# OOP in JS

## Objects

Brief Recap:

- In Lecture 3, we briefly introduced objects and discussed object creation using literals and accessing objects using dot and bracket notation.
- In this session, we will explore objects in more detail, focusing on their properties, methods, and additional concepts.

### Introduction to Objects:

- In JavaScript, objects are composite data types used to represent real-world entities or concepts.
- Objects are containers that store related data and functions together as properties and methods.
- Objects can be instantiated using two different approaches: object literals or constructor functions. In the following sections of the notes, we will delve deeper into the concept of constructor functions and discuss their usage in creating and initializing objects.

#### Object Creation Using Literals:

- Object literals provide a convenient way to create and initialize objects.
- Object literals consist of key-value pairs enclosed in curly braces {}.
- Each key-value pair represents a property, where the key is the property name and the value is the property value.
- Properties can hold various data types, including strings, numbers, booleans, arrays, or even other objects.

Example:

```
// Creating an object using object literal
var person = {
```

```
    name: "Peter",
    age: 25,
    address: "123 Street",
    hobbies: ["reading", "playing guitar"]
  };
```

- In ES5 object literals, methods are defined similarly to other properties. The function expressions are used to assign values to these properties.

For instance:

```
  var obj = {
    calculateArea: function (radius) {
        return Math.PI * radius * radius;
    },
    displayMessage: function (name) {
        console.log("Hello, " + name + "!");
    }
};
```

In the code above, the `obj` object has two methods: `calculateArea` and `displayMessage`. The `calculateArea` method calculates the area of a circle based on the given radius, while the `displayMessage` method logs a personalized greeting to the console.

- In ES6, a new syntax called method definitions was introduced to make method creation more concise:

```
const obj = {
    calculateArea(radius) {
        return Math.PI * radius * radius;
    },
    displayMessage(name) {
        console.log("Hello, " + name + "!");
    }
```

```
};
```

With the ES6 method definitions, we can directly define methods within the object literal without using the `function` keyword. The functionality remains the same as in the previous example.

## Accessing Objects Using Dot Notation:

- Once an object is created, we can access its properties using dot notation.
- Dot notation involves using the dot operator **(.)** followed by the property name to access the corresponding value.

Example:

```
// Accessing properties using dot notation
console.log(person.name);        // Output: Peter
console.log(person.age);         // Output: 25
console.log(person.address);     // Output: 123 Street
console.log(person.hobbies[0]);  // Output: reading
```

## Accessing Objects Using Bracket Notation:

- In addition to dot notation, JavaScript also allows accessing object properties using bracket notation.
- Bracket notation involves using square brackets **[]** with the property name inside.
- Bracket notation is useful when the property name contains special characters, spaces, or is determined dynamically at runtime.

Example:

```
// Accessing properties using bracket notation
console.log(person["name"]);        // Output: Peter
console.log(person["age"]);         // Output: 25
console.log(person["address"]);     // Output: 123 Street
console.log(person["hobbies"][0]);  // Output: reading
```

**Comparison between Dot Notation and Bracket Notation:**

- Both dot notation and bracket notation achieve the same result of accessing object properties.
- Dot notation is commonly used when the property name is known and valid as an identifier.
- Bracket notation is more versatile as it allows using variables or property names with special characters or spaces.
- Bracket notation is also used when the property name is dynamically determined.

It's important to note that dot notation and bracket notation are interchangeable, but bracket notation provides more flexibility in certain situations.

# `this` keyword

- In lecture 5, we discussed the keyword `'this'` in JavaScript and its general purpose. Now, let's delve deeper into the concept of `'this'` specifically in the context of JavaScript objects.
- In this section, we will provide a comprehensive explanation of `'this'` in object-oriented programming, highlighting its significance and providing clear examples.

**I. Recap: Understanding `'this'` in JavaScript**

- Previously, we learned that `'this'` refers to the current execution context in JavaScript.
- It allows us to access and manipulate object properties and methods within the scope of an object.

**II. `'this'` in Object Methods:**

- When used within an object method, `'this'` refers to the object itself.
- This allows seamless access to other properties and methods of the same object.

Let's illustrate this concept with an example:

```
const person = {
  name: 'Peter',
  age: 30,
  greet: function() {
    console.log('Hello, my name is ' + this.name + ' and I am ' +
this.age + ' years old.');
  }
};

person.greet(); // Output: Hello, my name is Peter and I am 30 years
old.
```

Here, `'this'` within the `'greet'` function allows for dynamic access to object properties, ensuring that the function can be applied to different objects while maintaining the correct context.

## Constructor Functions

- When creating objects in JavaScript, object literals serve as a convenient and straightforward approach.
- However, there are scenarios where object literals may not be suitable or efficient for object creation. This leads us to the introduction of constructor functions, which provide a powerful alternative for creating and initializing objects.

**So, why do we require the new concept of constructor functions?**

- Object literals have their limitations when creating multiple objects with similar properties and behaviors. Each object created with an object literal has its own properties and methods, resulting in duplicated code and inefficiency.
- Constructor functions address this limitation by providing a blueprint for creating multiple objects with shared properties and methods.

- By defining a constructor function, we can instantiate new objects that inherit properties and methods from the constructor's prototype.
- This approach promotes code reusability, reduces redundancy, and enables efficient object creation.

### Syntax:

1. Constructor functions are defined using function declarations or expressions.
2. By convention, the function name starts with a capital letter to indicate that it is a constructor.

Example:

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

## The Role of `'this'` Keyword in Constructor Functions:

**A. Context and Ownership:**

1. Inside the constructor function, the `'this'` keyword refers to the object being created.
2. `'this'` allows access to and assignment of object properties within the function.

**B. Assigning Properties:**

1. Using `'this.propertyName'`, properties can be assigned to the object being created.
2. In the example above, `'this.name'` and `'this.age'` assign the `'name'` and `'age'` properties to the created object.

**C. Defining Shared Methods:**

1. Constructor functions can define methods that will be shared among all instances of the object.
2. The `'greet'` method in the example is assigned to `'this.greet'`, making it accessible to all instances.

### Invoking Constructor Functions with the 'new' Keyword:

1. To create an object instance, the constructor function is invoked using the **`'new'`** keyword.
2. The **`'new'`** keyword creates a new object and binds '`this`' to that object inside the constructor function.

Example:

```
const person1 = new Person('Tommy', 30);
const person2 = new Person('Arthur', 25);

person1.greet(); // Output: Hello, my name is Tommy and I am 30
years old.
person2.greet(); // Output: Hello, my name is Arthur and I am 25
years old.
```

Understanding constructor functions allows for efficient code organization and the creation of reusable object templates.

# Prototype

### Object Prototype in JavaScript using Constructor Functions

- In JavaScript, constructor functions are used to create objects. When you define a constructor function, you can add properties and methods to its prototype object. Any objects created with that constructor will then inherit those properties and methods.

- The `**prototype**` property of a constructor function is not the same thing as an instance's prototype - rather, it's used to add properties and methods that will be inherited by instances created from the constructor.

## Accessing an Object's Prototype

The property of an object that points to its prototype is not called "prototype". Its name is not standard across browsers; however,proto can be used in practice for most browsers. The recommended way to access an object's prototype is through the `**Object.getPrototypeOf()**` method.

Example:

```
function Person(name) {
  this.name = name;
}
var bob = new Person('Bob');
console.log(Object.getPrototypeOf(bob)); // returns Person {}
```

## Prototypal Inheritance

Prototypal inheritance refers to how objects inherit from their prototypes. When you look up a property on an object (e.g., `bob.name`), if it doesn't exist on the object itself, JavaScript looks at its prototype chain until it finds what it needs.

Example:

```
function Animal() {}
Animal.prototype.move = function() { console.log('moving') };

function Dog() {}
Dog.prototype.bark = function() { console.log('woof!') };
Dog.prototype.__proto__ = Animal.prototype;

const fido = new Dog();
fido.move(); // logs 'moving'
fido.bark(); // logs 'woof!'
```

- Here we have set up prototypal inheritance between our `Animal` and `Dog` constructors so that dogs inherit both their own unique behaviors (`bark`) and those shared with animals (`move`).

- This works because when we call `.move()` on our instance of `Dog`, which doesn't actually have its own `.move()` method defined, JavaScript follows the chain up through `Dog`'s parent (`Animal`) until it finds one that does.

### Prototype in Array

Arrays in JavaScript are also objects, and therefore have a prototype. The **`Array.prototype`** object contains properties and methods that are available to all arrays created from the **`Array`** constructor function.

Example:

```
const myArr = [1, 2, 3];
console.log(myArr.map(x => x * 2)); // returns [2,4,6]
```

Here we've used the **`.map()`** method of our array instance to return a new array with each element doubled. This method is actually defined on the **`Array.prototype`**, so when we call it on our specific instance (`myArr`), JS looks up its prototype chain to find where `.map()` is defined (in this case: `Array.prototype`).

# Object.create

- In JavaScript, the `Object.create()` method is used to link objects together and establish prototype chains. It allows us to create a new object with a specified prototype object.
- The `Object.create()` method creates a new object and sets the specified object as its prototype.
- It provides a way to create objects that inherit properties and methods from a prototype object.

Example:

```
const personPrototype = {
    greet: function() {
      console.log(`Hello, my name is ${this.name}.`);
    }
  };
  const john = Object.create(personPrototype);
  john.name = "John";
  john.greet(); // Output: Hello, my name is John.
```

## call/apply/bind methods

- In JavaScript, the **`call()`**, **`apply()`**, and **`bind()`** methods are used to manipulate the execution context of functions and explicitly set the value of `this`. These methods provide flexibility and control over how functions are invoked.

1. **The `call()` Method:**

- The **`call()`** method is used to invoke a function and explicitly set the value of **`this`**.
- It accepts arguments individually, separated by commas.

Example:

```
  function greet(name) {
    console.log(`Hello, ${name}! I'm ${this.title}.`);
  }
  const person = {
    title: "Mr."
  };
  greet.call(person, "John"); // Output: Hello, John! I'm Mr.
```

2. **The `apply()` Method:**

- The **`apply()`** method is similar to **`call()`** but accepts arguments as an array or an array-like object.
- It invokes the function and sets the value of **`this`**.

Example:

```
function greet(name) {
  console.log(`Hello, ${name}! I'm ${this.title}.`);
}
const person = {
  title: "Mr."
};
greet.apply(person, ["John"]); // Output: Hello, John! I'm Mr.
```

3. **The `bind()` Method:**

- The **`bind()`** method creates a new function that has a specified **`this`** value and, optionally, initial arguments.
- It does not immediately invoke the function but instead returns a new function that can be called later.

Example:

```
function greet(name) {
  console.log(`Hello, ${name}! I'm ${this.title}.`);
}
const person = {
  title: "Mr."
};
const greetPerson = greet.bind(person);
greetPerson("John"); // Output: Hello, John! I'm Mr.
```

# Object and Array Destructuring

## Object Destructuring:

In JavaScript, object destructuring is a convenient way to extract values from an object and assign them to variables. It allows you to access and unpack properties from an object in a concise and structured manner.

**Syntax:**

```
const { prop1, prop2, ... } = object;
```

- **Some of the key points include the following:**

1. **Variable Assignment:** Object destructuring enables you to assign object properties to variables with the same name. For example:

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // Output: John
console.log(age); // Output: 30
```

2. **Renaming Variables:** You can assign object properties to variables with different names using the colon (:) syntax. This can be useful when the variable name conflicts with another identifier.

```
const { name: fullName, age: years } = person;
console.log(fullName); // Output: John
console.log(years); // Output: 30
```

3. **Default Values:** Object destructuring allows you to assign default values to variables if the corresponding object property is undefined.

```
const { name, age, city = 'Unknown' } = person;
console.log(name); // Output: John
console.log(age); // Output: 30
console.log(city); // Output: Unknown
```

- **Property value shorthand:**

  It allows you to omit the property key in an object literal if the variable name matches the key. This shorthand is also applicable during object destructuring:

  Example 1:

```
const { x, y } = { x: 11, y: 8 }; // x = 11; y = 8
```

  This shorthand syntax is equivalent to:

```
const { x: x, y: y } = { x: 11, y: 8 };
```

  Example 2:

  You can combine property value shorthand with default values:

```
const { x, y = 1 } = {}; // x = undefined; y = 1
```

## Array Destructuring

Similarly, array destructuring in JavaScript provides a concise way to extract values from an array and assign them to variables.

**Syntax:**

```
const [item1, item2, ...] = array;
```

**Some of the key points include the following:**

1. **Variable Assignment:** Array destructuring allows you to assign array elements to variables based on their order.

```
const numbers = [1, 2, 3];
const [a, b, c] = numbers;
console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(c); // Output: 3
```

2. **Ignoring Elements:** You can skip elements in the array by leaving empty slots (commas) in the destructuring pattern.

```
const [x, , z] = numbers;
```

```
console.log(x); // Output: 1
console.log(z); // Output: 3
```

3. **Rest Syntax:** The rest syntax (...) allows you to capture the remaining elements of an array into a new array.

```
const [first, ...rest] = numbers;
console.log(first); // Output: 1
console.log(rest); // Output: [2, 3]
```

4. **Default Values:** Array destructuring supports default values as well, which are assigned if an array element is undefined.

```
const [p = 0, q = 0, r = 0] = numbers;
console.log(p); // Output: 1
console.log(q); // Output: 2
console.log(r); // Output: 3
```

Object and array destructuring are powerful features in JavaScript that simplify the extraction of values from objects and arrays, providing cleaner and more readable code.

## Summarizing it

In this lecture, we have covered the following topics:

- Objects, Object Creation and Accessing: We explored the concept of objects in JavaScript. We discussed object creation using object literals, which provide a convenient way to create and initialize objects using key-value pairs. We also learned about accessing object properties using dot notation and bracket notation.
- The "this" Keyword in Objects: We learned that "this" refers to the object itself when used within an object method, allowing seamless access to other properties and methods of the same object.
- Constructor Functions: We introduced constructor functions as an alternative approach for creating and initializing objects.

- Object Prototype: We learned that properties and methods can be added to the prototype object of a constructor function, and any objects created with that constructor will inherit those properties and methods.
- Object.create: We briefly discussed the `Object.create()` method, which is used to link objects together and establish prototype chains. It allows for the creation of new objects that inherit properties and methods from a prototype object.
- Call/Apply/Bind Methods: We covered the call(), apply(), and bind() methods, which are used to manipulate the execution context of functions and explicitly set the value of "this".
- Object and Array Destructuring in JavaScript

# References


- Object Prototypes: **Link**
- Array Prototype: **Link**