Фундаментальные принципы объектно-ориентированного программирования на JavaScript

Первый шаг к рабочему и эффективному коду — разобраться в механизмах, которые лежат в основе языка. Например, некоторые функции в JavaScript не всегда ведут себя так, как того ожидает Объектно-ориентированное программирование (ООП) — это шаблон проектирования программного обеспечения, который позволяет решать задачи с точки зрения объектов и их взаимодействий. ООП обычно реализуется с помощью классов или прототипов. Большинство объектно-ориентированных языков (Java, C++, Ruby, Python и др.) используют наследование на основе классов. JavaScript реализует ООП через прототипное наследование. В этой статье мы рассмотрим оба эти подхода в JavaScript, обсудим их преимущества и недостатки, а также предложим альтернативу для разработки более модульных и масштабируемых приложений.

Что такое объект?

Принцип ООП заключается в том, чтобы составлять систему из объектов, решающих простые задачи, которые вместе составляют сложную программу. Объект состоит из приватных изменяемых состояний и функций (методов), которые работают с этими состояниями. У объектов есть определение себя (self, this) и поведение, наследуемое от чертежа, т.е. класса (классовое наследование) или других объектов (прототипное наследование).

Наследование — способ сказать, что эти объекты похожи на другие за исключением некоторых деталей. Наследование позволяет ускорить разработку за счёт повторного использования кода.

Классовое наследование

В классовом ООП классы являются чертежами для объектов. Объекты (или экземпляры) создаются на основе классов. Существует конструктор, который используется для создания экземпляра класса с заданными свойствами.

Например:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }
  getFullName() {
    return this.firstName + ' ' + this.lastName
  }
}
```

Здесь при помощи ключевого слова `class` из ES6 мы создаём класс `Person` со свойствами `firstName` и `lastName`, которые хранятся в `this`. Значения свойств задаются в конструкторе, а доступ к ним осуществляется в методе `getFullName()`.

Мы создаём экземпляр класса `Person` с именем `person` с помощью ключевого слова `new`:

```
let person = new Person('Dan', 'Abramov')
person.getFullName() //> "Dan Abramov"
// Мы можем использовать акцессор или получить доступ напрямую
person.firstName //> "Dan"
person.lastName //> "Abramov"
```

Объекты, созданные с помощью ключевого слова `new`, изменяемы. Другими словами, изменения в классе повлияют на все объекты, являющиеся экземплярами этого класса, а также на дочерние классы, которые его расширяют (`extends`).

Для расширения класса мы можем создать другой класс. Расширим класс `Person` с помощью класса `User`. `User` — это `Person` с почтой и паролем:

```
class User extends Person {
  constructor(firstName, lastName, email, password) {
    super(firstName, lastName)
    this.email = email
    this.password = password
   }
   getEmail() {
```

```
    return this.email
  }
  getPassword() {
    return this.password
  }
}
```

Выше мы создали класс `User`, расширяющий возможности `Person` путём добавления свойств `email` и `password` и функций доступа к ним. В функции `App()` ниже мы создаём объект нового класса `user`:

```
function App() {
  let user = new User('Dan',
                      'Abramov',
                      'dan@abramov.com',
                      'iLuvES6')
  user.getFullName() //> "Dan Abramov"
  user.getEmail() //> "dan@abramov.com"
  user.getPassword() //> "iLuvES6"
  user.firstName //> "Dan"
  user.lastName //> "Abramov"
  user.email //> "dan@abramov.com"
  user.password //> "iLuvES6"
}
```

Всё вроде бы хорошо работает, но использование классового подхода к наследованию привело к большому конструктивному недостатку: откуда пользователи класса `User` (например, `App`) могут знать, что у этого класса есть свойства `firstName` и `lastName` и функция `getFullName`? Одного взгляда на код класса `User` недостаточно для того, чтобы сказать что-либо о его родительском классе. В итоге приходится копаться в документации или искать нужный код по всей иерархии классов.

> *Как говорит Дэн Абрамов: "Проблема с наследованием заключается в том, что у потомков слишком высокий уровень доступа к деталям реализации каждого базового класса в иерархии и наоборот. После изменения требований рефакторинг иерархии классов настолько сложно провести, что она превращается в полную неразбериху со следами устаревших требований".*

Классовое наследование построено на создании связей через зависимости. На основе базовых классов (или суперклассов) создаются производные классы. Классовое наследование хорошо подходит для небольших и простых приложений, которые редко меняются и у которых не более одного уровня наследования (неглубокие деревья наследования позволяют избежать проблемы хрупкого базового класса) или совершенно разные сценарии использования. Однако по мере расширения иерархии такое наследование со временем будет невозможно поддерживать.

> *Эрик Эллиот описал, как классовое наследование может потенциально привести к провалу проекта, а в худшем случае — к провалу компании: "Как только у вас наберётся достаточно клиентов, использующих new, вы даже при желании не сможете изменить реализацию конструктора, а если попытаетесь, то сломаете весь чужой код".*

Когда много производных классов с очень разными функциями наследуются от одного базового класса, любое, казалось бы, безобидное изменение в базовом классе может привести к сбою в работе производных. За счёт усложнения кода и всего процесса создания продукта вы могли бы смягчить побочные эффекты, создав контейнер для инъекций зависимостей. Это обеспечило бы единый интерфейс для создания сервисов, потому что позволило бы абстрагироваться от подробностей создания. Есть ли способ получше?

Прототипное наследование

В прототипном наследовании классы не используются совсем. Вместо этого объекты создаются из других объектов. Мы начинаем с обобщённого объекта — прототипа. Прототип можно использовать для создания других объектов путём его клонирования или расширять его разными функциями.

Хотя в предыдущем разделе мы показали, как использовать `class` из ES6, классы в JavaScript не такие уж и классы:

```
typeof Person //> "function"
typeof User //> "function"
```

Классы в ES6 — на самом деле синтаксический сахар для существующего в JavaScript прототипного наследования. Под капотом при создании класса с помощью ключевого слова `new` создаётся новый объект функции с кодом из `constructor`.

По сути, JavaScript — прототипно-ориентированный язык.

> *Числа, строки, логические переменные (true и false), а также значения null и undefined в JavaScript относятся к простым типам данных. Всё остальное — объекты. Числа, строки и логические переменные похожи на объекты тем, что имеют методы, но в отличие от объектов они неизменны. Объекты в JavaScript имеют изменяемые ключевые коллекции. В JavaScript объектами являются массивы, функции, регулярные выражения, и, конечно, объекты также являются объектами.*
>
> Из книги Дугласа Крокфорда «JavaScript: сильные стороны»

Посмотрим на один из таких объектов, доступных в JavaScript «из коробки», — `Array`. Массивы (экземпляры `Array`) наследуются от `Array.prototype`, который включает в себя много методов, разделённых на акцессоры (не изменяют исходный массив), мутаторы (изменяют исходный массив) и итераторы (применяют функцию, переданную в качестве аргумента, на каждом элементе массива для создания нового).

Акцессоры:

- `Array.prototype.includes(e)` — возвращает `true`, если массив содержит элемент `e`, в противном случае — `false`.
- `Array.prototype.slice(i, j)` — возвращает новый массив, который является срезом исходного от индекса `i` до `j` включительно.

Мутаторы:

- `Array.prototype.push(e)` — помещает элемент `e` в конец массива.
- `Array.prototype.pop()` — удаляет последний элемент массива.
- `Array.prototype.splice(i, j)` — извлекает срез массива от индекса `i` до `j` включительно без сохранения исходного.

Мутаторы изменяют исходный массив. Метод `splice()` извлекает такой же срез, как и `slice()`, однако если вам нужно оставить исходный массив, то лучше выбрать `slice()`.

Итераторы:

- `Array.prototype.map(f)` — применяет функцию `f` на каждом элементе массива и создаёт новый массив с результатом вызова указанной функции.

- `Array.prototype.filter(f)` — создаёт новый массив со всеми элементами, прошедшими проверку, задаваемую в функции `f`.
- `Array.prototype.forEach(f)` — применяет функцию `f` на каждом элементе массива.

Методы `map()` и `forEach()` похожи тем, что они что-то делают со всеми элементами массива, но ключевая разница в том, что `map()` возвращает массив, а `forEach()` — ничего. В хороших практиках проектирования ПО всегда рекомендуют писать функции без побочных эффектов, т.е. не использовать void-функции. Метод `forEach()` никак не изменяет исходный массив, поэтому `map()` будет лучшим выбором, если вам нужно как-то преобразовать данные. Один из возможных вариантов использования `forEach()` — вывод в консоль для отладки:

```
let arr = [1,2,3]
arr.forEach(e => console.log(e))
arr //> [1,2,3]
```

Предположим, что мы хотим расширить прототип `Array` новым методом `partition()`, который делит массив на два новых в зависимости от предиката. Например, `[1,2,3,4,5]` становится `[[1,2,3], [4,5]]`, если предикат — «меньше либо равно 3». Вот как это можно реализовать:

```
Array.prototype.partition = function(pred) {
  let passed = []
  let failed = []
  for (let i = 0; i < this.length; i++) {
    if (pred(this[i])) {
      passed.push(this[i])
    } else {
     failed.push(this[i])
   }
  }
  return [ passed, failed ]
}
```

Теперь мы можем применить `partition()` на любом массиве:

```
[1,2,3,4,5].partition(e => e <= 3)
//> [[1, 2, 3], [4, 5]]
```

`[1,2,3,4,5]` называется литералом. Литерал — один из способов создания объекта. Также мы можем использовать фабричные функции или `Object.create()` для создания такого же массива:

```
// Литерал
[1,2,3,4,5]

// Фабричная функция
Array(1,2,3,4,5)

// Object.create
let arr = Object.create(Array.prototype)
arr.push(1)
arr.push(2)
arr.push(3)
arr.push(4)
arr.push(5)
```

Фабричная функция — это функция, которая принимает несколько аргументов и возвращает новый объект, состоящий из этих аргументов. В JavaScript любая функция может возвращать объект. Если она делает это без ключевого слова `new`, то её можно назвать фабричной. Такие функции всегда были привлекательны, так как они дают возможность легко создавать новые объекты, не вникая в сложности классов и ключевого слова `new`.

Выше мы создали массив `arr` с помощью `Object.create()` и поместили в него 5 элементов. `arr` доступны все функции прототипа `Array` вроде `map()`, `pop()`, `slice()` и даже `partition()`, которую мы недавно создали. Добавим ещё функциональности объекту `arr`:

```
arr.hello = () => "hello"
```

Время викторины! Что будет возвращено после запуска кода ниже?

```
arr.partition(e => e < 3) // №1

arr.hello() // №2

let foo = [1,2,3]
foo.hello() // №3

Array.prototype.bye = () => "bye"
arr.bye() // №4
foo.bye() // №5
```

Ответы:

- №1 вернёт `[[1,2],[3,4,5]]`, так как функция `partition()` определена для `Array`, от которого наследуется `arr`.
- №2 вернёт `"hello"`, поскольку мы создали новую функцию `hello()` для объекта `arr`, которая не принимает аргументов и возвращает строку `"hello"`.
- В случае с №3 будет выведена ошибка «TypeError: foo.hello is not a function». Так как `foo` — новый объект, созданный из прототипа `Array`, для которого не определена функция `hello()`, то и для `foo` она не определена.
- №4 и №5 вернут `"bye"`, поскольку строкой выше мы добавили в прототип `Array` новую функцию `bye()`, которую наследуют `arr` и `foo`. Любые изменения в прототипе влияют на объекты на его основе даже после их создания.

Мы разобрались с основами прототипов, поэтому вернёмся к предыдущему примеру и создадим `Person` и `User` с помощью прототипного наследования:

```
function Person(firstName, lastName) {
  this.firstName = firstName
  this.lastName = lastName
}
```

```
Person.prototype.getFullName = function () {
  return this.firstName + ' ' + this.lastName
}
```

Теперь мы можем использовать прототип `Person` таким образом:

```
let person = new Person('Dan', 'Abramov')
person.getFullName() //> Dan Abramov
```

`person` — объект. Если ввести `console.log(person)`, мы увидим следующее:

```
Person {
  firstName: "Dan",
  lastName: "Abramov",
  __proto__: {
    getFullName: f
    constructor: f Person(firstName, lastName)
  },
  __proto__: Object
}
```

Для `User` нам всего лишь нужно расширить класс `Person`:

```
function User(firstName, lastName, email, password) {
  // call super constructor:
  Person.call(this, firstName, lastName)
  this.email = email
  this.password = password
}
User.prototype = Object.create(Person.prototype);
User.prototype.setEmail = function(email) {
  this.email = email
}
User.prototype.getEmail = function() {
  return this.email
}
user.setEmail('dan@abramov.com')
```

`user` — объект. Если ввести `console.log(user)`, мы увидим следующее:

```
User {
  firstName: "Dan",
  lastName: "Abramov",
  email: "dan@abramov.com",
  password: "iLuvES6",
  __proto__: Person {
    getEmail: f ()
    setEmail: f (email)
    __proto__: {
      getFullName: f,
      constructor: f Person(firstName, lastName)
      __proto__: Object
    }
  }
}
```

Что будет, если мы захотим изменить функцию `getFullName()` для `User`? Как этот код повлияет на `person` и `user`?

```
User.prototype.getFullName = function () {
  return 'User Name: ' +
         this.firstName + ' ' +
         this.lastName
}
user.getFullName() //> "User Name: Dan Abramov"
person.getFullName() //> "Dan Abramov"
```

Как и ожидалось, на `person` это никак не отразилось.

Давайте добавим в `Person` атрибут `gender` и соответствующие геттер и сеттер:

```
Person.prototype.setGender = function (gender) {
  this.gender = gender
}
Person.prototype.getGender = function () {
  return this.gender
```

```
}
person.setGender('male')
person.getGender() //> male
user.getGender() //> возвращает undefined, хотя это функция
user.setGender('male')
user.getGender() //> male
```

Изменения затронули как `person`, так и `user`, поскольку `User` наследуется от `Person`, поэтому при изменении последнего меняется и `User`.

Паттерн «декоратор» из прототипного наследования не сильно отличается от классового.

Классы vs Прототипы

> *Дэн Абрамов говорит, что:*
> *Классы скрывают прототипное наследование в основе JS;*
> *Классы побуждают к использованию наследования, но лучше использовать композицию;*
> *Классы, как правило, не дают вам изменить первую плохую структуру проекта, которая пришла вам в голову.*

Вместо классовой иерархии лучше создайте несколько фабричных функций. Они могут вызывать друг друга по цепочке, настраивая своё поведение. Также вы можете научить «основную» фабричную функцию принимать «стратегию», настраивающую поведение других фабричных функций, и передавать её из остальных фабричных функций.

Путь третий: без ООП

Три краеугольных камня ООП — наследование, инкапсуляция и полиморфизм — мощные средства/концепции, но со своими недостатками.

Наследование

Наследование способствует повторному использованию кода, но зачастую приходится брать больше, чем нужно.

> *Джо Армстронг (создатель Erlang) высказал мысль об этом лучшим образом: «Проблема объектно-ориентированных языков заключается во всей их неявной среде, которую они всегда тянут за собой. Вы хотели банан, а получили гориллу, держащую банан, вместе со всеми джунглями».*

Так что делать, если мы получили больше, чем просили? Просто игнорировать то, что нам не нужно? Только в простых случаях. Если нам нужны классы, которые зависят от других классов, а те, в свою очередь, зависят от третьих, то нам придётся иметь дело со всем этим адом зависимостей, что сильно замедляет процессы сборки и отладки. К тому же приложения с такими длинными цепочками зависимостей плохо портируются.

Здесь присутствует проблема хрупкого базового класса, упомянутая выше. Не стоит ожидать, что всё будет идти как по маслу, когда мы соотносим реальные объекты и их классы. Наследование не будет к вам снисходительно, когда вам придётся рефакторить код, особенно базовый класс. Также оно ослабляет инкапсуляцию, ещё один краеугольный камень ООП:

Проблема в том, что если вы наследуете реализацию суперкласса, а затем меняете её, то эти изменения отзываются эхом во всей иерархии классов. В конечном итоге это может повлиять на все подклассы.

Инкапсуляция

Инкапсуляция защищает от влияния извне внутренние переменные каждого объекта. В идеале программа должна состоять из «островов объектов»: каждый из них со своими состояниями, передающий сообщения туда и обратно. Звучит как хорошая идея в том случае, если вы создаёте идеально распределённую систему, но на практике разработка такой программы сложна и вгоняет в определённые рамки.

Много реальных приложений требуют решения проблем с множеством составных частей. Когда вы выбираете объектно-ориентированный подход для разработки программы, вы столкнётесь с разными головоломками вроде «как распределить функциональность приложения между разными объектами?» или «как управлять взаимодействием и обменом данными между разными объектами?». В этой статье есть несколько интересных мыслей насчёт задач, стоящих при проектировании ООП-приложений:

> *Когда мы рассматриваем необходимую функциональность нашего кода, многие из поведений по своей сути являются общими проблемами и потому не относятся к какому-то конкретному типу данных. Тем не менее, эти установки нужно куда-то пристроить, поэтому в итоге мы создаём бессмысленные классы для их содержания. У всех этих бессмысленных сущностей есть привычка становиться*

ещё более бессмысленными: когда у меня есть много объектов Manager, мне приходится создавать ManagerManager.

И ведь так и есть. Такие ManagerManager классы можно зачастую увидеть в продакшне, который по задумке не должен был стать настолько сложным с течением времени.

Далее мы увидим альтернативу ООП — функциональную композицию, где вместо объектов используются функции.

Но перед этим поговорим о последнем краеугольном камне ООП.

Полиморфизм

Полиморфизм позволяет описывать поведение вне зависимости от типа данных. В ООП это означает создание класса или прототипа, который может быть адаптирован объектами, работающими с другими типами данных. Объекты, которые используют полиморфный класс/прототип, должны определить специфичное для типа данных поведение, чтобы всё заработало. Посмотрим на пример.

Предположим, что мы хотим создать общий (полиморфный) объект, который принимает какие-то данные и флаг состояния в качестве параметров. Если состояние говорит, что данные валидные (т.е. `status === true`), на данных можно применить функцию, результат которой будет возвращён вместе с флагом состояния. В противном случае мы не применим функцию и просто вернём данные и флаг.

Начнём с создания полиморфного объекта-прототипа `Maybe`:

```
function Maybe({data, status}) {
  this.data = data
  this.status = status
}
```

`Maybe` — это обёртка для данных. Чтобы обернуть их, мы добавили поле `status`, которое указывает на валидность данных.

Мы можем добавить в прототип функцию `apply()`, которая принимает функцию и применяет её на данных, если статус говорит, что они валидны:

```
Maybe.prototype.apply = function (f) {
  if(this.status) {
    return new Maybe({data: f(this.data), status: this.status})
```

```
  }
  return new Maybe({data: this.data, status: this.status})
}
```

Ещё можем добавить функцию, которая возвращает либо данные, либо сообщение, если с ними что-то не так:

```
Maybe.prototype.getOrElse = function (msg) {
  if(this.status) return this.data
  return msg
}
```

Теперь создадим на основе `Maybe` два объекта: `Number`:

```
function Number(data) {
  let status = (typeof data === 'number')
  Maybe.call(this, {data, status})
}
Number.prototype = Object.create(Maybe.prototype)
```

и `String`:

```
function String(data) {
  let status = (typeof data === 'string')
  Maybe.call(this, {data, status})
}
String.prototype = Object.create(Maybe.prototype)
```

Посмотрим на объекты в действии. Создадим функцию `increment()`, которая определена только для чисел, и `split()`, которая определена только для строк:

```
const increment = num => num + 1
const split = str => str.split('')
```

Так как JavaScript не типобезопасен, вам никто не запретит использовать `increment()` для строки или `split()` для числа. Вы просто увидите ошибку исполнения. Например:

```
let foop = 12
foop.split('')
```

При запуске выдаст TypeError.

Тем не менее если мы используем объекты `Number` и `String`, чтобы обернуть числа и строки до работы с ними, мы сможем предотвратить эти ошибки:

```
let numValid = new Number(12)
let numInvalid = new Number("foo")
let strValid = new String("hello world")
let strInvalid = new String(-1)

let a = numValid.apply(increment).getOrElse('TypeError!')
let b = numInvalid.apply(increment).getOrElse('TypeError Oh no!')
let c = strValid.apply(split).getOrElse('TypeError!')
let d = strInvalid.apply(split).getOrElse('TypeError :(')
```

Что будет выведено в консоль?

```
console.log({a, b, c, d})
```

Поскольку мы описали прототип `Maybe` таким образом, чтобы функция применялась на данных верного типа, результат будет таким:

```
{
 a: 13,
 b: 'TypeError Oh no!',
 c: [ 'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd' ],
 d: 'TypeError :('
}
```

Только что мы сделали что-то вроде монады (хотя мы и не реализовывали `Maybe` по всем законам монад). Монада `Maybe` — это обёртка, которая используется, когда данные могут не пройти проверку или отсутствовать, и вам не важно по какой причине. Как правило, такое случается при извлечении и проверке данных. `Maybe` обрабатывает ошибки при валидации данных или применении функции схожим с try-catch образом. Здесь вся обработка заключается в выводе в консоль, но мы легко можем переделать функцию `getOrElse()` так, чтобы она вызывала другую функцию-обработчик.

У некоторых языков вроде Haskell монада является встроенным типом, но в JavaScript вам придётся создавать свою реализацию. В ES6 появились `Promise` — монады для работы с задержкой. Иногда нам требуются данные, на получение которых требуется время. `Promise` дают возможность писать синхронный код, откладывая работу с данными до того момента, когда они станут доступными. Использование `Promise` — более «чистый» способ асинхронного программирования, чем callback-функции, использование которых может привести к ситуации, известной как «ад обратных вызовов».

Композиция

Как упоминалось ранее, существует нечто гораздо более простое, чем классы/прототипы — функциональная композиция. Её легко можно использовать снова, она инкапсулирует внутренние состояния, выполняет операции на любом типе данных и может быть полиморфной.

JavaScript позволяет легко объединить связанные функции и данные в объекте:

```
const Person = {
  firstName: 'firstName',
  lastName: 'lastName',
  getFullName: function() {
    return `${this.firstName} ${this.lastName}`
  }
}
```

Теперь мы можем использовать объект `Person` таким образом:

```
let person = Object.create(Person)

person.getFullName() //> "firstName lastName"

// Присваиваем значения переменным внутреннего состояния
person.firstName = 'Dan'
person.lastName = 'Abramov'

// Получаем к ним доступ
person.getFullName() //> "Dan Abramov"
```

Создадим объект `User`, склонировав объект `Person`, и добавим туда дополнительные данные и функции:

```
const User = Object.create(Person)
User.email = ''
User.password = ''
User.getEmail = function() {
  return this.email
}
```

Затем мы можем создать экземпляр `User` с помощью `Object.create()`:

```
let user = Object.create(User)
user.firstName = 'Dan'
user.lastName = 'Abramov'
user.email = 'dan@abramov.com'
user.password = 'iLuvES6'
```

Хитрость здесь заключается в использовании `Object.create()` для копирования. Объекты в JavaScript изменяемы, поэтому, когда вы используете присваивание для создания нового объекта и меняете второй объект, это изменяет и исходный объект!

За исключением чисел, строк и булевых значений в JavaScript всё является объектом:

```
// Неправильно
const arr = [1,2,3]
const arr2 = arr
arr2.pop()
arr //> [1,2]
```

Здесь мы использовали ключевое слово `const`, чтобы показать, что оно не защищает вас от изменения объектов. Объекты определяются их ссылкой, поэтому хоть `const` и не даёт переназначить `arr`, вы всё ещё можете его изменить.

Чтобы убедиться, что мы не передаём ссылку объекта, а копируем его, мы используем `Object.create()`.

Как и с кубиками Лего, мы можем создавать копии одного и того же объекта, настраивать их, совмещать и передавать другим объектам для увеличения их возможностей.

В качестве примера определим объект `Customer` с данными и функциями. Когда наш пользователь (`User`) станет клиентом (`Customer`), мы хотим добавить к объекту `user` всё, что есть в `Customer`:

```
const Customer = {
  plan: 'trial'
}
Customer.setPremium = function() {
  this.plan = 'premium'
}
```

Теперь мы можем добавить в объект `user` методы и поля `Customer`:

```
User.customer = Customer
user.customer.setPremium()
```

После выполнения этих двух строк объект `user` будет выглядеть так:

```
{
  firstName: 'Dan',
  lastName: 'Abramov',
  email: 'dan@abramov.com',
  password: 'iLuvES6',
  customer: { plan: 'premium', setPremium: [Function] }
}
```

Когда мы захотим добавить ещё больше возможностей, объекты высшего уровня нам с этим всегда помогут.

Как показано в примере выше, классовому наследованию нам стоит предпочесть композицию, так как она проще, выразительнее и более гибкая.

Заключение

Программистам часто приходится искать компромисс между повторным использованием кода и его масштабируемостью. Вероятно, использование классового ООП имеет смысл для корпоративного ПО, так как оно не сильно меняется. Поведение в ООП чётко прописано в абстрактных классах, но его можно в какой-то степени настроить во время создания экземпляров класса. Это способствует лучшему повторному использованию кода, что экономит разработчикам много времени. Тем не менее, если вы ожидаете, что в будущем много раз придётся дополнять код и даже пересматривать проект, тогда ООП в итоге будет мешать продуктивности разработчика и код станет нетестируемым и сильно связанным со средой.

Материал взят из источника – «Фундаментальные принципы объектно-ориентированного программирования на JavaScript»:
https://tproger.ru/translations/oop-js-fundamentals/