OOPs

JavaScript

What is OOPs

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of "objects".

- which can contain data (in the form of "properties") and code (in the form of "methods").
- OOP is a popular programming paradigm because it allows for modular, reusable code that can be easier to read, maintain, and scale.
- There are two types of OOP languages:
  - Class-Based languages like JAVA, C++.
  - Prototype-Based languages like JS.

Features Of OOPs

There are four rules or main pillars of Object-oriented programming language.

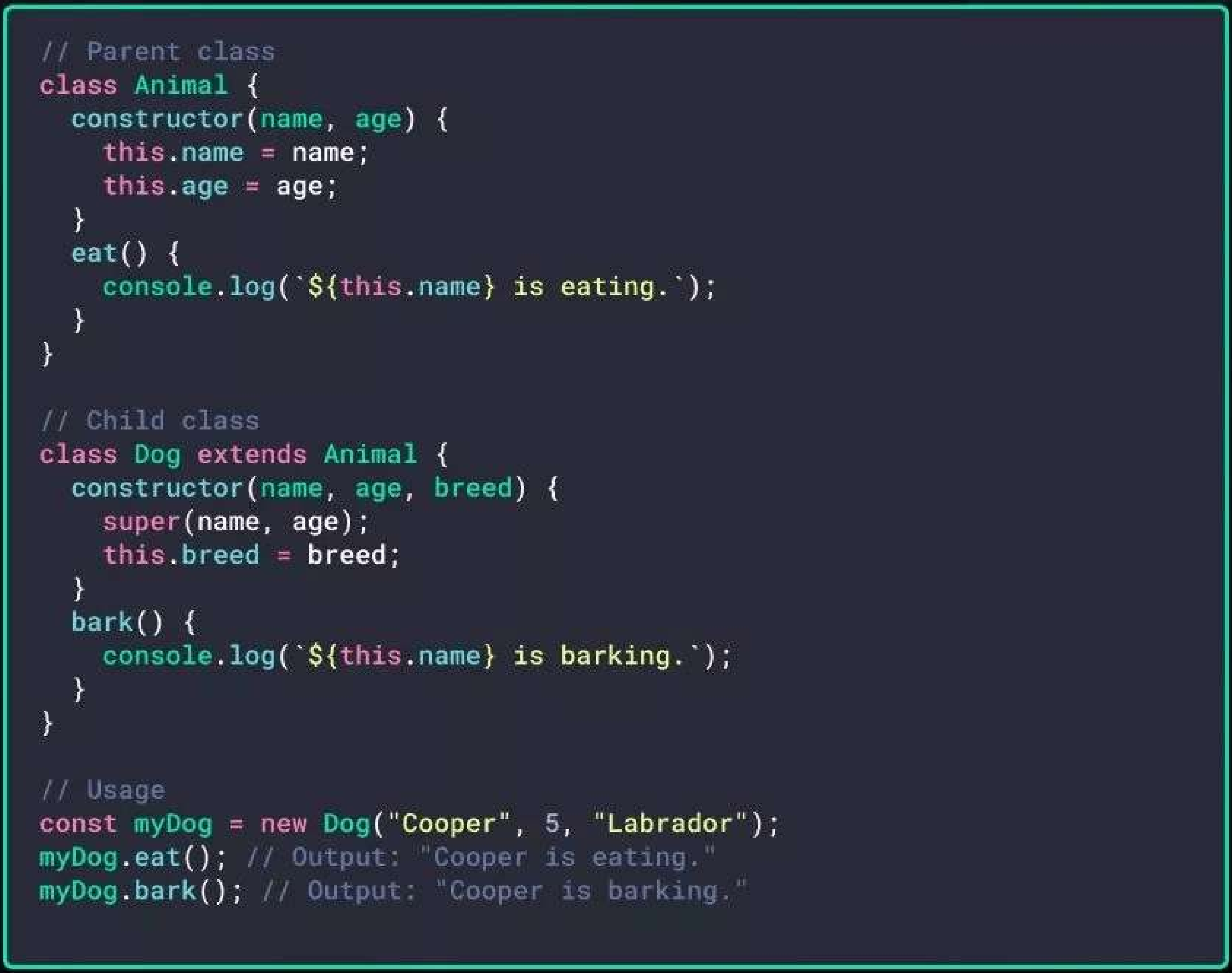
This defines how the data and actions associated with the data; are organized using code.

- OOPs Concepts
  - Objects
  - Classes
  - inheritance
  - polymorphism
  - encapsulation
  - abstraction
- In a previous post we discussed JavaScript classes, objects and properties.

Inheritance

Inheritance is a concept where one class (child class) inherits properties and methods from another class (parent class).

- In JavaScript, inheritance is achieved through the use of the extends keyword.



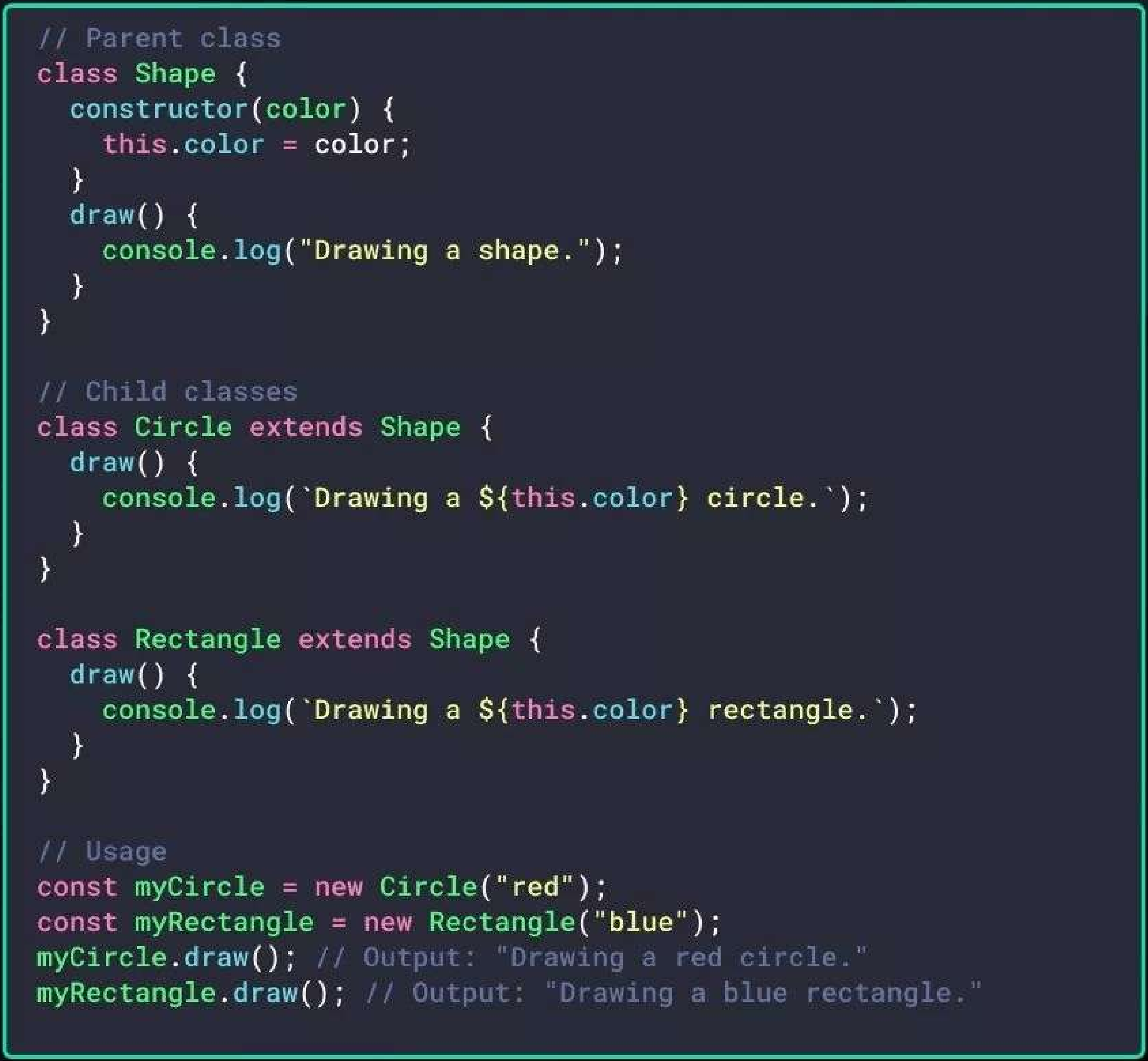
```
// Parent class
class Animal {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  eat() {
    console.log(`${this.name} is eating.`);
  }
}

// Child class
class Dog extends Animal {
  constructor(name, age, breed) {
    super(name, age);
    this.breed = breed;
  }
  bark() {
    console.log(`${this.name} is barking.`);
  }
}

// Usage
const myDog = new Dog("Cooper", 5, "Labrador");
myDog.eat(); // Output: "Cooper is eating."
myDog.bark(); // Output: "Cooper is barking."
```

Polymorphism

Polymorphism is the ability of objects to use the same function in different forms.

- This reduces repetition and makes the code snippet useful in many different cases.
- In JavaScript, polymorphism is achieved through method overriding or method overloading.
- Method overriding is where a subclass provides its own implementation of a method that is already defined in the parent class.
- Method overloading is where a class has multiple methods with the same name but different parameters.

Here's an example of polymorphism in JavaScript using method overriding

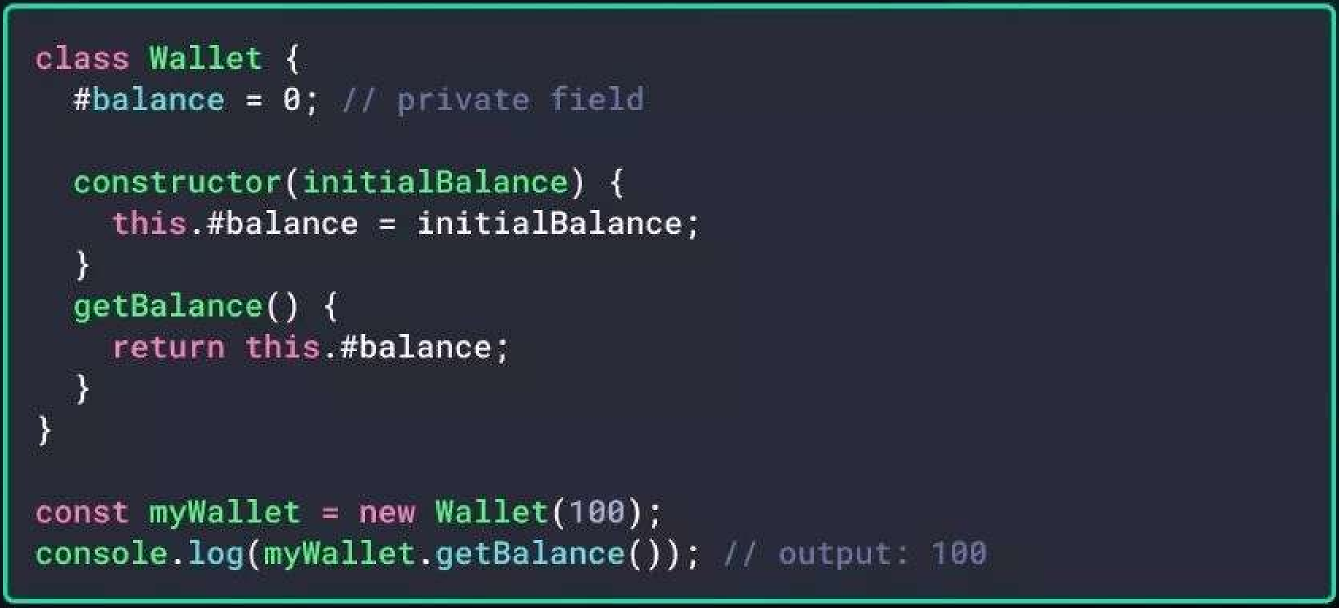


```
// Parent class
class Shape {
  constructor(color) {
    this.color = color;
  }
  draw() {
    console.log("Drawing a shape.");
  }
}

// Child classes
class Circle extends Shape {
  draw() {
    console.log(`Drawing a ${this.color} circle.`);
  }
}

class Rectangle extends Shape {
  draw() {
    console.log(`Drawing a ${this.color} rectangle.`);
  }
}

// Usage
const myCircle = new Circle("red");
const myRectangle = new Rectangle("blue");
myCircle.draw(); // Output: "Drawing a red circle."
myRectangle.draw(); // Output: "Drawing a blue rectangle."
```

Here both override the draw() method of the parent class, but provide their own implementation of it.

Encapsulation

Encapsulation is the practice of hiding the internal details of an object from the outside world.



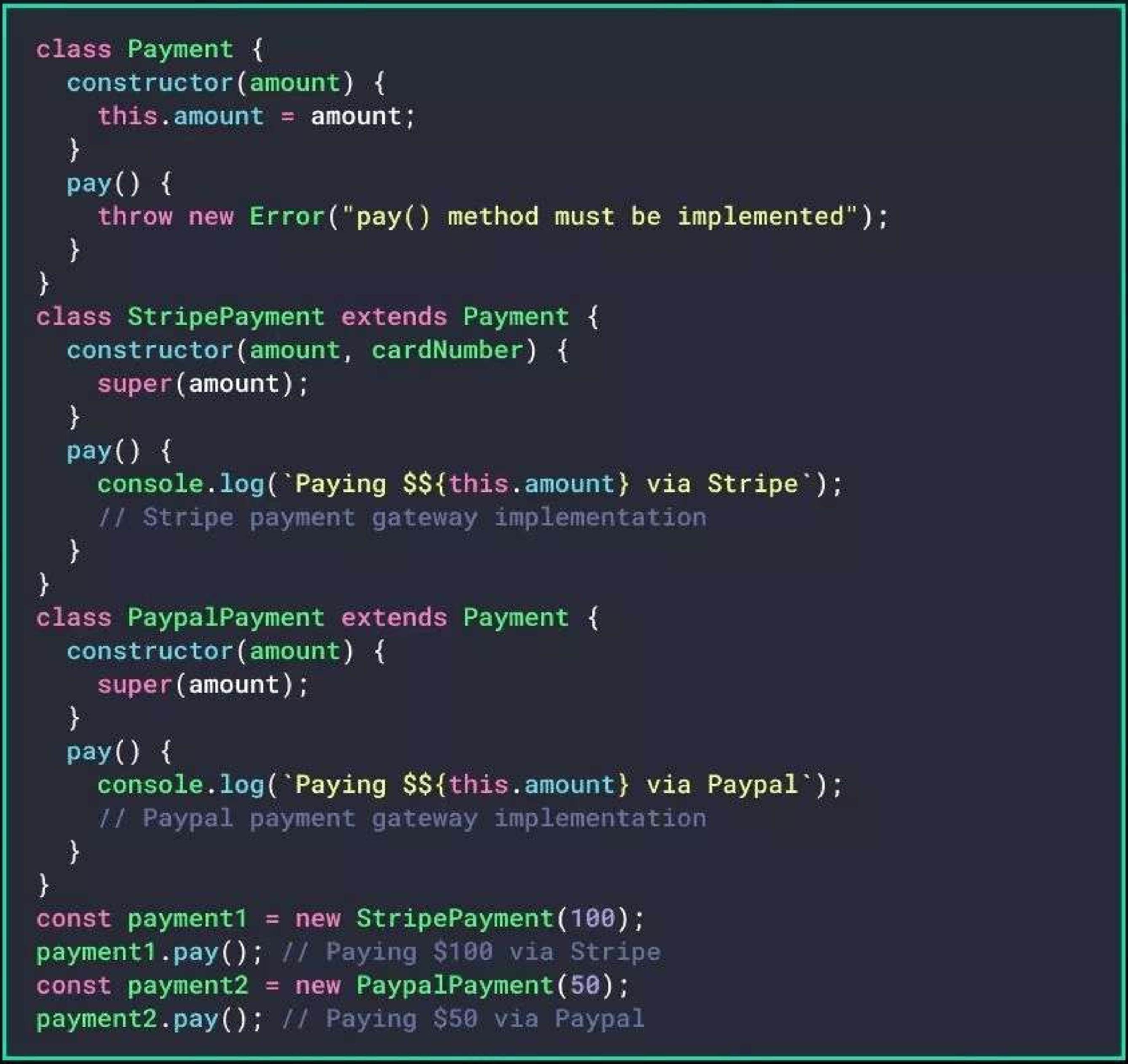
```
class Wallet {
  #balance = 0; // private field

  constructor(initialBalance) {
    this.#balance = initialBalance;
  }
  getBalance() {
    return this.#balance;
  }
}

const myWallet = new Wallet(100);
console.log(myWallet.getBalance()); // output: 100
```

- By encapsulating the #balance field within the Wallet class, we are preventing direct access to the #balance field from outside of the class.

- This is an example of how encapsulation can help to prevent unwanted modifications in a real-world scenario such as managing a wallet.

Abstraction

Abstraction is the process of hiding the implementation details while showing only the necessary information to the user.



```
class Payment {
  constructor(amount) {
    this.amount = amount;
  }
  pay() {
    throw new Error("pay() method must be implemented");
  }
}
class StripePayment extends Payment {
  constructor(amount, cardNumber) {
    super(amount);
  }
  pay() {
    console.log(`Paying $${this.amount} via Stripe`);
    // Stripe payment gateway implementation
  }
}
class PaypalPayment extends Payment {
  constructor(amount) {
    super(amount);
  }
  pay() {
    console.log(`Paying $${this.amount} via Paypal`);
    // Paypal payment gateway implementation
  }
}
const payment1 = new StripePayment(100);
payment1.pay(); // Paying $100 via Stripe
const payment2 = new PaypalPayment(50);
payment2.pay(); // Paying $50 via Paypal
```

- In this example, the Payment class is the abstract class that defines the interface for making payments.
- It has a pay method that is abstract and must be implemented by its subclasses.
- The StripePayment and PaypalPayment classes are concrete classes that implement the pay method for their respective payment gateways.
- As always, I hope you enjoyed the post and learned something new.
- If you have any queries then let me know in the comment box.

i post about Tech, Coding and Career

Follow for more content like this