# Функции

function add(a,b){

console.log(a+b);

};

Можно тот же код записать по-другому.

var add = function(a,b){

console.log(a + b);

};

Одним из больших плюсов Javascript является то, что функции в нем являются полноценными объектами. Настоящие First Class Citizen.

Приведенная выше функция работает с побочными эффектами, то есть изменяет состояние внешнего мира. Это выражается в использовании console.log().

А теперь рассмотрим пример чистой функции.

var add = function(a,b){

return a + b;

};

Чистые функции не производят побочных эффектов. Вы передаете в них какие-то данные, и они отдают вам данные обратно. Их очень просто анализировать. Их легче тестировать. Не надо проверять внешние зависимости. Поэтому в большинстве случаев чистые функции предпочтительнее функций с побочным эффектам.

Но, с другой стороны, программа состоящая исключительно из чистых функций не несет практического смысла. Она ничего не считывает и не выводит.

Поэтому логично будет писать программы таким образом, чтобы отделить чистые функции от функций с побочными эффектами и таким образом упростить себе жизнь.

Итак, первое правило функционального программирования — используем чистые функции.

# Функции высшего порядка

Поскольку функции — это объекты, то мы можем их передавать в другие функции. А функции высшего порядка, это функции, которые возвращают функции, или которые принимают функции в качестве параметра.

Вот простой пример функции, которая тоже возвращает функцию.

function makeAdder(base){

return function(num){

return base + num;

}

}

И пример ее использования.

var add2 = makeAdder(2);

add2(3); *//5*

add2(7); *//9*

А вот достаточно известный пример функции высшего порядка

var el = document.getElementById("btn");

el.addEventListener("click", function (event){

});

addEventListener в качестве параметра получает функцию. То есть addEventListener является функцией высшего порядка.

И функция-обработчик будет вызвана, когда произойдет какое-то событие.

Возможно вам более привычен другой вариант:

$("input[type=submit]").on("clink", function(event){

*// ...*

});

Итак еще раз определение:

ФВП — это функции, которые либо возвращают функции либо принимают функции в качестве параметров.

# Циклы

Под циклом будем понимать стандартное лобовое решение. Примерно такое

for(var i =0; i<n; ++1){

*//*

}

Или такое

while(n--){

*// ...*

}

*// ...*

Зачем мы используем циклы? Давайте разберем несколько стандартных вариантов использования, и увидим, что циклы — не всегда лучшее решение.

Первый вариант — обход массивов и списков

for(var i =0; l< arr.length; i<l; ++i){

console.log(arr[i]);

}

Обычно такой обход используется совместно с побочными эффектами. И обычно эти эффекты немного полезнее, чем простой вывод в консоль.

Второй вариант — вытаскивание данных из списков

var names = [];

for (var i =0; l= tweeps.length; i< l; ++i) {

names.push(tweeps[i].name);

}

В этом случае — список пользователей твиттера.

При помощи цикла мы получаем список имен наших пользователей

Еще один вариант использования — агрегация данных в списке:

var html = "";

for(var i =0; l = items.length, i<l, i++){

html += '<li>' + items[i] + '</li>';

}

this.list.innerHTML = html;

То есть мы агрегируем данные списка, и получаем на выходе другую структуру данных.

# foreach

Чем можно заменить подобный цикл?

for (var i =1; l = arr.length; i< l; ++i){

console.log(arr[i]);

}

Например foreach.

Array.prototype.forEach

arr.forEach(function(item){

console.log(item);

});

Вместо пробегать по списку руками, мы можем использовать

метод массива. Передадим туда функцию обрабатывающую каждый элемент и получим необходимый результат.

Но в чем *принципиальная* разница между этим

for (var i =1; l = arr.length; i< l; ++i){

console.log(arr[i]);

}

и этим

arr.forEach(function(item){

console.log(item);

});

К сожалению, синтаксис для описания функций в JS достаточно многословен, поэтому мы не получили значительной экономии в количестве написанного текста.

Но есть еще кое что. Глядя на код, можно сказать, чему уделяется внимание в каждой из реализаций.

Первый участок ориентирован на саму механику цикла. Взять число, увеличить его на единицу, получить элемент массива по индексу, произвести действие.

Второй пример гораздо проще для понимания. Мы делаем что-то с каждым элементом списка.

Во втором примере уровень абстракции гораздо выше. И он позволяет подходить к решению задачи с другой стороны.

Итак, для чего мы можем использовать циклы:

* Побочные эффекты
* Трансформация
* Фильтры
* Комбинирование элементов
* Еще куча вариантов

# map

Давайте рассмотрим еще одну функцию, которая есть в Javascript.

var names = [];

for( var i =0, l= tweeps.length, i< l; ++i){

names.push(tweeps[i].name);

}

Это абстракция, которая соответствует трансформации списка.

Используя map мы можем решить эту задачу гораздо проще

*//Array.prototype.map*

var names = tweeps.map(function (tweep){

return tweep.name;

});

Мы избавились от временных переменных, от описания цикла. Прямое и понятный код. И поскольку функция обработки достаточно короткая, мы можем уместить все в одну строку.

var names = tweeps.map(function(t){return t.name;});

другой пример:

var str = "mentioned by";

for(var i =0; l= tweeps.length; i < l; ++i){

str += tweeps[i].name;

if(i< tweeps.length-1) {str += ", "}

}

Достаточно корявый пример. Тут может быть куча ошибок с индексацией и получением элемента массива.

Давайте проанализируем, что мы реально делаем в этом примере:

* Вытаскиваем имена пользователей
* Объединяем имена пользователей (при этом в конце списка не должно быть запятой)
* Используем запятую в качестве разделителя

Перепишем, используя map и join

var str = "mentioned by " + tweeps.map(function(t){

return t.name;

}).join(", ");

Возможностей ошибиться стало гораздо меньше.

Но можно ли сделать лучше?

Давайте введем еще одну функцию высшего порядка, которую будем использовать для доступа к свойствам объектов.

Назовем ее *prop*

function prop(name){

return function (object){

return object[name];

}

}

На первый взгляд она достаточно бессмысленна. Мы передаем в нее имя,

а она возвращает нам функцию, куда передается объект из которого вытаскиваем необходимое поле.

Какое-то запутанное объяснение получилось. Давайте просто попробуем использовать эту функцию на реальной задаче.

var str = "Mentioned by " + tweeps.map(prop ("name")).join(", ");

Итак, еще один однострочник. Достаточно неплохая выразительность. А функция prop не так уж бесполезна.

# reduce

Это прабабушка для for, foreach, while и прочих подобных структур. Эта функция также известна под именем *fold*.

Опять начнем с примитивного примера.

var totalLength = 0;

for(var i=0; i< buffers.length; i++){

total.Length += buffers[i].length;

}

Просто суммируем длину буферов.

Какие шаги мы должны выполнить?

* Получить длины буферов
* Просуммировать длины

Use the function, Luke.

Сначала мы используем map, чтобы получить список, содержащий длины буферов

var totalLength = buffers.

map(function (buffer) {return buffer.length; })

А на втором шаге мы применим reduce, чтобы получить их сумму.

var totalLength = buffers.

map(function (buffer) {return buffer.length; }).

reduce(function(sum, curr){return sum+curr;}, 0);

Если вы не знакомы с reduce, то она работает очень просто. В нее передается функция-аккумулятор, которая будет применяться к каждому элементу и начальное значение для функции аккумулятора.

Как-то опять слишком сложно. Давайте просто посмотрим, что будет происходить, если мы применим reduce к простому списку.

[10, 5, 15, 10, 10].reduce(function(sum, curr){return sum+curr;}, 0);

*// [10, 5, 15, 10, 10]*

*// sum curr*

*// => 0, 10 => 10*

*// => 10, 5 => 15*

*// => 15, 15 => 30*

*// => 30, 10 => 40*

*// => 40, 10 => 50*

Итак, с помощью reduce мы можем легко просуммировать элементы списка.

Но у нас уже было что-то похожее. Сравните.

function (prev, curr){return prev + curr;}

и

function add(a,b){

return a+b;

}

Поэтому мы можем немного упростить нашу функцию по вычислению суммарной длины буферов.

var totalLength = buffers.

map(function (buffer) {return buffer.length; }).

reduce(add, 0);

Теперь стало яснее? reduce просто суммирует все элементы списка, применяя функцию add. Начальное значение суммы равно нулю. Что может быть проще?

Но на этом упрощения не заканчиваются. Сравните

function (buffer) {return buffer.length; }

и

prop("length")

Брюки превращаются…

var totalLength = buffers.

map(prop("length")).

reduce(add, 0);

В элегантные шорты.

Ну и, естественно, мы можем записать это в одну строку

var totalLength = buffers.map(prop("length")).reduce(add, 0);

Использование свертки (reduce) вместо циклов позволяет нам думать на другом уровне абстракции. Мы совершаем операции над списком, а не на уровне каждого элемента.

# Асинхронные вызовы

Но использование reduce для суммирования списков — очень упрощенный пример. Идея гораздо мощнее. Давайте разберем еще один пример.

Одна из проблем использования Javascript в браузере заключается в том, что все выполняется в одном потоке, и поэтому мы должны использовать коллбеки.

Задача.

* Загрузить несколько скриптов
* Склеить их
* сохранить порядок скриптов при склейке

То есть надо написать функцию примерно такого вида:

combine(["/jquery.js",

"/underscore.js",

"/backbone.js"], function(content){

*// content должен содержать все скрипты, склеенные в правильном порядке.*

});

Напишем реализацию функции combine. Сначала — лобовой подход.

function combine(scripts, callback){

var data [];

for(var i =0; l = scripts.length; i< l; ++i){

*// ....*

}

}

Для получения скриптов было бы логично использовать jQuery.ajax:

function combine(scripts, callback){

var data [];

for(var i =0; l = scripts.length; i< l; ++i){

jQuery.ajax({

url: scripts[i],

success : function(response){

*// ....*

}

});

}

}

Подобный код не будет тормозить браузер, поскольку запросы к серверу будут отправлены асинхронно. То есть при выполнении будет 3 параллельных запроса.

Напишем обработчик для успешного скачивания скрипта.

function combine(scripts, callback){

var data [];

for(var i =0; l = scripts.length; i< l; ++i){

jQuery.ajax({

url: scripts[i],

success : function(response){

data[i] = response;

if(data.length === scripts.length){

callback(data.join(""));

}

}

});

}

}

Вроде бы функция готова. Но есть два но.

Во-первых уродливо, во-вторых — оно не будет работать.

С чем тут могут быть проблемы? С областями видимости Javascript. В этом языке область видимость не поблочная, а функциональная. то есть все 3 функции будут видеть одно и то значение переменной i. Поскольку цикл отработает раньше, чем придут ответы от сервера, все три функции будут работать с i == 3;

Эта проблема решается стандартным способом — мы кэшируем значение переменной цикла. Но нельзя сказать, что код от этого стал красивее.

function combine(scripts, callback){

var data [];

for(var i =0; l = scripts.length; i< l; ++i){

(function (i){

jQuery.ajax({

url: scripts[i],

success : function(response){

data[i] = response;

if(data.length === scripts.length){

callback(data.join(""));

}

}

});

}(i));

}

}

Почти даже работает. Для того, чтобы избавиться от замыканий и хитрых переменных, можно использовать foreach

function combine(scripts, callback){

var data [];

scripts.forEach(function(script,i){

jQuery.ajax({

url: scripts[i],

success : function(response){

data[i] = response;

if(data.length === scripts.length){

callback(data.join(""));

}

}

});

});

}

}

Лучше, конечно, но все равно страшновато. Кстати, код все еще не будет работать правильно. Его можно допилить до работоспособного состояния, но это создаст дополнительные сложности и в разработке и в последующей поддержке.

# Continuation Passing Style

Для избавления от головной боли воспользуемся библиотекой

[github.com/caolan/async](http://github.com/caolan/async)

Для работы будем использовать такую вещь как CPS.

Звучит гораздо страшнее, чем есть на самом деле. Это функция, которая получает в качестве параметра другую функцию, и когда первая функция завершается, она вместо retrun вызывает функцию-параметр.

Обернем jQuery.ajax таким образом, чтобы получить требуемый результат.

function ajax(url, callback){

jQuery.ajax({url: url, success: callback});

}

Функция получает в качестве параметера callback, и мы не описали обработчик ошибок. В реальном коде он обязан быть, но для простоты изложения, мы о нем забудем.

Что же будет, если использовать библиотеку async? Получится что-то типа такого:

function combine(scripts, callback){

async.map(scripts, ajax, function(contents){

callback(contents.join(""));

});

}

Мы имеем готовую функцию map, работающую в асинхронном мире. Кстати, текущая реализация обеспечит правильный порядок склейки скриптов, в отличие от нашего лобового примера.

Сравните с тем, что было:

function combine(scripts, callback){

var data [];

for(var i =0; l = scripts.length; i< l; ++i){

(function (i){

jQuery.ajax({

url: scripts[i],

success : function(response){

data[i] = response;

if(data.length === scripts.length){

callback(data.join(""));

}

}

});

}(i));

}

}

Поскольку map для меня уже является естественным способом написания программ, я бы никогда не написал кода приведенного выше. Я бы думал, как приспособить map к асинхронному окружению. И если бы не было библиотеки async, то написал бы асинхронный map сам.

Функциональный подход позволяет гораздо проще смотреть на вещи. И реализовывать более красивые решения.

# Частичное применение функций

Еще одна идея, которая пришла из функционального программирования, и может быть очень полезной, если уметь ее правильно готовить.

В качестве примера мы будем создавать DOM элементы.

(Прим. переводчика: cull.dom — библиотека, автора, которую он создавал для одного из проектов. Но функции в ней очевидны и просты.)

var ul = cull.dom.el("ul"); *//document.createElement("ul")*

ul.nodeType === 1 *// true*

Также можно задавать атрибуты свойств.

var ul = cull.dom.el("ul", {className: "bands"});

И указывать дочерние элементы

var li = cull.dom.el("li", "Tom Waits");

var ul = cull.dom.el("ul", {className: "bands"}, li);

Если их использовать друг внутри друга, можно получить некое подобие DSL для HTML.

va ul = cull.dom.el("ul", className:"bands"},

cull.dom.el("li", "Tom Waits"));

А теперь все-таки приступим к обсуждению частичного применения функций. Помните один из первых примеров?

function makeAdder(base){

return function(num){

return base + num;

}

}

Он возвращает функцию, которая будет суммировать два числа. Естественно, при необходимости мы можем использовать именованные функции.

function makeAdder(base){

return function(num){

return add(base, num);

}

}

И теперь мы видим, что функция makeAdder берет функцию add и фиксирует один из ее аргументов. Вы получаете функцию сложения, в которой один из аргументов — константа

var add2 = cull.partial(add, 2);

add2(5); *//7*

Теперь мы получили достаточно интересную возможность — сделать наш DSL по созданию DOM элементов еще красивее.

var ul = cull.partial(cull.dom.el, "ul");

var li = cull.partial(cull.dom.el, "li");

И можем строить HTML списки приблизительно так

var list = ul({className: "bands"},

[li("Diamanda Galas"),

li("Руки вверх"),

li("John Zorn")]);

Если вы как и я, не любите программировать на уровне строковых переменных — это отличный способ упростить себе жизнь. У вас теперь будет работать автодополнение кода, и другие приятные вещи. А еще ваш код очень похож на обычный HTML.

И поскольку наш подход достаточно красив, мы можем создать функции для всех элементов документа заранее:

["a", "br", "code", "div", ...].forEach(function(tagName){

cull.dom.el[tagName] = cull.partial(cull.dom.el, tagName);

});

Таким образом, мы создадим функцию для каждого HTML элемента.

Конечно, пространство имен не всегда удобно использовать полностью, поэтому будем упрощать еще дальше.

var e = cull.dom.el;

var list = ul({className: "bands"},

[e.li("Pan Sonic"),

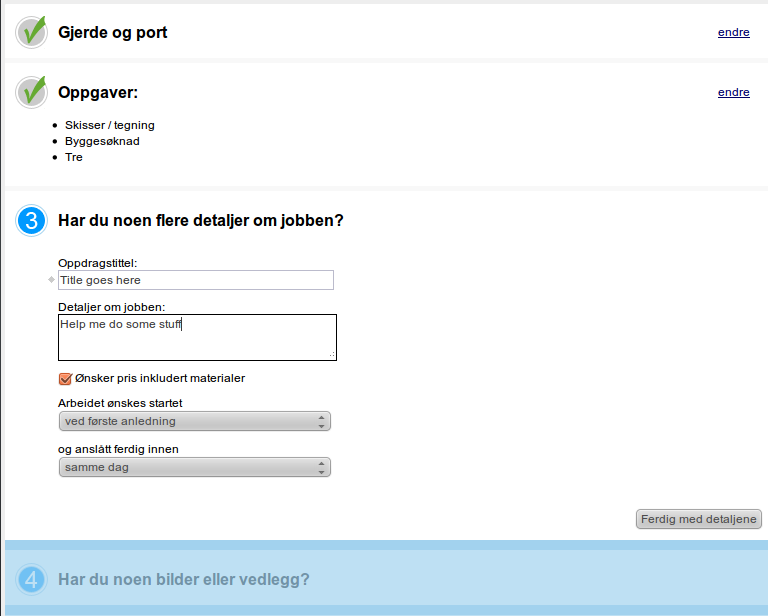
e.li("Веня Дркин"),

e.li("Muslimgauze")]);

Теперь мы не завязаны на глобальные переменные и функции, что хорошо.

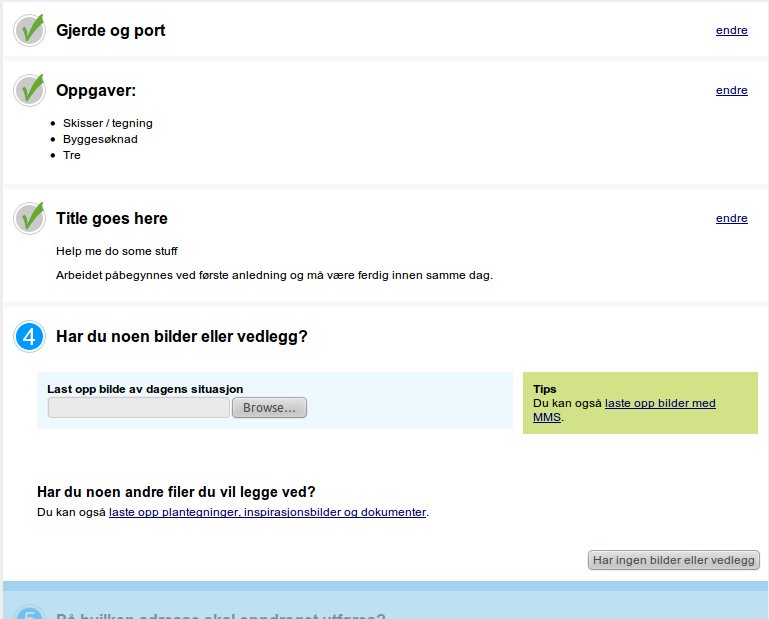
# Композиция функций

Вот еще пример простого приложения — опросника.



Необходимо ответить на каждый блок. Каждый блок содержит несколько вопросов. После ответа на один блок, мы переходим к следующему.

Каждый блок можно представить как панель, которая может быть в режиме вопроса, режиме результата или неактивной.



В каждой панели могут быть разные поля. Строковые, числовые, даты.

Поля могут быть в двух режимах — редактирования или результата.

Посмотрим, как можно подойти к решению этой задачи, используя функциональный подход.

Помните нашу любимую функцию prop?

tweeps.map(prop("name"));

У нее есть брат-близнец *func*.

tweeps.map(func("to.String"));

Она возвращает функцию, которую вы можете применять к объектам.

Теперь посчитаем результат каждого блока в опроснике

buildSummary: function(){

return div(this.components.map(func("buildSummary")));

}

Принцип должен быть очевиден. Мы возвращаем div, в котором будут элементы, созданные функцией buildSummary для каждого блока опросника.

В этом примере каждый компонент сам знает как представить свой результат. Но иногда панель должна отобразить результат специфическим образом.

Поэтому мы можем написать 2 функции: buildSummary и getSummary.

Первая — строит полное представление, включая html теги.

Вторая — возвращает объект, который содержит необходимые результаты.

И как только нам понадобилась хитрая обработка результатов, вся красота начала рушиться.

buildSummary: function(){

var div = document.createElement("div");

for(var i =0; l=this.components.length; i<l; ++i)

{

p = document.CreateElement("p");

p.innerHTML = this.components[i].getSummary().text;

div.appendChild(p);

}

return div;

}

Однако, мы уже достаточно функционально ориентированы, чтобы улучить этот кусок кода. Первое очевидное улучшение — применить foreach.

buildSummary : function(){

var div = document.createElement("div");

this.components.forEach(function(component){

var p = document.createElement("p");

p.innerHTML = component.getSummary().text;

div.appendChild(p);

});

return div;

}

Мы избавились от переменных цикла, но возможно ли использовать map?

buildSummary : function(){

return div(this.components.map(function(component){

var p = document.createElement("p");

p.innerHTML = component.getSummary().text;

return p;

}));

}

Коротко, но далеко до идеала. Основная проблема в этом выражении:

component.getSummary().text;

Проблема в том, тут происходит не одна, а целых три вещи:

1. Получение результата через getSummary()
2. Получение свойства text
3. Оборачивание результата в тег p

А как насчет нескольких функций map?

buildSummary: function() {

return div(this.components.

map(function(component){

return component.getSummary();

}).map(function(summary){

return summary.text;

}).map(function(text){

var p = document.createElement("p");

p.innerHTML = text;

return p;

}));

}

Функциональный стиль налицо, но выглядит страшно. И читать очень неудобно.

Но давайте глянем на код еще разок. Что у нас здесь?

return component.getSummary();

Здесь мы вызываем метод объекта. Но ведь мы создали специальную функцию для этого, func.

buildSummary: function() {

return div(this.components.

map(func("getSummary")).

map(function(summary){

return summary.text;

}).map(function(text){

var p = document.createElement("p");

p.innerHTML = text;

return p;

}));

}

А здесь?

function(summary){

return summary.text;

}

Мы получаем доступ к свойству объекта. И для этого тоже есть удобная функция.

buildSummary: function() {

return div(this.components.

map(func("getSummary")).

map(prop("text")).

map(function(text){

var p = document.createElement("p");

p.innerHTML = text;

return p;

}));

}

Остался последний участок.

function(text){

var p = document.createElement("p");

p.innerHTML = text;

return p;

}

Мы здесь создаем DOM элемент и устанавливаем его внутреннее свойство. У нас есть что-то похожее в нашем DSL, не правда ли?

buildSummary: function() {

return div(this.components.

map(func("getSummary")).

map(prop("text")).

map(p));

}

Теперь почти красиво. Но есть один нюанс. Мы делаем 3 прохода по списку. В каких-то случаях это может быть нормально, но в целом несколько неоптимально. Что же можно сделать?

Пора использовать композицию функций. Мы хотим заставить одну функцию делать то, что делают три.

var summarize = compose(

[p, prop("text"), func("getSummary")]);

## Как же нам реализовать compose?

По частям. Для начала создадим синонимы, чтобы не писать много кода.

var callGetSummary = func("getSummary");

var getText = prop("text");

var summarize = compose([p, getText, callGetSummary]);

Все просто и очевидно. Едем дальше. Разберем, что же происходит, когда мы вызываем функцию summarize.

### Шаг первый

var callGetSummary = func("getSummary");

var getText = prop("text");

var summarize = compose([p, getText, callGetSummary]);

*// summarize(obj);*

*// => callGetSummary(obj)*

Объект предеается в последнюю функцию из списка, а именно getSummary. Она возвращает нам объект типа summary. А этот объект передается в следующую функцию, getText

### Шаг второй

var callGetSummary = func("getSummary");

var getText = prop("text");

var summarize = compose([p, getText, callGetSummary]);

*// summarize(obj);*

*// => getText(callGetSummary(obj))*

В результате второго шага мы получим строку, которая содержится в свойстве text. А после этого строка попадет в функцию, которая создаст нам DOM объект p.

### Шаг третий

var callGetSummary = func("getSummary");

var getText = prop("text");

var summarize = compose([p, getText, callGetSummary]);

*// summarize(obj);*

*// => p(getText(callGetSummary(obj)))*

Это пример простой композиции, когда параметр передается из функцию в функцию последовательно. Можно создать композицию, когда параметр будет передаваться в каждую функцию, и на выходе будет список результатов. Или еще как-то.

Итак, вернемся к нашему многострадальному примеру.

builSummary: function() {

var summarize = compose(

[p, prop("text"), func("getSummary")]);

return div(this.components.map(summarize));

}

Сначала мы создали функцию вычисления результатов. А потом применили map.

При этом заметьте, что функция summarize абсолютно не знает, с каким объектом она работает. Это три различных абстракции, которые соединяются исключительно благодаря функции compose. Поэтому мы можем вынести summarize в отдельную сущность.

var summarize = compose(

[p, prop("text"), func("getSummary")]);

*// ...*

builSummary: function() {

return div(this.components.map(summarize));

}

Выглядит здорово и красиво, но что насчет производительности?

# Вопросы производительности

for — 5M операций в секунду

forEach — 1,5M операций в секунду

reduce — 1.5M операций в секунду

Работа с DOM — 50K операций в секунду

Так что беспокоиться стоит не о функциональном подходе, а о тормозах работы с DOM. Само собой, тут все зависит от вашей задачи, поэтому если сомневаетесь — делайте замеры. Особенно на мобильных устройствах.

# Заключение

Используем чистые функции

Используем функции высшего порядка (map, reduce).

Используем небольшие абстракции.

Много мелких абстракций могут легко собираться в одну большую мощную вещь.