(10.1.1) ES6 Intermediate - Class - Getter & Setter

Understanding Getter and Setter Methods in JavaScript
In object-oriented programming, encapsulation(כימוס) is a fundamental concept that involves bundling data (properties) and methods (functions) that operate on that data within a single unit, typically known as a class. JavaScript, despite being a prototype-based language, supports encapsulation through the use of getter and setter methods. These methods allow controlled access to an object's properties and enable validation and modification of data before it is accessed or assigned.

What are Getter and Setter Methods?
Getter and setter methods are special functions used to access and modify the properties of an object in a controlled manner. They provide an interface to read and write values to object properties, allowing for better control over the property's behavior and enabling data validation and manipulation.

Getter Methods
A getter method is a function that is used to retrieve the value of a specific property in an object. It is associated with a property and is invoked when that property is accessed. Getter methods are defined using the `get` keyword, followed by the desired property name.

```
const obj = { _value: 42, // Private property get value() {return this._value;
}};console.log(obj.value); // Output: 42
```

In this example, the `value` property is associated with a getter method that retrieves the value of the private property `_value`.

Setter Methods
A setter method is a function that is used to modify the value of a specific property in an object. It allows you to perform actions before setting a new value to a property, such as validation or formatting. Setter methods are defined using the `set` keyword, followed by the desired property name.

```
const obj = { _value: 42, // Private property set value(newValue) {if (newValue
>= 0) { this._value = newValue;} else { console.error('Value must be a non-
```

```
negative number.');} }};obj.value = 50; // Sets the value using the
setterconsole.log(obj.value); // Output: 50obj.value = -10; // Attempts to set a
negative value// Output: Value must be a non-negative number.
```

In this example, the `value` property is associated with a setter method that validates the input before modifying the private property `_value`.

Use Cases of Getter and Setter Methods

1. Data Validation

```
class Temperature {
constructor() {
this._celsius = 0; // Private property for temperature in Celsius
}

// Getter method to retrieve temperature in Celsius
get celsius() {
return this._celsius;
}

// Setter method to set temperature in Celsius with validation
set celsius(value) {
if (value >= -273.15) { // Absolute zero in Celsius
this._celsius = value;
} else {
console.error('Invalid temperature. Must be greater than or equal to
-273.15°C.');
}
}
}

const weather = new Temperature();
weather.celsius = 25; // Setting a valid temperature
console.log(`Current temperature: ${weather.celsius}°C`); // Output: Current
temperature: 25°C

weather.celsius = -300; // Attempting to set an invalid temperature
// Output: Invalid temperature. Must be greater than or equal to -273.15°C.
```

In this example, the getter and setter methods for the `celsius` property of the `Temperature` class ensure that temperature values are valid and within a reasonable range.

2. Computed Properties

```
class Rectangle {
```

```
constructor(width, height) {
this._width = width;
this._height = height;
}

// Getter method to compute and return the area
get area() {
return this._width * this._height;
}
}

const myRectangle = new Rectangle(5, 10);
console.log(`Area of the rectangle: ${myRectangle.area} square units`); //
Output: Area of the rectangle: 50 square units
```

In this example, the `area` getter method calculates and returns the area of a rectangle based on its `width` and `height` properties. This allows you to access the area as if it were a regular property.

3. Controlled Access

```
class BankAccount {
constructor(accountNumber, initialBalance) {
this._accountNumber = accountNumber;
this._balance = initialBalance;
}

// Getter method to retrieve the balance
get balance() {
return this._balance;
}

// Setter method to set the balance with additional logic
set balance(newBalance) {
if (newBalance >= 0) {
this._balance = newBalance;
} else {
console.error('Invalid balance. Balance cannot be negative.');
}
}

// Method to withdraw funds with validation
withdraw(amount) {
if (amount <= this._balance) {
this._balance -= amount;
console.log(`Withdrawn ${amount} units. New balance: ${this._balance}`);
} else {
console.error('Insufficient funds.');
}
```

```
}
}

const myAccount = new BankAccount('123456', 1000);
console.log(`Initial balance: ${myAccount.balance} units`); // Output: Initial
balance: 1000 units

myAccount.withdraw(500); // Withdraw funds
console.log(`Current balance: ${myAccount.balance} units`); // Output: Current
balance: 500 units

myAccount.balance = -200; // Attempting to set a negative balance
// Output: Invalid balance. Balance cannot be negative.
```

In this example, the `balance` getter and setter methods allow controlled access to the `balance` property of a bank account object. The `withdraw` method also ensures that funds are withdrawn only if the balance is sufficient. This controlled access helps maintain internal consistency in the bank account object.

Conclusion

Getter and setter methods in JavaScript are powerful tools that allow developers to control access to an object's properties, validate data, and add logic for property assignment. By leveraging these methods, you can enhance the security, robustness, and flexibility of your JavaScript applications. Understanding when and how to use getter and setter methods is a valuable skill for any JavaScript developer seeking to write clean, maintainable, and efficient code.