'Corolla');
myCar.drive(50);
console.log(myCar.readOdometer());  // Outputs: 50
```

In this example, `odometer` is private data encapsulated within the `createCar` function. It can't be accessed or modified directly from outside that function. Instead, it can only be modified through the `drive` method, and its current value can be read with the `readOdometer` method.

7.2 Encapsulation using ES6 Classes

With the introduction of ES6 classes, JavaScript now has built-in syntax for defining getter and setter methods, which can be used to create private properties and control how they are accessed and modified. Here's an example:

```
class Car {
    constructor(make, model) {
        this._make = make;
        this._model = model;
        this._odometer = 0;
    }

    get make() {
        return this._make;
    }

    get model() {
        return this._model;
    }

    drive(distance) {
        this._odometer += distance;
    }

    readOdometer() {
        return this._odometer;
    }
}
```

```
const myCar = new Car('Toyota', 'Corolla');
myCar.drive(50);
console.log(myCar.readOdometer());  // Outputs: 50
```

In this example, the underscore prefix (_) on _odometer, _make, and _model is a common convention to indicate that they are intended to be private properties. The drive method and readOdometer method are the only ways to modify and read the _odometer property. The make and model properties can only be read and not modified.

Keep in mind that this isn't true privacy. The properties can still be accessed and modified directly, and the underscore is just a convention to signal that they shouldn't be. ES6 introduced a new feature, JavaScript #private fields, to make properties truly private.

#_ Overview of JavaScript #private fields:

JavaScript introduced a new syntax for truly private class fields, denoted by a # sign before the field name. This syntax ensures that these fields are only accessible within the class they're defined. They're not accessible outside of the class, not even from instances of the class. This provides a stronger level of encapsulation compared to the underscore (_) convention.

Here's an example using private fields in the BankAccount class:

```
class BankAccount {
    #balance;

    constructor(initialDeposit) {
        this.#balance = initialDeposit;
    }

    deposit(amount) {
        this.#balance += amount;
    }

    withdraw(amount) {
        if (this.#balance >= amount) {
            this.#balance -= amount;
```

```
        } else {
            console.log("Insufficient funds for this withdrawal.");
        }
    }

    getBalance() {
        return this.#balance;
    }
}

let account = new BankAccount(100);
account.deposit(50);
account.withdraw(30);
console.log(account.getBalance());  // Outputs: 120
console.log(account.#balance);  // Error: Private field '#balance' must be
declared in an enclosing class
```

As you can see in this example, attempting to access the `#balance` field directly from outside the class results in a syntax error. This ensures that the `#balance` field can only be accessed or modified through the `deposit`, `withdraw`, and `getBalance` methods.

Section 8: Polymorphism

Polymorphism is a concept in OOP that describes the ability of objects of different classes to respond to the same method call in different ways. In other words, the same method can have different behaviors in different classes. In JavaScript, polymorphism is achieved through inheritance and method overriding.

8.1 Method Overriding

In the context of OOP in JavaScript, method overriding occurs when a method in a child class has the same name as one in its parent class. When the method is called on an instance of the child class, the child class's version of the method is executed.

Here's an example:

```
class Animal {
    makeSound() {
        console.log('The animal makes a sound');
    }
}

class Dog extends Animal {
    makeSound() {
        console.log('The dog barks');
    }
}

class Cat extends Animal {
    makeSound() {
        console.log('The cat meows');
    }
}

let dog = new Dog();
dog.makeSound();  // Outputs: 'The dog barks'

let cat = new Cat();
cat.makeSound();  // Outputs: 'The cat meows'
```

In this example, the Dog and Cat classes override the makeSound method of the Animal class. When makeSound is called on a Dog or Cat object, the overridden method in the respective child class is executed.

8.2 Super Keyword

Sometimes, you might want to execute the parent class's method before or after executing additional code in the child class's method. You can do this using the super keyword.

Here's an example:

```
class Dog extends Animal {
    makeSound() {
        super.makeSound();
        console.log('The dog barks');
    }
}
```

```
let dog = new Dog();
dog.makeSound();  // Outputs: 'The animal makes a sound' then 'The dog barks'
```

In this example, `super.makeSound()` calls the `makeSound` method of the `Animal` class. Then, the additional code in the `Dog` class's `makeSound` method is executed.

Section 9: Abstraction

Abstraction is a principle in object-oriented programming that deals with ideas rather than events. It is the process of exposing only the relevant and essential data to the users without showing unnecessary information. In JavaScript, abstraction can be achieved through classes, interfaces, and methods.

9.1 Abstraction with Classes

Abstraction can be achieved in JavaScript using classes. You can define a class that abstracts a certain concept or object, and then use that class throughout your code.

```
class Vehicle {
    constructor(name, type) {
        this.name = name;
        this.type = type;
    }

    start() {
        return `${this.name} engine started`;
    }
}

let vehicle = new Vehicle("Tesla", "Electric");
console.log(vehicle.start());  // Outputs: "Tesla engine started"
```

In this example, the `Vehicle` class is an abstraction of a vehicle. It has properties like `name` and `type`, and methods like `start`. This class can be used to create any type of vehicle and start it.

9.2 Abstraction with Methods

Another way to achieve abstraction in JavaScript is by providing methods that perform complex operations behind the scenes, while presenting a simple interface to the user.

```
class BankAccount {
    constructor(balance = 0) {
        this.balance = balance;
    }

    deposit(amount) {
        this.balance += amount;
        return this.balance;
    }

    withdraw(amount) {
        if (amount > this.balance) {
            return "Insufficient funds";
        } else {
            this.balance -= amount;
            return this.balance;
        }
    }
}

let account = new BankAccount(100);
console.log(account.deposit(50));  // Outputs: 150
console.log(account.withdraw(30));  // Outputs: 120
```

In this example, the `deposit` and `withdraw` methods of the `BankAccount` class are abstractions. They perform the necessary calculations to update the balance of the account, and the user doesn't need to know the details of how they work in order to use them.

Section 10: Advanced Topics

10.1 Composition vs Inheritance

Composition and inheritance are two ways to reuse code across objects in JavaScript.

Inheritance involves creating a parent class that contains shared code, and then creating child classes that inherit this code. We've discussed this extensively in the previous sections. One downside to inheritance is that it can lead to tightly coupled code and can be difficult to manage as classes grow and start to have different requirements.

Composition, on the other hand, is a way of building complex objects by combining simpler ones. In JavaScript, composition can be achieved by adding functions to the prototype of an object, or by using `Object.assign()` to combine objects.

Here's an example of composition:

```
const canEat = {
    eat: function() {
        console.log("Eating");
    }
};

const canWalk = {
    walk: function() {
        console.log("Walking");
    }
};

const person = Object.assign({}, canEat, canWalk);

person.eat();  // Outputs: "Eating"
person.walk();  // Outputs: "Walking"
```

In this example, we're composing a `person` object from the `canEat` and `canWalk` objects, rather than inheriting from a parent class.

10.2 Prototype vs Class Inheritance

In JavaScript, you can achieve inheritance in two ways: through prototype chains or through class inheritance introduced in ES6.

Prototype inheritance involves creating an object that serves as a prototype, and then creating new objects that inherit properties and methods from this prototype.

Class inheritance, on the other hand, involves creating a parent class that contains shared properties and methods, and then creating child classes that inherit from this parent class.

In reality, class inheritance is built on top of prototype inheritance. When you create a class and use the `extends` keyword, JavaScript sets up a prototype chain under the hood.

10.3 Factory Functions vs Constructor Functions vs Classes

Factory functions, constructor functions, and classes are all ways to create objects in JavaScript.

A factory function is a function that returns an object when you call it. Here's an example:

```
function createPerson(name, age) {
    return {
        name: name,
        age: age
    };
}

let person = createPerson("Alice", 25);
```

A constructor function is a function that is used with the `new` keyword to create new objects. Here's an example:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

let person = new Person("Alice", 25);
```

A class, as you've seen in the previous sections, is a way to define a blueprint for creating objects. Here's an example:

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}

let person = new Person("Alice", 25);
```

In each case, we're achieving the same end result: we're creating a `person` object with a `name` and `age`. The method you choose to create objects really depends on your specific needs and the conventions of your codebase.

Section 11: Modules

Modules in JavaScript are reusable pieces of code that can be exported from one program and imported for use in another program. They allow you to encapsulate code into separate files, organize related code together, and avoid polluting the global namespace.

JavaScript didn't originally have built-in support for modules, but with the introduction of ES6 (ES2015), JavaScript now natively supports module functionality. Here's how you can create and use modules in modern JavaScript:

1.1 Exporting from a Module

You can use the `export` keyword to export functions, objects, or primitive values from a module so that they can be used in other programs. You can use named exports or a default export.

Named export (allows multiple per module):

```
// mathFunctions.js
export function add(x, y) {
    return x + y;
}

export function subtract(x, y) {
    return x - y;
}
```

Default export (only one per module):

```
// MyModule.js
export default function() { console.log("I'm the default export!"); }
```

1.2 Importing from a Module

You can use the `import` keyword to import functions, objects, or values that were exported from another module.

Import named exports:

```
// main.js
import { add, subtract } from './mathFunctions.js';

console.log(add(2, 2));  // 4
console.log(subtract(2, 2));  // 0
```

Import default export:

```
// main.js
import myDefaultFunction from './MyModule.js';

myDefaultFunction();  // I'm the default export!
```