Что такое функциональное программирование?

О том, что такое функциональное программирование, можно почитать в Википедии. А именно, речь там идёт о том, что функциональное программирование — это парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних. Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемости этого состояния.

Сейчас мы, на примерах, разберём некоторые идеи функционального программирования.

Чистые функции

Чистые функции — это первая фундаментальная концепция, которую нужно изучить для того, чтобы понять сущность функционального программирования.

Что такое «чистая функция»? Что делает функцию «чистой»? Чистая функция должна отвечать следующим требованиям:

- Она всегда возвращает, при передаче ей одних и тех же аргументов, один и тот же результат (такие функции также называют детерминированными).
- Такая функция не обладает побочными эффектами.

Рассмотрим первое свойство чистых функций, а именно тот факт, что они, при передаче им одних и тех же аргументов, всегда возвращают один и тот же результат.

Аргументы функций и возвращаемые ими значения

Представим себе, что нам надо создать функцию, которая вычисляет площадь круга. Функция, которая не является чистой, принимала бы, в качестве параметра, радиус круга (`radius`), после чего возвращала бы значение вычисления выражения `radius * radius * PI` :

```
const PI = 3.14;

function calculateArea(radius) {
  return radius * radius * PI;
}

calculateArea(10); // возвращает 314
```

Почему эту функцию нельзя назвать чистой? Дело в том, что она использует глобальную константу, которая не передаётся ей в качестве аргумента.

Теперь представьте себе, что некие математики пришли к выводу о том, что значением

константы `PI` должно являться число `42`, из-за чего было изменено значение этой константы.

Теперь функция, не являющаяся чистой, при передаче ей того же входного значения, числа `10`, вернёт значение `10 * 10 * 42 = 4200`. Получается, что использование здесь такого же, как в прошлом примере, значения параметра `radius`, приводит к возврату функцией другого результата. Исправим это:

```
const PI = 3.14;

function calculateArea(radius, pi) {
  return radius * radius * pi;
}

calculateArea(10, PI); // возвращает 314
```

Теперь мы, вызывая эту функцию, всегда будем передавать ей аргумент `pi`. Как результат, функция будет работать только с тем, что передано ей при вызове, не обращаясь к глобальным сущностям. Если проанализировать поведение этой функции, то можно прийти к следующим выводам:

- Если функции передают аргумент `radius`, равный `10`, и аргумент `pi`, равный `3.14`, она всегда будет возвращать один и тот же результат — `314`.
- При вызове её с аргументом `radius`, равным `10` и аргументом `pi`, равным `42`, она всегда будет возвращать `4200`.

Чтение файлов

Если наша функция выполняет чтение файлов, то чистой она не будет. Дело в том, что содержимое файлов может меняться.

```
function charactersCounter(text) {
  return `Character count: ${text.length}`;
}

function analyzeFile(filename) {
  let fileContent = open(filename);
  return charactersCounter(fileContent);
}
```

Генерирование случайных чисел

Любая функция, которая полагается на генератор случайных чисел, не может быть чистой.

```
function yearEndEvaluation() {
  if (Math.random() > 0.5) {
    return "You get a raise!";
  } else {
    return "Better luck next year!";
  }
}
```

Теперь поговорим о побочных эффектах.

Побочные эффекты

Примером побочного эффекта, который может проявиться при вызове функции, является модификация глобальных переменных или аргументов, передаваемых функциям по ссылке.

Предположим, нам нужно создать функцию, которая принимает целое число и выполняет увеличение этого числа на 1. Вот как может выглядеть реализация подобной идеи:

```
let counter = 1;

function increaseCounter(value) {
  counter = value + 1;
}

increaseCounter(counter);
console.log(counter); // 2
```

Тут имеется глобальная переменная `counter`. Наша функция, не являющаяся чистой, получает это значение в виде аргумента и перезаписывает его, добавляя к его прежнему значению единицу.

Глобальная переменная меняется, подобное в функциональном программировании не приветствуется.

В нашем случае модификации подвергается значение глобальной переменной. Как в этих условиях сделать функцию `increaseCounter()` чистой? На самом деле, это очень просто:

```
let counter = 1;

function increaseCounter(value) {
  return value + 1;
}

increaseCounter(counter); // 2
console.log(counter); // 1
```

Как видите, функция возвращает `2`, но при этом значение глобальной переменной `counter` не меняется. Тут можно сделать вывод о том, что функция возвращает переданное ей значение, увеличенное на `1`, при этом ничего не изменяя.

Если следовать двум вышеописанным правилам написания чистых функций, это приведёт к тому, что в программах, созданных с использованием таких функций, будет легче ориентироваться. Окажется, что каждая функция будет изолирована и не будет оказывать воздействия на внешние по отношению к ней части программы.

Чистые функции стабильны, единообразны и предсказуемы. Получая одни и те же входные данные, такие функции всегда возвращают один и тот же результат. Это избавляет программиста от попыток учесть возможность возникновения ситуаций, в которых передача функции одних и тех же параметров приводит к разным результатам, так как подобное при использовании чистых функций просто невозможно.

Сильные стороны чистых функций

Среди сильных сторон чистых функций можно отметить тот факт, что код, написанный с их использованием, легче тестировать. В частности, не нужно создавать неких объектов-заглушек. Это позволяет выполнять модульное тестирование чистых функций в различных контекстах:

- Если функции передаётся параметр A — ожидается возврат значения B.
- Если функции передаётся параметр C — ожидается возврат значения D.

В качестве простого примера этой идеи можно привести функцию, которая принимает массив чисел, и при этом ожидается, что она увеличит на единицу каждое число этого массива, вернув новый массив с результатами:

```
let list = [1, 2, 3, 4, 5];

function incrementNumbers(list) {
  return list.map(number => number + 1);
}
```

Здесь мы передаём функции массив чисел, после чего используем метод массивов `map()`, который позволяет модифицировать каждый элемент массива и формирует новый массив, возвращаемый функцией. Вызовем функцию, передав ей массив `list`:

```
incrementNumbers(list); // возвращает [2, 3, 4, 5, 6]
```

От этой функции ожидается, что, приняв массив вида `[1, 2, 3, 4, 5]`, она возвратит новый массив `[2, 3, 4, 5, 6]`. Именно так она и работает.

Иммутабельность

Иммутабельность некоей сущности можно описать как то, что с течением времени она не меняется, или как невозможность изменений этой сущности.

Если иммутабельный объект пытаются изменить — сделать этого не удастся. Вместо этого нужно будет создать новый объект, содержащий новые значения.

Например, в JavaScript часто используется цикл `for`. В ходе его работы, как показано ниже, применяются мутабельные переменные:

```
var values = [1, 2, 3, 4, 5];
var sumOfValues = 0;

for (var i = 0; i < values.length; i++) {
  sumOfValues += values[i];
}

sumOfValues // 15
```

На каждой итерации цикла меняется значение переменной `i` и значение глобальной переменной (её можно считать состоянием программы) `sumOfValues`. Как в подобной ситуации поддерживать неизменность сущностей? Ответ лежит в использовании рекурсии.

```
let list = [1, 2, 3, 4, 5];
let accumulator = 0;

function sum(list, accumulator) {
  if (list.length == 0) {
    return accumulator;
  }

  return sum(list.slice(1), accumulator + list[0]);
}

sum(list, accumulator); // 15
list; // [1, 2, 3, 4, 5]
accumulator; // 0
```

Тут имеется функция `sum()`, которая принимает массив чисел. Эта функция вызывает сама себя до тех пор, пока массив не опустеет (это базовый случай нашего рекурсивного алгоритма). На каждой такой «итерации» мы добавляем значение одного из элементов массива к параметру функции `accumulator`, не затрагивая при этом глобальной переменной `accumulator`. При этом глобальные переменные `list` и `accumulator` остаются неизменными, до и после вызова функции в них хранятся одни и те же значения.

Надо отметить, что для реализации подобного алгоритма можно воспользоваться методом массивов `reduce`. Об этом мы поговорим ниже.

В программировании распространена задача, когда нужно, на основе некоего шаблона объекта, создать его окончательное представление. Представьте, что у нас есть строка, которую нужно преобразовать в вид, подходящий для использования в качестве части URL, ведущего к некоему ресурсу.

Если решить эту задачу, используя Ruby и задействовав принципы ООП, то мы сначала создадим класс, скажем, назвав его `UrlSlugify`, после чего создадим метод этого класса `slugify!`, который используется для преобразования строки.

```
class UrlSlugify
  attr_reader :text

  def initialize(text)
    @text = text
  end

  def slugify!
    text.downcase!
    text.strip!
    text.gsub!(' ', '-')
  end
end

UrlSlugify.new(' I will be a url slug   ').slugify! # "i-will-be-a-url-slu
g"
```

Алгоритм мы реализовали, и это замечательно. Тут мы видим императивный подход к программированию, когда мы, обрабатывая строку, расписываем каждый шаг её трансформации. А именно — сначала приводим её символы к нижнему регистру, потом убираем ненужные пробелы, и, наконец, меняем оставшиеся пробелы на тире.

Однако в ходе такого преобразования происходит мутация состояния программы.

Справиться с проблемой мутации можно, выполняя композицию функций или объединение вызовов функций в цепочку. Другими словами, результат, возвращаемый функцией, будет использован как входные данные для следующей функции, и так — для всех функций, объединённых в цепочку. При этом исходная строка меняться не будет.

```
let string = " I will be a url slug   ";

function slugify(string) {
  return string.toLowerCase()
    .trim()
    .split(" ")
    .join("-");
}
```

```
slugify(string); // i-will-be-a-url-slug
```

Здесь мы используем следующие функции, представленные в JavaScript стандартными методами строк и массивов:

- `toLowerCase` : преобразует символы строки к нижнему регистру.
- `trim` : убирает пробельные символы из начала и конца строки.
- `split` : разбивает строку на части, помещая слова, разделённые пробелами, в массив.
- `join` : формирует на основе массива со словами строку, слова в которой разделены тире.

Эти четыре функции и позволяют создать функцию для преобразования строки, не меняющую саму эту строку.

Ссылочная прозрачность

Создадим функцию `square()` , возвращающую результат умножения числа на это же число:

```
function square(n) {
  return n * n;
}
```

Это чистая функция, которая всегда, для одного и того же входного значения, будет возвращать одно и то же выходное значение.

```
square(2); // 4
square(2); // 4
square(2); // 4
// ...
```

Например, сколько бы ей ни передавали число `2` , эта функция всегда будет возвращать число `4` . В результате оказывается, что вызов вида `square(2)` можно заменить числом `4` . Это означает, что наша функция обладает свойством ссылочной прозрачности.

В целом, можно сказать, что если функция неизменно возвращает один и тот же результат для одних и тех же передаваемых ей входных значений, она обладает ссылочной прозрачностью.

Чистые функции + иммутабельные данные = ссылочная прозрачность

Задействовав идею, вынесенную в заголовок этого раздела, можно мемоизировать функции. Предположим, у нас имеется такая функция:

```
function sum(a, b) {
  return a + b;
}
```

Мы вызываем её так:

```
sum(3, sum(5, 8));
```

Вызов `sum(5, 8)` всегда даёт `13`. Поэтому вышеприведённый вызов можно переписать так:

```
sum(3, 13);
```

Это выражение, в свою очередь, всегда даёт `16`. Как результат, его можно заменить числовой константой и мемоизироватьего.

Функции как объекты первого класса

Идея восприятия функций как объектов первого класса заключается в том, что такие функции можно рассматривать как значения и работать с ними как с данными. При этом можно выделить следующие возможности функций:

- Ссылки на функции можно хранить в константах и переменных и через них обращаться к функциям.
- Функции можно передавать другим функциям в качестве параметров.
- Функции можно возвращать из других функций.

То есть, речь идёт о том, чтобы рассматривать функции как значения и обращаться с ними как с данными. При таком подходе можно комбинировать различные функции в процессе создания новых функций, реализующих новые возможности.

Представьте, что у нас имеется функция, которая складывает переданные ей два числовых значения, после чего умножает их на `2` и возвращает то, что у неё получилось:

```
function doubleSum(a, b) {
  return (a + b) * 2;
}
```

Теперь напишем функцию, которая вычитает из первого переданного ей числового значения второе, умножает то, что получилось, на `2`, и возвращает вычисленное значение:

```
function doubleSubtraction(a, b) {
  return (a - b) * 2;
}
```

Эти функции имеют схожую логику, различаются они лишь тем, какие именно операции они производят с переданными им числами. Если мы можем рассматривать функции как значения и передавать их как аргументы другим функциям, это означает, что мы можем создать функцию, которая принимает и использует другую функцию, описывающую особенности выполняемых вычислений. Эти рассуждения позволяют нам выйти на следующие конструкции:

```
function sum(a, b) {
  return a + b;
}

function subtraction(a, b) {
  return a - b;
}

function doubleOperator(f, a, b) {
  return f(a, b) * 2;
}

doubleOperator(sum, 3, 1); // 8
doubleOperator(subtraction, 3, 1); // 4
```

Как видите, теперь у функции `doubleOperator()` имеется параметр `f`, а функция, которую он представляет, используется для обработки параметров `a` и `b`.
Функции `sum()` и `substraction()`, передаваемые функции `doubleOperator()`, фактически, позволяют управлять поведением функции `doubleOperator()`, меняя его в соответствии с реализованной в них логикой.

Функции высшего порядка

Говоря о функциях высшего порядка мы имеем в виду функции, которые характеризуются хотя бы одной из следующих особенностей:

- Функция принимает другую функцию в качестве аргумента (таких функций может быть и несколько).
- Функция возвращает другую функцию в качестве результата своей работы.

Возможно, вы уже знакомы со стандартными методами JS-массивов `filter()`, `map()` и `reduce()`. Поговорим о них.

Фильтрация массивов и метод filter()

Предположим, у нас есть некая коллекция элементов, которую мы хотим отфильтровать по какому-то атрибуту элементов этой коллекции и сформировать новую коллекцию. Функция `filter()` ожидает получить какой-то критерий оценки элементов, на основе которого она и определяет, нужно или не нужно включать некий элемент в результирующую коллекцию. Этот критерий задаёт передаваемая ей функция, которая возвращает `true` в том случае, если функция `filter()` должна включить элемент в итоговую коллекцию, а в противном случае возвращает `false`.

Представим, что у нас имеется массив целых чисел и мы хотим отфильтровать его, получив новый массив, в котором содержатся только чётные числа из исходного массива.

Императивный подход

При применении императивного подхода к решению этой задачи средствами JavaScript нам нужно реализовать следующую последовательность действий:

- Создать пустой массив для новых элементов (назовём его `evenNumbers`).
- Перебрать исходный массив целых чисел (назовём его `numbers`).
- Поместить чётные числа, обнаруженные в массиве `numbers`, в массив `evenNumbers`.

Вот как выглядит реализация этого алгоритма:

```
var numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var evenNumbers = [];

for (var i = 0; i < numbers.length; i++) {
  if (numbers[i] % 2 == 0) {
    evenNumbers.push(numbers[i]);
  }
}

console.log(evenNumbers); // (6) [0, 2, 4, 6, 8, 10]
```

Кроме того, мы можем написать функцию (назовём её `even()`), которая, если число является чётным, возвращает `true`, а если нечётным — `false`, после чего передать её методу массива `filter()`, который, проверив с её помощью каждый элемент массива, сформирует новый массив, содержащий лишь чётные числа:

```
function even(number) {
  return number % 2 == 0;
}

let listOfNumbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
listOfNumbers.filter(even); // [0, 2, 4, 6, 8, 10]
```

Вот, кстати, решение одной интересной задачи, касающейся фильтрации массива, которое я выполнил, работая над задачами по функциональному программированию на Hacker Rank. По условию задачи нужно было отфильтровать массив целых чисел, выведя лишь те его элементы, которые меньше заданного значения `x`.

Императивное решение этой задачи на JavaScript может выглядеть так:

```
var filterArray = function(x, coll) {
  var resultArray = [];

  for (var i = 0; i < coll.length; i++) {
    if (coll[i] < x) {
      resultArray.push(coll[i]);
    }
  }

  return resultArray;
}

console.log(filterArray(3, [10, 9, 8, 2, 7, 5, 1, 3, 0])); // (3) [2, 1, 0]
```

Суть императивного подхода заключается в том, что мы расписываем последовательность действий, выполняемых функцией. А именно — мы описываем перебор массива, сравнение текущего элемента массива с `x` и помещение этого элемента в массив `resultArray` в том случае, если он проходит проверку.

Декларативный подход

Как перейти к декларативному подходу решения этой задачи и соответствующему использованию метода `filter()`, являющегося функцией высшего порядка? Например, это может выглядеть так:

```
function smaller(number) {
  return number < this;
}

function filterArray(x, listOfNumbers) {
  return listOfNumbers.filter(smaller, x);
}
```

```
let numbers = [10, 9, 8, 2, 7, 5, 1, 3, 0];

filterArray(3, numbers); // [2, 1, 0]
```

Возможно, вам в этом примере необычным покажется использование ключевого слова `this` в функции `smaller()`, но ничего сложного тут нет. Ключевое слово `this` представляет собой второй аргумент метода `filter()`. В нашем примере это — число `3`, представленное параметром `x` функции `filterArray()`. На это число и указывает `this`.

Такой же подход можно использовать и в том случае, если в массиве хранятся сущности, обладающие достаточно сложной структурой, например — объекты. Предположим, у нас имеется массив, хранящий объекты, содержащие имена людей, представленные свойством `name`, и сведения о возрасте этих людей, представленные свойством `age`. Вот как выглядит такой массив:

```
let people = [
  { name: "TK", age: 26 },
  { name: "Kaio", age: 10 },
  { name: "Kazumi", age: 30 }
];
```

Мы хотим этот массив отфильтровать, выбрав из него только те объекты, которые представляют собой людей, чей возраст превысил `21` год. Вот как можно решить эту задачу:

```
function olderThan21(person) {
  return person.age > 21;
}

function overAge(people) {
  return people.filter(olderThan21);
}

overAge(people); // [{ name: 'TK', age: 26 }, { name: 'Kazumi', age: 30 }]
```

Здесь у нас имеется массив с объектами, представляющими людей. Мы проверяем элементы этого массива с помощью функции `olderThan21()`. В данном случае мы, при проверке, обращаемся к свойству `age` каждого элемента, проверяя, превышает ли значение этого свойства `21`. Данную функцию мы передаём методу `filter()`, который и фильтрует массив.

Обработка элементов массивов и метод map()

Метод `map()` используется для преобразования элементов массивов. Он применяет к каждому элементу массива переданную ему функцию, после чего строит новый массив, состоящий из изменённых элементов.

Продолжим эксперименты с уже знакомым вам массивом `people`. Теперь мы не собираемся фильтровать этот массив, основываясь на свойстве объектов `age`. Нам нужно сформировать на его основе список строк вида `TK is 26 years old`. Строки, в которые превращаются элементы, при таком подходе будут строиться по шаблону `p.name is p.age years old`, где `p.name` и `p.age` — это значения соответствующих свойств элементов массива `people`.

Императивный подход к решению этой задачи на JavaScript выглядит так:

```
var people = [
  { name: "TK", age: 26 },
  { name: "Kaio", age: 10 },
  { name: "Kazumi", age: 30 }
];

var peopleSentences = [];

for (var i = 0; i < people.length; i++) {
  var sentence = people[i].name + " is " + people[i].age + " years old";
  peopleSentences.push(sentence);
}

console.log(peopleSentences); // ['TK is 26 years old', 'Kaio is 10 years o
ld', 'Kazumi is 30 years old']
```

Если прибегнуть к декларативному подходу, то получится следующее:

```
function makeSentence(person) {
  return `${person.name} is ${person.age} years old`;
}

function peopleSentences(people) {
  return people.map(makeSentence);
}

peopleSentences(people); // ['TK is 26 years old', 'Kaio is 10 years old',
 'Kazumi is 30 years old']
```

Собственно говоря, основная мысль тут заключается в том, что с каждым элементом исходного массива нужно что-то сделать, после чего — поместить его в новый массив.

Вот ещё одна задача с Hacker Rank, которая посвящена обновлению списка. А именно, речь идёт о том, чтобы поменять значения элементов существующего числового массива на их абсолютные значения. Так, например, при обработке массива `[1, 2, 3, -4, 5]` он приобретёт вид `[1, 2, 3, 4, 5]` так как абсолютное значение `-4` равняется `4`.

Вот пример простого решения этой задачи, когда мы перебираем массив и меняем значения

его элементов на их абсолютные значения.

```
var values = [1, 2, 3, -4, 5];

for (var i = 0; i < values.length; i++) {
  values[i] = Math.abs(values[i]);
}

console.log(values); // [1, 2, 3, 4, 5]
```

Тут для преобразования значений элементов массива использован метод `Math.abs()`, изменённые элементы записываются туда же, где они были до преобразования.

Это решение не является примером функционального подхода к программированию.

Первое, что надо вспомнить в связи с этим решением, заключается в том, что выше мы говорили о важности иммутабельности. Эта концепция делает результаты работы функций предсказуемыми и ведёт к стабильной работе функций. При таком подходе, решая нашу задачу, нужно создать новый массив, содержащий абсолютные значения элементов исходного массива.

Второй вопрос, который стоит себе задать в этой ситуации, касается метода массивов `map()`. Почему бы не воспользоваться им?

Вооружившись этими идеями, я решил поэкспериментировать с методом `abs()`, взглянуть на то, как он обрабатывает разные числа.

```
Math.abs(-1); // 1
Math.abs(1); // 1
Math.abs(-2); // 2
Math.abs(2); // 2
```

Как видно, он возвращает положительные числа, представляющие собой абсолютное значение передаваемых ему чисел.

После того, как мы поняли принцип преобразования числа к его абсолютному значению, мы можем использовать `Math.abs()` в качестве аргумента метода массива `map()`. Помните о том, что функции высшего порядка могут принимать другие функции и использовать их? Метод `map()` является именно такой функцией. Вот как решение нашей задачи будет выглядеть теперь:

```
let values = [1, 2, 3, -4, 5];

function updateListMap(values) {
  return values.map(Math.abs);
}
```

```
updateListMap(values); // [1, 2, 3, 4, 5]
```

Уверен, никто не поспорит с тем, что оно, в сравнении с предыдущим вариантом, получилось куда более простым, предсказуемым и понятным.

Преобразование массивов и метод reduce()

В основу метода `reduce()` положена идея преобразования массива к единственному значению путём комбинации его элементов с использованием некоей функции.

Распространённым примером применения этого метода является нахождение общей суммы по некоему заказу. Представьте, что речь идёт об интернет-магазине. Покупатель добавляет в корзину товары `Product 1`, `Product 2`, `Product 3` и `Product 4`. После этого нам надо найти общую стоимость этих товаров.

Если воспользоваться для решения этой задачи императивным подходом, то нужно перебрать список товаров из корзины и сложить их стоимости. Например, это может выглядеть так:

```
var orders = [
  { productTitle: "Product 1", amount: 10 },
  { productTitle: "Product 2", amount: 30 },
  { productTitle: "Product 3", amount: 20 },
  { productTitle: "Product 4", amount: 60 }
];

var totalAmount = 0;

for (var i = 0; i < orders.length; i++) {
  totalAmount += orders[i].amount;
}

console.log(totalAmount); // 120
```

Если воспользоваться для решения этой задачи методом массивов `reduce()`, то мы можем создать функцию (`sumAmount()`), используемую для вычисления суммы элементов массива, после чего передать её методу `reduce()`:

```
let shoppingCart = [
  { productTitle: "Product 1", amount: 10 },
  { productTitle: "Product 2", amount: 30 },
  { productTitle: "Product 3", amount: 20 },
  { productTitle: "Product 4", amount: 60 }
];

const sumAmount = (currentTotalAmount, order) => currentTotalAmount + orde
r.amount;
```

```
function getTotalAmount(shoppingCart) {
  return shoppingCart.reduce(sumAmount, 0);
}

getTotalAmount(shoppingCart); // 120
```

Тут имеется массив `shoppingCart`, представляющий собой корзину покупателя, функция `sumAmount()`, которая принимает элементы массива (объекты `order`, при этом нас интересуют их свойства `amount`), и текущее вычисленное значение суммы их стоимостей — `currentTotalAmount`.

При вызове метода `reduce()`, выполняемого в функции `getTotalAmount()`, ему передаётся функция `sumAmount()` и начальное значение счётчика, которое равняется `0`.

Ещё один способ решения нашей задачи заключается в комбинации методов `map()` и `reduce()`. Что имеется в виду под их «комбинацией»? Дело тут в том, что мы можем использовать метод `map()` для преобразования массива `shoppingCart` в массив, содержащий лишь значения свойств `amount` хранящихся в этом массиве объектов, а затем воспользоваться методом `reduce()` и функцией `sumAmount()`. Вот как это выглядит:

```
const getAmount = (order) => order.amount;
const sumAmount = (acc, amount) => acc + amount;

function getTotalAmount(shoppingCart) {
  return shoppingCart
    .map(getAmount)
    .reduce(sumAmount, 0);
}

getTotalAmount(shoppingCart); // 120
```

Функция `getAmount()` принимает объект и возвращает только его свойство `amount`. После обработки массива с использованием метода `map()`, которому передана эта функция, получается новый массив, который выглядит как `[10, 30, 20, 60]`. Затем, с помощью `reduce()`, мы находим сумму элементов этого массива.

Совместное использование методов filter(), map() и reduce()

Выше мы поговорили о том, как работают функции высшего порядка, рассмотрели методы массивов `filter()`, `map()` и `reduce()`. Теперь, на простом примере, рассмотрим использование всех трёх этих функций.

Продолжим пример с интернет-магазином. Предположим, корзина покупателя сейчас выглядит так:

```
let shoppingCart = [
  { productTitle: "Functional Programming", type: "books", amount: 10 },
  { productTitle: "Kindle", type: "eletronics", amount: 30 },
  { productTitle: "Shoes", type: "fashion", amount: 20 },
  { productTitle: "Clean Code", type: "books", amount: 60 }
]
```

Нам нужно узнать стоимость книг в заказе. Вот какой алгоритм можно предложить для решения этой задачи:

- Отфильтровать массив по значению свойства `type` его элементов, учитывая то, что нас интересует значение этого свойства `books`.
- Преобразовать полученный массив в новый, содержащий лишь стоимости товаров.
- Сложить все стоимости товаров и получить итоговое значение.

Вот как выглядит код, реализующий этот алгоритм:

```
let shoppingCart = [
  { productTitle: "Functional Programming", type: "books", amount: 10 },
  { productTitle: "Kindle", type: "eletronics", amount: 30 },
  { productTitle: "Shoes", type: "fashion", amount: 20 },
  { productTitle: "Clean Code", type: "books", amount: 60 }
]

const byBooks = (order) => order.type == "books";
const getAmount = (order) => order.amount;
const sumAmount = (acc, amount) => acc + amount;

function getTotalAmount(shoppingCart) {
  return shoppingCart
    .filter(byBooks)
    .map(getAmount)
    .reduce(sumAmount, 0);
}

getTotalAmount(shoppingCart); // 70
```