3. **Inheritance:** Inheritance allows a class to inherit properties and methods from another class. This helps to create a new class based on an existing class, allowing for code reuse and a more hierarchical class structure. For example, you could have a `Vehicle` class and create a `Car` class that inherits from `Vehicle`. The `Car` class will have all the properties and methods of `Vehicle`, and you can also add specific ones to `Car`.

Inheritance allows a class to inherit properties and methods from another class. This helps in creating a new class based on an existing class, enabling code reuse and the creation of a hierarchical class structure.

## Example: Inheritance

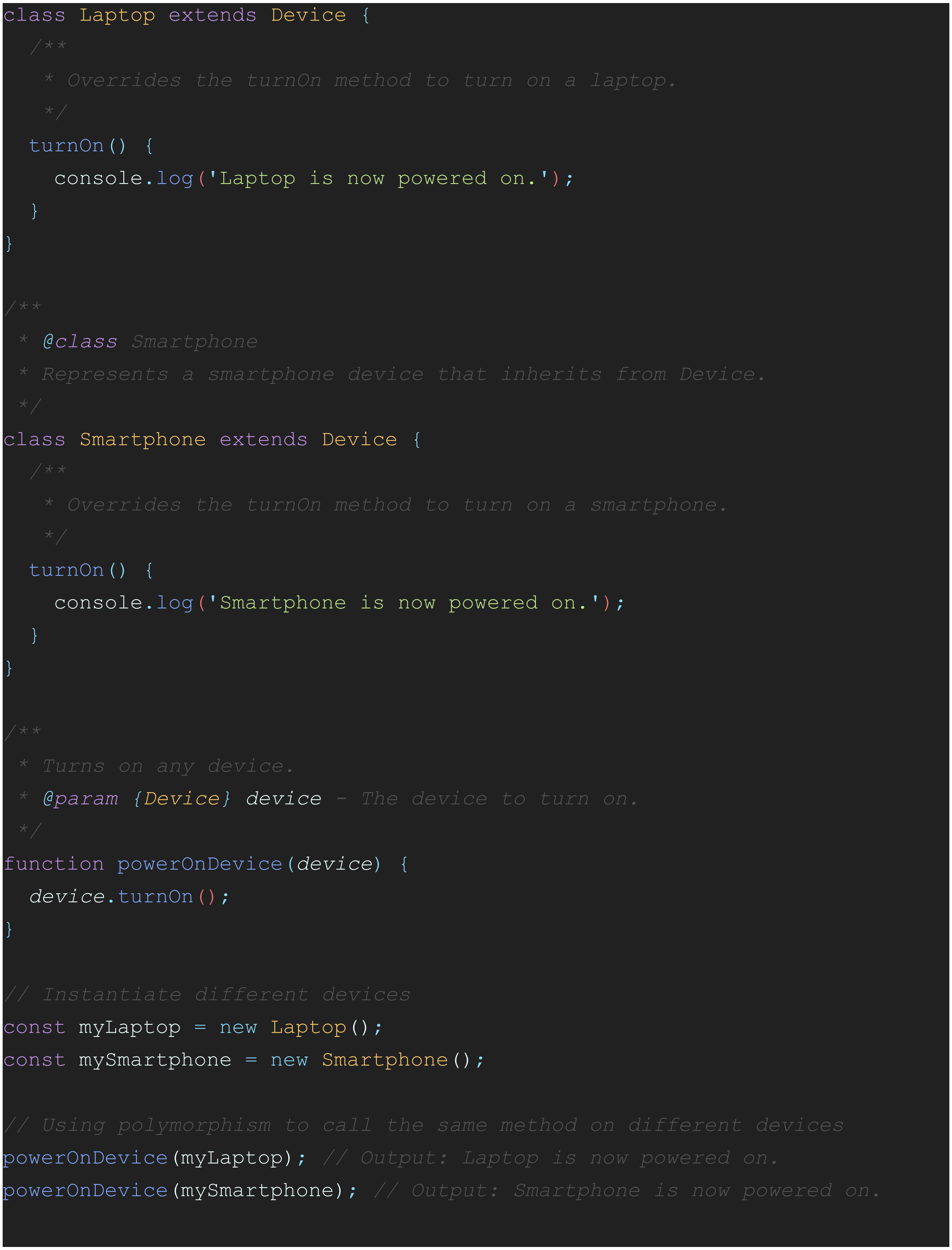
Let's create a base class called `Animal` and a derived class called `Dog` that inherits from `Animal`.

### JavaScript Code Example

```javascript
// Define the Animal class (base class)
class Animal {
    // Constructor method to initialize the properties of a new Animal
object
    constructor(name) {
        this.name = name; // Property: the name of the animal
    }

    // Method: a function that describes an action related to the animal
    speak() {
        console.log(`${this.name} makes a sound.`);
    }
}

// Define the Dog class (derived class) that inherits from Animal
class Dog extends Animal {
    // Constructor method to initialize the properties of a new Dog object
    constructor(name, breed) {
        // Call the parent class's constructor using 'super'
        super(name);
        this.breed = breed; // Additional property specific to Dog
    }

    // Method: overrides the speak method from the Animal class
    speak() {
        console.log(`${this.name} barks.`);
    }
}

// Create an object (instance) of the Dog class
const myDog = new Dog('Buddy', 'Golden Retriever');

// Access the properties of the object
console.log(myDog.name);  // Output: Buddy
console.log(myDog.breed); // Output: Golden Retriever

// Call the method of the object
myDog.speak(); // Output: Buddy barks.
```

**Polymorphism:**
Polymorphism means "many shapes" and allows objects of different classes to be treated as objects of a common superclass. It allows a single interface to represent different underlying data types. For example, a `Device` class might have a method `turnOn()`, and the `Laptop` and `Smartphone` classes, which inherit from `Device`, might implement `turnOn()` differently. However, you can still call `turnOn()` on any `Device` object, whether it's a laptop or a smartphone.

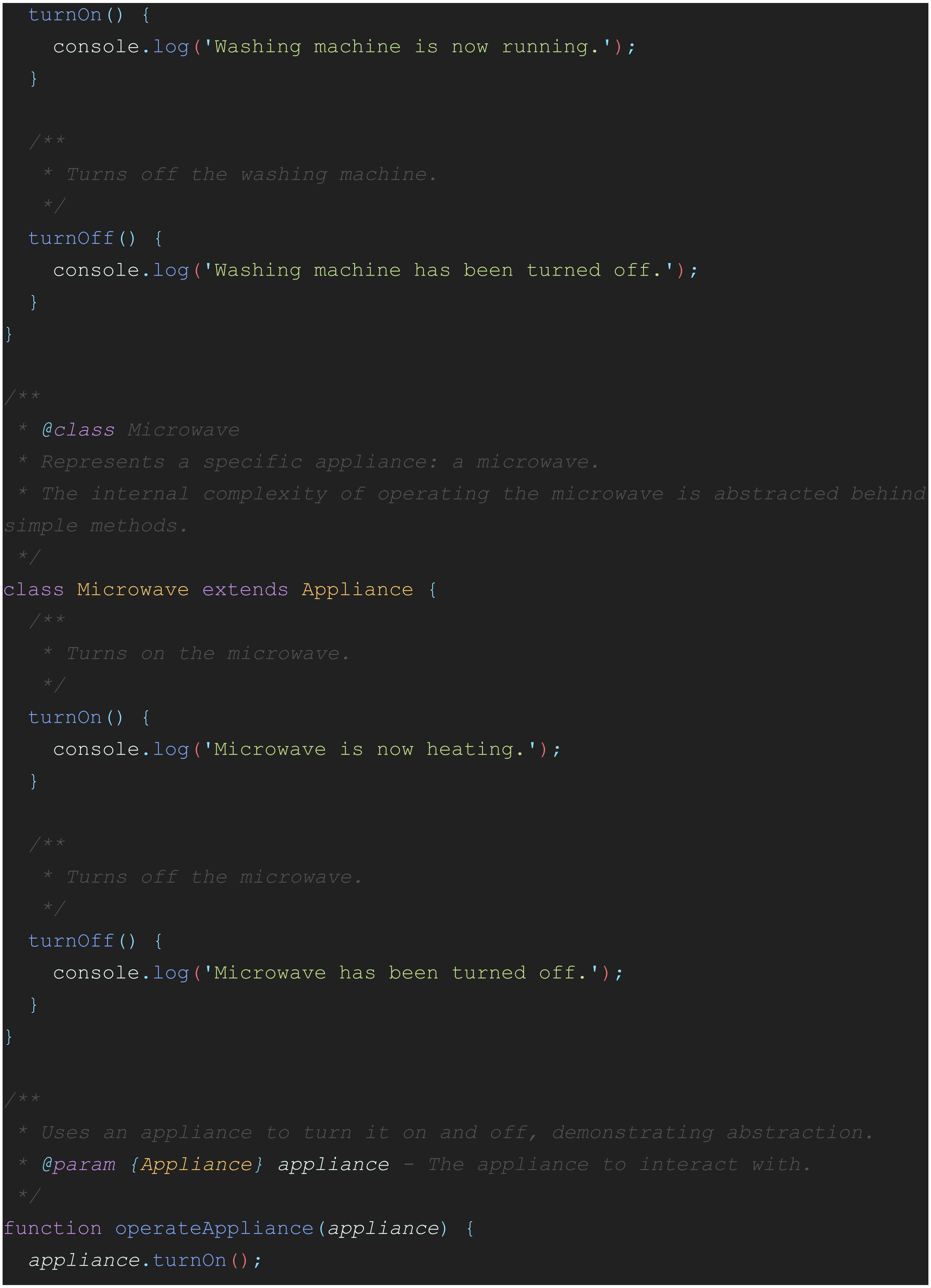
```
/**
 * @class Device
 * Represents a general device.
 */
class Device {
  /**
   * Turns on the device. This method is meant to be overridden.
   * @abstract
   */
  turnOn() {
    throw new Error('Method turnOn() must be implemented');
  }
}
/**
 * @class Laptop
 * Represents a laptop device that inherits from Device.
 */
```



```
class Laptop extends Device {
  /**
   * Overrides the turnOn method to turn on a laptop.
   */
  turnOn() {
    console.log('Laptop is now powered on.');
  }
}

/**
 * @class Smartphone
 * Represents a smartphone device that inherits from Device.
 */
class Smartphone extends Device {
  /**
   * Overrides the turnOn method to turn on a smartphone.
   */
  turnOn() {
    console.log('Smartphone is now powered on.');
  }
}

/**
 * Turns on any device.
 * @param {Device} device - The device to turn on.
 */
function powerOnDevice(device) {
  device.turnOn();
}

// Instantiate different devices
const myLaptop = new Laptop();
const mySmartphone = new Smartphone();

// Using polymorphism to call the same method on different devices
powerOnDevice(myLaptop); // Output: Laptop is now powered on.
powerOnDevice(mySmartphone); // Output: Smartphone is now powered on.
```

5. **Abstraction:** Abstraction means hiding the complex reality while exposing only the essential parts. It reduces programming complexity and effort by hiding the unnecessary details from the user. For example, when you use a smartphone, you interact with its interface (the screen), without needing to understand the complex hardware and software working inside the device.

```
/**
 * @class Appliance
 * Represents an abstract appliance.
 * This class hides the complexity of internal appliance workings and exposes essential methods.
 */
class Appliance {
  /**
   * Turns on the appliance. This method abstracts the internal workings.
   * @abstract
   */
  turnOn() {
    throw new Error('Method turnOn() must be implemented');
  }

  /**
   * Turns off the appliance. This method abstracts the internal workings.
   * @abstract
   */
  turnOff() {
    throw new Error('Method turnOff() must be implemented');
  }
}

/**
 * @class WashingMachine
 * Represents a specific appliance: a washing machine.
 * The internal logic of operating a washing machine is abstracted behind simple methods.
 */
class WashingMachine extends Appliance {
  /**
   * Turns on the washing machine.
   */
```



```
  turnOn() {
    console.log('Washing machine is now running.');
  }

  /**
   * Turns off the washing machine.
   */
  turnOff() {
    console.log('Washing machine has been turned off.');
  }
}

/**
 * @class Microwave
 * Represents a specific appliance: a microwave.
 * The internal complexity of operating the microwave is abstracted behind simple methods.
 */
class Microwave extends Appliance {
  /**
   * Turns on the microwave.
   */
  turnOn() {
    console.log('Microwave is now heating.');
  }

  /**
   * Turns off the microwave.
   */
  turnOff() {
    console.log('Microwave has been turned off.');
  }
}

/**
 * Uses an appliance to turn it on and off, demonstrating abstraction.
 * @param {Appliance} appliance - The appliance to interact with.
 */
function operateAppliance(appliance) {
  appliance.turnOn();
  // Simulate some operation
  appliance.turnOff();
}

// Instantiate different appliances
const myWashingMachine = new WashingMachine();
const myMicrowave = new Microwave();

// Using abstraction to operate appliances without knowing internal details
operateAppliance(myWashingMachine); // Output: Washing machine is now running. Washing machine has been turned off.
operateAppliance(myMicrowave);      // Output: Microwave is now heating. Microwave has been turned off.
```

**Why Do We Use OOP?**

1. **Improved Code Organization:** OOP helps in organizing the code better by grouping related properties and methods into objects. This makes the code easier to understand, manage, and debug.

2. **Reusability:** Through inheritance, you can create new classes that reuse, extend, or modify the behavior defined in other classes. This avoids code duplication and enhances maintainability.

3. **Modularity:** By encapsulating the related data and behavior into objects, you make the code modular. You can independently develop, test, and maintain different parts of an application.

4. **Scalability:** OOP makes it easier to scale applications. As applications grow, the modular structure of OOP helps in managing the increasing complexity. You can add new functionality with minimal changes to existing code.

5. **Flexibility Through Polymorphism:** With polymorphism, you can write code that works on objects of different classes but shares a common superclass. This flexibility makes it easier to introduce new classes without modifying existing code.

6. **Abstraction for Simplicity:** By using abstraction, OOP allows programmers to focus on what the object does instead of how it does it. This simplifies programming and helps in managing complexity.

## Conclusion

Object-Oriented Programming is a powerful paradigm that can help you write more structured, reusable, and maintainable code. By thinking in terms of objects and their interactions, you can model real-world systems more naturally and handle the complexity of large applications more effectively. As we dive deeper into OOP in JavaScript, you'll see how these concepts come to life and how they can be applied to solve real-world problems in a more elegant way.