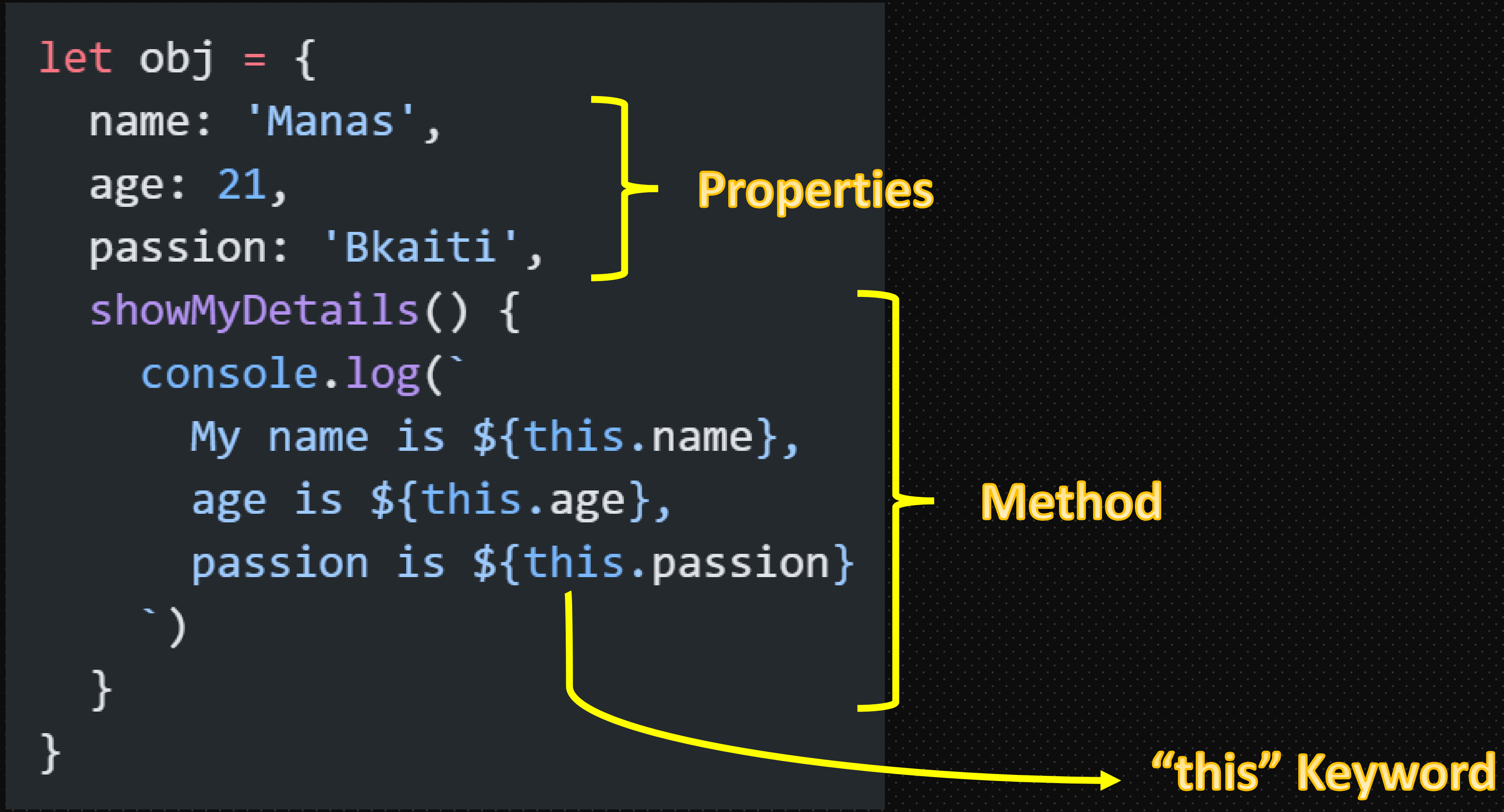
Build Some Base



```
let obj = {
  name: 'Manas',
  age: 21,
  passion: 'Bkaiti',
  showMyDetails() {
    console.log(`
      My name is ${this.name},
      age is ${this.age},
      passion is ${this.passion}
    `)
  }
}
```

Properties

Method

“this” Keyword

The different ways to create and use objects in JavaScript — these are the foundations for understanding OOP in JS:

1. Object Literal

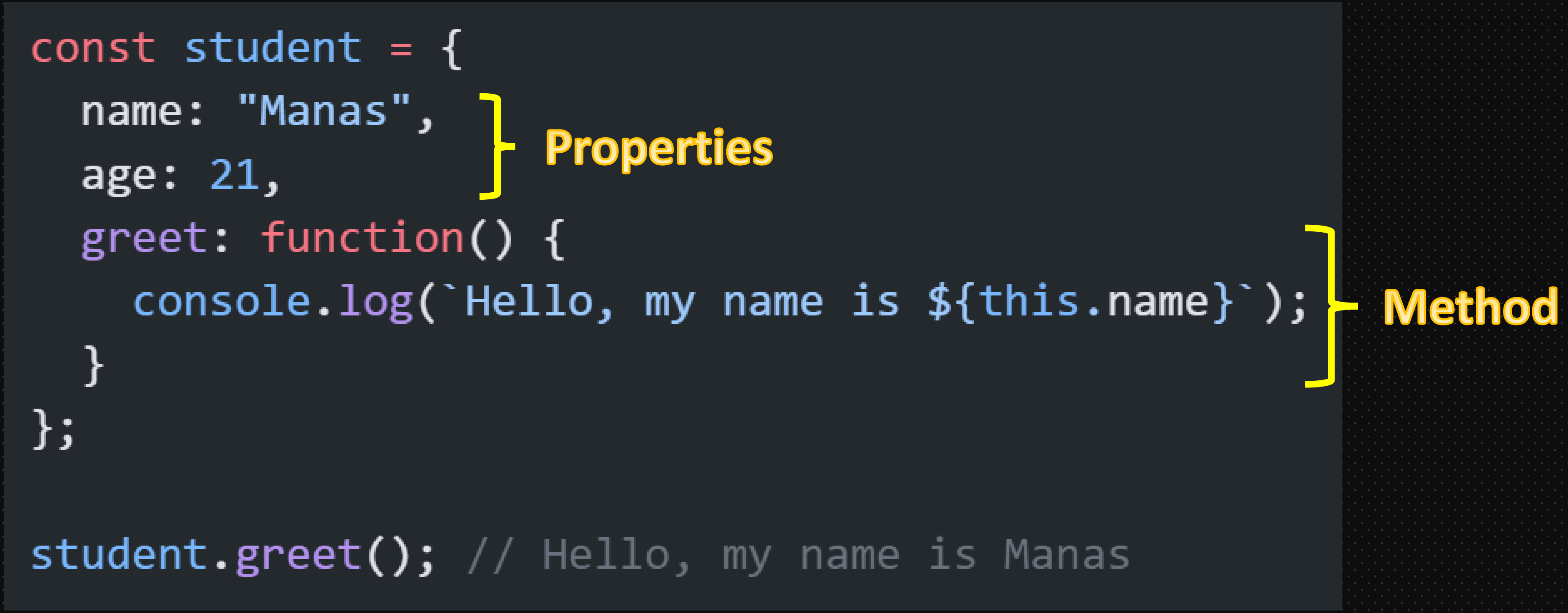
2. Factory Function

3. Constructor Function

4. Class Syntax (ES6)

1. Object Literal

- ❑ Simplest and most common way to create an object.
- ❑ Used when creating a single, specific object.

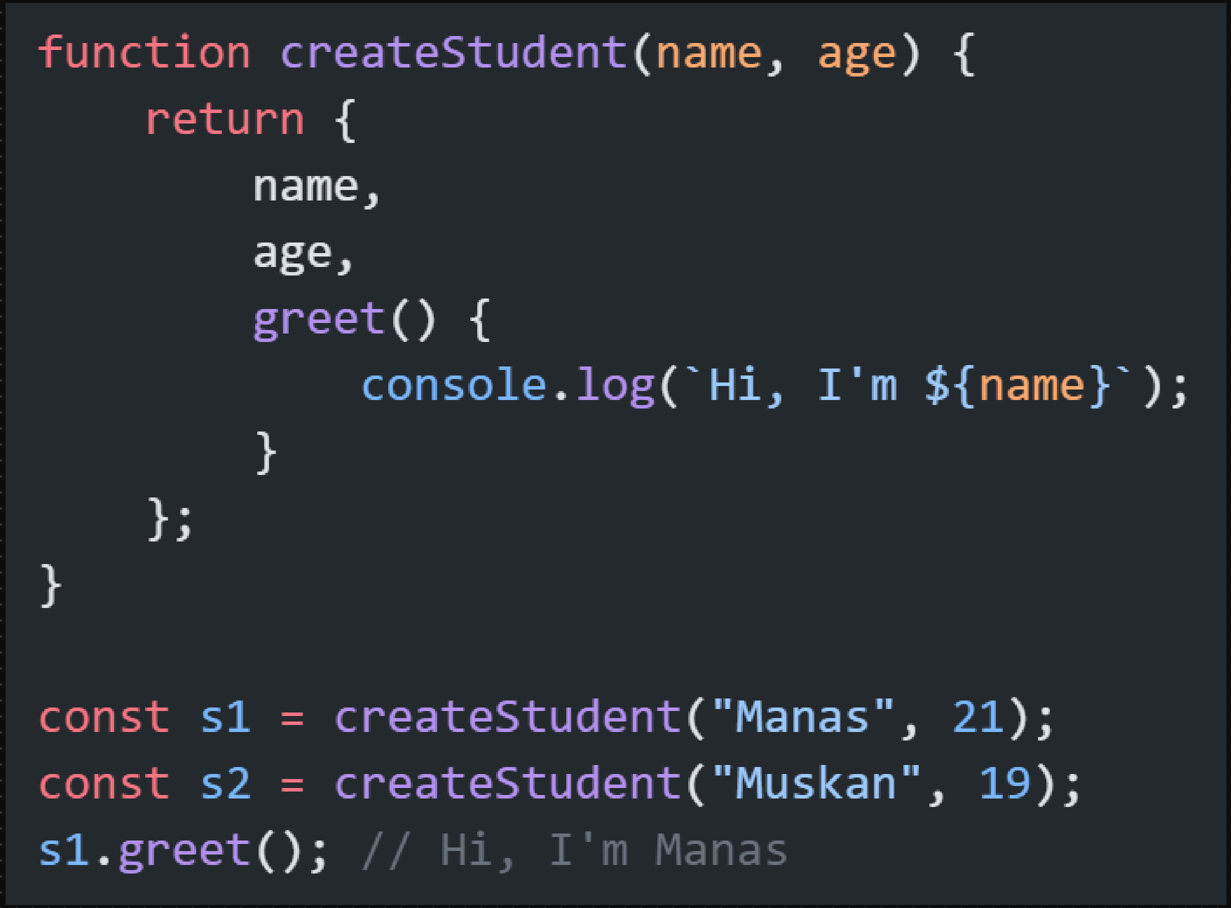


```javascript
const student = {
  name: "Manas",
  age: 21,
  greet: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

student.greet(); // Hello, my name is Manas
```

2. Factory Function

❑ A function that returns a new object.

❑ Great for creating multiple similar objects without classes.



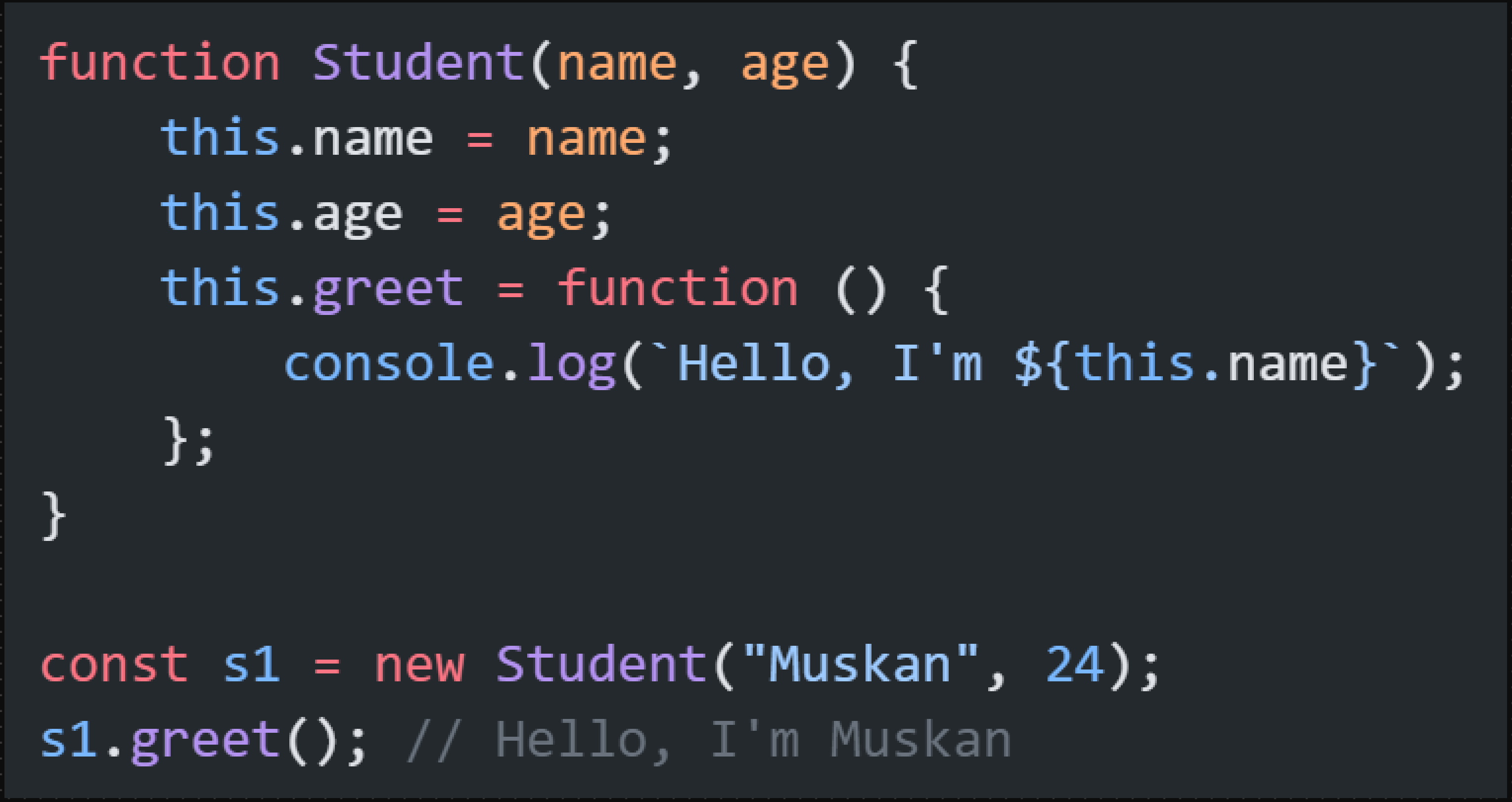
```javascript
function createStudent(name, age) {
    return {
        name,
        age,
        greet() {
            console.log(`Hi, I'm ${name}`);
        }
    };
}

const s1 = createStudent("Manas", 21);
const s2 = createStudent("Muskan", 19);
s1.greet(); // Hi, I'm Manas
```

Doesn’t involve prototypes by default (unless you manually set them).

3. Constructor Function

□ Uses the new keyword.

□ Before class syntax was introduced in ES6, this was the standard way to create "object blueprints."



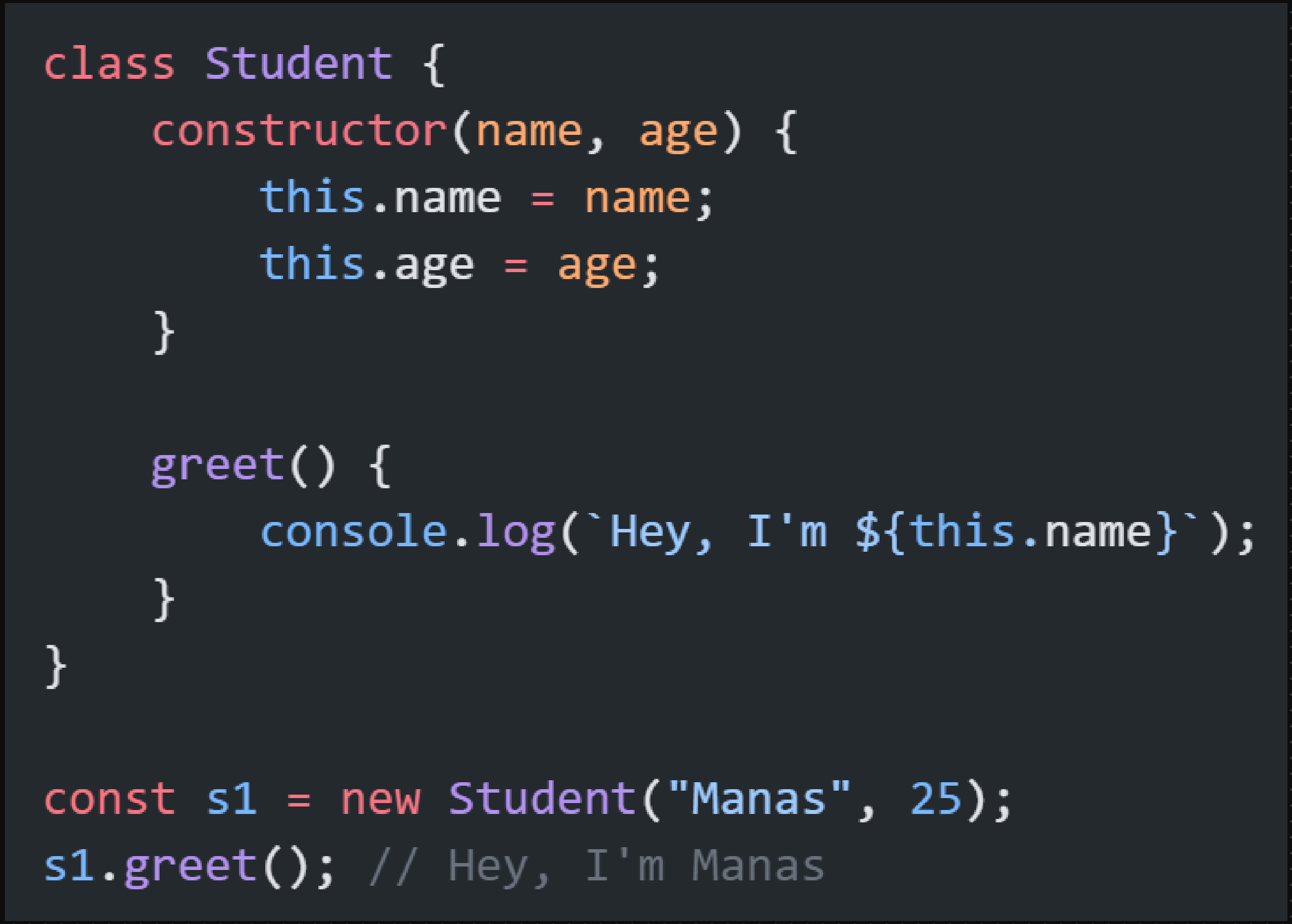
```
function Student(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function () {
        console.log(`Hello, I'm ${this.name}`);
    };
}

const s1 = new Student("Muskan", 24);
s1.greet(); // Hello, I'm Muskan
```

Automatically sets up a link to Student.prototype.

4. Class Syntax (ES6)

□ A modern, cleaner syntax for creating constructor functions.

□ Internally still works like constructor functions.



```javascript
class Student {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log(`Hey, I'm ${this.name}`);
    }
}

const s1 = new Student("Manas", 25);
s1.greet(); // Hey, I'm Manas
```

“this” keyword

❑ Its value depends on how the function or method is called.

❑ In OOP, this refers to the object that is calling the method.

❑ It's used to access the current instance's properties or methods.

❑ Arrow functions don’t have their own this — they inherit this from the surrounding (lexical) scope.

“new” keyword

❑ In JavaScript, the new keyword is used to create an instance of an object from a constructor function or class.

❑ It's like saying: “Make me a new object from this blueprint (function or class).”

prototype:

❑ In JavaScript OOP, it allows us to share methods between all instances of a class or constructor function, making code memory-efficient.

❑ JavaScript is a prototypal-based (or prototype-based) language.

How it works?

❑ Every object has an internal link to another object called its prototype.

❑ When you access a property or method, JavaScript looks for it in the object.

  ○ If not found, it climbs the prototype chain to find it.

🤔 So, what about class in JS?

❑ JavaScript introduced the class keyword in ES6, but:

  ○ class is just syntactic sugar — underneath, it's still using prototypes.

Object Oriented Programming:

Object-Oriented Programming (OOPs) in JavaScript is a programming paradigm based on the concept of objects.

These objects encapsulate both data (attributes) and the functions that operate on that data (methods).

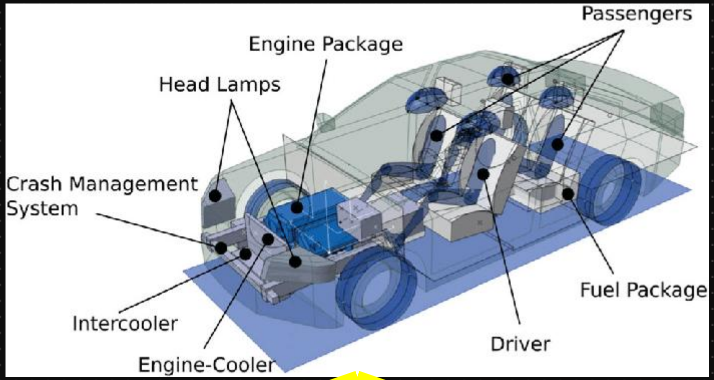
JavaScript, while not a purely class-based language like Java or C++, is heavily object-oriented and supports OOP principles through its prototype-based model and class syntax.

Class :

A Blueprint or Template, encapsulates data (properties) and functions (methods)

Object:

instance of a class, Each object has its own unique set of values for its properties.

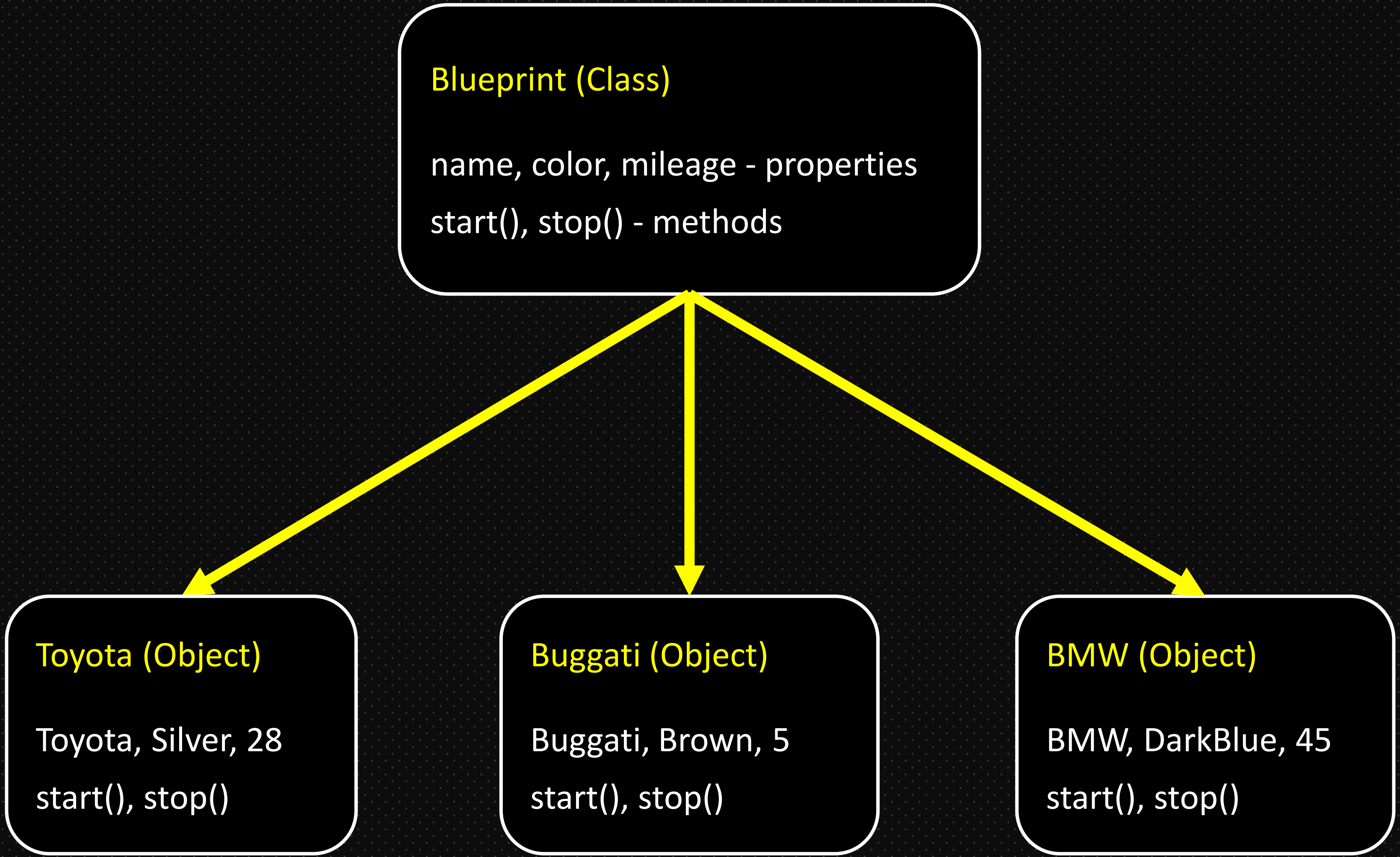


Blueprint (Class)

Toyota (Object)

Bugatti (Object)

BMW (Object)



Blueprint (Class)

name, color, mileage - properties

start(), stop() - methods

Toyota (Object)

Toyota, Silver, 28

start(), stop()

Buggati (Object)

Buggati, Brown, 5

start(), stop()

BMW (Object)

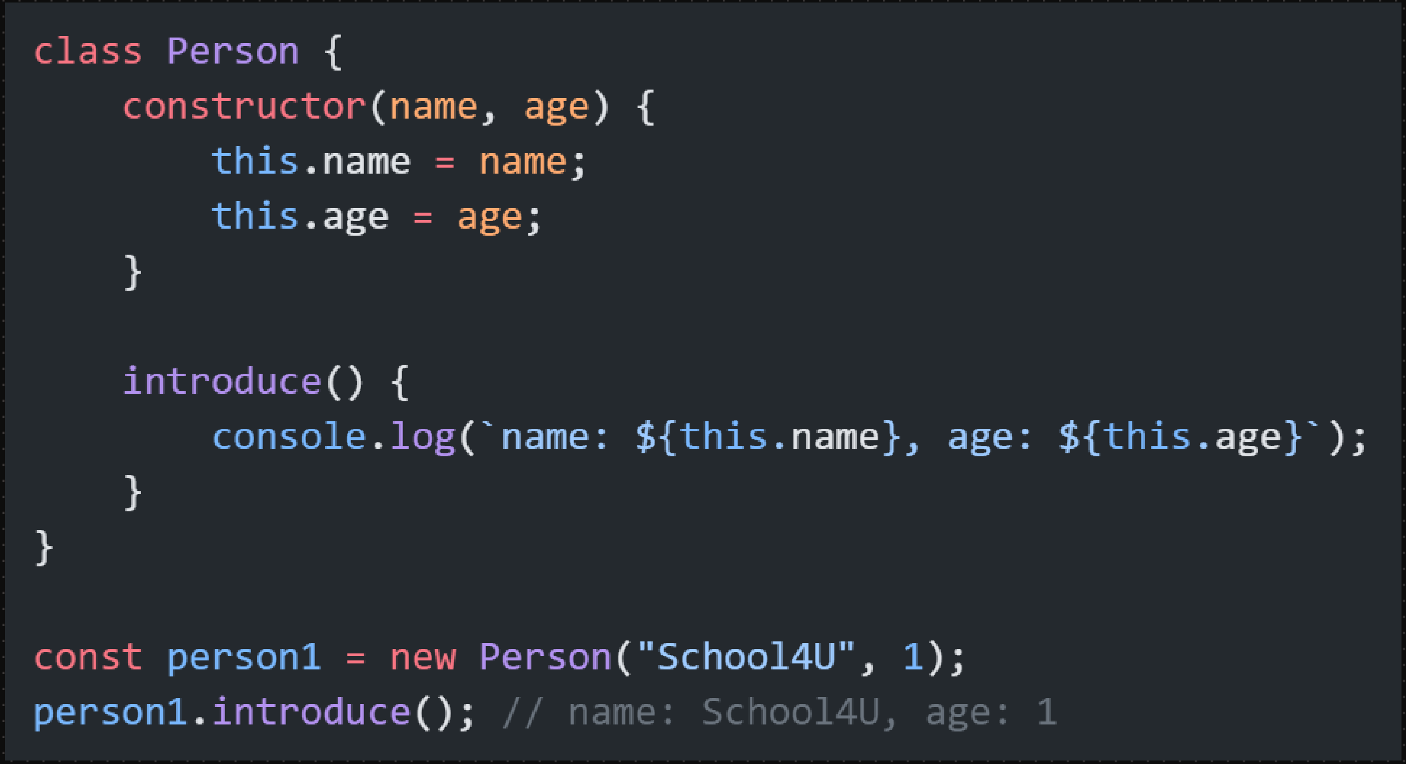
BMW, DarkBlue, 45

start(), stop()

Constructor

❑ A constructor is a special method within a class that is automatically called when a new object instance of that class is created.

❑ It is primarily used to initialize object properties with specific values or perform setup tasks for the object.



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

    introduce() {
        console.log(`name: ${this.name}, age: ${this.age}`);
    }
}

const person1 = new Person("School4U", 1);
person1.introduce(); // name: School4U, age: 1
```

Key characteristics of constructors:

❑ Purpose: To create and initialize objects.

❑ Automatic invocation: Called automatically when an object is created using the new keyword.

❑ Initialization: Sets initial values for object properties.

❑ Implicit constructor: If a class does not have a constructor, JavaScript provides a default empty constructor.

❑ Derived class constructor: If a derived class does not have a constructor, it calls the parent constructor passing along any arguments.

Four pillars of OOP:

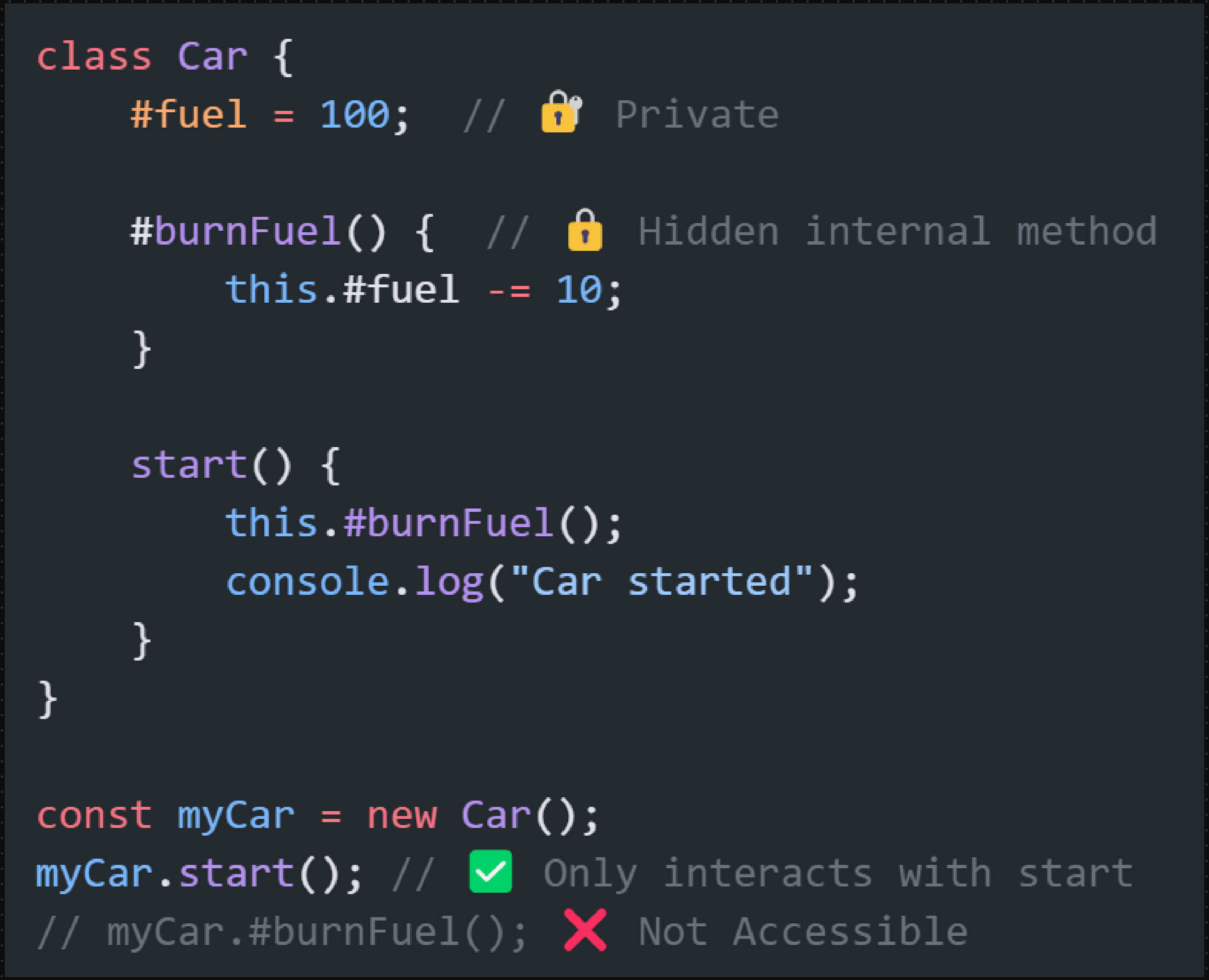
- Abstraction – hiding complexity and showing only the essential features.

- Encapsulation – hiding data inside objects and provide security.

- Inheritance – using properties and methods from another object/class.

- Polymorphism – same method behaving differently based on the object.

Abstraction:



```javascript
class Car {
    #fuel = 100;  // 🔐 Private

    #burnFuel() {  // 🔒 Hidden internal method
        this.#fuel -= 10;
    }

    start() {
        this.#burnFuel();
        console.log("Car started");
    }
}

const myCar = new Car();
myCar.start(); // ✅ Only interacts with start
// myCar.#burnFuel(); ❌ Not Accessible
```

Abstraction means hiding complex implementation details and showing only the essential features to the user.

Goal: Hiding complexity (what is irrelevant).

Encapsulation:

❑ It means wrapping data (properties) and methods (functions) together into a single unit, usually a class or an object, and restricting direct access to some of the components.

Why Encapsulation?

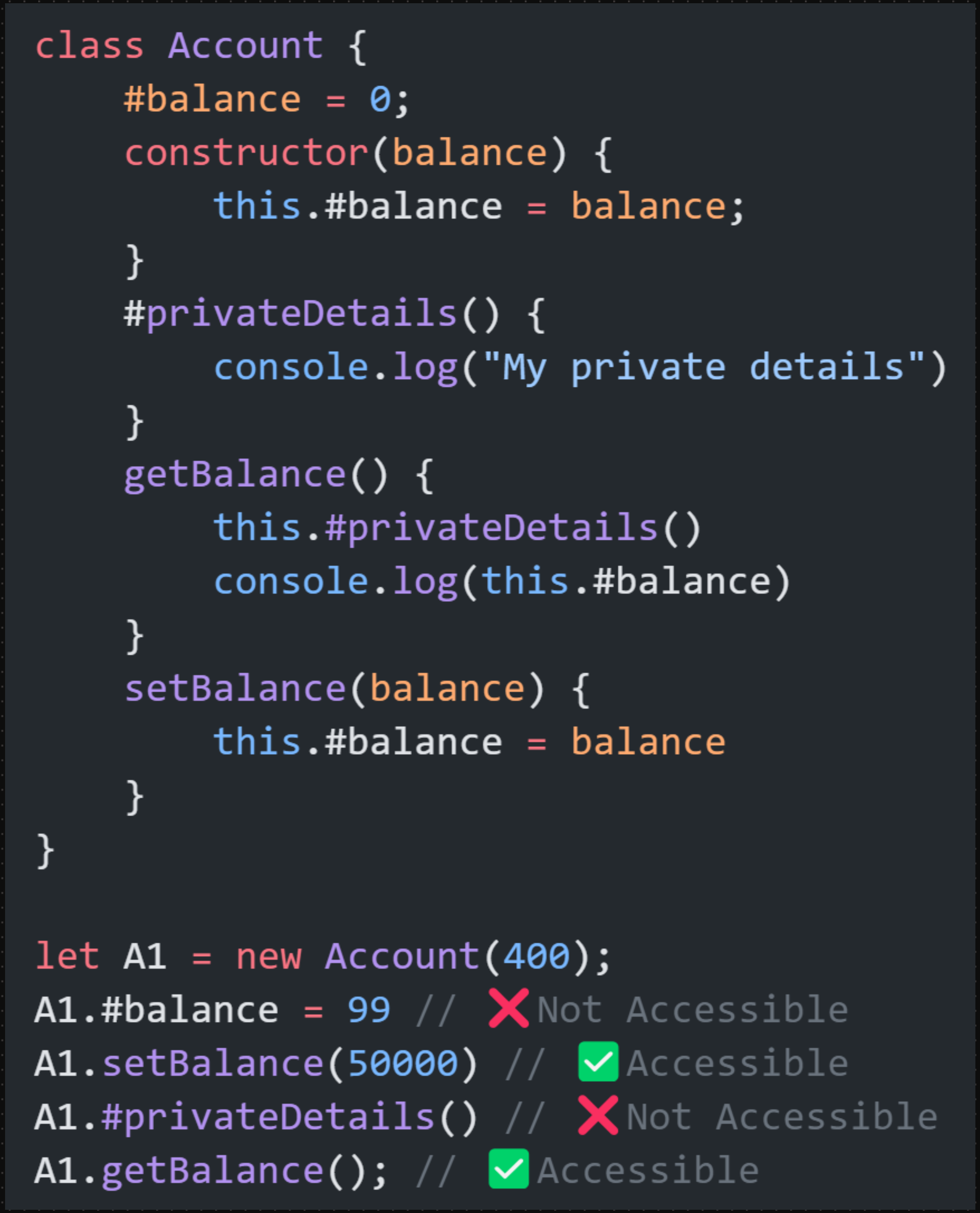
❑ Protects data from unauthorized access

❑ Prevents misuse of code

❑ Makes code easier to maintain

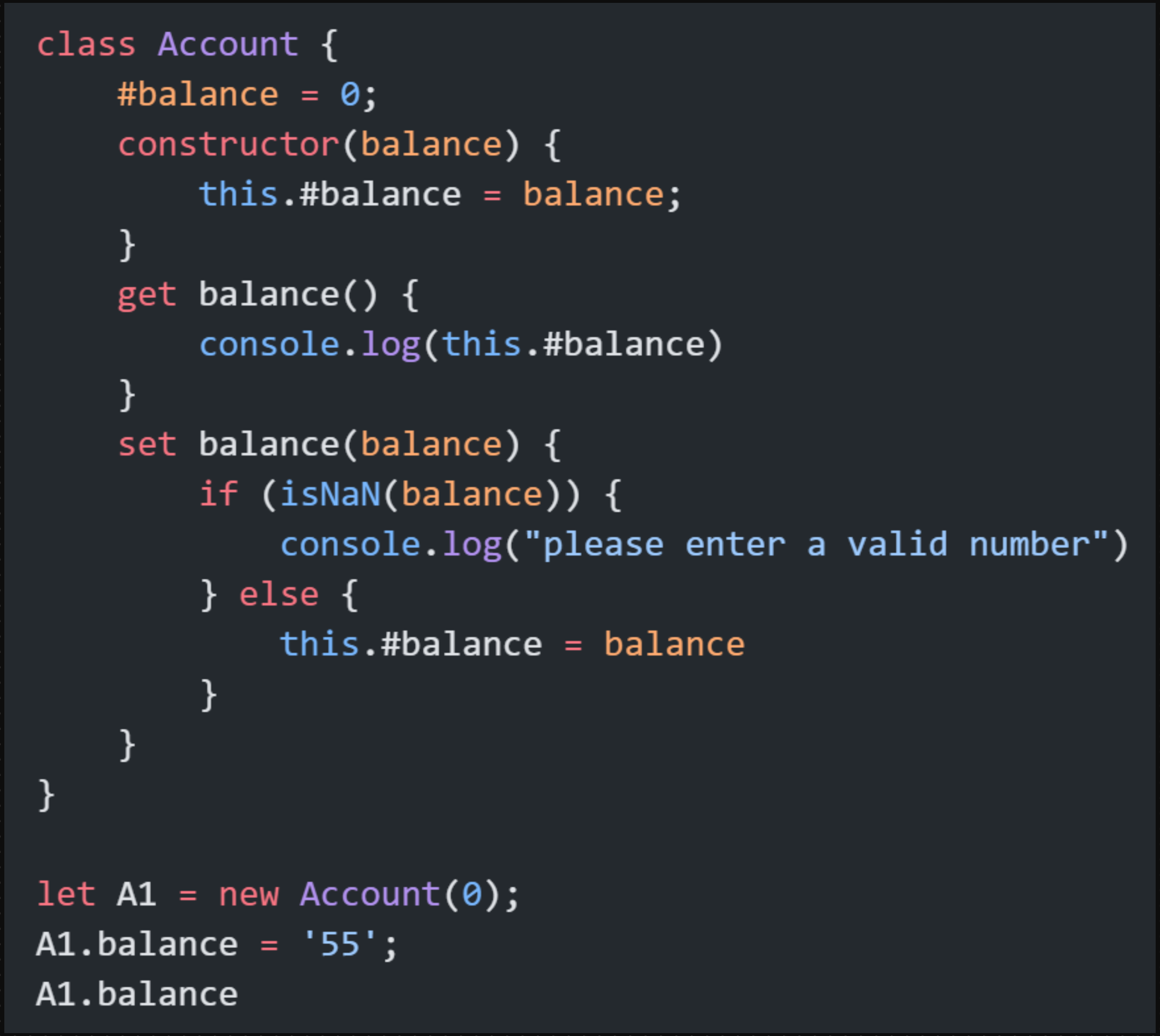
❑ Supports data hiding

Goal: Hide internal details and only expose what's necessary.



```
class Account {
    #balance = 0;
    constructor(balance) {
        this.#balance = balance;
    }
    #privateDetails() {
        console.log("My private details")
    }
    getBalance() {
        this.#privateDetails()
        console.log(this.#balance)
    }
    setBalance(balance) {
        this.#balance = balance
    }
}

let A1 = new Account(400);
A1.#balance = 99 // ❌Not Accessible
A1.setBalance(50000) // ✅Accessible
A1.#privateDetails() // ❌Not Accessible
A1.getBalance(); // ✅Accessible
```

- ❑ Encapsulation hides internal details
- ❑ Use # for private class fields
- ❑ Use getters/setters for controlled access



```
class Account {
    #balance = 0;
    constructor(balance) {
        this.#balance = balance;
    }
    get balance() {
        console.log(this.#balance)
    }
    set balance(balance) {
        if (isNaN(balance)) {
            console.log("please enter a valid number")
        } else {
            this.#balance = balance
        }
    }
}

let A1 = new Account(0);
A1.balance = '55';
A1.balance
```

get and set:

❑ They allow you to control how a property is read or written — like a security gate for your variables.

❑ You can check values before setting them

❑ Hide sensitive data

❑ Access methods like regular properties (obj.name)

Abstraction v/s Encapsulation:

Concept	What It Hides	What It Shows
Abstraction	The *process* / logic	A *simple interface*
Encapsulation	The *data* / internal state	Only what’s allowed to access

❑ Use abstraction to make the system easy to use.

❑ Use encapsulation to make the system safe and secure.

Inheritance:

❑ Inheritance is an OOP concept where one class (child) can acquire properties and methods of another class (parent).

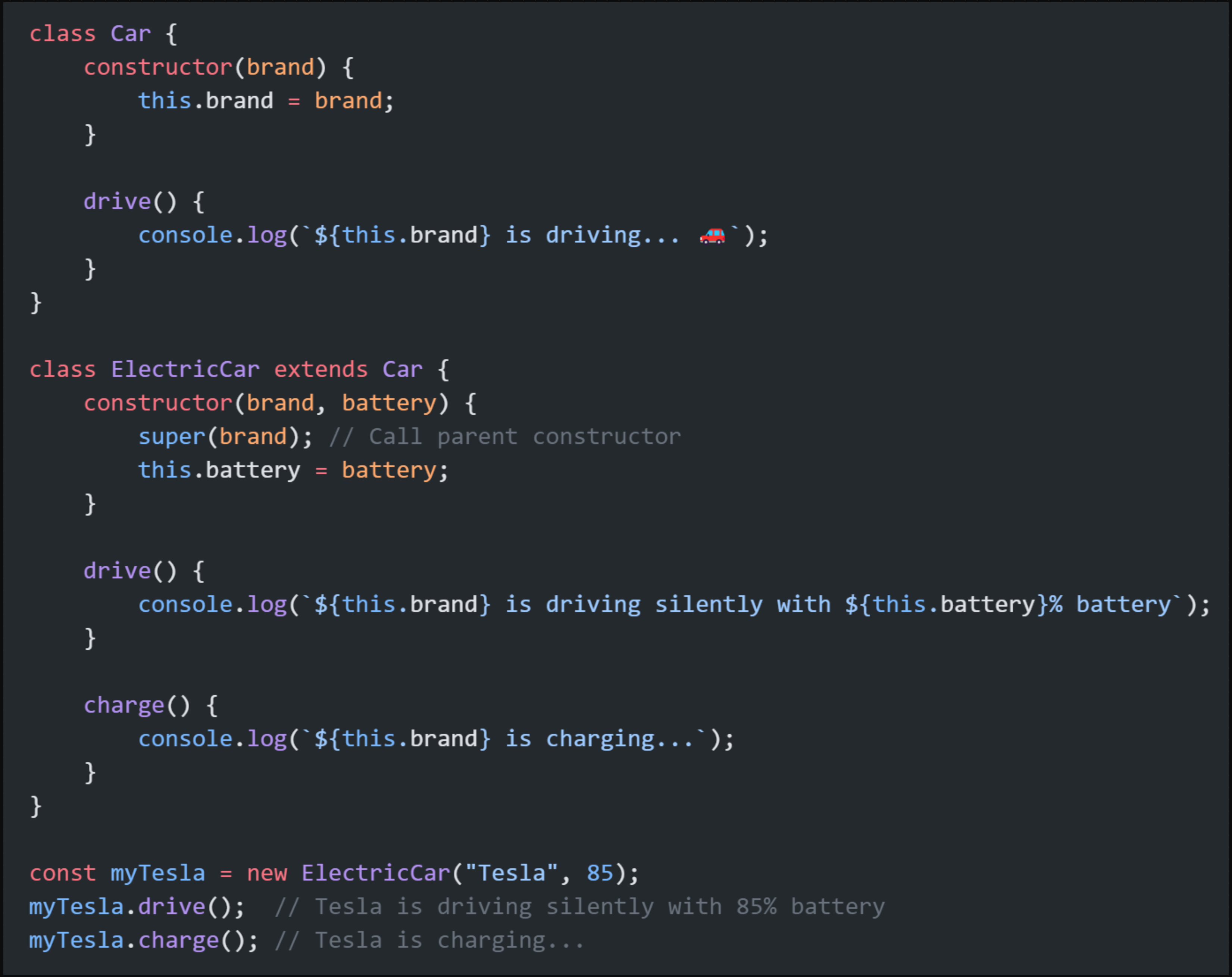
Why Use Inheritance?

❑ Reuse existing code

❑ Create logical relationships (is-a)

❑ Reduce duplication

❑ Easier maintenance and scalability



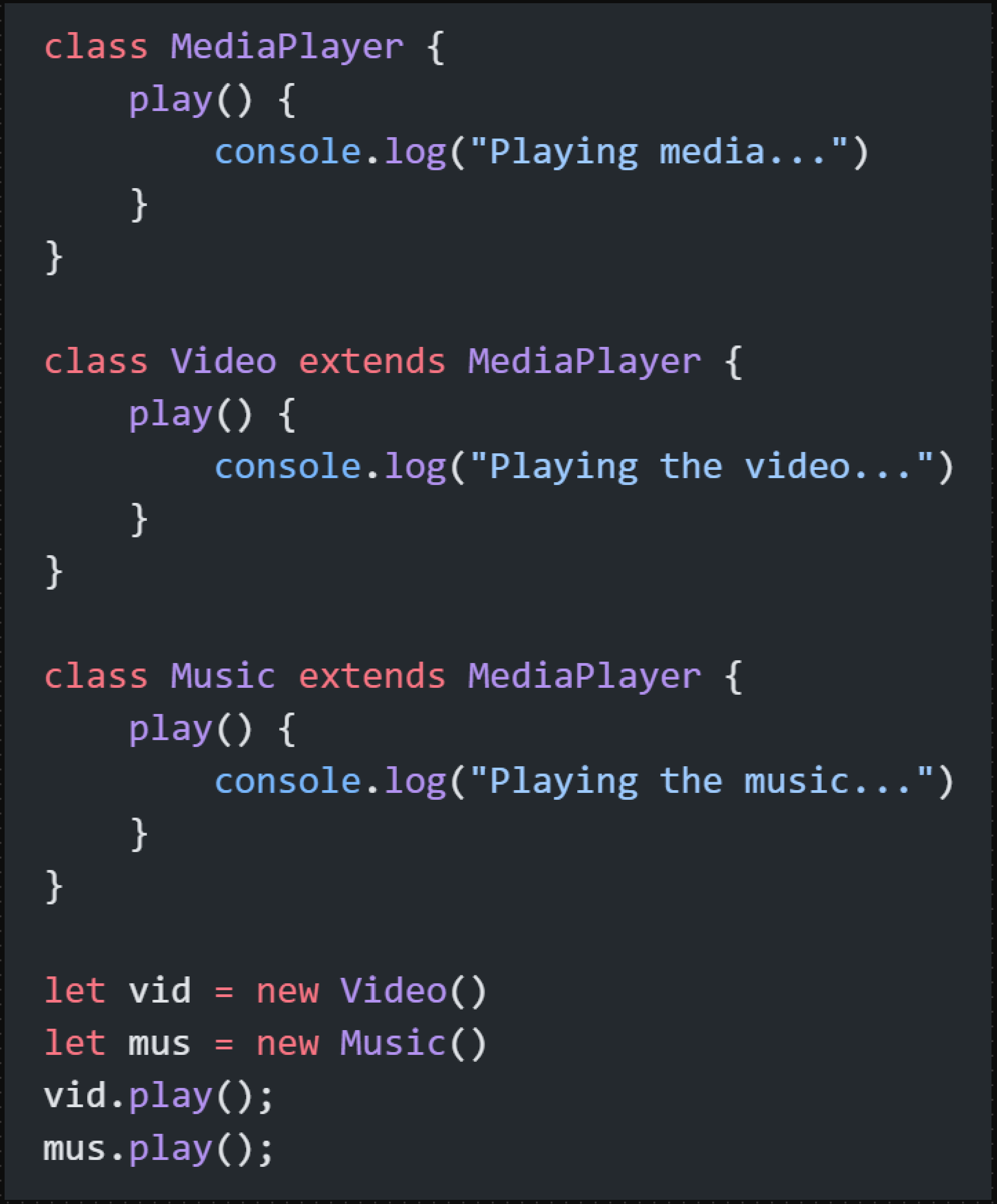
```javascript
class Car {
    constructor(brand) {
        this.brand = brand;
    }

    drive() {
        console.log(`${this.brand} is driving... 🚗`);
    }
}

class ElectricCar extends Car {
    constructor(brand, battery) {
        super(brand); // Call parent constructor
        this.battery = battery;
    }

    drive() {
        console.log(`${this.brand} is driving silently with ${this.battery}% battery`);
    }

    charge() {
        console.log(`${this.brand} is charging...`);
    }
}

const myTesla = new ElectricCar("Tesla", 85);
myTesla.drive();  // Tesla is driving silently with 85% battery
myTesla.charge(); // Tesla is charging...
```

Polymorphism:

- ❑ Poly = many, morph = forms, Polymorphism = many forms
- ❑ It allows different classes to define methods with the same name but different behavior. (or we can say that has more than one form)

Imagine a play() button:

- ❑ On a video, it plays the video
- ❑ On a music player, it plays the music
- ❑ On a game, it starts the game



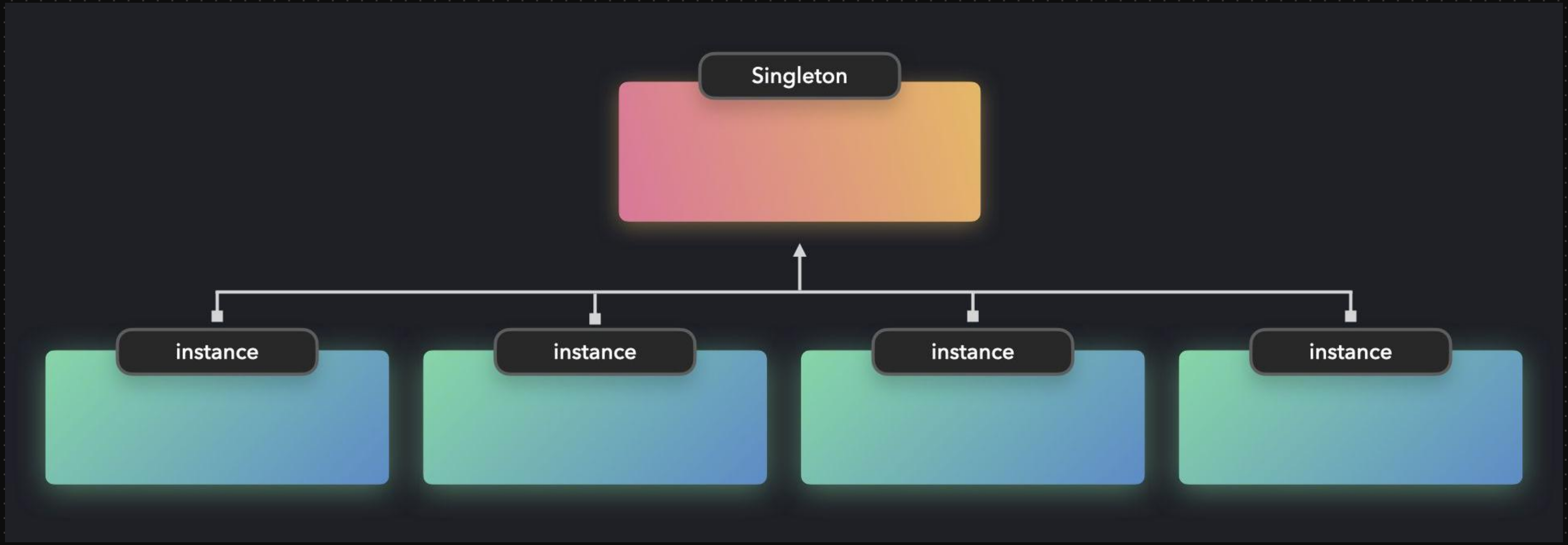
```
class MediaPlayer {
    play() {
        console.log("Playing media...")
    }
}

class Video extends MediaPlayer {
    play() {
        console.log("Playing the video...")
    }
}

class Music extends MediaPlayer {
    play() {
        console.log("Playing the music...")
    }
}

let vid = new Video()
let mus = new Music()
vid.play();
mus.play();
```

Both Video and Music override the play() method from MediaPlayer.

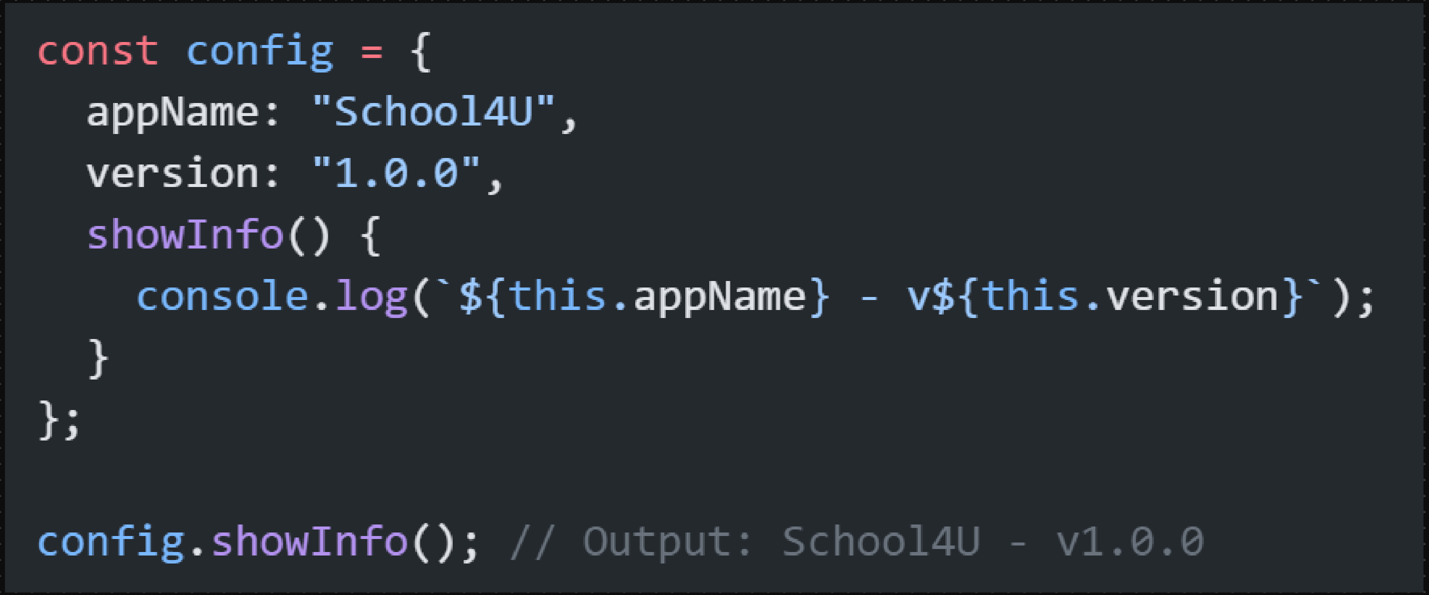
Singleton Object

- ❑ A Singleton Object is an object that is created only once and used everywhere in your code.

- ❑ It ensures that only one instance of that object exists during the lifetime of the application.



Example 1: Object Literal (Most Basic Singleton)



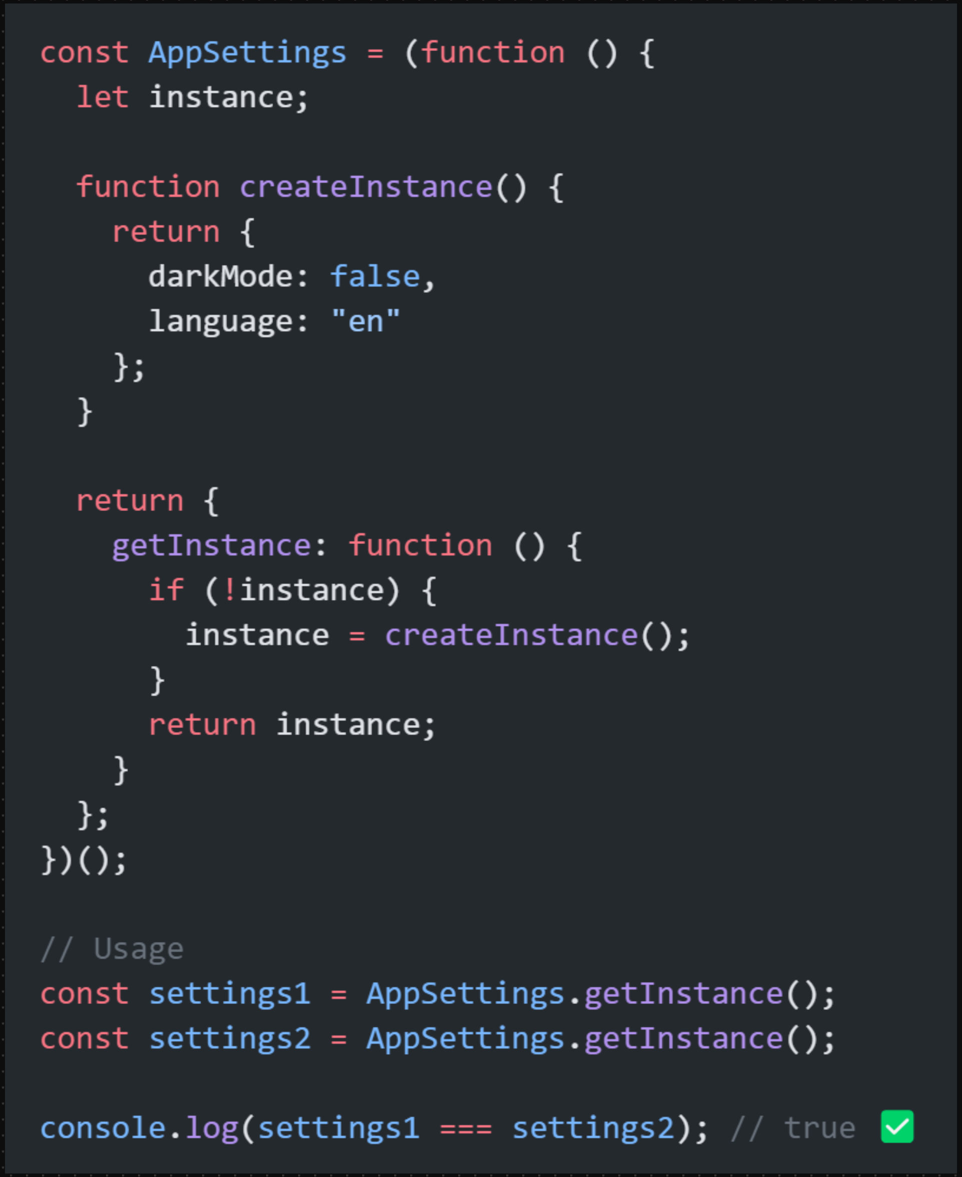
```
const config = {
  appName: "School4U",
  version: "1.0.0",
  showInfo() {
    console.log(`${this.appName} - v${this.version}`);
  }
};

config.showInfo(); // Output: School4U - v1.0.0
```

❑ config is a singleton object created using object literal {}.

❑ You can't accidentally create another version of it.

❑ You reuse the same config object wherever needed.

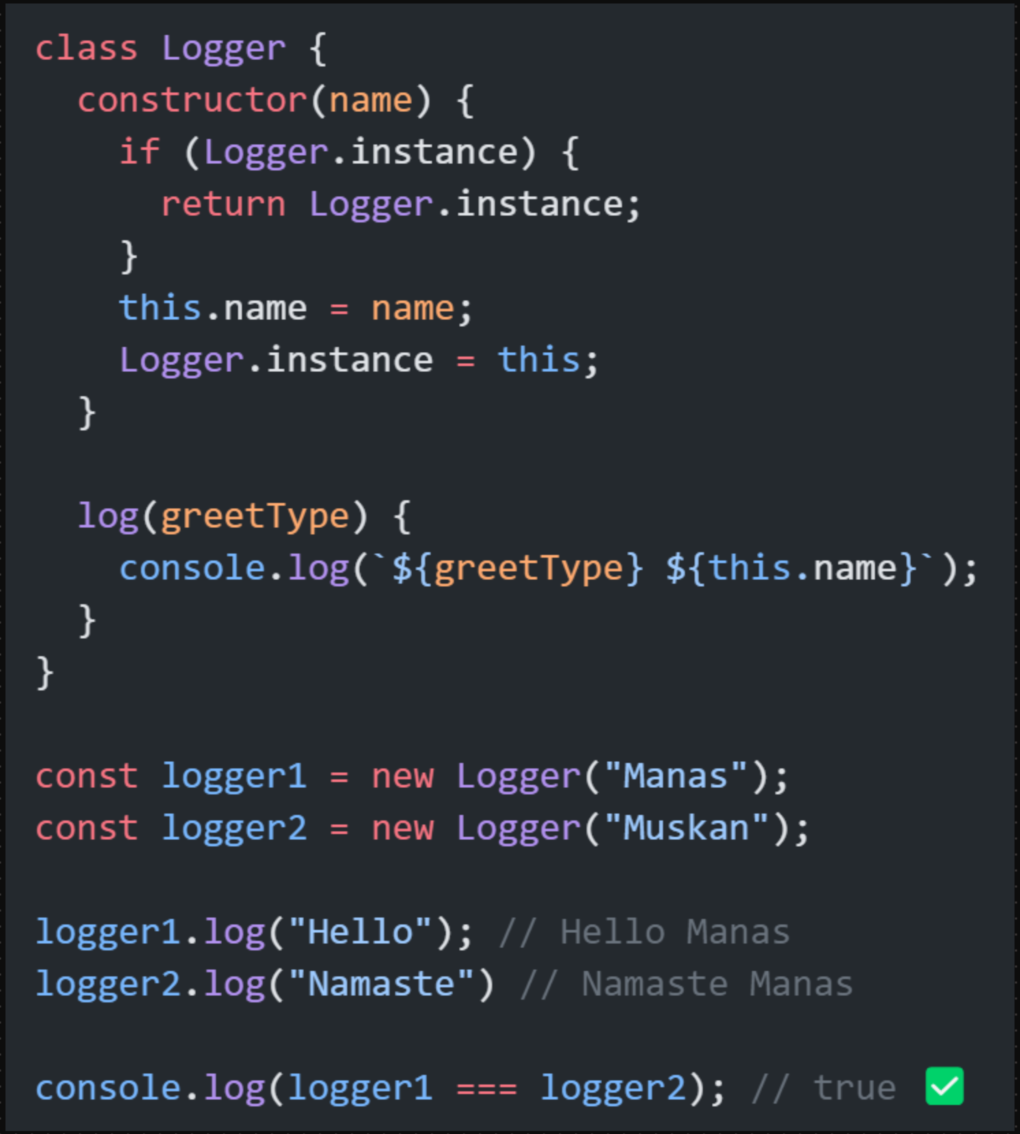
Example 2: Singleton Using Function (Closure)



```javascript
const AppSettings = (function () {
  let instance;

  function createInstance() {
    return {
      darkMode: false,
      language: "en"
    };
  }

  return {
    getInstance: function () {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

// Usage
const settings1 = AppSettings.getInstance();
const settings2 = AppSettings.getInstance();

console.log(settings1 === settings2); // true ✅
```

- □ AppSettings is a self-invoking function that returns an object with a getInstance() method.
- □ The instance is created only once, then reused.
- □ Both settings1 and settings2 are same object.

Example 3: Singleton with Class (ES6 Style)



```
class Logger {
  constructor(name) {
    if (Logger.instance) {
      return Logger.instance;
    }
    this.name = name;
    Logger.instance = this;
  }

  log(greetType) {
    console.log(`${greetType} ${this.name}`);
  }
}

const logger1 = new Logger("Manas");
const logger2 = new Logger("Muskan");

logger1.log("Hello"); // Hello Manas
logger2.log("Namaste") // Namaste Manas

console.log(logger1 === logger2); // true ✅
```

□ In this class, we store the first created instance as Logger.instance.

□ If another object is created using new Logger(), it will return the same instance.

□ It prevents creating multiple copies.

Why use Singleton?

- To avoid multiple copies of the same object.

- To maintain a single shared state.

- Useful for things like:

    - App settings
    - Database Connections
    - Authentication state
    - Logger services