ES6 Classes

Classes in JS

Classes in JavaScript provide a convenient and object-oriented way to define and create objects with similar behavior and properties.

Why use classes?

- While constructor functions have been used to create objects in JavaScript for a long time, they're not as intuitive or easy to read as classes. Classes also provide a more concise and consistent syntax for working with inheritance.
- Classes are just an alternative way of creating objects and dealing with object-oriented programming in JavaScript - but one that many developers find more convenient.
- However, the existence of constructor functions in JavaScript prior to the introduction of classes makes them still relevant. They are compatible with older codebases and are still widely used in various scenarios.

Class Declaration:

A class declaration defines a new class using the `` `class` `` keyword. It consists of a class name and body containing constructor and method definitions.

Syntax:

```
class ClassName {
    constructor(parameters) {
      // Initialize object properties
    }
    method1() {
      // Method definition
    }
```

```
    method2() {
      // Method definition
    }
    // Additional methods...
  }
```

Insights into the syntax of class declaration in JavaScript.

- **Constructor:** The constructor method is a special method used for initializing class instances. It is invoked automatically when a new object is created from the class. It is defined using the `` `constructor` `` keyword.
- **Methods:** Methods are functions defined within a class. They are shared by all instances of the class and can be accessed through the instance objects. Methods do not require the `` `function` `` keyword.

Class Expression:

Similar to function expressions, classes can also be defined using class expressions. Class expressions can be named or unnamed.

Syntax:

1. **Unnamed class expression**

```
const ClassName = class {
    constructor(parameters) {
      // Initialize object properties
    }
    method1() {
      // Method definition
    }
    method2() {
      // Method definition
    }
    // Additional methods...
```

```
};
```

2. Named class expression

```
const ClassName = class ClassName {
  constructor(parameters) {
    // Initialize object properties
  }
  method1() {
    // Method definition
  }
  method2() {
    // Method definition
  }
  // Additional methods...
};
```

Hoisting and Execution of Strict Mode:

- Unlike function declarations, class declarations are not hoisted. They need to be declared before they are used. Class expressions, on the other hand, behave like variables and can be hoisted.
- When using classes, the **`use strict`** directive is implicitly applied, ensuring a stricter mode of JavaScript execution within the class.

Encapsulation

- Encapsulation is a fundamental principle of object-oriented programming that promotes data hiding and ensures that the internal implementation details of an object are inaccessible from outside the object.
- JavaScript, being a flexible and dynamic language, traditionally lacks built-in support for private properties and methods.

- However, with the introduction of new versions of JavaScript, specifically ECMAScript 2019 (ES10) and later, private properties and methods can be achieved using the `#` symbol.

Private Properties with the `#` Symbol:

- Encapsulation using private properties and methods with the `#` symbol provides a more robust and secure approach to object-oriented programming in JavaScript, enabling better data protection, code organization, and controlled access to internal functionality.
- The `#` symbol before a property name in a class or object denotes that the property is private and cannot be accessed or modified from outside the class or object. It restricts direct access to the property, providing encapsulation.

Example:

```
class MyClass {
    #privateProperty = 42;
    #privateMethod() {
      return 'This is a private method';
    }
    publicMethod() {
      console.log(this.#privateProperty); // Accessing private
property
      console.log(this.#privateMethod()); // Invoking private method
    }
  }
  const myObj = new MyClass();
  myObj.publicMethod(); // Output: 42, "This is a private method"
```

```
console.log(myObj.#privateProperty); // Error: SyntaxError

console.log(myObj.#privateMethod()); // Error: SyntaxError
```

In the example above, `#privateProperty` and `#privateMethod()` are private within the `MyClass` class. They can only be accessed and invoked from within the class itself, not from outside.

- It's important to note that private properties and methods are unique to each instance of the class. Each instance has its own private property values and private methods.
- The use of private properties and methods helps in achieving better encapsulation by hiding implementation details and preventing accidental modification or access from external code.

Benefits of Encapsulation and Private Properties:

1. **Information Hiding:** Private properties and methods encapsulate implementation details, hiding them from the outside world. This protects sensitive data and prevents direct manipulation, reducing the risk of unintended side effects.
2. **Controlled Access:** Encapsulation allows developers to control access to internal state and behavior. Only the necessary methods and properties are exposed as part of the public interface, ensuring more predictable and maintainable code.
3. **Code Organization**: Encapsulation promotes modular and organized code. Private properties and methods keep implementation-specific details hidden, allowing developers to focus on the public interface and higher-level functionality.

Inheritance

- Inheritance is a key concept in object-oriented programming that allows objects to inherit properties and methods from a parent class. It enables code

reuse, promotes code organization, and facilitates the creation of specialized subclasses.

- In other words, Inheritance is the practice of creating new objects based on existing ones, inheriting properties and methods from their parent objects.
- In JavaScript, inheritance can be achieved using the `super` keyword, the `extends` keyword, and function overriding.

The `extends` and `super` Keywords:

- The `extends` keyword is used to create a child class that inherits from a parent class in JavaScript. It establishes a hierarchical relationship between classes, allowing the child class to inherit the properties and methods of the parent class.
- The `super` keyword is primarily used within the constructor of a child class to call the constructor of the parent class. It enables the child class to access and initialize properties defined in the parent class.
- However, it's worth noting that the `super` keyword can also be used to call other methods defined in the parent class, not just the constructor. This allows the child class to leverage and extend the functionality provided by the parent class.

Example:

```
class Animal {

  constructor(name) {

    this.name = name;

  }

}

class Dog extends Animal {
```

```
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
}
const myDog = new Dog('Max', 'Labrador');
console.log(myDog);
```

- In the example above, the `Animal` class serves as the parent class, and the `Dog` class extends it using the **`extends`** keyword.
- The **`super`** keyword is used in the `Dog` class's constructor to call the `Animal` class's constructor and pass the `name` parameter.
- This allows the `Dog` class to inherit the `name` property from the `Animal` class.
- The final line outputs the `myDog` object, allowing you to see the values of the `name` and `breed` properties.

Function Overriding

- Function overriding is the ability of a child class to provide a different implementation for a method that is already defined in its parent class.
- When a child class overrides a method, the implementation in the child class is executed instead of the parent class's implementation.

Example:

```
class Shape {
  draw() {
    console.log('Drawing a shape.');
```

```
  }
}
class Circle extends Shape {
  draw() {
    console.log('Drawing a circle.');
  }
}
const myShape = new Shape();
myShape.draw(); // Output: Drawing a shape.
const myCircle = new Circle();
myCircle.draw(); // Output: Drawing a circle.
```

In the example above, the `Circle` class extends the `Shape` class and overrides the `draw()` method. When `draw()` is called on a `Circle` instance, the overridden implementation in the `Circle` class is executed instead of the one in the `Shape` class.

Inheritance in Constructor Functions

- In JavaScript, inheritance can also be achieved using constructor functions and the prototype chain. Constructor functions serve as blueprints for creating objects and can be used to establish inheritance relationships between classes.
- To implement inheritance in constructor functions, the `**prototype**` property is used to define shared properties and methods that will be inherited by all instances of the class. The `**prototype**` object of the parent constructor function is assigned to the `**prototype**` property of the child constructor function, creating a prototype chain.

Example:

```
function Vehicle(make) {
  this.make = make;
}

Vehicle.prototype.start = function() {
  console.log(`Starting the ${this.make} vehicle.`);
};

function Car(make, model) {
  Vehicle.call(this, make);
  this.model = model;
}
Car.prototype = Object.create(Vehicle.prototype);
const myCar = new Car('Toyota', 'Camry');
myCar.start(); // Output: Starting the Toyota vehicle.
```

- In this example, we have a `Vehicle` constructor function that sets the `make` property and defines a `start` method on its prototype.
- The `Car` constructor function extends `Vehicle` by calling `Vehicle.call(this, make)` in its own constructor. This ensures that the `make` property is properly set for each `Car` instance.
- To establish the inheritance relationship, we use `Object.create(Vehicle.prototype)` to create a new object with `Vehicle.prototype` as its prototype. This new object is assigned to `Car.prototype`, allowing `Car` instances to inherit properties and methods from `Vehicle`.
- Finally, we create an instance of `Car` named `myCar` with the make 'Toyota' and model 'Camry'. We can call the `start` method on `myCar`, which invokes the inherited `start` method from `Vehicle`, resulting in the output "Starting the Toyota vehicle."

Static keyword in JS

- The `static` keyword is used to define static properties and methods within a class. Static members are associated with the class itself rather than its instances.
- They are accessed and invoked directly on the class, without the need to create an object. They are accessed using the class name followed by the dot notation.
- Here's an example that demonstrates the usage of static properties and the `static` keyword:

```
class Circle {
  static PI = 3.14159;

  constructor(radius) {
    this.radius = radius;
  }

  calculateArea() {
    return Circle.PI * this.radius * this.radius;
  }

  static formatNumber(number) {
    return number.toFixed(2);
  }
}

const myCircle = new Circle(5);

console.log(myCircle.calculateArea()); // Output: 78.53975
console.log(Circle.formatNumber(5)); // Output: 5.00
console.log(myCircle.PI); // Output: undefined
console.log(Circle.PI); // Output: 3.14159
```

- In the example, we have a `Circle` class with a static property `PI` representing the mathematical constant π.
- The `calculateArea` method uses this static property to calculate the area of the circle.
- The `formatNumber` method is also declared as static and can be directly accessed on the class. It formats a given number to two decimal places.
- When we create an instance of the `Circle` class (`myCircle`) and call the `calculateArea` method, it correctly calculates the area using the static property `PI`.
- However, when we try to access the static property `PI` through the instance `myCircle`, it returns undefined. Static properties are not accessible through instances; they are only accessible through the class itself.
- To access the static property `PI`, we use the class name `Circle` followed by the dot notation (`Circle.PI`). This returns the value of the static property, which is 3.14159.

Getter and Setter

- Getters and Setters are special methods that allow controlled access to object properties. They provide a way to define custom behavior when getting or setting the values of properties.
- Getters are used to retrieve the value of a property, while setters are used to set the value of a property.
- They provide control over property access and manipulation, allowing for more robust and controlled data handling within objects.

Key Points about Getters and Setters:

1. **Getter:**

a. A getter is a method that is used to retrieve the value of a property.

b. It is defined using the `get` keyword followed by the method name.

c. Getters do not accept any parameters.

d. Getters are accessed like regular properties, without using parentheses.

e. Getters can be useful for performing computations or returning modified values based on existing properties.

2. **Setter:**

a. A setter is a method that is used to set the value of a property.

b. It is defined using the `set` keyword followed by the method name.

c. Setters accept a single parameter, representing the value to be set.

d. Setters are accessed like regular properties, using the assignment operator (=).

e. Setters can be useful for performing validation or implementing side effects when assigning values to properties.

3. **Syntax:**

- **Getter syntax:**

```
`get propertyName() { /* code to return value */ }`
```

- **Setter syntax:**

```
`set propertyName(value) { /* code to set value */ }`
```

4. **Usage and Benefits:**

a. Getters and setters provide an interface to access and modify object properties in a controlled manner.

b. They allow encapsulation of data, enabling validation or manipulation of property values before setting or retrieving them.

c. Getters and setters can be used to implement computed properties, where the value is derived from other properties or calculations.

d. They provide a way to handle edge cases and ensure consistency in data manipulation within an object.

Example:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  get uppercaseName() {
    return this.name.toUpperCase();
  }
  set setName(newName) {
    this.name = newName;
  }
}
const alice = new Person("Alice");
console.log(alice.uppercaseName); // logs "ALICE"
alice.setName = "Bob"; // sets the name property to "Bob"
```

```
console.log(alice.uppercaseName); // logs "BOB"
```

- The example defines a `Person` class with a `name` property. It has a getter method `uppercaseName` that returns the name property in uppercase.
- The setter method `setName` allows updating the `name` property. An instance of `Person` named `alice` is created with the name "Alice".
- By accessing `alice.uppercaseName`, it returns the uppercase name "ALICE". Setting `alice.setName` to "Bob" updates the name property.
- Accessing `alice.uppercaseName` again returns the updated uppercase name "BOB".

Built-in Objects in JS

Built-in objects in JavaScript provide a set of predefined functionalities and properties that can be used in your code without the need for additional declarations or external libraries. These objects offer a wide range of capabilities to work with various data types, perform mathematical operations, manipulate dates, and more. Let's explore a few important built-in objects:

1. **Date Object:**
    a. The **`Date`** object is used for working with dates and times in JavaScript.
    b. It provides methods to create, manipulate, and format dates.
    c. You can create a new instance of **`Date`** using the **`new Date()`** constructor.

    ❖ **Date Formats:**

    ○ The **`Date`** object supports various date formats, such as:

- ISO 8601 format: "YYYY-MM-DDTHH:mm:ss.sssZ"
- Short date format: "MM/DD/YYYY"
- Long date format: "Month DD, YYYY"
- Custom formats using the **`toLocaleDateString()`** method.

❖ **Common Methods:**

- **`getFullYear()`:** Returns the year (4 digits) of the date.
- **`getMonth()`:** Returns the month (0-11) of the date.
- **`getDate()`:** Returns the day of the month (1-31).
- **`getDay()`**: Returns the day of the week (0-6), starting from Sunday.
- **`getHours()`, `getMinutes()`, `getSeconds()`:** Returns the respective time components.
- **`toString()`:** Returns a string representation of the date.

Example:

```
const currentDate = new Date();
console.log(currentDate.toString()); // Output: Wed Jun 14 2023
12:34:56 GMT+0530 (India Standard Time)
```

2. **Math Object:**
   a. The **`Math`** object provides mathematical operations and constants.
   b. It contains various methods for common mathematical calculations, such as **`Math.sqrt()`, `Math.pow()`, `Math.abs()`, `Math.sin()`, `Math.cos()`**, etc.
   c. Constants like **`Math.PI`** and **`Math.E`** are also available.

Example:

```
  console.log(Math.sqrt(16)); // Output: 4
  console.log(Math.PI); // Output: 3.141592653589793
```

3. Random Number Generation:

a. JavaScript provides the Math.random() method to generate random numbers between 0 and 1 (exclusive).
b. You can multiply it by a desired range and apply appropriate rounding or truncation to get random integers within a specific range.

Example:

```
const randomNumber = Math.floor(Math.random() * 10); // Generates a random integer between 0 and 9
console.log(randomNumber);
```

JSON

- JSON (JavaScript Object Notation) is a lightweight data interchange format widely used in web development for data transmission between client and server, storing configuration settings, and exchanging data with APIs.
- It provides a simple and readable format for representing structured data.
- Let's explore JSON and its related methods in more detail:

1. JSON Format:

- JSON represents data using a combination of object notation and key-value pairs. It resembles the syntax of JavaScript objects and arrays.
- Object Notation:
    - Data is structured as a collection of key-value pairs, enclosed in curly braces {}. Key-value pairs are separated by a colon (:), and each pair is separated by a comma (,).
    - JSON values can be strings, numbers, booleans, arrays, objects, or null.

Example:

```
const person = {

  "name": "Chris Evans",

  "age": 30,

  "isStudent": false

};
```

- Array Notation:
    - Data can also be represented as an ordered list using square brackets `[]`.

Example:

```
const people = [

  {

    "name": "Peter Parker",

    "age": 30,

    "isStudent": false

  },

  {

    "name": "Jane Smith",

    "age": 25,

    "isStudent": true

  }

];
```

2. **File Extension:**

- JSON files typically have the `.json` extension.

- They are commonly used for data storage and exchange between applications.

In JavaScript, JSON is a built-in object that offers methods for parsing and stringifying JSON data.

1. **JSON.parse():**

- The `**JSON.parse()**` method converts a JSON string into a JavaScript object.
- It takes a valid JSON string as input and returns a corresponding JavaScript object.

Example:

```
const jsonString = '{"name":"John","age":30,"city":"New York"}';

const person = JSON.parse(jsonString);

console.log(person.name); // Output: "John"
```

2. **JSON.stringify():**

- The `**JSON.stringify()**` method converts a JavaScript object into a JSON string.
- It takes an object as input and returns a string representation of the object in JSON format.

Example:

```
const person = {

  name: "John",

  age: 30,

  city: "New York"

};
```

```
  const jsonString = JSON.stringify(person);

  console.log(jsonString); // Output:
'{"name":"John","age":30,"city":"New York"}'
```

Shallow and Deep Copy in JS

- In JavaScript, when working with objects and arrays, it's important to understand the concepts of shallow copy and deep copy.
- It's important to choose the appropriate copy method based on the specific requirements of your use case.
- Shallow copy is suitable when you want to create a new object or array with the same references to nested objects or arrays.
- Deep copy, while not always straightforward, is necessary when you need to create a completely independent copy that does not affect the original object or array.

Let's explore what they mean and how they can be achieved using different methods:

Shallow Copy

A shallow copy creates a new object or array and copies the references of the original values. This means that if the original object contains nested objects or arrays, the shallow copy will still reference the same nested objects or arrays.

Methods for Shallow Copy:

1. **Spread Operator:**

- The spread operator (`...`) can be used to create a shallow copy of an array or object.
- Shallow copy only affects the first layer of the object or array. If there are nested objects or arrays, they will still be referenced, and modifying them in the shallow copy will affect the original object or array.

Example:

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

const shallowCopyArray = [...originalArray];

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

const shallowCopyObject = { ...originalObject };
```

2. **Object.assign():**

- The `Object.assign()` method can be used to create a shallow copy of an object.

Example:

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

const shallowCopyObject = Object.assign({}, originalObject);
```

Deep Copy

A deep copy creates a completely new object or array with its own set of values. Any changes made to the deep copy will not affect the original object or array, and vice versa.

Methods for Deep Copy:

1. **JSON.stringify() and JSON.parse():**

- The combination of `JSON.stringify()` and `JSON.parse()` can be used to create a deep copy of an object or array.

Example:

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

const deepCopyObject = JSON.parse(JSON.stringify(originalObject));
```

Note: Although `JSON.stringify()` and `JSON.parse()` are commonly used for deep copying, they have limitations. They cannot handle functions, undefined values, or circular references. Additionally, if the object or array contains complex data types

like dates or regular expressions, they may be converted to strings during the stringification process and lose their original type.

Summarizing it

Let's summarize what we have learned in this Lecture:

- ES6 classes: Introduced for class-based object creation and inheritance.
- Encapsulation: Concept of bundling data and methods within a class.
- Inheritance: Mechanism for inheriting properties and methods from parent classes.
- Static keyword: Used to define static properties and methods in classes.
- Getter and Setter: Special methods for property retrieval and assignment.
- Built-in objects: JavaScript provides objects like Date, Math, and Array for common operations.
- JSON: Lightweight format for data interchange and storage.
- Deep and Shallow Copy: Techniques for creating copies of objects and arrays with varying levels of independence.

References

- Standard built-in objects: **Link**
- Working with JSON: **Link**