WORKING WITH OBJECT ORIENTED PROGRAM

IN JAVASCRIPT

OOP KEY CONCEPTS

Object-oriented programming (OOP) : is a programming paradigm based on the concept of "objects," which can contain data and code that manipulates that data.

Key concepts of OOP include:

1. **Classes:** Blueprints for creating objects.
2. **Objects:** Instances of classes.
3. **Encapsulation:** Bundling data and methods that operate on the data within one unit, and restricting access to some of the object's components.
4. **Inheritance:** Mechanism where one class can inherit traits from another.
5. **Polymorphism:** Ability to process objects differently based on their class or data type.
6. **Abstraction:** Simplifying complex reality by modeling classes based on essential qualities.

Sangarananthan P
@Sangarananthan

this Keyword:

Contextual usage of this in functions, objects, and global scope: this refers to the object in context, varying based on where it is called.

javascript Copy code

```javascript
// Global Scope
console.log(this); // In browser, `this` refers to the global window object

// Function Context
function globalFunction() {
    console.log(this); // In non-strict mode, `this` refers to the global object
}
globalFunction();

// Object Context
const obj = {
    name: 'Object',
    showThis: function() {
        console.log(this); // `this` refers to the object `obj`
    }
};
obj.showThis();

// Constructor Function
function Person(name) {
    this.name = name; // `this` refers to the newly created instance
}

const person1 = new Person('Alice');
console.log(person1.name); // Alice
```

Sangarananthan P
@Sangarananthan

Save

Object Creation Patterns:

- Object literals: Creating objects using a concise syntax with key-value pairs

javascript Copy code

```javascript
const objLiteral = {
    name: 'Object Literal',
    sayHello: function() {
        console.log('Hello from ' + this.name);
    }
};
objLiteral.sayHello();
```

- Factory functions: Functions that return new objects without using the new keyword.

javascript Copy code

```javascript
function createPerson(name) {
    return {
        name: name,
        sayHello: function() {
            console.log('Hello from ' + this.name);
        }
    };
}
const personFactory = createPerson('Factory Function');
personFactory.sayHello();
```

Sangarananthan P
@Sangarananthan

Save

- Constructor functions: Functions designed to be used with the new keyword to create objects.

javascript Copy code

```javascript
function PersonConstructor(name) {
    this.name = name;
    this.sayHello = function() {
        console.log('Hello from ' + this.name);
    };
}

const personConstructor = new PersonConstructor('Constructor Function');
personConstructor.sayHello();
```

Constructor Functions:

- Constructor functions with 'new' create instances by setting 'this' to the newly created object within the function.

javascript Copy code

```javascript
function Car(model) {
    this.model = model;
}

const car1 = new Car('Tesla');
const car2 = new Car('BMW');

console.log(car1.model); // Tesla
console.log(car2.model); // BMW
```

Sangarananthan P
@Sangarananthan

Prototypes | Prototype Chain:

- Accessing and modifying prototypes: Changing the prototype properties of objects to share common behaviors.
- Custom methods on built-in prototypes: Adding new methods to JavaScript's built-in object prototypes like Array or String.

javascript Copy code

```javascript
function Dog(name) {
    this.name = name;
}

Dog.prototype.bark = function() {
    console.log(this.name + ' says Woof!');
};

const dog1 = new Dog('Buddy');
dog1.bark(); // Buddy says Woof!

// Custom methods on built-in prototypes
Array.prototype.last = function() {
    return this[this.length - 1];
};

const numbers = [1, 2, 3];
console.log(numbers.last()); // 3
```

Sangarananthan P
@Sangarananthan

Save

Prototype Inheritance:

- Inheritance using constructor functions: Achieving inheritance by setting a constructor function's prototype to an instance of another constructor.
- Setting up inheritance with Object.create: Creating a new object with the specified prototype using Object.create.

javascript Copy code

```javascript
function Animal(name) {
    this.name = name;
}

Animal.prototype.speak = function() {
    console.log(this.name + ' makes a sound.');
};

function Cat(name) {
    Animal.call(this, name);
}

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

Cat.prototype.speak = function() {
    console.log(this.name + ' meows.');
};

const cat = new Cat('Whiskers');
cat.speak(); // Whiskers meows.
```

Sangarananthan P
@Sangarananthan

Save

Class Syntax:

- Defining classes: Using the class keyword to create class-based structures.

javascript Copy code

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

    sayHello() {
        console.log('Hello from ' + this.name);
    }
}

const personClass = new PersonClass('Class Syntax');
personClass.sayHello();
```

- Class inheritance with extends and super: Using extends to create a subclass and super to call the parent class's constructor.

javascript Copy code

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

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

class DogClass extends AnimalClass {
    constructor(name) {
        super(name);
    }

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

const dogClass = new DogClass('Buddy');
dogClass.speak(); // Buddy barks.
```

Sangarananthan P
@Sangarananthan

Modifiers (Access Control):

- Public fields and methods: Properties and methods accessible from outside the class.

javascript Copy code

```javascript
class PublicExample {
    constructor(name) {
        this.name = name; // Public field
    }

    greet() {
        console.log('Hello ' + this.name); // Public method
    }
}

const publicExample = new PublicExample('Alice');
publicExample.greet(); // Hello Alice
```

- Protected fields and methods: Fields and methods intended to be accessible only within the class and its subclasses.

javascript Copy code

```javascript
class ProtectedExample {
    constructor(name) {
        this._name = name; // Protected field convention
    }

    _greet() { // Protected method convention
        console.log('Hello ' + this._name);
    }
}

const protectedExample = new ProtectedExample('Alice');
protectedExample._greet(); // Hello Alice (Conventionally protected)
```

- Private fields and methods using closures: Restricting access using closures to create private data and methods.

javascript Copy code

```javascript
function PrivateExample(name) {
    let privateName = name; // Private field

    this.getName = function() { // Public method
        return privateName;
    };

    this.setName = function(newName) { // Public method
        privateName = newName;
    };
}

const privateExample = new PrivateExample('Alice');
console.log(privateExample.getName()); // Alice
privateExample.setName('Bob');
console.log(privateExample.getName()); // Bob
```

Sangarananthan P
@Sangarananthan

Save

Encapsulation:

- Restricting access to private data: Hiding internal object details using closures or private fields.
- Using public methods to manipulate private data: Providing controlled access to private data through public methods.

javascript Copy code

```javascript
class EncapsulationExample {
    #privateField; // Private field

    constructor(value) {
        this.#privateField = value;
    }

    getPrivateField() {
        return this.#privateField;
    }

    setPrivateField(value) {
        this.#privateField = value;
    }
}

const encapsulationExample = new EncapsulationExample('secret');
console.log(encapsulationExample.getPrivateField()); // secret
encapsulationExample.setPrivateField('new secret');
console.log(encapsulationExample.getPrivateField()); // new secret
```

Sangarananthan P
@Sangarananthan

Save

Abstraction:

- Abstract methods in base classes: Defining methods in a base class that must be implemented by subclasses.
- Implementing abstract methods in subclasses: Providing specific implementations for abstract methods in derived classes.

javascript Copy code

```javascript
class AbstractBase {
    constructor() {
        if (new.target === AbstractBase) {
            throw new TypeError('Cannot construct AbstractBase instances directly');
        }
    }

    abstractMethod() {
        throw new Error('Abstract method must be implemented');
    }
}

class ConcreteClass extends AbstractBase {
    abstractMethod() {
        console.log('Abstract method implemented in ConcreteClass');
    }
}

const concreteInstance = new ConcreteClass();
concreteInstance.abstractMethod(); // Abstract method implemented in ConcreteClass
```

Sangarananthan P
@Sangarananthan

Save

Inheritance:

- ES5 Syntax: Constructor functions and Object.create: Using constructor functions and Object.create for inheritance in ES5.

javascript Copy code

```javascript
function AnimalES5(name) {
    this.name = name;
}

AnimalES5.prototype.speak = function() {
    console.log(this.name + ' makes a sound.');
};

function DogES5(name) {
    AnimalES5.call(this, name);
}

DogES5.prototype = Object.create(AnimalES5.prototype);
DogES5.prototype.constructor = DogES5;

DogES5.prototype.speak = function() {
    console.log(this.name + ' barks.');
};

const dogES5 = new DogES5('Buddy');
dogES5.speak(); // Buddy barks.
```

Sangarananthan P
@Sangarananthan

Save

- ES6 Syntax: Class-based inheritance: Using the class syntax and extends keyword for inheritance in ES6.

javascript Copy code

```javascript
class AnimalES6 {
    constructor(name) {
        this.name = name;
    }

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

class DogES6 extends AnimalES6 {
    speak() {
        console.log(this.name + ' barks.');
    }
}

const dogES6 = new DogES6('Buddy');
dogES6.speak(); // Buddy barks.
```

Sangarananthan P
@Sangarananthan

Save

Polymorphism:

- Method overriding: Redefining a method in a subclass that exists in the parent class.
- Treating objects of different classes as instances of the parent class: Using a unified interface to interact with objects of different subclasses.

javascript Copy code

```javascript
class Bird {
    fly() {
        console.log('Bird is flying');
    }
}

class Eagle extends Bird {
    fly() {
        console.log('Eagle is soaring high');
    }
}

class Sparrow extends Bird {
    fly() {
        console.log('Sparrow is flying low');
    }
}

const birds = [new Bird(), new Eagle(), new Sparrow()];
birds.forEach(bird => bird.fly());
// Output:
// Bird is flying
// Eagle is soaring high
// Sparrow is flying low
```

Sangarananthan P
@Sangarananthan

Save

Follow For More

Sangarananthan P

@Sangarananthan

Save