In JavaScript, inheritance is the mechanism by which one object acquires the properties and methods of another object. It is one of the core features of object-oriented programming (OOP) and is used to promote code reuse and create hierarchical relationships between classes. JavaScript supports inheritance through prototype-based inheritance and class-based inheritance (ES6+). Let's look at both approaches with detailed examples.

1. Prototype-based Inheritance (Traditional Inheritance)

In JavaScript, every object has a prototype, which is another object that it can inherit properties and methods from. This is how JavaScript implements inheritance.

Example 1: Inheriting properties and methods using prototypes

Javascript code:-

```
// Constructor function for Animal
function Animal(name) {
    this.name = name;
}

// Adding a method to Animal's prototype
Animal.prototype.speak = function() {
    console.log(this.name + ' makes a sound');
};

// Constructor function for Dog, inheriting from Animal
function Dog(name, breed) {
    Animal.call(this, name); // Call the parent constructor (Animal) to
inherit its properties
    this.breed = breed;
}

// Inheriting methods from Animal using the prototype chain
Dog.prototype = Object.create(Animal.prototype);

// Ensuring Dog's constructor points to Dog, not Animal
Dog.prototype.constructor = Dog;

// Adding a method specific to Dog
Dog.prototype.bark = function() {
    console.log(this.name + ' barks');
};

// Create a new Dog instance
const dog1 = new Dog('Buddy', 'Golden Retriever');

// Accessing inherited method from Animal
dog1.speak();  // "Buddy makes a sound"

// Accessing method specific to Dog
dog1.bark();   // "Buddy barks"
```

Explanation:

- `Animal` is a constructor function that creates an object with a `name` property.
- `Animal.prototype.speak` is a method that is shared by all instances of `Animal`.
- `Dog` is another constructor function, which inherits from `Animal` using `Object.create(Animal.prototype)`. This sets the prototype of `Dog` to be an object that inherits from `Animal`'s prototype.
- `Dog.call(this, name)` ensures that the `Dog` constructor also calls the `Animal` constructor to inherit the `name` property.
- `Dog.prototype.bark` adds a method specific to `Dog`.
- We create an instance `dog1` of `Dog`, which can use both `Animal`'s `speak` method and `Dog`'s `bark` method.

2. Class-based Inheritance (ES6 Class Syntax)

In ECMAScript 6 (ES6), JavaScript introduced the `class` syntax to simplify object creation and inheritance. This provides a more familiar, clean way to work with inheritance, similar to classical OOP languages.

Example 2: Inheriting properties and methods using ES6 classes

Javascript code

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

    speak() {
        console.log(this.name + ' makes a sound');
    }
}

// Defining the Dog class which extends Animal
class Dog extends Animal {
    constructor(name, breed) {
        super(name);  // Call the parent class constructor to inherit
properties
        this.breed = breed;
    }

    bark() {
        console.log(this.name + ' barks');
    }
}

// Create a new Dog instance
const dog2 = new Dog('Max', 'Bulldog');

// Accessing inherited method from Animal
```

```
dog2.speak();  // "Max makes a sound"

// Accessing method specific to Dog
dog2.bark();   // "Max barks"
```

Explanation:

- `Animal` is defined as a class with a constructor that sets the `name` property and a `speak` method.
- `Dog` extends `Animal`, meaning `Dog` is a subclass of `Animal`. This allows `Dog` to inherit all properties and methods from `Animal`.
- `super(name)` calls the parent class (`Animal`) constructor to initialize the `name` property.
- `Dog` can also have its own methods, like `bark`, which are specific to the `Dog` class.
- `dog2` is an instance of `Dog`, which can use both the inherited `speak` method and the `bark` method defined in `Dog`.

3. Overriding Methods in Inheritance

You can override inherited methods to provide custom behavior in the subclass.

Example 3: Overriding inherited methods

Javascript code:-

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

    speak() {
        console.log(this.name + ' makes a sound');
    }
}

// Defining the Dog class which extends Animal
class Dog extends Animal {
    constructor(name, breed) {
        super(name);
        this.breed = breed;
    }

    speak() {
        console.log(this.name + ' barks');
    }
}

// Create a new Dog instance
const dog3 = new Dog('Rex', 'Labrador');

// The Dog class overrides the speak method
dog3.speak();  // "Rex barks"
```

Explanation:

- In this example, the `Dog` class overrides the `speak` method from the `Animal` class to provide custom behavior specific to dogs (barking instead of just making a sound).
- When `dog3.speak()` is called, it uses the overridden method in the `Dog` class instead of the one in `Animal`.

4. Inheritance with Getter and Setter Methods

JavaScript allows you to use getter and setter methods in classes, which can also be inherited.

Example 4: Using getter and setter in inheritance

Javascript code:-

```
// Defining the Animal class
class Animal {
    constructor(name) {
        this._name = name; // Using a private property
    }

    get name() {
        return this._name;
    }

    set name(value) {
        this._name = value;
    }

    speak() {
        console.log(this.name + ' makes a sound');
    }
}

// Defining the Dog class which extends Animal
class Dog extends Animal {
    constructor(name, breed) {
        super(name);
        this.breed = breed;
    }

    speak() {
        console.log(this.name + ' barks');
    }
}

// Create a new Dog instance
const dog4 = new Dog('Charlie', 'Beagle');
dog4.speak();  // "Charlie barks"

// Using getter and setter
dog4.name = 'Max';
console.log(dog4.name);  // "Max"
```

Explanation:

- The `Animal` class has a getter and setter for the `name` property. The setter allows modification of the `_name` private variable, and the getter allows access to it.
- The `Dog` class inherits these methods and can access and modify the `name` property via the getter and setter, in addition to its own methods.

Summary:

- **Prototype-based inheritance**: Objects inherit directly from other objects through prototypes. The method is more flexible but can be less clear.
- **Class-based inheritance** (introduced in ES6): This is syntactic sugar over prototype-based inheritance and is easier to understand for developers familiar with class-based OOP.
- **Overriding methods**: You can override inherited methods to change the behavior in the subclass.
- **Getter/Setter**: These methods allow controlled access to object properties and can be inherited like regular methods.

In both approaches, inheritance allows objects to share behavior and state, reducing code duplication and improving maintainability.