Lesson Plan

ES6 Classes

PW SKILLS

Topics

- Class Syntax
- Class constructor and initialization
- Hoisting in class
- Getters and setters
- Class inheritance
- Method overriding
- Static properties and method
- Syntactic sugar over constructor functions
- Classical inheritance vs prototypical inheritance

JavaScript, though not a purely object-oriented language by design, offers powerful features for building object-oriented applications. In this module, we will explore some key concepts in JavaScript OOP

Class Syntax

ES6 introduced classes as a more syntactic sugar-like way to define object blueprints. A class encapsulates data (properties) and behavior (methods) associated with objects.

First, look at the class in javascript

```javascript
class Meal {
  constructor(name, type) {
    this.name = name;
    this.type = type;
  }

  describe() {

    console.log(`This is a ${this.type} meal named ${this.name}.`);
  }
}

const meal1 = new Meal('Breakfast', 'healthy');
meal1.describe(); // Output: This is a healthy meal named Breakfast.
```

Class constructor and initialization

The constructor method is a special function called when creating a new object from a class. It's used to initialize the object's properties.

Example of class constructor -
From the above code examples, In the Meal class, the constructor takes name and type arguments, assigns them to this.name and this.type (properties of the new object), effectively initializing them.

Hoisting in class

Unlike functions, class declarations are not hoisted. You cannot use a class before it's declared.

For example

```
Unset
// This will throw an error:
console.log(Meal); // ReferenceError: Person is not defined

class Meal{
  // ...
}
```

Getters and setters

Getters and setters are special methods that allow you to control access to and modification of an object's properties.

For example, let's modify the above Meal class with the help of Getters and Setters

```
Unset
class Meal {
  constructor(name, type) {
    this._name = name;
    this._type = type;
  }

  // Getter for name
  get name() {
    return this._name;
  }

  // Setter for name
  set name(newName) {
    this._name = newName;
  }

  // Getter for type
  get type() {
    return this._type;
  }
```

```
  // Setter for type
  set type(newType) {
    this._type = newType;
  }

  describe() {
    console.log(`This is a ${this._type} meal named ${this._name}.`);
  }
}

const meal1 = new Meal('Breakfast', 'healthy');
console.log(meal1.name); // Output: Breakfast
console.log(meal1.type); // Output: healthy

meal1.name = 'Brunch'; // Using the setter for name
meal1.type = 'delicious'; // Using the setter for type

cons
```

Class inheritance

Classes can inherit properties and methods from parent classes using the extends keyword. This promotes code reusability and creates a hierarchy of related objects.

Let's take a simple example illustration of class inheritance

Unset

```
// parent class
class PW {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello ${this.name}`);
  }
}

// inheriting parent class
class Skills extends PW {}

let student1 = new Skills('PW skills');
student1.greet();
//output - Hello PW skills
```

Method overriding

In inheritance, a child class can redefine (override) a method inherited from the parent class. This allows for specialized behavior in child objects.

```
Unset
// Animal class
class Animal {
  constructor() {}

  // Method in Animal class
  speak() {
    console.log('Animal speaks');
  }
}

// Dog class inheriting from Animal
class Dog extends Animal {
  constructor() {
    super(); // Call Animal constructor
  }

  // Override the method in Dog class
  speak() {
    console.log('Woof woof!');
  }
}

// Creating instances
let animalObj = new Animal();
let dogObj = new Dog();

// Calling methods
animalObj.speak(); // Outputs: Animal speaks
dogObj.speak(); // Outputs: Woof woof!
```

From the above example, the Animal serves as the parent class, and the Dog is the child class inheriting from the Animal. We've changed the method name from sayHello to speak to better fit the animal context. Additionally, the override method in the Dog class now represents a dog's way of speaking.

Static properties and method

Static properties and methods belong to the class itself, not to individual objects. They are accessed using the class name, not through object instances.

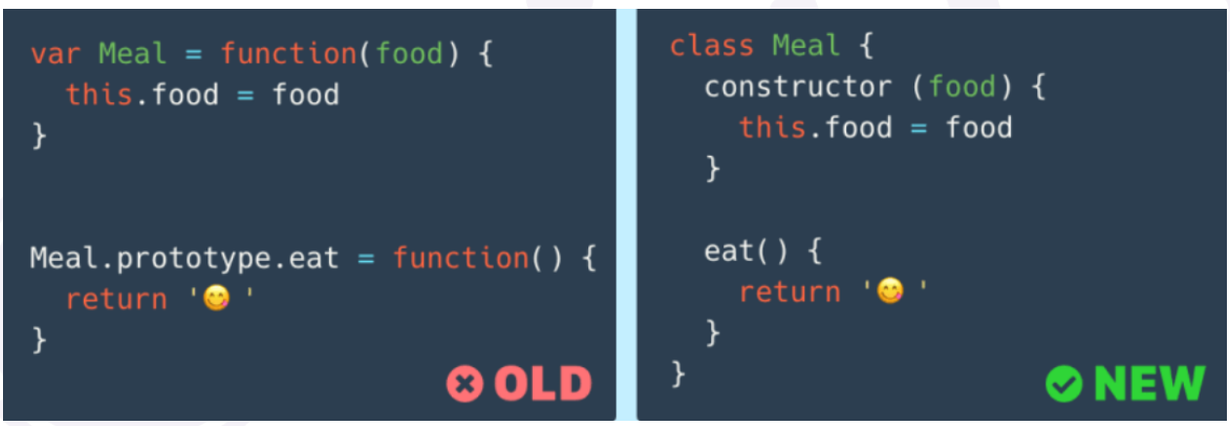
```
Unset
class Car {
  static MAX_SPEED = 120; // Static property

  static calculateSpeed(distance, time) {
    return distance / time;
  }
}

console.log(Car.MAX_SPEED); // Output: 120
const speed = Car.calculateSpeed(100, 2); // Call static method
console.log(speed); // Output: 50
```

Syntactic sugar over constructor functions

In inheritance, a child class can redefine (override) a method inherited from the parent class. This allows for specialized behavior in child objects.



Classes offer a cleaner and more readable syntax compared to traditional constructor functions for defining object blueprints.

Traditional Constructor Function Approach

```
Unset
// Parent constructor function
function Parent(name) {
  this.name = name;
}

// Method in parent constructor function
Parent.prototype.sayHello = function () {
  console.log('Hello, my name is ' + this.name);
};

// Creating an instance
var parentObj = new Parent('John');

// Calling the method
parentObj.sayHello(); // Outputs: Hello, my name is John
```

ES6 Class Syntax Approach

```
Unset
// Parent class
class Parent {
  constructor(name) {
    this.name = name;
  }

  // Method in parent class
  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

// Creating an instance
let parentObj = new Parent('John');

// Calling the method
parentObj.sayHello(); // Outputs: Hello, my name is John
```

Classical inheritance vs prototypical inheritance

Object-oriented programming (OOP) is a powerful paradigm for structuring code. Inheritance, a core concept in OOP, allows you to create new objects (child objects) based on existing objects (parent objects). However how inheritance works can differ between programming languages. In JavaScript, we have two main approaches: classical inheritance and prototypal inheritance.

Let's dive into their key differences:

Feature	Classical Inheritance	Prototypal Inheritance
Building Block	Classes	Objects
Inheritance Structure	Strict parent-child hierarchy	Flexible prototype chain
Prototype Modification	Affects all child objects	Only affects objects created after modification
JavaScript Implementation	Simulated using prototypes	Native inheritance model
Code readability	More familiar syntax with Class keyword	Less familiar syntax