

In JavaScript, composition and inheritance are two fundamental design principles used to build and structure objects. Here's a differentiation between the two, along with their respective pros, cons, and scenarios in which each is preferable.

## Inheritance

Inheritance is a mechanism where a new class (child class) is derived from an existing class (parent class), inheriting its properties and methods.

Example:

```
```javascript
```

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog('Rex');
dog.speak(); // Rex barks.
```

```
```
```

Pros:

1. Code Reusability: Inherited code can be reused in multiple classes.
2. Hierarchical Relationships: Clearly defines a parent-child relationship, making it easier to understand and maintain.
3. Polymorphism: Allows objects to be treated as instances of their parent class, enabling polymorphic behavior.

Cons:

1. Tight Coupling: Child classes are tightly coupled to parent classes, making changes to the parent class risky.
2. Fragile Base Class Problem: Changes in the base class can inadvertently break derived classes.

3. Inflexibility: Inheritance structures can be rigid, making it difficult to adapt to new requirements.

When to Use:

- When there is a clear hierarchical relationship and shared behavior between classes.
- When you need to leverage polymorphism.
- When you want to reuse common functionality in a predictable manner.

## Composition in FP

In functional programming, composition is a fundamental concept that involves combining simple functions to build more complex ones. This allows for more modular, reusable, and maintainable code. In JavaScript, functional composition is a powerful technique for creating pipelines of data transformations.

### Key Concepts of Functional Composition in JavaScript

1. Pure Functions: Functions that return the same output for the same input and have no side effects.
2. Higher-Order Functions: Functions that take other functions as arguments or return them as results.
3. Function Composition: The process of combining two or more functions to produce a new function.

### Function Composition

Function composition in JavaScript typically involves creating a new function by chaining together multiple functions. This is often achieved using utility functions like `compose` or `pipe`.

Example with `compose`:

```
```javascript
    const compose = (...functions) => input =>
        functions.reduceRight((acc, fn) => fn(acc), input);

    const add = x => x + 1;
    const multiply = x => x * 2;

    const addThenMultiply = compose(multiply, add);

    console.log(addThenMultiply(5)); // Output: 12
...`
```

In this example, `addThenMultiply` is a compose function that first adds 1 to the input and then multiplies the result by 2.

Example with `pipe`:

The `pipe` function is similar to `compose`, but it applies functions from left to right:

```
```\njavascript\n\n    const pipe = (...functions) => input =>\n        functions.reduce((acc, fn) => fn(acc), input);\n\n    const add = x => x + 1;\n    const multiply = x => x * 2;\n\n    const addThenMultiply = pipe(add, multiply);\n\n    console.log(addThenMultiply(5)); // Output: 12\n...`
```

## Benefits of Function Composition

1. Modularity: Functions can be reused and combined in different ways, promoting code reuse.
2. Readability: Composing functions can make the code more declarative and easier to understand.
3. Maintainability: Smaller functions are easier to test and debug, and changes to individual functions don't affect the composed logic.

## Practical Examples

### Example 1: Data Transformation Pipeline

```
```\njavascript\n\n    const capitalize = str => str.charAt(0).toUpperCase() + str.slice(1);\n    const reverse = str => str.split("").reverse().join("");\n    const exclaim = str => `${str}!`;\n\n    const transform = pipe(capitalize, reverse, exclaim);\n\n    console.log(transform('hello')); // Output: Olleh!\n...`
```

In this example, the `transform` function capitalizes a string, reverses it, and then adds an exclamation mark.

## Example 2: Processing an Array of Data

```
```javascript
    const filterEven = arr => arr.filter(x => x % 2 === 0);
    const double = arr => arr.map(x => x * 2);
    const sum = arr => arr.reduce((acc, x) => acc + x, 0);

    const processNumbers = pipe(filterEven, double, sum);

    console.log(processNumbers([1, 2, 3, 4, 5, 6])); // Output: 24
...
```
```

Here, `processNumbers` filters even numbers, doubles them, and then sums the result.

## Conclusion

In functional programming, composition allows you to build complex behavior from simple, reusable functions. By using higher-order functions and composing them, you can create more modular, readable, and maintainable code. This approach leverages the power of functions as first-class citizens in JavaScript, making it a key aspect of functional programming in the language.

## Composition in OOP

Composition involves building complex objects by combining simpler objects, often through aggregation or delegation.

Example:

```
```javascript
class Engine {
  start() {
    console.log('Engine started.');
  }
}

class Car {
  constructor() {
    this.engine = new Engine();
  }

  start() {
    this.engine.start();
    console.log('Car started.');
  }
}
```
```

```

    }

    const car = new Car();
    car.start(); // Engine started. Car started.
...

```

#### Pros:

1. Flexibility: Objects can be composed in different ways without altering their structures.
2. Loose Coupling: Components are loosely coupled, making changes easier and safer.
3. Reusability: Smaller, reusable components can be combined to create more complex functionality.

#### Cons:

1. Complexity: Managing multiple components can introduce complexity.
2. Indirection: Composition often requires additional indirection, which can make code harder to follow.

#### When to Use:

- When you need to build complex objects from simpler, reusable components.
- When the problem domain involves multiple behaviors that can change independently.
- When you want to favor flexibility and extensibility over a rigid hierarchy.

#### Comparison

| Feature               | Inheritance                                       | Composition                                     |
|-----------------------|---------------------------------------------------|-------------------------------------------------|
| <b>Relationship</b>   | "Is-a" (Hierarchical)                             | "Has-a" (Component-based)                       |
| <b>Coupling</b>       | Tight                                             | Loose                                           |
| <b>Flexibility</b>    | Less Flexible                                     | More Flexible                                   |
| <b>Reusability</b>    | Reuse through inheritance                         | Reuse through composition                       |
| <b>Modification</b>   | Changes in base class affect derived class        | Changes in components don't affect other parts  |
| <b>Complexity</b>     | Simple to understand hierarchical structure       | Can introduce complexity in managing components |
| <b>Usage Scenario</b> | Clear hierarchical relationships, shared behavior | Multiple behaviors that change independently    |

#### Conclusion

- Use Inheritance when there is a clear hierarchical relationship and shared behavior that you want to reuse.
- Use Composition when you need flexibility, loose coupling, and the ability to easily combine different behaviors.

By understanding the strengths and weaknesses of both approaches, you can choose the most appropriate one for your specific use case, leading to more maintainable and scalable code.