vivasoft

Learnathon 2.0 - JS Class 5

OOP concept in JS

M. Firoz Ahmed | Software Engineer (L-I)

Learnathon - 2.0

vivasoft

What is OOP?

Object-Oriented Programming (OOP) is a fundamental programming style that structures code around objects (class instances).

- Encapsulation: bundling of data and methods into a single unit ,
- Inheritance: ability of objects to inherit properties and methods from parent objects,
- Polymorphism: capacity for different objects to respond to the same method calls in a way that suits their type,
- Abstraction: simplifying complex systems by focusing on essential features while hiding unnecessary details

vivasoft

Class in JS

The class keyword was introduced in ES6. It provides a more structured and syntactically clear way to define blueprints for creating objects. Classes serve as templates for object creation, encapsulating both properties and methods. By using classes, developers can create objects with shared characteristics more easily.

vivasoft

Class in JS

```
class Car {
  constructor(brand) {
    this.brand = brand;
  }

  startEngine() {
    console.log(`Starting the engine of ${this.brand}.`);
  }
}

const car = new Car('Toyota');
car.startEngine();
```

vivasoft

Encapsulation

```
function BankAccount(accountHolder, initialBalance) {
  this.name = accountHolder;
  this.balance = initialBalance;

  this.getBalance = function () {
    return this.balance;
  };

  this.deposit = function (amount) {
    if (amount > 0) {
      this.balance += amount;
      return `Deposited ${amount} dollars. New balance: ${this.balance} dollars.`;
    } else {
      return 'Invalid deposit amount.';
    }
  };

  this.withdraw = function (amount) {
    if (amount > 0 && amount <= this.balance) {
      this.balance -= amount;
      return `Withdrawn ${amount} dollars. New balance: ${this.balance} dollars.`;
    } else {
      return 'Invalid withdrawal amount or insufficient funds.';
    }
  };
}
```

```
const acc1 = new BankAccount('Alice', 1000);
const acc2 = new BankAccount('Bob', 500);

console.log(acc1.getBalance());
console.log(acc2.deposit(200));
console.log(acc2.withdraw(700));
```

vivasoft

Inheritance

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

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

function Dog(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
Dog.prototype.bark = function () {
  console.log(`${this.name} barks.`);
};
```

```
const animal = new Animal("Generic Animal");
const dog = new Dog("Buddy", "Golden Retriever");

animal.speak();
dog.speak();
dog.bark();
```

vivasoft

Polymorphism

```
function Shape() {}

Shape.prototype.getArea = function () {
  return "This is a generic shape and doesn't have a specific area.";
};

function Circle(radius) {
  this.radius = radius;
}

Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle;
Circle.prototype.getArea = function () {
  return Math.PI * this.radius ** 2;
};


function Square(sideLength) {
  this.sideLength = sideLength;
}

Square.prototype = Object.create(Shape.prototype);
Square.prototype.constructor = Square;
Square.prototype.getArea = function () {
  return this.sideLength ** 2;
};
```

```
const myCircle = new Circle(5);
const mySquare = new Square(4);

console.log(`Circle Area:${myCircle.getArea()}`);
console.log(`Square Area:${mySquare.getArea()}`);
```

vivasoft

Abstraction

```
function InternetBanking(mobile, initBalance) {
  this.mobile = mobile;
  this.balance = initBalance;

  this.checkValidNumber = function (number) {
    // Process here
  }

  this.checkAmountValidity = function (amount) {
    // Process here
  }

  this.addAmount = function (amount, number) {
    // Process here
  }

  this.transferBalance = function (amount, toNumber) {
    let res = true;
    res &= this.checkValidNumber(toNumber);
    res &= this.checkAmountValidity(amount);
    res &= this.addAmount(amount, toNumber);
    if (res) {
      console.log('Successfully balance transferred.');
    } else {
      console.log('Balance transfer failed.');
    }
  }
}
```

```
const mobileBanking = new InternetBanking('01111111111', 15000);
mobileBanking.transferBalance(7000, '01222222222222');
```

vivasoft

Abstraction

```
function Animal(name) {
  if (this.constructor === Animal) {
    throw new Error("Cannot instantiate an abstract class.");
  }

  this.name = name;
}

Animal.prototype.makeSound = function () {
  throw new Error("Abstract method 'makeSound' must be implemented by subclasses.");
};

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

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
Dog.prototype.makeSound = function () {
  console.log(`${this.name} barks.`);
};
```

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

Cat.prototype = Object.create(Animal.prototype);
Cat.prototype.constructor = Cat;
Cat.prototype.makeSound = function () {
  console.log(`${this.name} meows.`);
};
```

```
const dog = new Dog("Buddy");
const cat = new Cat("Whiskers");

dog.makeSound();
cat.makeSound();
```

vivasoft

Factory Pattern

The Factory Pattern is a design pattern that addresses the problem of object creation. It provides a centralized interface for creating objects.

- call factory methods to create objects instead of directly invoking constructors or class instantiation,
- promotes loose coupling between client code and object creation,
- makes it easier to modify or extend the creation process without affecting the rest of the application.

vivasoft

Factory Pattern

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

const person = createPerson('Mr. X', 30);
person.sayHello();
```

vivasoft

Constructor Pattern

JavaScript used constructor functions for creating objects before classes were introduced. These functions are invoked using the ‘new’ keyword, creating instances of objects with properties and methods.

- act as blueprints for object instantiation,
- crucial in achieving encapsulation, inheritance, and code organization.

vivasoft

Constructor Pattern using Function

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

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

const person = new Person('Mr. X', 30);
person.sayHello();
```

vivasoft

Constructor Pattern using Class

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

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

const person = new Person("Mr. X", 30);
person.sayHello();
```

vivasoft

New Keyword in JS

The new keyword plays a vital role in object creation. When used with constructor functions or class constructors, it initiates the object instantiation process. It

- creates a new object,
- sets the prototype chain to link the object to its constructor's prototype, and
- assigns this to refer to the new object.

Without new, this would point to the global object, leading to unexpected behavior. The new keyword ensures proper object creation and linkage, facilitating the use of constructor patterns and classes.

vivasoft

Getter/Setter method in JS

Getter and setter methods provide controlled access to object properties.

- Getter methods retrieve property values,
- Setter methods enable controlled modification of property values.

They are defined within classes or objects using the get and set keywords. They are valuable for enforcing data validation, maintaining data integrity, and controlling access to object state.

vivasoft

Getter/Setter method in JS

```
const car = {
  _name: "Ford",

  get getName() {
    return this._name;
  },

  set setName(name) {
    this._name = name;
  }
};

console.log(car.getName);
car.setName = "Toyota";
console.log(car.getName);
```

```
const car = {
    name: "Ford",
};

Object.defineProperty(car, "getName", {
    get: function () {
        return this.name;
    },
});

Object.defineProperty(car, "setName", {
    set: function (value) {
        this.name = value;
    },
});

console.log(car.getName);
car.setName = "Toyota";
console.log(car.getName);
```

vivasoft

Prototype and Prototypical Inheritance

JavaScript employs a prototype-based inheritance model. Each object has a prototype property that serves as a reference to another object. When a property or method is accessed, JavaScript first looks for it on the object itself. If not found, it follows the prototype chain, searching through linked objects until it locates the property or method. This mechanism enables prototypical inheritance, where objects can inherit properties and methods from their prototypes.

vivasoft

Prototype and Prototypical Inheritance

```
function Car(color, name) {
  this.color = color;
  this.name = name;

  this.getColor = function () {
    return this.color;
  };

  this.getName = function () {
    return this.name;
  };
}

const car1 = new Car("red", "BMW");
const car2 = new Car("black", "Ford");

console.log(car1, car2);
```

```
function Car(color, name) {
    this.color = color;
    this.name = name;
}

Car.prototype.getColor = function () {
    return this.color;
};

Car.prototype.getName = function () {
    return this.name;
};

const car1 = new Car("red", "BMW");
const car2 = new Car("black", "Ford");

console.log(car1, car2);
```

vivasoft

Multi-level Inheritance

Multi-level inheritance is a type of inheritance where objects inherit properties and methods from a hierarchy of parent objects. In this hierarchy, an object can have a parent object, and that parent object can, in turn, have its parent, forming a chain of inheritance. This concept allows for the creation of complex class structures where objects inherit characteristics from multiple levels of parent objects.

vivasoft

Multi-level Inheritance

```
const grandfather = {
    eyeColor: "black",
    grandfatherProperty: function () {
        console.log("Grandfather's property");
    },
};

const father = {
    body: "slim",
    __proto__: grandfather,
    fatherProperty: function () {
        console.log("Father's property");
    },
};

const son = {
    __proto__: father,
    sonProperty: function () {
        console.log("Son's Property");
    },
};

console.log(father.eyeColor);
console.log(son.eyeColor);
console.log(son.body);
son.grandfatherProperty();
```

```
class Grandfather {
    eyeColor = "black";
    grandfatherProperty() {
        console.log("Grandfather's property");
    }
}

class Father extends Grandfather {
    body = "slim";
    fatherProperty() {
        console.log("Father's property");
    }
}

class Son extends Father {
    sonsProperty() {
        console.log("Son's Property");
    }
}

const father = new Father();
console.log(father.eyeColor);

const son = new Son();
console.log(son.eyeColor);
console.log(son.body);
son.grandfatherProperty();
```

vivasoft

Questions?

Learnathon - 2.0

vivasoft

Thank You 

Learnathon - 2.0