Web development basics

OOP in JS

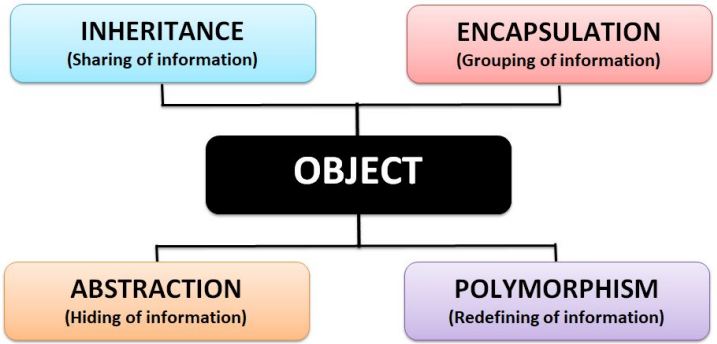
Introduction to OOP in JavaScript

OOP in JavaScript stands for **Object-Oriented Programming**. It's a **programming paradigm** that focuses on organizing code into objects that represent real-world entities, making the code more **readable**, **reusable** and **scalable**.

By using OOP in JavaScript, developers can build more complex and organized codebases, improving their productivity and the quality of the code they produce.

The 4 Pillars of OOP programming

1) **Abstraction**: The ability to hide implementation details and present a simplified interface to the user.
2) **Encapsulation**: Wrapping code and data into a single unit, making it easier to maintain and manage.
3) **Inheritance**: The ability to extend and reuse code, reducing duplicated effort and improving code maintainability.
4) **Polymorphism**: The ability to write code that can operate on multiple types of objects, making it more flexible and adaptable.



Explaining “this”

1) ES5 Functions: In ES5, the value of this is determined by how the function is called. If a function is called as a method of an object, this refers to the object. If a function is called as a standalone function, this refers to the global object (i.e., window in the browser or global in Node.js).

```
const person = {
  name: "John Doe",
  greet: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

person.greet(); // Output: Hello, my name is John Doe

const greet = person.greet;
greet(); // Output: Hello, my name is undefined (because this refers to the global object)
```

Explaining “this” 2

2) ES6 Arrow Functions: In ES6, arrow functions have a different this binding behavior than traditional functions. The value of this inside an arrow function is lexically scoped, meaning it takes the value of this from its surrounding context. This means that the value of this inside an arrow function is not affected by how the function is called.

```
const person = {
  name: "John Doe",
  greet: () => {
    console.log(`Hello, my name is ${this.name}`);
  }
};

person.greet(); // Output: Hello, my name is undefined (because this refers to the global object)

const greet = person.greet;
greet(); // Output: Hello, my name is undefined (because this refers to the global object)
```

Defining Classes in JavaScript

Classes in JavaScript can be defined using a class syntax introduced in ECMAScript 6 or using a **constructor** function. Example of defining class using class syntax:

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

  greet() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}
```

Creating Objects from Classes, Accessing Properties and Methods

Once a class is defined, you can create objects from it using the **new** keyword. Example:

```
const john = new Person("John Doe", 30);
const jane = new Person("Jane Doe", 28);
```

Accessing **Properties** and **Methods**: You can access the properties and methods of objects created from a class using the **dot notation**.

```
console.log(john.name); // Output: John Doe
john.greet(); // Output: Hello, my name is John Doe and I am 30 years old.
```

Constructor Functions

A **constructor** function is a special type of function that is used to create and initialize objects. The constructor function is automatically called when an object is created from a class or when the new keyword is used to create an instance of an object.

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.greet = function() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}
```

Setting Properties with the Constructor Function

The constructor function can be used to **set** properties on the newly created object. In the example above, name and age are properties that are set when an object is created from the Person constructor function. In this example, an object john is created from the Person constructor function and its properties name and age are set to "John Doe" and 30 respectively. These properties can be accessed using the dot notation.

```
const john = new Person("John Doe", 30);
console.log(john.name); // Output: John Doe
console.log(john.age); // Output: 30
```

Classwork for basic class usage

1) Create a Student class with a constructor that takes two parameters: name and grade.
2) Create properties for name and grade in the class and set them using the constructor.
3) Create a method displayInfo in the class that logs "Name: [name], Grade: [grade]" to the console.
4) Create another method addGrade in the class that takes a parameter value and adds it to the grade property.

Classwork for basic class usage - solution

```
class Student {
  constructor(name, grade) {
    this.name = name;
    this.grade = grade;
  }

  displayInfo() {
    console.log(`Name: ${this.name}, Grade: ${this.grade}`);
  }

  addGrade(value) {
    this.grade += value;
  }
}

const student1 = new Student("John Doe", 80);
student1.displayInfo(); // Output: Name: John Doe, Grade: 80
student1.addGrade(10);
student1.displayInfo(); // Output: Name: John Doe, Grade: 90
```

What is Inheritance

1) **Inheritance** is a concept in Object-Oriented Programming (OOP) that allows an object to inherit properties and behavior from its parent object. This means that a **child object** can access and use all of the properties and methods of its parent object. Inheritance is a way to **reuse** code and **prevent duplication** of code.

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

  eat() {
    console.log(`${this.name} is eating.`);
  }
}

class Dog extends Animal {
  bark() {
    console.log(`${this.name} is barking.`);
  }
}

const myDog = new Dog("Buddy");
myDog.eat(); // Output: Buddy is eating.
myDog.bark(); // Output: Buddy is barking.
```

Implementing Inheritance in JavaScript

2) In JavaScript, inheritance is implemented using the prototype object of an object. A child object can inherit properties and methods from its parent object by setting its prototype to be the parent object.

```
var animal = {
  name: "",
  eat: function() {
    console.log(`${this.name} is eating.`);
  }
};

var dog = Object.create(animal);
dog.name = "Buddy";
dog.bark = function() {
  console.log(`${this.name} is barking.`);
};

dog.eat(); // Output: Buddy is eating.
dog.bark(); // Output: Buddy is barking.
```

The extends Keyword

3) The **extends** keyword is a convenient way to implement inheritance in ECMAScript 6 (ES6) and later. The extends keyword is used to specify that one class should inherit from another class.

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

  eat() {
    console.log(`${this.name} is eating.`);
  }
}

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

  bark() {
    console.log(`${this.name} is barking.`);
  }
}

const myDog = new Dog("Buddy", "Labrador");
myDog.eat(); // Output: Buddy is eating.
myDog.bark(); // Output: Buddy is barking.
```

Overriding Methods

4) **Overriding Methods** in JavaScript: Overriding methods refer to a process in OOP where a subclass provides its own implementation of a method that is already defined in its parent class. This allows the subclass to inherit properties and methods from its parent class, but still provide its own custom implementation of specific methods.

```
class Vehicle {
  drive() {
    return "Driving a Vehicle";
  }
}

class Car extends Vehicle {
  drive() {
    return "Driving a Car";
  }
}

let myCar = new Car();
console.log(myCar.drive());
// Output: "Driving a Car"
```

What is Prototype-Based Inheritance

1) **Prototype-based inheritance** is a way of implementing inheritance in JavaScript where objects can inherit properties and methods from other objects. Instead of using classes to define objects and their relationships, objects can inherit from other objects directly through the use of **prototypes**.

```
let animal = {
  makesSound: function() {
    return "Some sound";
  }
};

let dog = Object.create(animal);
dog.bark = function() {
  return "Woof!";
};

console.log(dog.makesSound()); // Output: "Some sound"
console.log(dog.bark()); // Output: "Woof!"
```

The prototype Property

2) Every object in JavaScript has a special property called **prototype** that points to the object it **inherits** from. This property is used to create the **prototype chain** that determines the **inheritance hierarchy** of objects in JavaScript.

```
let animal = {
  makesSound: function() {
    return "Some sound";
  }
};

let dog = Object.create(animal);
dog.bark = function() {
  return "Woof!";
};

console.log(dog.prototype); // Output: Object { makesSound: [Function] }
```

Understanding the Prototype Chain

3) The **prototype chain** is a chain of objects that are linked to each other through the prototype property. When an object is asked for a property or method that it doesn't have, it will check its prototype, and so on, until it finds what it's looking for or until there are no more prototypes to check.

```
let animal = {
  makesSound: function() {
    return "Some sound";
  }
};

let dog = Object.create(animal);
dog.bark = function() {
  return "Woof!";
};

console.log(dog.makesSound()); // Output: "Some sound"
console.log(dog.hasOwnProperty("makesSound")); // Output: false
```

Classwork - extending and inheritance

1) Create a class called "Vehicle" with properties "make" and "model".
2) Create a class called "Car" that extends "Vehicle" and adds a property called "doors".
3) Create a Car object and print the "make", "model" and "doors" properties.

Classwork - extending and inheritance - solution§

```
// Define class Vehicle
class Vehicle {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }
}

// Define class Car that extends Vehicle
class Car extends Vehicle {
  constructor(make, model, doors) {
    super(make, model);
    this.doors = doors;
  }
}
```

```
// Create a Car object
let myCar = new Car("Toyota", "Camry", 4);

// Print the properties of myCar
console.log("Make: " + myCar.make);
console.log("Model: " + myCar.model);
console.log("Doors: " + myCar.doors);
```

Homework

1) Create a class called "Animal" with properties "species" and "age".
2) Add a method called "increaseAge" that increases the age of the animal by 1.
3) Create another class called "Dog" that extends "Animal" and adds a property called "breed".
4) Override the "increaseAge" method in the "Dog" class to also increase the "age" of the dog by 2.
5) Create a Dog object, increase its age multiple times and print the species, breed, and age of the dog.