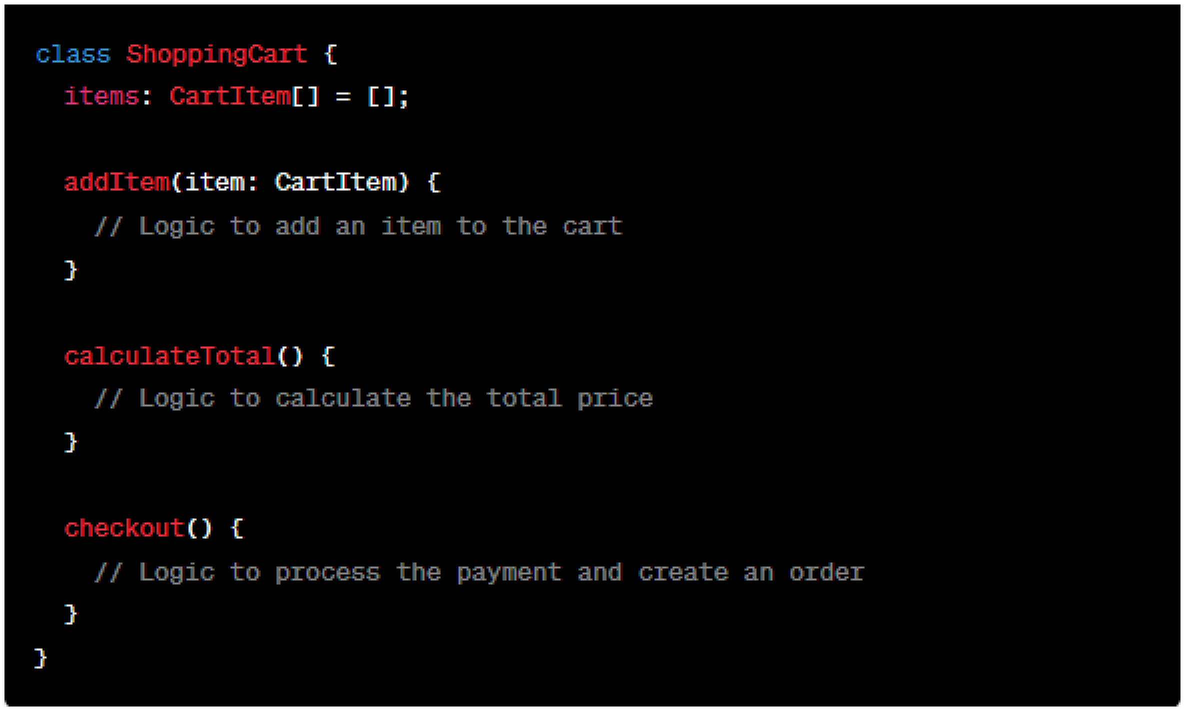
Understanding the SOLID Principles in TypeScript

When it comes to writing clean, maintainable, and scalable code, the SOLID principles are like a set of golden rules for developers. SOLID is an acronym, and each letter stands for a fundamental software design principle. In this blog, we'll explore what each principle means and provide examples using TypeScript, a popular programming language.

1. Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a class should have only one reason to change. In other words, a class should have a single responsibility. This helps keep your code modular and easy to understand.

Suppose we are building a simple online store application. We can have a class for the shopping cart:



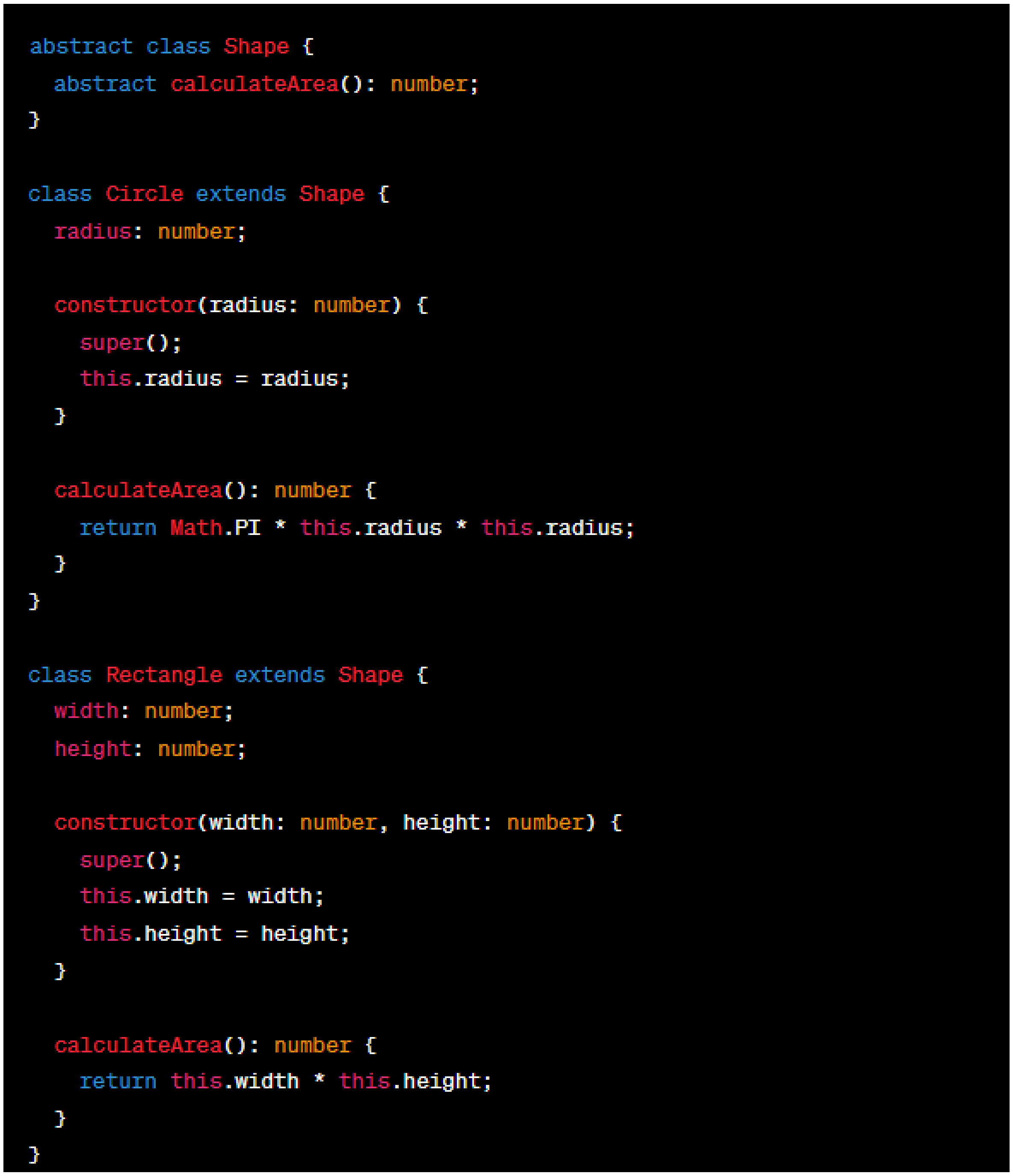
```
class ShoppingCart {
  items: CartItem[] = [];

  addItem(item: CartItem) {
    // Logic to add an item to the cart
  }

  calculateTotal() {
    // Logic to calculate the total price
  }

  checkout() {
    // Logic to process the payment and create an order
  }
}
```

In this example, the ShoppingCart class has multiple responsibilities: adding items, calculating totals, and processing payments. To follow the SRP, we can split it into separate classes, each with its own responsibility.

2. Open/Closed Principle (OCP)

The Open/Closed Principle suggests that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means you can add new features without changing existing code.

Let’s say we have a Shape class and want to calculate the area of different shapes. We can use inheritance and extension to achieve the OCP:



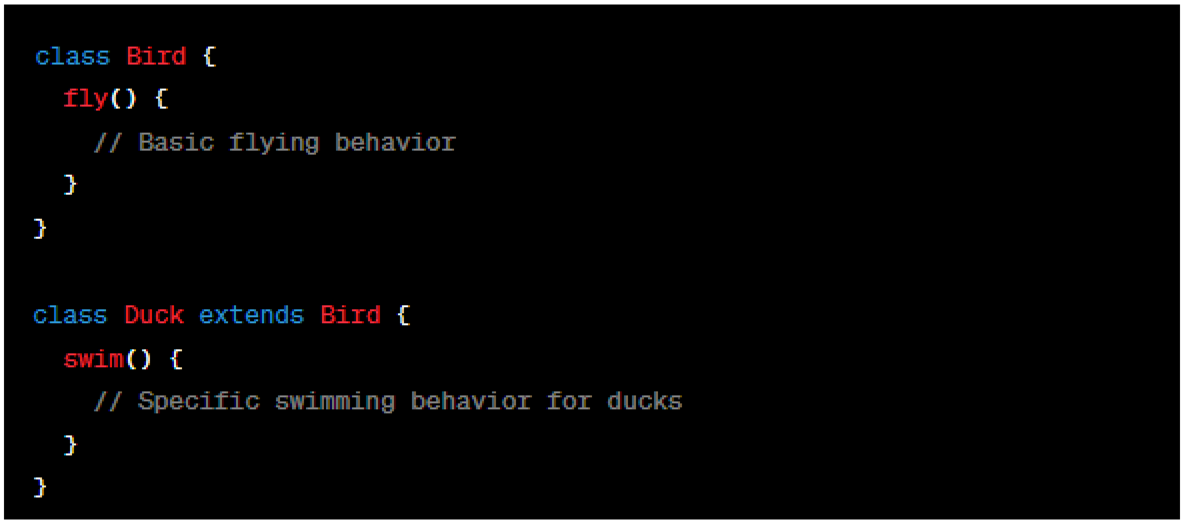
```typescript
abstract class Shape {
  abstract calculateArea(): number;
}

class Circle extends Shape {
  radius: number;

  constructor(radius: number) {
    super();
    this.radius = radius;
  }

  calculateArea(): number {
    return Math.PI * this.radius * this.radius;
  }
}

class Rectangle extends Shape {
  width: number;
  height: number;

  constructor(width: number, height: number) {
    super();
    this.width = width;
    this.height = height;
  }

  calculateArea(): number {
    return this.width * this.height;
  }
}
```

Here, we can easily add more shapes (e.g., Triangle) without modifying the existing Shape or Circle/Rectangle classes.

3. Liskov Substitution Principle (LSP)

The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In simple terms, if you have a class hierarchy, derived classes should be able to substitute their base classes seamlessly.

Suppose we have a Bird class and a Duck subclass:



```
class Bird {
  fly() {
    // Basic flying behavior
  }
}

class Duck extends Bird {
  swim() {
    // Specific swimming behavior for ducks
  }
}
```

According to LSP, we can safely use a Duck wherever we expect a Bird, and it should not break the code.

4. Interface Segregation Principle (ISP)

The Interface Segregation Principle suggests that clients should not be forced to depend on interfaces they do not use. In other words, it’s better to have smaller, focused interfaces rather than large, monolithic ones.

Suppose we have an interface for a multimedia player:

```
interface MediaPlayer {
  playAudio(): void;
  playVideo(): void;
  showLyrics(): void;
}
```

If a class only needs to play audio, it shouldn’t be forced to implement the ‘playVideo()’ and ‘showLyrics()’ methods. We can split the interface into smaller ones:

```
interface AudioPlayer {
  playAudio(): void;
}

interface VideoPlayer {
  playVideo(): void;
}

interface LyricsDisplay {
  showLyrics(): void;
}
```

This way, classes can implement the interfaces that are relevant to them, promoting flexibility.

5. Dependency Inversion Principle (DIP)

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions. It encourages the use of interfaces or abstract classes to decouple components.

Consider a class that sends notifications:

```
class NotificationService {
  sendEmail(message: string) {
    // Logic to send an email
  }
}
```

To follow DIP, we can create an interface for notification and make the ‘NotificationService’ depend on that interface:

```
interface NotificationSender {
  send(message: string): void;
}

class NotificationService {
  private sender: NotificationSender;

  constructor(sender: NotificationSender) {
    this.sender = sender;
  }

  sendNotification(message: string) {
    this.sender.send(message);
  }
}
```

Now, we can easily switch between different notification senders (e.g., email, SMS) without modifying the ‘NotificationService’ class.

In conclusion, the SOLID principles provide a set of guidelines for writing maintainable and flexible code. By following these principles in TypeScript projects, we can improve code quality and make your software more adaptable to changes and enhancements.