OOP (Object Oriented Programming) JavaScript

Object & Encapsulation:

- ## Why OOP?

    i. Example of unorganized Data:

```
const johnFName = 'John';
const johnLName = 'Doe';
const getJohnFullName = () => `${johnFName} ${johnLName}`;
console.log(getJohnFullName()); // Output: John Doe
```

- ii. Organizing Data into manageable and reusable components such as Objects.

```
const johnObject = {
  fName: 'John',
  lName: 'Doe',
  age: 20,
  getFullName: () => `${johnObject['fName']} ${johnObject.lName}`,
  getFullNameAndAge: function () {
    return `${this['fName']} ${this.lName} ${this.age}`;
  },
};
console.log(johnObject); // Output: { fName: 'John', lName: 'Doe', age: 20, getFullName: [F
console.log(johnObject.fName); // Output: John
console.log(johnObject['lName']); // Output: Doe
console.log(johnObject.getFullName()); // Output: John Doe
console.log(johnObject.getFullNameAndAge()); // Output: John Doe 20
```

- **Encapsulation:**

  i. Allows data and methods to be bundled together in an object.

  ii. Organizing and protecting the internal workings of an object.

  iii. Ensuring that its state and behavior are accessed and modified only through well-defined in

```
const johnFunc = function (fName, lName) {
  return {
    getFullName: function () {
      return `${fName} ${lName}`;
    },
    getNameAndAge: function (age) {
      return `${fName} ${lName} ${age}`;
    },
  };
};

console.log(johnFunc('John', 'Doe').getFullName()); // Output: John Doe
console.log(johnFunc('John', 'Doe').getNameAndAge('20')); // Output: John Doe 20
```

- # What is This?

  i. the Window this

```
this.location.replace('https://www.google.com'); // redirect to google
this.location.reload(); // reload the page
this.open('http://localhost:3000'); // open a new tab with the given URL
```

- ii. This in regular VS arrow functions.

```
const regular = function () {
  console.log(this); // this refers to the global object or undefined in strict mode
};
const arrow = () => {
  console.log(this); // this refers to the parent object
};
const objFunc = {
  label: 'I am an Object',
  regular,
  arrow,
};

regular(); // Output: global object
objFunc.regular(); // Output: objFunc
arrow(); // Output: global object
objFunc.arrow(); // Output: global object
```

- iii. There is much more to the "this" keyword.

READ MORE
READ MORE
learn more about:

```
call() apply() bind()
```

- ## Creating Object with the keyword Object( )

```
const john = new Object();

john['fName'] = 'John';

john.lName = 'Doe';

john.getFullName = function () {
  return `${this['fName']} ${this['lName']}`;
};

console.log(john.getFullName()); // Output: John Doe
```

- ## Creating an Object using spreading & destructuring.

```
const john = {};

john.fName = 'John';

john.lName = 'Doe';

john.getFullName = function () {
  return `${this['fName']} ${this['lName']}`;
};

const karl = { ...john, fName: 'Karl' };

console.log(karl.getFullName()); // Output: Karl Doe
```

- **Creating Object with Constructor functions.**

```
function Person(fName, lName) {
  this.fName = fName;
  this.lName = lName;
  this.getFullName = function () {
    return `${this.fName} ${this.lName}`;
  };
}

const john = new Person('John', 'Doe');
const karl = new Person('Karl', 'Doe');

console.log(john.getFullName()); // Output: John Doe
console.log(karl.getFullName()); // Output: Karl Doe
```

Prototypes

- **Prototype property and Constructor property.**

```
function Person(fName, lName) {
  this.fName = fName;
  this.lName = lName;
}

const john = new Person('John', 'Doe');
const karl = new Person('Karl', 'Doe');

john.getFullName = function () {
  return `${this.fName} ${this.lName}`;
};

console.log(john.getFullName()); // Output: John Doe
console.log(karl.getFullName()); // Output: TypeError: karl.getFullName is not a function
```

- **Example about Prototype property and constructor property.**

```
function Person(fName, lName) {
  this.fName = fName;
  this.lName = lName;
}

const john = new Person('John', 'Doe');
const karl = new Person('Karl', 'Doe');

Person.prototype.getFullName = function () {
  return `${this.fName} ${this.lName}`;
};

console.log(john.getFullName()); // Output: John Doe
console.log(karl.getFullName()); // Output: Karl Doe
```

- **Adding custom methods to the JS Object.**

```
Object.prototype.x = function () {
  console.log(this);
};

const a = '1';
const b = 1;
const c = [];
const e = true;
const d = (element) => element;

a.x(); // Output: Sting {'1'}
b.x(); // Output: Number {1}
c.x(); // Output: []
e.x(); // Output: Boolean {true}
d('Hello').x(); // Output: String{'Hello'}
```

- **Changing existing method behavior in prototype chain.**

```
const numbersArr = [1, 2, 3, 4, 5, 6];

numbersArr.push(7);
console.log(numbersArr); // Output: [1, 2, 3, 4, 5, 6, 7]

Array.prototype.push = function () {
  this.reverse();
};

numbersArr.push(10);
console.log(numbersArr); // Output: [7, 6, 5, 4, 3, 2, 1]

Array.prototype.crazyFunc = function (x) {
  this.shift();
  this.pop();
  this.reverse();
  this.push(x + 1);
};

numbersArr.crazyFunc(7);
numbersArr.push(1);
console.log(numbersArr); // Output: [ 6, 5, 4, 3, 2, 8, 2 ]
```

Abstraction & Inherited

- **Abstraction:**

i. Allows to represent complex systems in a simplified manner.

ii. Provides a way to create abstract models that can be easily used.

iii. In JavaScript, abstraction can be achieved using classes and inheritance.

- ## Inheritance:

  i. Allows objects to acquire properties and methods from a parent or base class.

  ii. Promotes code reuse, modularity, and the ability to create specialized classes.

- ## Example of Abstraction & Inheritance (Single-Level):

```
const Person = function (fName, LName) {
  this.fName = fName;
  this.LName = LName;
  this.getFullName = function () {
    return `${this.fName} ${this.LName}`;
  };
};

const john = new Person('John', 'Doe');

console.log(john.getFullName()); // Output: John Deo
```

Let's create an Object that inherits all Person's prototypes and add some

```
function Employ(fName, LName, email) {
  Person.call(this, fName, LName); // call Person's arguments by using .call

  // Person.apply(this, arguments); // Or you can use .apply( )

  this.email = email;

  this.getEmployData = function () {
    return `${this.fName} ${this.LName} ${this.email}`;
  };
}

const Michel = new Employ('Michel', 'James', 'MJ@google.com');

console.log(Michel.getEmployData()); // Output: Michel James
```

Classes & Polymorphism

- ## Polymorphism:

Allows objects of different classes to be treated as interchangeable, based on a common interface

- ## Example of Polymorphism in Classes

  i. **Basic Shape class with one function that calculate the area of the sh**

```
class Shape {
  calculateArea() {}
}
```

  ii. **Rectangle is a subclass of Shape. It has it's own calculateArea( ) me**

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

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

const rectangle = new Rectangle(4, 5);

console.log(rectangle.calculateArea()); // Output: 20
```

iii. **Circle is also a subclass of Shape. It has it's own calculateArea( ) me**

```
class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  calculateArea() {
    return Math.PI * this.radius * this.radius;
  }
}

const circle = new Circle(3);

console.log(circle.calculateArea()); // Output: 28.274333882308138
```

iv. **Circumference is a subclass of Circle. It has a method to calculate tl Circumference of a circle CircleCircumference( ).**

```
class Circumference extends Circle {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  circleCircumference(radius) {
    return 2 * Math.PI * radius;
  }
}

const circle = new Circle(3);

console.log(circle.circleCircumference()); // Output:  18.84954
```

- ## Static Properties and Methods

```
class Person {
  constructor(fName, lName) {
    Object.assign(this, { fName, lName });
  }

  getFullName() {
    return `${this.fName} ${this.lName} `;
  }

  static workTime() {
    const hour = new Date().getHours();
    if (hour > 17 || hour < 9) {
      return 'Out of work';
    } else {
      return 'At work';
    }
  }
}

console.log(Person.workTime()); // Output: At work

const John = new Person('John', 'Doe');
console.log(John.getFullName()); // Output: John Doe
```

- ## Accessors Getters & Setters

  i. Allow to define the behavior for accessing and modifying object properties.

  ii. Provide a way to control the reading and writing of object data and enable encapsulation.

```
class Person {
  #tel; // private property
  constructor(fName, lName, tel) {
    Object.assign(this, { fName, lName });
    this.#tel = tel;
  }

  // Normal function
  getFullName() {
    return `${this.fName} ${this.lName}`;
  }

  // Getter
  get fullName() {
    return `${this.fName} ${this.lName}`;
  }

  // Setter
  set fullName(value) {
    const [fName, lName] = value.split(' ');
    this.fName = fName;
    this.lName = lName;
  }
}

const john = new Person('John', 'Doe', '0151231234');

john.fName = 'Mike';
john.tel = '4567890'; // will add a new key to the Object
john['#tel'] = 456; // will change private property directly

console.log(john); // Output: User { fName: 'Mike', lName: 'Doe', '#tel': 456 }
console.log(john.getFullName()); // Output: Mike Doe
console.log(john.fullName()); // Output: Error: john.fullName is not a function

john.fullName = 'Gorge Dwo'; // Setter: Will set the name using the setter method
console.log(john.fullName); // Gorge Dwo
```

Additional Notes:

- ## Shallow VS Deep copying

    i. Object.assign() is a shallow copy, static method, and mutable

    ii. Object.create() is a deep copy, instance method, and immutable

    iii. Object.setPrototypeOf() is a deep copy, instance method, and mutable

    iv. Object.defineProperty() is a deep copy, instance method, and mutable

Shallow VS Deep copying

i. Arrow functions do not have prototype property.

ii. Arrow functions do not have their own this. The value of this inside an arrow function remair throughout the lifecycle of the function and is always bound to the value of this in the closes parent function.

iii. Arrow functions cannot be used as constructors and will throw an error when used with new

iv. Arrow functions cannot be used as methods on objects. They cannot be used as object pro

v. Arrow functions cannot be used as generators.

vi. const f = function(){} === f(){} !== const f = ()=>{}