Encapsulation in JavaScript

Encapsulation

Encapsulation is the concept of bundling the data (state) and methods (functions) that operate on the data into a single unit known as a class. It restricts access to some of the object's components, providing a way to control the data's visibility and behavior.

Benefits of Encapsulation

- **Data Protection:** It helps protect the integrity of an object's data by controlling how it can be accessed and modified.
- **Abstraction:** Encapsulation hides the internal implementation details of a class, allowing you to interact with objects using a simple interface, which is useful for managing complexity.
- **Maintainability:** It makes code easier to maintain and refactor because changes to the internal implementation of a class do not affect the code that uses the class.
- **Reduced Bugs:** By controlling access to data and enforcing constraints through setter methods, you can reduce the chances of introducing bugs and invalid states.

Defining Private Properties and Methods:

To define a private property or method, use the # symbol followed by the property or method name within a class. Private properties are typically used to store internal state, while private methods encapsulate logic.

```
    class Person {
        #name; // Private property
        #age; // Private property
        constructor(name, age) {
          this.#name = name;
          this.#age = age;
        }
        #incrementAge() { // Private method
          this.#age++;
        }
        sayHello() {
          console.log(`Hello, my name is ${this.#name} and I am ${this.#age}
years old.`);
        }
      }
```

Accessing Private Properties and Methods Inside the Class:

Inside the class, private properties and methods can be accessed directly like any other property or method. They are in the same scope as other class members.

```
class Person {
    #name;
    #age;
    constructor(name, age) {
      this.#name = name;
      this.#age = age;
    }
    greet() {
      console.log(`Hello, my name is ${this.#name}`);
      this.#incrementAge(); // Accessing private method
    }
    #incrementAge() {
      this.#age++;
    }
  }
```

Accessing Private Properties and Methods Outside the Class:

Private properties and methods **cannot be accessed directly from outside the class**. Attempting to do so will result in a TypeError. You can only access them through public methods like getter and setter.

Getter- A method that returns the property of a class.

Setter - A method that is used to manipulate the property of the class.

Getter Methods for Private Properties:

To provide controlled access to private properties, you can define public-getter methods within the class. These getter methods allow you to retrieve the values of private properties.

```
class Person {

    #name;

    #age;

    constructor(name, age) {

      this.#name = name;

      this.#age = age;

    }

    getName() {

      return this.#name; // Getter method for private property

    }
  }

  const person = new Person('Alice', 30);

  console.log(person.getName()); // Outputs: Alice
```

This way, you can retrieve the value of the private property #name without exposing it directly.

Setter Methods for Private Properties:

To modify the values of private properties, you can define public setter methods within the class. These setter methods allow you to update the values of private properties with validation logic if needed.

```
  class Person {

      #name;

      constructor(name) {

        this.#name = name;

      }

      setName(newName) {

        if (newName.length >= 3) {

          this.#name = newName; // Setter method for private property

        }

      }

    }

    const person = new Person('Alice');

    person.setName('Bob');

    console.log(person.getName()); // Outputs: Bob
```

The setName method checks the length of the new name and only updates the private property if the condition is met.

Summary:

In summary, encapsulation in JavaScript using ES6 classes involves defining classes, using naming conventions for property and method visibility, and employing getter and setter methods to control access to class properties. It helps in creating well-structured and maintainable code by encapsulating data and behavior within classes.