OOJS study
What is OOJS?
Possible ways to implement class
Static class, Properties declaration
ECMAScript6
Difference between let, var & const
JavaScript Classes
Arrow functions
Import, Export, async, await Functions
Extra Points
Difference between == & ===, != & !===

- **What is OOJS?**
  - "OOJS" stands for "Object-Oriented JavaScript." It refers to the use of object-oriented programming (OOP) principles in JavaScript, a language that supports both OOP and other programming paradigms. OOP is a way of organizing and structuring code using objects and classes.
- **Constructor Functions:**
  - In traditional JavaScript, you can create objects using constructor functions. These functions are called with the new keyword to create instances of objects. Properties and methods can be defined inside the constructor function.

Object-Oriented JavaScript (OOJS)

- Object-Oriented JavaScript (OOJS) is a programming paradigm that allows you to model real-world entities and their interactions in your JavaScript code. It brings the principles of object-oriented programming (OOP) to JavaScript, making it easier to manage and organize complex code.
- In OOJS, you work with objects, which are instances of classes. Classes define the structure and behavior of objects, while objects represent specific instances of those classes. JavaScript supports OOP features like

inheritance, encapsulation, and polymorphism, making it a versatile language for building large-scale applications.

Implementing Classes in JavaScript

- There are several ways to implement classes in JavaScript. Here are two common approaches:
- **1. Constructor Functions**
- Constructor functions are a traditional way to create classes in JavaScript. They are functions that initialize new objects with properties and methods. Here's an example:
-
- function Person(name, age) {
- this.name = name;
- this.age = age;
-
- this.sayHello = function() {
- console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
- };
- }
-
- const person1 = new Person("Alice", 30);
- person1.sayHello();
-
-
- In this example, `Person` is a constructor function that creates `Person` objects with `name` and `age` properties and a `sayHello` method.

2. ES6 Classes

-
- With the introduction of ES6 (ECMAScript 2015), JavaScript introduced a more structured way to define classes using the `class` keyword. Here's an example using ES6 classes:
-
-
- class Animal {

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

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

const cat = new Animal("Cat");
cat.speak();
```

ES6 classes offer a cleaner syntax for defining classes and are widely adopted in modern JavaScript development.

3. Static Class Members

Static members belong to the class itself rather than individual instances of the class. In JavaScript, you can define static methods and properties using the `static` keyword within a class. Here's an example:

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

  static subtract(a, b) {
    return a - b;
  }
}

const sum = MathUtils.add(5, 3);
const difference = MathUtils.subtract(10, 4);
```

In this example, `add` and `subtract` are static methods of the `MathUtils` class, which means you can call them without creating an instance of the class.

❖ **Properties Declaration**

❖

❖ In JavaScript, you can declare class properties directly within the constructor or using class fields (available in modern JavaScript). Here's an example using both approaches:

❖

❖

❖ class Car {
❖   constructor(make, model) {
❖   this.make = make; // Property declared within the constructor
❖   this.model = model;
❖   }
❖   year = 2023; // Class field for a default value
❖ }
❖ const myCar = new Car("Toyota", "Camry");
❖ console.log(`My car is a ${myCar.year} ${myCar.make} ${myCar.model}.`);
❖ In this example, `make` and `model` are properties declared within the constructor, while `year` is declared as a class field with a default value.
❖ In JavaScript, class members are not inherently private. However, you can achieve a level of privacy by using certain conventions and techniques. Here are some ways to implement private members in a JavaScript class:

❖

❖ **Private Fields (ES6 and later):**

❖

❖ Starting with ES6, JavaScript introduced a way to declare private class fields using the `#` symbol. Private fields are accessible only within the class where they are declared. Here's an example:

❖

❖ class Person {
❖   #age; // Private field
❖
❖   constructor(name, age) {
❖   this.name = name;
❖   this.#age = age; // Private field initialization
❖   }

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

const person1 = new Person("Alice", 30);
person1.sayHello();
console.log(person1.#age); // This will result in an error, as #age is private


In this example, `#age` is a private field, and it cannot be accessed or modified from outside the `Person` class.
```

ECMAScript 6 (ES6) ➖

ECMAScript 6, often referred to as ES6 or ECMAScript 2015, is a significant update to the JavaScript language specification. It introduced numerous features and improvements to make JavaScript more powerful, expressive, and developer-friendly. This document provides a deep dive into some of the key features of ES6.

1. Block-Scoped Declarations with `let` and `const`

ES6 introduced two new ways to declare variables, `let` and `const`, which have block-level scope. This means they are confined to the nearest enclosing block, making it easier to manage variable scope in your code.

```
if (true) {
  let x = 10; // Block-scoped variable
  const y = 20; // Block-scoped constant
}

console.log(x); // ReferenceError: x is not defined
console.log(y); // ReferenceError: y is not defined
```

2. Arrow Functions

Arrow functions provide a concise syntax for defining functions. They automatically capture the `this` value from their enclosing context, which is helpful when working with functions as arguments or in classes.

```
const add = (a, b) => a + b;

const double = (x) => x * 2;
```

3. Template Literals

Template literals allow you to embed expressions inside strings using backticks (`). This makes string interpolation and multiline strings more convenient.

```
const name = "Alice";

console.log(`Hello, ${name}!`);
```

4. Enhanced Object Literals

ES6 introduced enhancements to object literals, allowing for shorthand property and method definitions, making object creation more concise.

```
const x = 10;

const y = 20;
```

```
const obj = {
  x,         // Shorthand property
  y,         // Shorthand property
  calculate() { // Shorthand method
    return this.x + this.y;
  }
};
```

5. Destructuring Assignment

Destructuring assignment allows you to extract values from objects and arrays into variables, making it more convenient to work with complex data structures.

```
const person = { name: "Bob", age: 30 };
const { name, age } = person;
const numbers = [1, 2, 3, 4, 5];
const [first, second] = numbers;
```

6. Classes

ES6 introduced a class syntax for defining constructor functions and prototypes more explicitly. It provides a more structured way to create and extend classes.

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

```
}
```

7. Modules

ES6 introduced a module system, allowing you to organize your code into separate files and import/export functionality between them, improving code maintainability and reusability.

```
// In module.js

export function add(a, b) {

  return a + b;

}

// In main.js

import { add } from './module';
```

8. Promises

Promises provide a cleaner way to handle asynchronous operations in JavaScript, making it easier to manage callbacks and avoid callback hell.

```
fetch('https://api.example.com/data')

  .then(response => response.json())

  .then(data => console.log(data))

  .catch(error => console.error(error));
```

9. Default Parameters and Rest Parameters

ES6 introduced default parameter values for functions and rest parameters, making it easier to work with variable-length argument lists.

```
function greet(name = "Guest") {

  console.log(`Hello, ${name}!`);

}

function sum(...numbers) {

  return numbers.reduce((acc, val) => acc + val, 0);
```

```
}
```

10. Spread Operator

The spread operator (`...`) allows you to spread elements of an array or properties of an object into another array or object, facilitating data manipulation.

```
const arr1 = [1, 2, 3];

const arr2 = [...arr1, 4, 5]; // Spread operator for arrays

console.log(arr2); // [1, 2, 3, 4, 5]

const obj1 = { x: 1, y: 2 };

const obj2 = { ...obj1, z: 3 }; // Spread operator for objects

console.log(obj2); // { x: 1, y: 2, z: 3 }
```

Async / Await

Promises and `async/await` in JavaScript

Promises and `async/await` are two mechanisms in JavaScript for working with asynchronous operations, making it easier to manage and reason about asynchronous code. Here's an explanation of both concepts:

Promises:

1. What are Promises?

Promises are a way to represent the eventual completion (or failure) of an asynchronous operation. They provide a more structured approach to handling asynchronous tasks than traditional callbacks.

2. Promise States:

Promises have three possible states:

- **Pending**: Initial state, neither fulfilled nor rejected.

- **Fulfilled**: The operation completed successfully, and the result is available.

- **Rejected**: The operation encountered an error, and an error reason is available.

3. Creating a Promise:

You can create a Promise using the `Promise` constructor, which takes a function (often referred to as an executor) with `resolve` and `reject` functions as parameters.

```
const myPromise = new Promise((resolve, reject) => {

    // Asynchronous code here

    if (/* Operation succeeded */) {

      resolve(result); // Resolve with a value

    } else {

      reject(error);  // Reject with an error

    }

  });
```

4. Using `.then()` and `.catch()`:

You can attach callbacks to a Promise using `.then()` to handle successful results and `.catch()` to handle errors.

```
  myPromise
```

```
  .then(result => {
    // Handle success
  })
  .catch(error => {
    // Handle error
  });
```

5. Chaining Promises:

Promises can be chained together using `.then()`, allowing you to perform a series of asynchronous operations in a specific sequence.

```
asyncTask1()
  .then(result1 => asyncTask2(result1))
  .then(result2 => asyncTask3(result2))
  .catch(error => handleError(error));
```

`async/await`:

1. What is `async/await`?

`async/await` is a more recent addition to JavaScript (ES2017) that simplifies working with Promises. It allows you to write asynchronous code in a more synchronous and readable manner.

2. Using `async` Function:

To declare an `async` function, simply prefix the function declaration with the `async` keyword. An `async` function always returns a Promise.

```
async function myAsyncFunction() {
  // Asynchronous code here
```

```
}
```

3. `await` Keyword:

Inside an `async` function, you can use the `await` keyword before a Promise to pause the execution of the function until the Promise is resolved. It allows you to work with Promises as if they were synchronous.

```
async function getData() {
    try {
      const result = await fetch('https://api.example.com/data');
      const data = await result.json();
      // Process data here
    } catch (error) {
      // Handle errors
    }
}
```

4. Error Handling with `try/catch`:

You can use `try/catch` to handle errors within `async` functions, making error handling more straightforward.

5. Benefits of `async/await`:

- **Improved readability:** Asynchronous code looks more like synchronous code.
- Error handling is simplified with `try/catch`.
- Easier to debug and reason about asynchronous flows.

In JavaScript, `==` and `!=` are loose equality operators, while `===` and `!==` are strict equality operators. They are used to compare values, but they behave differently in terms of type coercion. Here's the difference between them:

Loose Equality (`==`) and Inequality (`!=`):

- `==` **(Equality Operator):** This operator compares values for equality after performing type coercion if necessary. It tries to convert the operands to the same type before making the comparison.

  Example:

  5 == "5" // true, because it coerces the string "5" to a number and then compares them

- **`!=` (Inequality Operator):** This operator checks if two values are not equal after performing type coercion if necessary.

  Example:

  5 != "6" // true, because it coerces the string "6" to a number and then compares them

Strict Equality (`===`) and Inequality (`!==`):

- `===` (Strict Equality Operator): This operator checks for equality without performing type coercion. It only returns `true` if both the value and the type of the operands are the same.

  Example:

  5 === 5 // true, because both the value and type are the same

- `!==` (Strict Inequality Operator): This operator checks for inequality without performing type coercion. It returns `true` if either the value or the type of the operands is different.

  Example:

  5 !== "5" // true, because the type is different (number vs. string)

- Use `===` and `!==` when you want to ensure both the value and type are the same for comparison. This is often considered safer and more predictable in JavaScript.

- Use `==` and `!=` when you want to perform type coercion and are more lenient in your comparison. However, be cautious when using them, as they can lead to unexpected results, especially when comparing values of different types.