SUNSTONE

JavaScript

LECTURE 12 NOTES

Classes and Modules

- **What are Classes?**
  - In JavaScript, classes are a way to define objects and create reusable blueprints for making instances of those objects. They provide a syntactical sugar on top of JavaScript's existing prototype-based inheritance.
  - In JavaScript, classes are a special type of function. We can define the class just like function declarations and function expressions.
  - The JavaScript class contains various class members within a body including methods or constructors. The class is executed in strict mode. So, the code containing the silent error or mistake throws an error.
  - Here's an example of how to define a class in JavaScript:

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

  - In the above example, we define a Person class with a constructor method and a greet method. The constructor method is called when a new instance of the class is created and is used to initialize the object's properties. The greet method is a regular function that can be called on instances of the Person class.
  - To create a new instance of the Person class, you can use the new keyword:

```
const person1 = new Person('John Doe', 25);
person1.greet();
// Output: Hello, my name is John Doe and I am 25 years old.
```

- You can also define class methods and class properties that are shared across all instances of the class, rather than being specific to each instance. Class methods can be defined using the static keyword:

```
class Person {
  static count = 0;
  constructor(name, age) {
    this.name = name;
    this.age = age;
    Person.count++;
  }
  static getCount() {
    console.log(`Total number of persons: ${Person.count}`);
  }
}
const person1 = new Person('John Doe', 25);
const person2 = new Person('Jane Smith', 30);
Person.getCount(); // Output: Total number of persons: 2
```

- In the above example, the count property is a class property that keeps track of the number of instances created. The **getCount** method is a static method that can be called on the Person class itself, rather than on instances of the class.

- **Defining and Using Classes**

Step 1: Defining a Class

- To define a class in JavaScript, you can use the class keyword followed by the name of the class. Inside the class, you can define a constructor method and other methods.

```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }
  calculateArea() {
    return this.width * this.height;
  }
  calculatePerimeter() {
    return 2 * (this.width + this.height);
  }
}
```

- In the above example, we define a Rectangle class with a constructor that takes width and height as parameters. The class also has two methods, calculateArea() and calculatePerimeter(), which perform calculations based on the dimensions of the rectangle.

Step 2: Creating Instances

- To create instances (objects) of the class, you can use the new keyword followed by the class name and any necessary arguments for the constructor.

```
const rectangle1 = new Rectangle(5, 10);
const rectangle2 = new Rectangle(3, 6);
```

- In this example, we create two instances of the Rectangle class, rectangle1 and rectangle2, with different dimensions.

Step 3: Accessing Object Properties and Methods

- Once you have created instances of the class, you can access their properties and call their methods using the dot notation.

```
console.log(rectangle1.width);                // Output: 5

console.log(rectangle1.calculateArea());      // Output: 50

console.log(rectangle2.height);               // Output: 6

console.log(rectangle2.calculatePerimeter()); // Output: 18
```

- Here, we access the properties width and height of rectangle1 and rectangle2, and call their respective methods to calculate the area and perimeter.

Step 4: Class Inheritance (Optional)

- JavaScript also supports class inheritance, where you can create a new class based on an existing class. The new class inherits the properties and methods of the existing class and can add its own unique properties and methods.

```
class Square extends Rectangle {

  constructor(side) {

    super(side, side); // Call the parent class constructor

  } }
```

- In this example, we define a Square class that extends the Rectangle class. The Square class has a constructor that takes a side parameter and calls the parent class constructor using the super keyword.

```
const square = new Square(4);

console.log(square.calculateArea());          // Output: 16

console.log(square.calculatePerimeter());     // Output: 16
```

- We create an instance of the Square class and use its inherited methods `calculateArea()` and `calculatePerimeter()` from the Rectangle class.

- **Static and Instance Members**

In JavaScript, static and instance members are two different types of properties and methods that can be defined in a class. Let's explore each of them:

1. **Instance Members:** Instance members are properties and methods that belong to individual instances (objects) of a class. Each instance has its own copy of these members. They are defined without the static keyword within the class.

```
class MyClass {
  constructor() {
    this.instanceProperty = 123;
  }
  instanceMethod() {
    console.log('This is an instance method');
  }
}
const obj1 = new MyClass();const obj2 = new MyClass();
console.log(obj1.instanceProperty);     // Output: 123
obj1.instanceMethod();     // Output: "This is an instance method"
console.log(obj2.instanceProperty);     // Output: 123
obj2.instanceMethod();     // Output: "This is an instance method"
```

   - In the example above, `instanceProperty` and `instanceMethod` are instance members of the `MyClass` class. Each instance of `MyClass` has its own copy of these members.

2. **Static Members:** Static members are properties and methods that belong to the class itself rather than its instances. They are shared among all instances and can be accessed directly in the class using the static keyword.

```
class MyClass {
  static staticProperty = 'Hello, I am a static property';
  static staticMethod() {
    console.log('This is a static method');
  } }
console.log(MyClass.staticProperty);     // Output: "Hello, I am a
static property"MyClass.staticMethod();
// Output: "This is a static method"
```

- In the example above, `staticProperty` and `staticMethod` are static members of the `MyClass` class. They are accessed directly on the class itself, without creating an instance.

- ***Note that static members cannot be accessed through instances of the class.***

  For example:

```
const obj = new MyClass();
console.log(obj.staticProperty);  //Output: undefined
obj.staticMethod();         //Error: obj.staticMethod is not a function
```

- Static members are shared across all instances and are commonly used for utility functions or constants that are not dependent on any specific instance.

- **Inheritance and Polymorphism**

  - In JavaScript, inheritance and polymorphism can be achieved using prototype-based inheritance. The prototype chain allows objects to inherit properties and methods

from other objects, enabling the concepts of inheritance and polymorphism.

- **Inheritance:** Inheritance allows objects to inherit properties and methods from a parent object or class. In JavaScript, this is accomplished by setting up the prototype chain.

  - Here's an example of how to create inheritance in JavaScript:

```
// Parent class
function Animal(name) {
  this.name = name;
}
// Method in the parent class
Animal.prototype.speak = function() {
  console.log(this.name + ' makes a sound.');
};
// Child class inheriting from the parent class
function Dog(name) {
  Animal.call(this, name);
}
// Set up the prototype chain
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
// Method in the child class
Dog.prototype.speak = function() {
  console.log(this.name + ' barks.');
};
// Create instances and invoke methods
const animal = new Animal('Animal');
```

```
animal.speak();          // Output: "Animal makes a sound."

const dog = new Dog('Dog');

dog.speak();             // Output: "Dog barks."
```

- In the example above, the Animal class acts as the parent class, while the Dog class inherits from it using Object.create(Animal.prototype). By doing this, the Dog instances have access to both their own methods and the inherited methods from the Animal class.

➢ **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass, thereby sharing the same interface or method signature.

- Here's an example of how polymorphism can be achieved in JavaScript:

```
// Parent class

function Shape() {}

// Method in the parent class

Shape.prototype.draw = function() {

  console.log('Drawing a generic shape.');

};

// Child classes inheriting from the parent class

function Circle() {}

function Square() {}

// Set up the prototype chain

Circle.prototype = Object.create(Shape.prototype);

Circle.prototype.constructor = Circle;

Square.prototype = Object.create(Shape.prototype);

Square.prototype.constructor = Square;
```

```
// Method in the child class, overriding the parent method
Circle.prototype.draw = function() {
  console.log('Drawing a circle.');
};
Square.prototype.draw = function() {
  console.log('Drawing a square.');
};
// Create instances and invoke the method
const shapes = [new Circle(), new Square()];
shapes.forEach(function(shape) {
  shape.draw();
});
```

- In the example above, the Shape class acts as the superclass, while the Circle and Square classes inherit from it. Each subclass overrides the draw method to provide its own implementation. By treating the Circle and Square objects as Shape objects, we can invoke the draw method on each of them, resulting in polymorphic behavior. The appropriate method implementation is based on the actual type of the object at runtime.

  Output:

```
Drawing a circle.
Drawing a square.
```

- As you can see, the draw method is called on each object, and the appropriate implementation is executed based on the object's type.

- **ES6 Modules and Import/Export Statements**

  - ES6 (ECMAScript 2015) introduced a new module system that allows JavaScript code to be organized into separate files and imported/exported as needed.
  - This module system provides a more structured and scalable approach to building large-scale JavaScript applications.
  - ES6 modules use the import and export statements to control the visibility and accessibility of code between different modules. Here's how they work:

➢ **Exporting from a Module:**

  - You can export functions, variables, or objects from a module using the export keyword.
  - There are two primary ways to export code:

    1. **Default export:** You can have one default export per module, representing the main functionality or value that the module provides. It is denoted using the default keyword.

```
// export a default function

export default function add(a, b) {

return a + b;

}
```

    2. **Named export:** You can export multiple variables, functions, or objects as named exports from a module. Each named export needs to be explicitly specified using the export keyword.

```
// export named variables

export const PI = 3.14;

export const MAX_VALUE = 100;

// export named functions
```

```
export function multiply(a, b) {
return a * b;
}
// export named object
export const person = {
name: "John",
age: 30
};
```

➢ **Importing into a Module:**

- To use code from other modules, you need to import it using the import keyword.
- You can import default exports and named exports using different syntax.

1. **Importing a default export:**

```
import add from "./math"; // imports the default export from "./math"
console.log(add(2, 3)); // uses the imported default function
```

2. **Importing named exports:**

```
import { PI, MAX_VALUE, multiply, person } from "./math"; // imports specific named exports
console.log(PI);          // uses the imported named variable
console.log(MAX_VALUE);
console.log(multiply(2, 3)); // uses the imported named function
console.log(person.name); // uses the imported named object
```

3. **Importing all named exports as an object:**

```
import * as math from "./math";

// imports all named exports as "math" object

console.log(math.PI);

// uses the imported named variable

console.log(math.MAX_VALUE);

console.log(math.multiply(2, 3));

// uses the imported named function

console.log(math.person.name);

// uses the imported named object
```

- It's important to note that ES6 modules use a static import/export system, which means imports and exports are resolved at module initialization. They are not dynamically evaluated at runtime. Additionally, modules are loaded asynchronously, making them suitable for use in both web browsers and server-side JavaScript environments like Node.js.