OBJECT OBJECT-ORIENTED PROGRAMMING (OOP)

Object-oriented programming (OOP) in JavaScript is an approach to programming where you structure your code around objects. An object is a self-contained unit that combines data and functions (known as methods) that operate on that data.

CLASS

- This ***blueprint*** or ***design*** of an object is called ***class***
- A class is a blueprint for creating objects with similar properties and behaviors. JavaScript classes are introduced in ***ECMAScript 2015 (ES6)*** and provide a more structured way to implement object-oriented programming concepts.

  Syntax:
  class <ClassName>
  {
  Properties;
  Methods ;
  }

- *Any element that defined in class known as members of the class.*

e.g.

```
class Person {

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

 sayHello() {
  console.log(`Hello, my name is ${this.name}`);
 }
}

const john = new Person('John', 25);
john.sayHello();
```

THIS KEYWORD

This keyword refers to the context within which a function is executed. It is a special variable that is automatically defined for every function and represents the object that the function is being called on or the object that is currently being operated on.

The value of this is determined at runtime, based on how a function is called or invoked.

e.g.

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

```
  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }
}
```

CREATION OF OBJECT

create objects from the class using the **new** keyword followed by the class name and any necessary arguments for the constructor.

e.g.

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

  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }

  getAge()  {
    return this.age;
  }
}

// Create objects from the Person class
const john = new Person('John', 25);
const jane = new Person('Jane', 30);

// Access object properties and call methods
console.log(john.name); // Output: John
console.log(jane.getAge());
john.sayHello();
```

VISIBILITY MODE :

In JavaScript, there is no built-in support for visibility modes like public, private, or protected as found in some other object-oriented programming languages such as Java or C++. However, you can achieve similar encapsulation and control over object properties and methods using various techniques and conventions.

Public: Properties and methods that are intended to be accessible from outside the object are usually declared without any special prefixes or naming conventions. They are accessible using the dot notation.

```
class      Person      {
  constructor(name) {
    this.name = name; // Public property
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name}`); // Public method
```

```
  }
}

const john = new Person('John');
console.log(john.name); // Accessing public property
john.sayHello(); // Calling public method
```

Private: Properties and methods that are intended to be private, not accessible from outside the object, are typically denoted with a prefix or naming convention, such as an underscore #.

e.g.

```
class Person {
 #name  //private member

 constructor(name) {
  this.#name = name; // Private property
 }

 #privateMethod() {
  console.log('This is a private method'); // Private method
 }

 publicMethod() {
  console.log(`Hello, my name is ${this.name}`); // Public method accessing private property
  this.#privateMethod(); // Public method calling private method
 }
}

const john = new Person('John');
// console.log(john.name); // Accessing private property (not recommended)
john.publicMethod()
//  john.#privateMethod()
```

CONSTRUCTOR

In JavaScript, the constructor is a special method that is used for initializing objects created from a class. It is automatically called when a new object is instantiated using the new keyword.

e.g.

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

const john = new Person('John', 25);
console.log(john.name); // Output: John
console.log(john.age); // Output: 25
```

Note:

In JavaScript, a class can have only one constructor method***. If you try to define multiple constructor methods within a class, it will result in a syntax error.***

This limitation ensures that there is a single designated method responsible for initializing the object's properties and performing any necessary setup tasks.

e.g.

```
<script>
 class complex{
  #real
  #image
  constructor(real, image){
   this.real = real
   this.image = image
  }
  get real(){
   return this._real;
  }
  set real(real){
   this._real = real;
  }
  // get image(){
  //  return this._real;
  // }
  get image(){
   return this._image;
  }
  set image(image){
   this._image = image;
  }

  addition(other){
   const temp = new complex()
   temp.real= this.real + other.real
   temp.image= this.image + other.image
   return temp
  }
  display(){
   console.log(this.real + " + " + this.image +"i")
  }


 }

 const c1 = new complex(10,20)
 const c2 = new complex(30,40)
 const c3 = c1.addition(c2)
 c3.display()
```

```
</script>
```

STATIC MEMBER FUNCTION / STATIC METHOD

To declare a static function in a JavaScript class, you can use the static keyword before the function declaration. Static functions are associated with the class itself rather than the instances created from it. They can be accessed directly on the class without needing to create an object.

e.g.

```
class MathUtils {
 static add(a, b) {
 return a + b;
 }

 static subtract(a, b) {
  return a - b;
 }
}
console.log(MathUtils.add(5, 3)); // Output: 8
console.log(MathUtils.subtract(10, 4)); // Output: 6
```

Note:

1. Static methods also have this in JavaScript, but it is for class context.
2. They can only access other class context members.

STATIC DATA MEMBER

In JavaScript, a static data member is a property that is associated with the class itself rather than with individual instances of the class. It is shared among all instances of the class and can be accessed and modified without creating an instance of the class.

e.g

```
<script>
 class Emp{
 #empno
 #name
 #salary
  static total_salary =0
  constructor(empno, name , salary){
  this.empno = empno
   this.name = name
   this.salary=salary
   Emp.total_salary = Emp.total_salary + this.salary
  }

  display(){
   console.log(`empno : ${this.empno},  name : ${this.empno}, salary: ${this.salary} `)
  }
```

```
  static display_total_salary(){
    console.log(`total salary is ${Emp.total_salary}`)
  }

}

Emp.display_total_salary();

const e1 = new Emp(101,"ram",20000)
e1.display()
const e2 = new Emp(102,"shyam",40000)
e1.display()
const e3 = new Emp(103,"ghanshyam",10000)


e3.display()

Emp.display_total_salary();
</script>.
```

GETTER & SETTER

The get and set keywords are used to define special methods called getters and setters. They are part of the JavaScript Object Property API and provide a way to access and modify object properties.

The ***get keyword*** is used to define a getter method for a property. Getters are used to retrieve the value of a property. When the property is accessed, the corresponding getter method is automatically invoked.

The ***set keyword*** is used to define a setter method for a property. Setters are used to assign a value to a property. When a value is assigned to the property, the corresponding setter method is automatically invoked.


```
class Person {
// name
constructor() {
  this.name = "dsaf"; // Property without underscore prefix
}

// Getter
get name() {
  return this._name;
}
```

```
  // Setter
  set name(name) {
    this._name = name;
  }
}

const p = new Person();


console.log(p.name); // Calls the getter
p.name = "ram!"; // Calls the setter
console.log(p.name); // Calls the getter
```

PROTOTYPE PROPERTY

Every object in JavaScript has a built-in property, which is called its prototype. The prototype is itself an object, so the prototype will have its own prototype, making what's called a prototype chain. The chain ends when we reach a prototype that has null for its own prototype.

Note:

When you try to access a property of an object: if the property can't be found in the object itself, the prototype is searched for the property. If the property still can't be found, then the prototype's prototype is searched, and so on until either the property is found, or the end of the chain is reached, in which case undefined is returned. You can check what the is the prototype of myObjectby using. : **Object.getPrototypeOf(myObject); // Object { }**

INHERITANCE (PROTOTYPAL INHERITANCE)

Prototypal inheritance in JavaScript provides a flexible and powerful way to reuse and share code, enabling object-oriented programming paradigms. It allows objects to inherit properties and methods from other objects. It is achieved through the prototype chain, where objects are linked to a prototype object, and they inherit properties and methods from that prototype.

Note:

If a property or method is not found in an object, JavaScript will look up the prototype chain until it finds the property or reaches the end of the chain.

e.g.

```
<script>
  class Shape {
    #color
    constructor(color) {
      this.color =color
    }

    getColor()  {
      return this.color;
    }
```

```
}

class Rectangle extends Shape {
 constructor(width, height) {
 super("red");
  this.width = width;
  this.height = height;
 }

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

 calculatePerimeter() {
  return 2 * (this.width + this.height);
 }
}

// Create an instance of Rectangle
const rectangle = new Rectangle(5, 10);

// Access properties and methods
console.log(rectangle.getColor()); // Outputs 'red'
console.log(rectangle.calculateArea()); // Outputs 50
console.log(rectangle.calculatePerimeter()); // Outputs 30
</script>
```

SUPER KEYWORD:

In JavaScript, the super keyword is used to call functions or access properties of a parent class (also known as the superclass) from within a subclass. It is primarily used in the context of inheritance to invoke the superclass's constructor, methods, or access its properties.

The super keyword can be used in two main ways:

Calling the superclass's constructor:

When defining a constructor in a subclass, you can use ***super()*** to call the constructor of the superclass. This is useful for initializing inherited properties or performing setup tasks defined in the superclass.

e.g.

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

```
  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Calling the constructor of the superclass
    this.breed = breed;
  }

  speak() {
    super.speak(); // Calling the speak() method of the superclass
    console.log(`${this.name} is a ${this.breed}.`);
  }
}

let dog = new Dog("Buddy", "Golden Retriever");
dog.speak();
```

ACCESSING SUPERCLASS METHODS:

The super keyword can also be used to access methods or properties of the superclass from within the subclass. This allows you to invoke overridden methods of the superclass.

e.g.

```
 class Shape {
  constructor(color) {
  this.color = color;
  }

  getInfo() {
    return `This shape is ${this.color}.`;
  }
}

class Square extends Shape {
  constructor(color, sideLength) {
  super(color);
    this.sideLength = sideLength;
  }

  getInfo() {
    let shapeInfo = super.getInfo(); // Calling the getInfo() method of the superclass
    return `${shapeInfo} It has a side length of ${this.sideLength}.`;
  }
}

let square = new Square("red", 5);
```

```
console.log(square.getInfo());
```

POLYMORPHISM

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different types to be treated as if they belong to a common superclass or implement a shared interface. JavaScript, being a dynamically typed language, supports polymorphism through its prototype-based object model.

METHOD OVERRIDING:

Method overriding occurs when a subclass provides a different implementation of a method that is already defined in its superclass. This allows the subclass to redefine the behavior of the method while maintaining the same method signature.

e.g.

```
class Shape {
  draw() {
    console.log("Drawing a shape");
  }
}

class Circle extends Shape {
  draw() {
    console.log("Drawing a circle");
  }
}

class rectangle extends Shape {
  draw() {
    console.log("Drawing a ractangle");
  }
}

let shape = new Circle();
shape.draw(); // Output: "Drawing a circle"

shape = new rectangle();
shape.draw(); // Output: "Drawing a rectangle"
```

METHOD OVERLOADING:

Method overloading refers to defining multiple methods with the same name but different parameters. JavaScript does not directly support method overloading, as it does not consider the number or types of arguments when resolving function calls. However, you can emulate method overloading by checking the number or types of arguments manually.

```
class Calculator {
```

```
  add(a, b) {
    if (typeof a === "number" && typeof b === "number") {
      return a + b;
    } else if (typeof a === "string" && typeof b === "string") {
      return a.concat(b);
    }
  }
}

let calc = new Calculator();
console.log(calc.add(2, 3)); // Output: 5
console.log(calc.add("Hello, ", "world!")); // Output: "Hello, world!"
```

DUCK TYPING:

JavaScript follows the principle of "duck typing," which means that the suitability of an object for a particular operation is determined by the presence of specific methods or properties, rather than its explicit type.

e.g.

```
function makeSound(animal) {
    animal.sound();
  }

  let duck = {
    sound() {
      console.log("Quack!");
    },
  };

  let cat = {
    sound() {
      console.log("Meow!");
    },
  };

  makeSound(duck); // Output: "Quack!"
  makeSound(cat); // Output: "Meow!"
```

INTERFACES:

JavaScript doesn't have explicit interfaces like some other programming languages, you can define and implement object interfaces to achieve polymorphism. An interface defines a contract specifying the methods that an object must implement.

e.g.

```
// Interface definition
  class Printable {
```

```
  print() {
    throw new Error("Method 'print' must be implemented.");
  }
}

// Implementing the interface
class Book extends Printable {
constructor(title) {
    super();
    this.title = title;
  }

  print() {
    console.log(`Printing book: ${this.title}`);
  }
}

class Magazine extends Printable {
  constructor(name) {
    super();
    this.name = name;
  }

  print() {
    console.log(`Printing magazine: ${this.name}`);
  }
}

let p = new Book("The Catcher in the Rye");
p.print(); // Output: "Printing book: The Catcher in the Rye"
p = new Magazine("Time");
p.print(); // Output: "Printing magazine: Time"

// p = new Printable()
// p.print()
```