Agenda

- Prototype Inheritance
- class
- Array
- Spread and Rest Operator
- DOM

Prototype Inheritance

- In JavaScript, prototype inheritance is a mechanism that allows objects to inherit properties and methods from other objects.
- It is the core of JavaScript's object-oriented nature, even with the introduction of the modern class syntax in ES6, which is "syntactical sugar" over the same underlying prototype system.
- Key concepts of prototypal inheritance

1. Prototypes: Every JavaScript object has an internal, hidden property called [[Prototype]], which is a link to another object. That linked object is the first object's prototype.
2. Prototype chain: When you try to access a property or method on an object, JavaScript first looks for it on the object itself. If it isn't found, it searches the object's prototype. This continues up the "prototype chain" until the property is found or the end of the chain is reached. The end of the chain is Object.prototype, which has a prototype of null.
3. Inheritance is delegation: This system is sometimes described as delegation rather than traditional inheritance. A child object does not get its own copy of the parent's properties and methods. Instead, it delegates calls to the properties and methods stored on the parent object (its prototype).

Working with constructor functions

- Before the class keyword was introduced, prototype inheritance was commonly achieved using constructor functions.

```
// A "parent" constructor function
function Person(name) {
  this.name = name;
}

// Add a method to the prototype of Person
Person.prototype.display = function() {
  console.log(`Name - ${this.name}`);
};

// A "child" constructor function
function Employee(id,name, salary) {
  Person.call(this, name); // Call the parent constructor to set 'name'
  this.id = id;
  this.salary = salary;
}

// Set Employee's prototype to inherit from Person's prototype
// This creates the prototype chain: e1 -> Employee.prototype -> Person.prototype
-> Object.prototype
Object.setPrototypeOf(Employee.prototype, Person.prototype);

// Create a new instance
const e1 = new Employee(1, "Anil", 10000);

// The Employee object does not have its own `display` method, but it can call the
one from its prototype
e1.display(); // Output: Name - Anil
```

Key Features of Prototype

1. Underlying mechanism: Even when using the modern class syntax, JavaScript is fundamentally a prototype-based language.
2. Code reuse: Prototypes are a powerful way to share methods and properties among objects without needing to create separate copies for each instance.
3. Dynamic and flexible: The prototype chain can be modified at runtime, allowing for dynamic behavior. However, this feature should be used with caution, as it can lead to unexpected side effects.

class

- Introduced in ECMAScript 2015 (ES6), the class keyword is a syntactic sugar over JavaScript's existing prototype-based inheritance.
- While classes do not change the fundamental nature of JavaScript objects, they provide a much cleaner and more familiar syntax for creating object blueprints, inheriting from other classes, and using object-oriented programming (OOP) principles.
- Key features of JavaScript classes

1. constructor() method: A special method called automatically when a new object is created using the new keyword. It is used to initialize the object's properties.
2. new keyword: Creates a new instance of a class, triggering the constructor() method. Attempting to call a class without new will throw an error.
3. Methods: Functions defined inside the class body, which provide behavior for objects created from the class.
4. Inheritance: Classes support inheritance using the extends keyword. This allows a new class (the child) to inherit the properties and methods of an existing class (the parent).

- The super() method is used inside a child class's constructor to call the parent's constructor.
- Static methods: Defined with the static keyword, these methods are called on the class itself, not on an instance of the class. They are useful for utility functions.

```
class Utils {
  static logMessage(message) {
    console.log(message);
  }
}
```

```
Utils.logMessage("Hello from a static method!");
```

- Private fields: Since ECMAScript 2022, private fields can be declared using a hash (#) prefix. This makes a property or method only accessible from inside the class itself, ensuring better encapsulation.

```
class BankAccount {
  #balance; // This is a private field

  constructor(initialBalance) {
    this.#balance = initialBalance;
  }

  deposit(amount) {
    this.#balance += amount;
  }

  getBalance() {
    return this.#balance;
  }
}
```

Array

- An array in JavaScript is a special variable that can hold multiple values.
- Arrays allow you to store, access, and manipulate lists of data efficiently.
- array declaration can be done in two ways:

1. Using Square Brackets (Recommended)
2. Using the Array Constructor

```
let fruits = ["Apple", "Banana", "Mango"];
let fruits = new Array("Apple", "Banana", "Mango");
```

- ☐ are preferred because they are more concise.
- Array elements are accessed using index numbers, starting from 0.

The Spread Operator (...)

- The spread operator unpacks elements from arrays, objects, or other iterables.
- It is useful for creating new copies of data structures without changing the original.

- With Arrays

```
// create a new array with a copy of an existing one.
const fruits = ['apple', 'banana', 'cherry'];
const vegetables = ['carrot', 'broccoli'];

// Combine arrays
const produce = [...fruits, ...vegetables];
console.log(produce); // Output: ["apple", "banana", "cherry", "carrot",
"broccoli"]

// Copy an array
const fruitsCopy = [...fruits];
fruitsCopy.push('grape');
console.log(fruits); // Output: ["apple", "banana", "cherry"]
console.log(fruitsCopy); // Output: ["apple", "banana", "cherry", "grape"]
```

- with Object

- The spread operator copies properties from one object to another.
- When merging objects, properties from the object that is spread last will overwrite earlier properties if they have the same key.

```
const person = { name: 'Alice', age: 30 };
const location = { city: 'New York', country: 'USA' };

// Merge objects
const profile = { ...person, ...location };
console.log(profile); // Output: { name: "Alice", age: 30, city: "New York",
country: "USA" }

// Update or override properties
const updatedProfile = { ...profile, age: 31 };
console.log(updatedProfile); // Output: { name: "Alice", age: 31, city: "New
York", country: "USA" }
```

- In function calls

```
 function add(n1, n2) {
    console.log(n1 + n2)
}

const arr = [10, 20]
add(...arr)
```

The Rest Operator (...)

The rest operator is used in function definitions to collect an indefinite number of arguments into a single array.

- It can also be used in destructuring assignments.

- In function parameters

- The rest parameter allows you to create functions that accept any number of arguments.
- The rest parameter must be the last parameter in the function definition.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4, 5)); // Output: 15
console.log(sum(10, 20)); // Output: 30
```

- In array destructuring

- When destructuring an array, the rest operator can collect all remaining elements into a new array.

```
const [first, second, ...restOfTheNumbers] = [10, 20, 30, 40, 50];

console.log(first); // Output: 10
console.log(second); // Output: 20
console.log(restOfTheNumbers); // Output: [30, 40, 50]
```

- In object destructuring

- In object destructuring, the rest operator gathers the remaining properties into a new object.

```
const person = {
  name: 'Bob',
  age: 42,
  city: 'Chicago',
  job: 'Programmer'
};

const { name, ...otherInfo } = person;

console.log(name); // Output: "Bob"
console.log(otherInfo); // Output: { age: 42, city: "Chicago", job: "Programmer" }
```

Window Object

- It represents an open window in the browser. It is browser's object(not JS object) which is created automatically
- It is a global object with lot of properties and methods

DOM (Document Object Model)

- When a webpage is loaded the browser creates DOM of the page
- It is the data representation of the objects that comprise the structure and content of a document on the web.
- DOM represents an HTML document in memory
- The DOM represents the document as nodes and objects so that programming languages can interact with the page.
- In both cases, it is the same document but the Document Object Model (DOM) representation allows it to be manipulated.
- console.dir(document) will display all the properties and methods from the document
- It is a tree like structure (window-> document -> html -> and all its sub nodes)

DOM Manipulation

Selection

1. Selecting with id
   - document.getElementById("myId") (#)
2. Selecting with class
   - document.getElementsByClassName("myClass") (.)
   - returns HTML collection an array of objects
3. Selecting with tag
   - document.getElementsByTagName("tagName")
   - returns HTML collection an array of objects
4. Query Selector
   - used to select the id,name and class automatically
   - document.querySelector("myId/myClass/tag")
     - returns first element
   - document.querySelectorAll("myId/myClass/tag")
     - returns a NodeList

Properties

1. tagName
   - returns tag for element nodes
2. innerText
   - returns text content of the element and all its children
   - It represents only the text part
3. innerHTML
   - returns the plain text or html contents in the elements
   - It represents text as well as any element/tag inside sit
4. textContent
   - returns textual content even for hidden elements

Attribute

1. getAttribite("attr")
   - to get the attribute value
2. setAttribute("attr",value)

  - to set the attribute value

Style

node.style - It helps to style the elements i.e apply css on it

Insert elements

1. node.append(e)
   - add at the end of the node (inside)
2. node.prepend(e)
   - add at the start of the node (inside)
3. node.before(e)

- add before the node (outside)

4. node.after(e)

- add after the node (outside)

Delete elements

- node.delete(e)
  - Used to delete the node