JavaScript Classes: Comprehensive Guide

JavaScript Classes provide a way to create reusable object templates using a modern syntax. This guide covers the basics, advanced features, practical examples, exercises, and multiple-choice questions to help you master JavaScript classes.

What are Classes in JavaScript?

Classes are syntactic sugar over JavaScript's prototype-based inheritance model. They allow you to define object templates with properties and methods.

Basic Syntax

```
class ClassName {
  constructor(param1, param2) {
    this.property1 = param1;
    this.property2 = param2;
  }
  method1() {
    console.log(this.property1);
  }
}
```

- **`constructor`**: A special method for initializing object properties.
- **`this`**: Refers to the instance of the class.

Example: Define and Instantiate a Class

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hi, my name is ${this.name} and I'm ${this.age} years
old.`);
  }
}
const person1 = new Person("Alice", 30);
person1.greet(); // Output: Hi, my name is Alice and I'm 30 years old.
```

Class Methods and Properties

1. **Instance Methods:** Methods available on instances of the class.
2. **Static Methods:** Methods that belong to the class itself, not the instances.

Example: Static Methods

```
class MathHelper {
  static add(a, b) {
    return a + b;
  }
}
console.log(MathHelper.add(5, 3)); // Output: 8
```

Inheritance with Classes

Classes support inheritance through the `extends` keyword. The `super` keyword calls the parent class's constructor or methods.

Example: Class Inheritance

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}
class Dog extends Animal {
  speak() {
    super.speak();
    console.log(`${this.name} barks.`);
  }
}
const dog = new Dog("Rex");
dog.speak();
// Output:
// Rex makes a noise.
// Rex barks.
```

Private and Public Fields

JavaScript classes can define private and public fields.

Example: Private Fields

```
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
// console.log(account.#balance); // Error: Private field '#balance'
must be declared in an enclosing class
```

- **`#balance`**: Denotes a private field accessible only within the class.

Getters and Setters

Getters and setters are used to define accessors for properties.

Example: Using Getters and Setters

```
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, 20);
console.log(rect.area); // Output: 200
rect.dimensions = { width: 5, height: 15 };
console.log(rect.area); // Output: 75
```

Exercises

Exercise 1: Create a Vehicle Class

1. Create a class `Vehicle` with properties `make` and `model`.
2. Add a method `getDetails` that prints `make` and `model`.

Solution:

```
class Vehicle {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }
  getDetails() {
    console.log(`Make: ${this.make}, Model: ${this.model}`);
  }
}
const car = new Vehicle("Toyota", "Corolla");
car.getDetails(); // Output: Make: Toyota, Model: Corolla
```

Exercise 2: Implement a Calculator Class

1. Create a class `Calculator` with static methods `add`, `subtract`, `multiply`, and `divide`.
2. Test the methods with sample inputs.

Solution:

```
class Calculator {
  static add(a, b) {
    return a + b;
  }
  static subtract(a, b) {
    return a - b;
  }
  static multiply(a, b) {
    return a * b;
  }
  static divide(a, b) {
    return b !== 0 ? a / b : "Cannot divide by zero";
  }
}
console.log(Calculator.add(5, 3)); // Output: 8
console.log(Calculator.divide(10, 2)); // Output: 5
```

Exercise 3: Inheritance

1. Create a parent class `Employee` with properties `name` and `position`.
2. Create a child class `Manager` that inherits from `Employee` and adds a method `announce`.

Solution:

```
class Employee {
  constructor(name, position) {
    this.name = name;
    this.position = position;
  }
  details() {
    console.log(`${this.name} works as a ${this.position}`);
  }
}
class Manager extends Employee {
  announce() {
    console.log(`${this.name} is a manager!`);
  }
}
const manager = new Manager("John", "Manager");
manager.details(); // Output: John works as a Manager
manager.announce(); // Output: John is a manager!
```

Multiple-Choice Questions

Question 1:

What is the purpose of the `constructor` in a class?

1. To define class methods.
2. To initialize properties of the class.
3. To extend another class.
4. To create static methods.

Answer: 2. To initialize properties of the class.

Question 2:

Which keyword is used to inherit from another class in JavaScript?

1. `implements`
2. `extends`
3. `inherits`
4. `prototype`

Answer: 2. `extends`

Question 3:

How do you define a private field in a JavaScript class?

1. Use the `private` keyword.
2. Use the # symbol before the field name.
3. Prefix the field name with `_`.

4. Declare the field inside a method.

Answer: 2. Use the # symbol before the field name.

Best Practices for Using Classes

1. **Encapsulation:** Use private fields and methods to hide implementation details.
2. **DRY Principle:** Use inheritance to avoid duplicate code.
3. **Static Methods:** Use static methods for utility functions that don't depend on instance data.
4. **Meaningful Names:** Use meaningful class and method names for readability.