**Clean code 13 principles**

**1. Don’t Repeat Yourself (DRY)**

This principle suggests that code should not have unnecessary duplication. Instead, it should be organized in a way that avoids redundancy and makes it easy to maintain. For example, instead of writing the same calculation in multiple places in the code, create a function that performs the calculation and call that function from the different places where the calculation is needed.

// bad example  
let total = 0;  
for (let i = 0; i < prices.length; i++) {  
 total += prices[i];  
}  
console.log(total);  
for (let i = 0; i < prices.length; i++) {  
 total += prices[i];  
}  
console.log(total);  
// good example  
function calculateTotal(prices) {  
 let total = 0;  
 for (let i = 0; i < prices.length; i++) {  
 total += prices[i];  
 }  
 return total;  
}  
console.log(calculateTotal(prices));  
console.log(calculateTotal(prices));

**2. Write Everything Twice (WET)**

This is an opposite principle of DRY. It suggest that if you find yourself copy-pasting code multiple times, anticipating the identical code forking in different directions later on, having WET code may make that future change easier.

class Dog {  
 name = "Dog";  
 move() {  
 console.log("Dog is moving");  
 // future implementation  
 // console.log("Dog is trotting");  
 }  
}  
  
class Cat {  
 name = "Cat";  
 move() {  
 console.log("Cat is moving");  
 // future implementation  
 // console.log("Cat is sneaking");  
 }  
}  
// with DRY, we would have a Animal with move method  
// however, dog and cat could be moving in different ways,   
// so we want to keep them seperate following WET

**3. Single Responsibility Principle (SRP)**

Each module or function should have only one reason to change. For example, instead of having a function that handles multiple tasks, split it up into multiple functions, each with a single responsibility.

// bad example  
function processData(data) {  
 // validate data  
 // save data to database  
 // send data to another system  
 // log data  
}  
  
// good example  
function validateData(data) {  
 // validate data  
}  
function saveData(data) {  
 // save data to database  
}  
function sendData(data) {  
 // send data to another system  
}  
function logData(data) {  
 // log data  
}

**4. Open/closed Principle (OCP)**

A module or function should be open for extension but closed for modification. For example, instead of modifying an existing class to add new functionality, create a new class that extends the original class and add the new functionality there.

// bad example  
class Rectangle {  
 constructor(width, height) {  
 this.width = width;  
 this.height = height;  
 }  
  
setWidth(width) {  
 this.width = width;  
 }  
 setHeight(height) {  
 this.height = height;  
 }  
 getArea() {  
 return this.width \* this.height;  
 }  
}  
// new requirement: add a getPerimeter method  
class Rectangle {  
 constructor(width, height) {  
 this.width = width;  
 this.height = height;  
 }  
 setWidth(width) {  
 this.width = width;  
 }  
 setHeight(height) {  
 this.height = height;  
 }  
 getArea() {  
 return this.width \* this.height;  
 }  
 getPerimeter() {  
 return 2 \* (this.width + this.height);  
 }  
}  
// good example  
class Rectangle {  
 constructor(width, height) {  
 this.width = width;  
 this.height = height;  
 }  
 setWidth(width) {  
 this.width = width;  
 }  
 setHeight(height) {  
 this.height = height;  
 }  
 getArea() {  
 return this.width \* this.height;  
 }  
}  
class RectangleWithPerimeter extends Rectangle {  
 getPerimeter() {  
 return 2 \* (this.width + this.height);  
 }  
}

**5. Liskov Substitution Principle (LSP)**

Objects of a superclass should be able to be replaced with objects of a subclass without altering the correctness of the program. For example, a subclass should be able to replace its parent class without breaking the program.

class Bird {  
 fly() {  
 console.log("I am flying");  
 }  
}  
  
class Ostrich extends Bird {  
 fly() {  
 throw "I can't fly";  
 }  
}  
let bird = new Bird();  
bird.fly(); // Output: "I am flying"  
let ostrich = new Ostrich();  
ostrich.fly(); // Output: "I can't fly"  
let birds = [new Bird(), new Ostrich()];  
for (let bird of birds) {  
 bird.fly();  
}  
// Output: "I am flying" "I can't fly"

**6. Interface Segregation Principle (ISP)**

A client should not be forced to implement interfaces it doesn’t use. For example, instead of having a monolithic interface with many methods, split it up into smaller, more specific interfaces.

// bad example  
interface Shape {  
 draw();  
 resize();  
 rotate();  
 getArea();  
}  
  
class Circle implements Shape {  
 draw() {...}  
 resize() {...}  
 rotate() {...}  
 getArea() {...}  
}  
// good example  
interface Drawable {  
 draw();  
}  
interface Resizable {  
 resize();  
}  
interface Rotatable {  
 rotate();  
}  
interface Measurable {  
 getArea();  
}  
class Circle implements Drawable, Resizable, Rotatable, Measurable {  
 draw() {...}  
 resize() {...}  
 rotate() {...}  
 getArea() {...}  
}

**7. Dependency Inversion Principle (DIP)**

High-level modules should not depend on low-level modules. Both should depend on abstractions. For example, instead of having a high-level module depend on a specific implementation of a low-level module, have it depend on an abstraction of the low-level module.

// bad example  
class UserController {  
 constructor() {  
 this.userRepository = new MySQLUserRepository();  
 }  
 ...  
}  
  
// good example  
class UserController {  
 constructor(userRepository) {  
 this.userRepository = userRepository;  
 }  
 ...  
}  
  
let mysqlUserRepository = new MySQLUserRepository();  
let userController = new UserController(mysqlUserRepository);

**8. Keep It Simple, Stupid (KISS)**

This principle suggests that code should be as simple as possible, and should avoid unnecessary complexity. For example, instead of using a complex algorithm to solve a problem, use a simpler one that gets the job done.

// bad example  
function calculateDistance(point1, point2) {  
 let xDiff = point2.x - point1.x;  
 let yDiff = point2.y - point1.y;  
 let distance = Math.sqrt((xDiff \*\* 2) + (yDiff \*\* 2));  
 return distance;  
}  
  
// good example  
function calculateDistance(point1, point2) {  
 let xDiff = point2.x - point1.x;  
 let yDiff = point2.y - point1.y;  
 return Math.abs(xDiff) + Math.abs(yDiff);  
}

**9. You Aren’t Gonna Need It (YAGNI)**

This principle suggests that code should not be written until it is actually needed, as it can add unnecessary complexity and make the code harder to maintain. For example, instead of adding a feature that may be needed in the future, focus on the features that are needed now.

// bad example  
class Order {  
 constructor() {  
 this.items = [];  
 this.discount = null;  
 }  
}  
  
// good example  
class Order {  
 constructor() {  
 this.items = [];  
 }  
}

**10. Fail Fast**

This principle suggests that code should fail as early as possible, so that issues can be identified and resolved quickly. For example, instead of waiting until the end of a function to check for errors, check for errors as soon as possible.

// bad example  
function divide(a, b) {  
 let result = a / b;  
 return result;  
}  
  
console.log(divide(1, 0)); // Output: Infinity  
// good example  
function divide(a, b) {  
 if (b === 0) {  
 throw new Error("Cannot divide by zero");  
 }  
 let result = a / b;  
 return result;  
}  
console.log(divide(1, 0)); // Output: Error: Cannot divide by zero

**11. Law of Demeter (LoD)**

This principle suggests that an object should only communicate with its immediate neighbors and should not reach into the internal state of other objects. For example, instead of accessing the internal state of an object, use a method to get the information you need.

// bad example  
class Order {  
 constructor() {  
 this.items = [];  
 }  
}  
  
class OrderItem {  
 constructor(name, price) {  
 this.name = name;  
 this.price = price;  
 }  
}  
let order = new Order();  
let item1 = new OrderItem("item1", 10);  
let item2 = new OrderItem("item2", 20);  
order.items.push(item1);  
order.items.push(item2);  
let total = 0;  
for (let item of order.items) {  
 total += item.price;  
}  
console.log(total);  
// good example  
class Order {  
 constructor() {  
 this.items = [];  
 }  
 addItem(item) {  
 this.items.push(item);  
 }  
 getTotal() {  
 let total = 0;  
 for (let item of this.items) {  
 total += item.price;  
 }  
 return total;  
 }  
}  
class OrderItem {  
 constructor(name, price) {  
 this.name = name;  
 this.price = price;  
 }  
}  
let order = new Order();  
let item1 = new OrderItem("item1", 10);  
let item2 = new OrderItem("item2", 20);  
order.addItem(item1);  
order.addItem(item2);  
console.log(order.getTotal());

**12. Command Query Separation (CQS)**

It is a principle that suggests that methods should either be command methods that change the state of an object, or query methods that return information about an object, but not both. For example, instead of having a method that both changes the state of an object and returns a value, have separate methods for changing the state and returning the value.

// bad example  
class Order {  
 constructor() {  
 this.items = [];  
 }  
  
 addItem(item) {  
 this.items.push(item);  
 return this.items.length;  
 }  
}  
let order = new Order();  
console.log(order.addItem({ name: "item1", price: 10 })); // Output: 1  
  
// good example  
class Order {  
 constructor() {  
 this.items = [];  
 }  
 addItem(item) {  
 this.items.push(item);  
 }  
 getItemCount() {  
 return this.items.length;  
 }  
}  
let order = new Order();  
order.addItem({ name: "item1", price: 10 });  
console.log(order.getItemCount()); // Output: 1

**13. Composition over Inheritance**

It suggests that code should favor composition over inheritance, as composition allows for greater flexibility and easier maintenance. For example, instead of inheriting properties and methods from a parent class, compose objects with the properties and methods they need.

// bad example  
class Vehicle {  
 constructor() {  
 this.speed = 0;  
 }  
 startEngine() {  
 this.speed = 10;  
 }  
}  
  
class Car extends Vehicle {  
 constructor() {  
 super();  
 this.numWheels = 4;  
 }  
}  
let car = new Car();  
console.log(car.numWheels); // Output: 4  
console.log(car.speed); // Output: 0  
car.startEngine();  
console.log(car.speed); // Output: 10  
// good example  
class Engine {  
 start() {  
 this.speed = 10;  
 }  
}  
class Car {  
 constructor() {  
 this.numWheels = 4;  
 this.engine = new Engine();  
 }  
}  
let car = new Car();  
console.log(car.numWheels); // Output: 4  
console.log(car.engine.speed); // Output: undefined  
car.engine.start();  
console.log(car.engine.speed); // Output: 10