14. Factories and Classes

Factories and classes are two approaches to creating objects and structures in JavaScript. Each has its own purpose and advantages.

Factories: Factories are functions that generate and return objects. These functions act as "factories" to create instances of objects with specific properties and methods. Factories are a flexible way to create objects in JavaScript as they can customize object creation based on the provided arguments.

```
function createPerson(name, age) {
    return {
        name: name,
        age: age,
        greet: function() {
                    console.log(`Hello, I'm ${this.name} and I'm
${this.age} years old.`);
        }
    };
}

const person1 = createPerson("John", 30);
person1.greet();
// Prints "Hello, I'm John and I'm 30 years old."
```

Classes: Classes are a concept introduced in ES6 that allows creating objects using a more object-oriented syntax. Classes serve as a blueprint for creating objects and define properties and methods that all instances will share. Although in JavaScript, classes are "syntactic sugar" over the prototype system, they provide a more structured way of working with objects.

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

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

const person2 = new Person("Jane", 25);
person2.greet();
// Prints "Hello, I'm Jane and I'm 25 years old."
```

In summary:

- **Factories:** Use functions to create and customize objects.
- **Classes:** Define object blueprints with shared properties and methods.

The choice between factories and classes depends on your needs and preferences. Factories are more flexible and versatile, while classes provide a more object-oriented structure and are especially useful when working with inheritance and similar objects.

15. this, call, apply, and bind

this, call, apply, and bind are concepts and methods in JavaScript related to managing the value of this in the context of a function. Here's a description of each of them:

this: In JavaScript, this refers to the object that is the current "execution context" in a function. The value of this can change depending on how a function is called and where it is in the code.

call and apply: Both methods allow temporarily changing the value of this in a function and then executing it. The main difference between call and apply lies in how arguments are passed:

call: Invokes a function with a specific value for this and individually passed arguments.

```
function greet(name) {
    console.log(`Hello, ${name}! My name is ${this.name}`);
}

const person = { name: 'John' };

greet.call(person, 'Maria'); // Prints: "Hello, Maria! My name is
John"
```

apply: Similar to call, but arguments are passed as an array.

```
function greet(name) {
    console.log(`Hello, ${name}! My name is ${this.name}`);
}

const person = { name: 'John' };

greet.apply(person, ['Maria']);
// Prints: "Hello, Maria! My name is John"
```

bind: The bind method returns a new function where the value of this is fixed to the provided value, and the initial arguments (if any) are "bound" to the new function.

```
function greet(name) {
    console.log(`Hello, ${name}! My name is ${this.name}`);
}

const person = { name: 'John' };
const greetPerson = greet.bind(person);

greetPerson('Maria'); // Prints: "Hello, Maria! My name is John"
```

In summary:

- **this:** Refers to the current execution context.
- **call:** Invokes a function with a specific value for this and individually passed arguments.
- **apply:** Similar to call, but arguments are passed as an array.
- **bind:** Creates a new function with this and arguments fixed.

These methods are useful for controlling the context of this in different situations, especially when working with functions that are part of objects or when you need to manipulate how a function is invoked.

16. new, Constructor, instanceof, and Instances

These concepts are related to the creation and use of objects through constructors in JavaScript.

new: new is a keyword used to create a new instance of an object from a constructor function. When used with a constructor function, new creates a new object and assigns the value of this within the function to the new object. Then, the constructor function can initialize properties and methods on that object.

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

const person1 = new Person("John", 30);
```

Constructor: A constructor is a function used to create and configure objects. Constructors often follow a naming convention with the first letter capitalized. Inside the constructor, you can initialize properties and methods of the object that will be created with new.

instanceof: instanceof is an operator used to check if an object is an instance of a specific class or constructor. It returns true if the object is an instance of the given class, otherwise, it returns false.

```
console.log(person1 instanceof Person); // Returns true
```

Instances: An instance is a unique object created from a constructor. Each time you use new with a constructor, a new instance of the object is created with its own properties and methods.

```
const person2 = new Person("Maria", 25);
```

In summary:

- **new:** A keyword used to create instances of objects from constructors.
- **Constructor:** A function used to create and configure objects.
- **instanceof:** An operator to check if an object is an instance of a class or constructor.
- **Instances:** Unique objects created from constructors.

Constructors and instances are essential for object-oriented programming in JavaScript, allowing the creation of objects with shared properties and behaviors.

17. Prototypal Inheritance and Prototype Chain

Prototypal inheritance and the prototype chain are fundamental concepts in JavaScript for achieving code reuse and establishing relationships between objects.

Prototypal Inheritance: In JavaScript, prototypal inheritance is a mechanism by which an object can inherit properties and methods from another object called its "prototype." Instead of traditional classes, JavaScript uses prototypal inheritance to create relationships between objects. When a property or method is looked up in an object, if it's not found in the object itself, it is searched in its prototype and in the ascending prototype chain.

Prototype Chain: The prototype chain is a series of links between prototype objects. Each object has an internal link to its prototype, and that prototype, in turn, may have its own prototype. This chain continues until it reaches the base object, which is the final prototype in the chain.

Example of Prototypal Inheritance and Prototype Chain:

```
// Define an "Animal" constructor
function Animal(name) {
    this.name = name;
}

// Add a "greet" method to the prototype of "Animal"
Animal.prototype.greet = function() {
    console.log(`Hello, I'm a ${this.name}`);
};
```

```
// Define a "Dog" constructor that inherits from "Animal"
function Dog(name, breed) {
 Animal.call(this, name); // Call the "Animal" constructor
 this.breed = breed;
}

// Establish prototypal inheritance
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Add an additional method to "Dog"
Dog.prototype.bark = function() {
 console.log("Woof woof!");
};

// Create an instance of "Dog"
const myDog = new Dog("Max", "Labrador");
myDog.greet(); // Prints "Hello, I'm Max"
myDog.bark(); // Prints "Woof woof!"
```

In this example:

- We create the Animal constructor with a greet method.
- We create the Dog constructor that inherits from Animal and has a bark method.
- Prototypal inheritance is established using Object.create(), and we ensure that the Dog constructor points correctly.
- We create an instance of Dog, and we can access methods from both Animal and Dog.

Prototypal inheritance and the prototype chain are crucial for understanding how objects work and code reuse in JavaScript.

18. Object.create and Object.assign

Both Object.create and Object.assign are important methods in JavaScript used for working with objects, but they serve different purposes.

Object.create: Object.create is a method used to create a new object and set its prototype (parent object). It allows creating objects that inherit properties and methods from another object. It's especially useful for implementing prototypal inheritance in JavaScript.

```
const newObj = Object.create(prototype);
```

Example of Object.create:

```
const animal = {
    sound: "Makes a sound",
    makeSound: function() {
        console.log(this.sound);
    }
};

const dog = Object.create(animal);
dog.sound = "Woof woof";
dog.makeSound(); // Prints "Woof woof"
```

Object.assign: Object.assign is a method used to copy enumerable properties from one or more source objects to a target object. If there are properties with the same name in the target object, they will be overwritten. It's useful for combining objects or cloning objects.

```
Object.assign(targetObject, sourceObject1, sourceObject2, ...);
```

Example of Object.assign:

```
const target = {};
const source1 = { name: "John", age: 30 };
const source2 = { city: "New York" };

Object.assign(target, source1, source2);
console.log(target);
// Prints { name: "John", age: 30, city: "New York" }
```

In summary:

- **Object.create:** Creates a new object with a specified prototype.
- **Object.assign:** Combines or copies enumerable properties from source objects to a target object.

Object.create is useful for establishing prototypal inheritance relationships, while Object.assign is useful for combining properties from multiple objects or for cloning objects.

19. map, reduce, and filter

These are three high-level methods provided by JavaScript for working with arrays in a more functional and elegant way. Each one has a specific purpose and is widely used for transforming, filtering, and reducing data in arrays.

map: The map method is used to transform each element of an array into a new array by applying a function to each element. It returns a new array with the results of applying the function to each original element in the same order.

```
const numbers = [1, 2, 3, 4];
const doubles = numbers.map(number => number * 2);
// doubles is now [2, 4, 6, 8]
```

filter: The filter method is used to create a new array with all elements that pass a test (meet a condition) provided by a function. It returns a new array with the elements that satisfy the condition.

```
const numbers = [1, 2, 3, 4, 5, 6];
const evens = numbers.filter(number => number % 2 === 0);
// evens is now [2, 4, 6]
```

reduce: The reduce method is used to reduce an array to a single cumulative value. It applies a function to an accumulator and each element in the array (from left to right), reducing the array to a single value.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, number) => accumulator +
number, 0);
// sum is 15
```

20. Pure Functions, Side Effects, State Mutation, and Event Propagation

These concepts are related to functional programming, data manipulation, and interaction in JavaScript. Here's an explanation of each one:

Pure Functions: Pure functions are functions that produce the same result for the same arguments and have no side effects on the environment. This means they don't modify external variables, perform input/output operations, and don't depend on external mutable data.

Characteristics of pure functions:

- **Determinism:** Same input, same result.
- **No side effects:** They don't alter data outside the function.

Side Effects: Side effects are observable changes outside a function due to its execution. These changes can include modifying external variables, accessing external resources (network, files), and manipulating the DOM.

State Mutation: State mutation occurs when mutable data, such as objects or arrays, is modified directly instead of creating new ones. It can lead to unwanted side effects and make tracking changes difficult.

Event Propagation: In the context of the DOM and interactions on a web page, event propagation refers to the flow of an event through the hierarchy of elements. Events can propagate from the target element to ancestor elements or vice versa.

Phases of Event Propagation:

- **Capture:** The event descends from the root element to the target.
- **Target:** The event reaches the target element.
- **Bubbling:** The event ascends from the target to the root element.

These concepts are fundamental for writing more predictable, maintainable, and scalable code. The focus on pure functions and avoiding side effects contributes to functional programming, while understanding state mutation is crucial for working with changing data. Event propagation is essential for managing interaction in web applications.

21. Closures

Closures are an important concept in programming, referring to a function's ability to access and remember variables from its lexical scope even after that function has finished execution and left that scope. Closures enable the creation of functions that maintain access to variables even when they are no longer in the scope of the function that created them.

```
function counter() {
    let count = 0;

    function increment() {
        count++;
        console.log(count);
    }

    return increment;
}

const counter1 = counter();
counter1(); // Prints: 1
counter1(); // Prints: 2=
```

In this example, the counter function returns the increment function. The count variable is retained in the lexical scope of the counter function but remains accessible through the increment function even after the counter function has finished execution. This is made possible by closures.

Closures are particularly useful for creating functions with private states, maintaining encapsulation in JavaScript. They are also used in patterns like modules and callback functions in events.

22. High Order Functions

High Order Functions are functions that can accept other functions as arguments and/or return functions as results. These functions are a fundamental part of functional programming and enable the creation of more modular, reusable, and expressive code. High Order Functions make it easier to use common programming patterns such as mapping, filtering, and reducing data.

Here are some examples and key concepts related to High Order Functions:

Functions as Arguments: You can pass a function as an argument to another function. This allows customizing the behavior of the receiving function.

```
function performOperation(operation, a, b) {
    return operation(a, b);
}

function add(x, y) {
    return x + y;
}

function subtract(x, y) {
    return x - y;
}

const resultAddition = performOperation(add, 5, 3);
// Result: 8
const resultSubtraction = performOperation(subtract, 10, 4);
// Result: 6
```

Functions as Results: You can return a function from another function. This allows creating specialized and parameterized functions.

```
function multiplier(factor) {
    return function (number) {
        return number * factor;
    };
}

const double = multiplier(2);
const triple = multiplier(3);

const resultDouble = double(5); // Result: 10
const resultTriple = triple(4); // Result: 12
```

Mapping (Map): The map method of an array applies a function to each element of the array and returns a new array with the results.

```
const numbers = [1, 2, 3, 4];
const duplicates = numbers.map(number => number * 2);
// Result: [2, 4, 6, 8]
```

Filtering (Filter): The filter method of an array creates a new array with elements that pass a specified condition.

```
const numbers = [1, 2, 3, 4, 5];
const odds = numbers.filter(number => number % 2 !== 0);
// Result: [1, 3, 5]
```

Reducing (Reduce): The reduce method of an array accumulates elements by applying a reducing function and returns a single result.

```
const numbers = [1, 2, 3, 4];
const totalSum = numbers.reduce((accumulator, number) =>
accumulator + number, 0); // Result: 10
```

23. Recursion

Recursion is a programming concept where a function calls itself to solve a problem. Essentially, it is a technique in which a function breaks down into smaller, more manageable problems until it reaches a base case that can be solved directly. Recursion is particularly useful for solving problems that have an intrinsic recursive structure.

Here's a simple example of a recursive function that calculates the factorial of a number:

```
function factorial(n) {
    if (n === 0 || n === 1) {
        return 1; // Base case: factorial of 0 and 1 is 1
    } else {
        return n * factorial(n - 1); // Recursive call
    }
}

const result = factorial(5); // Result: 120 (5 * 4 * 3 * 2 * 1)
```

In this example, the factorial function calls itself with a reduced argument in each call until it reaches the base case, where the result is returned. Recursion can also have side effects, such as the use of the call stack, so it's essential to ensure a clear base case and that recursion converges toward it.

Some classic problems solved with recursion include mathematical calculations (factorials, Fibonacci numbers), tree and graph traversal, and divide and conquer in algorithms. However, it's crucial to use recursion carefully, as improper use can lead to performance issues and stack overflow errors.

24. Collections and Generators

Collections and generators are important concepts in JavaScript that help handle and manipulate sets of data efficiently and flexibly. Here's a description of both:

Collections: Collections are data structures that allow storing and organizing multiple elements. In JavaScript, some of the most common collections are:

Arrays: Ordered lists of elements that can contain any data type. Arrays have zero-based indices to access elements.

```
const numbers = [1, 2, 3, 4];
const fruits = ['apple', 'orange', 'banana'];
```

Objects: Collections of key-value pairs where keys are strings (or symbols in ES6) and values can be any data type.

```
const person = { name: 'John', age: 30 };
```

Maps: Structures similar to objects but allow using any value as a key and maintain the insertion order.

```
const map = new Map();
map.set('name', 'Mary');
map.set('age', 25);
```

Sets: Collections of unique and non-duplicated values.

```
const set = new Set();
set.add('red');
set.add('green');
set.add('red'); // Not added, as 'red' is already in the set
```

Generators: Generators are special functions that allow pausing and resuming their execution at a specific point. They enable creating a custom lazy and iterative control flow. They are defined using the function* keyword and use the yield statement to pause the function and return values.

```
function* counter() {
    let num = 0;
    while (true) {
        yield num++;
    }
}

const generator = counter();
console.log(generator.next().value); // Prints: 0
console.log(generator.next().value); // Prints: 1
```

Generators are useful when you need to generate a sequence of values on-demand, rather than generating all values at once. This can be beneficial for handling large data streams or iterating over complex data structures.

Both collections and generators are powerful tools in JavaScript that allow you to handle and manipulate data more efficiently and flexibly.

25. Promises

Promises are a design pattern and feature in JavaScript that allows for a more elegant and efficient handling of asynchronous operations, such as network requests, file read/write, and other tasks that may take time and block execution.

Promises were introduced to address the issue of "callback hell" (excessive callback nesting) and to make asynchronous code more readable and manageable.

Creating a Promise: You can create a promise using its constructor. The constructor takes a function with two parameters, resolve and reject, which are functions to fulfill or reject the promise.

```
const promise = new Promise((resolve, reject) => {
    // Perform an asynchronous operation here
    // If the operation is successful, call resolve(value)
    // If the operation fails, call reject(error)
});
```

Handling Promises: You can chain methods to a promise to handle the result. These methods include then to handle resolution and catch to handle rejection.

```
promise.then(value => {
    // Do something with the resolved value
}).catch(error => {
    // Handle the error if the promise is rejected
});
```

Chaining Promises: You can chain multiple promises using the then method, allowing for sequential asynchronous operations.

```
doSomethingAsync()
    .then(result1 => doAnotherAsyncThing(result1))
    .then(result2 => doMoreAsyncThings(result2))
    .then(finalResult => console.log(finalResult))
    .catch(error => console.error(error));
```

Handling Multiple Promises: The Promise.all method allows you to handle an array of promises and returns a new promise that is resolved when all promises in the array have been resolved.

```
const promise1 = fetchData1();
const promise2 = fetchData2();

Promise.all([promise1, promise2])
    .then(results => {
        const result1 = results[0];
        const result2 = results[1];
        // Do something with the results
    })
    .catch(error => console.error(error));
```

Promises are a powerful and flexible way to handle asynchronous operations in JavaScript. However, as JavaScript evolved, more advanced concepts like async/await were introduced, providing a clearer and more readable syntax for working with asynchronous operations.

26. async/await

async/await is a feature introduced in ECMAScript 2017 (ES8) that simplifies asynchronous programming in JavaScript. It provides a cleaner and more readable syntax for working with promises and asynchronous operations. With async/await, you can write code that looks synchronous but efficiently handles asynchronous operations.

Async Functions: A function declared with the async keyword automatically returns a promise. You can use the await keyword within an async function to wait for the resolution of a promise without blocking the main thread's execution.

```
async function fetchData() {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    return data;
}

fetchData()
    .then(data => console.log(data))
    .catch(error => console.error(error));
```

In this example, fetchData is an asynchronous function that waits for the response of an API request and then awaits the conversion of that response into a JSON object. The code looks like a synchronous flow, even though it's handling asynchronous operations.

Error Handling: You can use the try/catch block with async/await to handle errors more naturally.

```
async function fetchData() {
    try {
        const response = await
fetch('https://api.example.com/data');
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error:', error);
    }
}
```

Chaining Promises with async/await: You can chain async functions using await, making the code more readable and avoiding excessive nesting.

```
async function fetchAndProcessData() {
    const data = await fetchData();
    const result = processData(data);
    return result;
}
```

async/await is a powerful and elegant way to handle asynchronous operations in JavaScript. Although it is newer than promises, it has gained popularity due to its simpler and easier-to-understand syntax. However, it's essential to remember that async/await still relies on underlying promises, so understanding how promises work is crucial to make the most out of async/await.

27. Data Structures

Data structures are organized and efficient ways to store and manage data in a program. These structures allow operations such as insertion, search, modification, and deletion of data to be performed efficiently. In JavaScript, there are several data structures that can be used for different purposes.

Arrays: Arrays are ordered collections of elements. Each element in an array has a numeric index indicating its position. Arrays are used to store lists of elements and are suitable for accessing elements by their index.

```
const fruits = ['apple', 'orange', 'banana'];
```

Objects: Objects are collections of key-value pairs. Keys are strings, and values can be of any type. Objects are useful for representing entities with properties and methods.

```
const person = {
    name: 'John',
    age: 30,
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};
```

Maps: Maps are collections of key-value pairs where keys can be of any type. Unlike objects, maps maintain the insertion order and allow keys of any type.

```
const map = new Map();
map.set('name', 'Maria');
map.set(100, 'one hundred');
```

Sets: Sets are collections of unique and non-duplicated values. They are useful when you need to maintain a list of elements without duplicates.

```
const set = new Set();
set.add('red');
set.add('green');
set.add('red'); // Not added, as 'red' is already in the set
```

Queues and Stacks: Queues and stacks are data structures for managing elements in order. Queues follow the FIFO (First In, First Out) principle, while stacks follow the LIFO (Last In, First Out) principle.

```
// Example of a queue
const queue = [];
queue.push('a');
queue.push('b');
const firstElement = queue.shift(); // 'a'

// Example of a stack
const stack = [];
stack.push('x');
stack.push('y');
const lastElement = stack.pop(); // 'y'
```

These are just some of the basic data structures in JavaScript. The choice of the right data structure depends on the specific requirements of your program and the operations you need to perform on the stored data.

28. Costly Operations and Big O Notation

Costly operations refer to actions in a program that consume a significant amount of resources, such as time and memory. These operations can negatively impact the performance and efficiency of your program. One way to measure and compare the efficiency of different algorithms or operations is through Big O notation.

Big O Notation: Big O notation is a way to express the time or space complexity of an algorithm in terms of the input size. It helps understand how the execution time or memory usage increases as the size of the problem grows. It is used to evaluate the relative performance of different algorithms based on the input.

Some common Big O notations include:

- **O(1) (Constant Time):** The operation does not depend on the size of the input. Example: accessing elements in an array by index.
- **O(log n) (Logarithmic Time):** The operation becomes slower as the size of the input increases, but not proportionally. Example: binary search in a sorted array.
- **O(n) (Linear Time):** The execution time or memory usage increases linearly with the size of the input. Example: traversing all elements in an array.

- **O(n log n) (Linearithmic Time):** Common in efficient sorting algorithms like Merge Sort and Quick Sort.
- **O(n^2), O(n^3), ... (Quadratic, Cubic Time):** The execution time increases quadratically, cubically, etc., in relation to the size of the input. Example: nested loops.
- **O(2^n), O(n!) (Exponential, Factorial Time):** These notations indicate exponential or factorial growth in execution time and are generally considered inefficient for large input sizes.

Choosing efficient algorithms and data structures is crucial to minimize costly operations. In many cases, algorithms with lower Big O notations are sought to improve performance. However, it's essential to consider context and other factors that may influence the choice of the right algorithm.

29. Algorithms

Algorithms are ordered sets of steps or instructions that solve a problem or perform a specific task. In programming, algorithms are fundamental for addressing a wide variety of challenges, from data manipulation to decision-making and optimization. A good algorithm should be efficient, clear, and capable of solving the given problem.

Here are some examples of algorithms and areas where they are applied:

Sorting: Sorting algorithms are used to rearrange elements in a sequence according to certain rules. Examples of sorting algorithms include Quick Sort, Merge Sort, and Bubble Sort.

Searching: Searching algorithms are used to find a specific element within a collection of data. Examples include binary search and linear search.

Graphs and Trees: Algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS) are used to traverse and search in graphs and trees.

Recursion: Recursive algorithms solve problems by breaking them down into smaller subproblems and solving each subproblem recursively. Examples include factorial calculations and Fibonacci numbers.

Dynamic Programming: This approach divides a problem into smaller subproblems and solves each subproblem only once, storing its results to avoid duplicates. It is used in optimization problems, such as efficiently calculating the Fibonacci sequence.

Backtracking: This approach is used to explore all possible solutions to a problem, backtracking and retrying if a partial solution doesn't work. It is applied in problems like Sudoku and the N-Queens problem.

Linear Algebra: Algorithms like Gaussian Elimination are used to solve systems of linear equations.

Cryptography: Cryptographic algorithms are used to secure communication and protect information. Examples include encryption algorithms like AES and RSA.

Optimization: Optimization algorithms seek to find the best solution among multiple options. Examples include genetic algorithms and search optimization algorithms.

Artificial Intelligence: Algorithms like decision trees and machine learning algorithms are used for decision-making and prediction in artificial intelligence systems.

Algorithms are an essential part of programming and are used in various contexts to solve problems effectively and efficiently. Each algorithm has advantages and disadvantages depending on the problem and available resources, so choosing the right approach for each situation is crucial.

30. Inheritance, Polymorphism, and Code Reusability

Inheritance: Inheritance is a fundamental concept in object-oriented programming (OOP) that allows the creation of new classes based on existing classes. A class that inherits from another (called a base class or superclass) acquires its attributes and methods, enabling code reuse and functionality extension.

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

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

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

    speak() {
        console.log(`${this.name} barks.`);
    }
}

const myDog = new Dog('Max', 'Labrador');
myDog.speak(); // Prints: "Max barks."
```

In this example, Dog inherits from Animal and extends its functionality. This promotes code reuse and allows adding dog-specific behaviors in the subclass.

Polymorphism: Polymorphism is the ability of objects from different classes to respond to the same method call differently. It allows treating objects from different classes as if they were objects of the same base class, simplifying design and interaction between objects.

```
class Shape {
    area() {
        return 0;
    }
}

class Square extends Shape {
    constructor(side) {
        super();
        this.side = side;
    }

    area() {
        return this.side * this.side;
    }
}
```