5-Classes

Chapter 13: Understanding Classes

Classes are blueprints for creating objects that share similar properties and methods. They allow you to structure your code in an organized, reusable way, making JavaScript especially powerful for larger projects.

13.1 What is a Class?

A class in JavaScript defines a template for creating objects, encapsulating data (properties) and behavior (methods). Each object created from a class is an **instance** of that class.

13.2 Defining a Class

You can define a class using the `class` keyword.

Example:

```
class Person {
  constructor(name, age) {
    this.name = name; // Property
    this.age = age;   // Property
  }

  greet() { // Method
    console.log(`Hello, my name is ${this.name}, and I am ${this.age} years old.`);
  }
}

// Creating an instance
const rick = new Person("Rick", 30);
rick.greet(); // Output: Hello, my name is Rick, and I am 30 years old.
```

Key Points:

- **Constructor:** The `constructor` method is called when a new instance of the class is created. It initializes the object's properties.
- **Properties:** Store data specific to each instance.
- **Methods:** Define behaviors that instances can perform.

13.3 Adding Methods to a Class

A class can include multiple methods to define behaviors.

Example:

```javascript
class Calculator {
  add(a, b) {
    return a + b;
  }

  subtract(a, b) {
    return a - b;
  }
}

const calc = new Calculator();
console.log(calc.add(5, 3)); // Output: 8
console.log(calc.subtract(10, 7)); // Output: 3
```

13.4 Working with Static Methods

Static methods belong to the class itself, not the instances, and are called directly on the class.

Example:

```javascript
class MathUtils {
  static square(num) {
    return num * num;
  }
}

console.log(MathUtils.square(4)); // Output: 16
```

13.5 Inheritance: Extending Classes

Inheritance allows a class to inherit properties and methods from another class. Use the `extends` keyword to create a subclass.

Example:

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  makeSound() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  makeSound() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog("Buddy");
dog.makeSound(); // Output: Buddy barks.
```

Key Concepts:

- **Parent Class:** The base class being inherited from.
- **Child Class:** The class that extends the parent class.
- **Method Overriding:** The child class can redefine methods from the parent class.

13.6 Getters and Setters

Getters and setters allow you to access or update properties with additional logic.

Example:

```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }

  get area() {
    return this.width * this.height;
  }

  set dimensions({ width, height }) {
    this.width = width;
    this.height = height;
  }
}

const rect = new Rectangle(10, 5);
console.log(rect.area); // Output: 50
rect.dimensions = { width: 20, height: 10 };
console.log(rect.area); // Output: 200
```

13.7 Practical Example: Creating a Game Character

Example:

```
class Character {
  constructor(name, health, attackPower) {
    this.name = name;
    this.health = health;
    this.attackPower = attackPower;
  }

  attack(target) {
    console.log(`${this.name} attacks ${target.name} for ${this.attackPower} damage!`);
    target.health -= this.attackPower;
  }

  displayStats() {
    console.log(`${this.name}: ${this.health} HP`);
  }
}

class Warrior extends Character {
  constructor(name, health, attackPower, shield) {
    super(name, health, attackPower); // Call parent constructor
    this.shield = shield;
  }

  defend(damage) {
    const reducedDamage = damage - this.shield;
    this.health -= reducedDamage > 0 ? reducedDamage : 0;
    console.log(`${this.name} defends! ${this.health} HP remaining.`);
  }
}

// Creating instances
const hero = new Warrior("Aragon", 100, 15, 5);
const monster = new Character("Goblin", 50, 10);

// Simulating battle
hero.attack(monster);
monster.displayStats();
monster.attack(hero);
hero.defend(monster.attackPower);
hero.displayStats();
```

13.8 Advanced Features

1. **Private Fields and Methods:** Use the `#` syntax to define private properties and methods that are only accessible within the class.
   Example:

```
class BankAccount {
  #balance = 0;

  deposit(amount) {
    this.#balance += amount;
  }

  getBalance() {
    return this.#balance;
  }
}

const account = new BankAccount();
account.deposit(100);
console.log(account.getBalance()); // Output: 100
```

2. **Polymorphism:** Create methods in child classes that have the same name as in the parent class but behave differently.
3. **Abstract Classes and Interfaces (via Prototypes):** While JavaScript doesn’t have built-in abstract classes or interfaces, you can simulate them with prototypes or frameworks.

13.9 When to Use Classes

- Organize related data and behaviors.
- Reuse code through inheritance and modular design.
- Simplify development for large-scale applications, such as games or web apps.