Объекты в javascript

Введение

На предыдущих занятиях мы программировали на базе простых данных и структур. Такой тип программирования называется процедурным. Но в JavaScript мы уже использовали совершенно чуждую для этого подхода сущность – объект. При этом, мы пока даже не догадывались, что используем объекты. Но на этом уроке мы научимся осознанно создавать и применять объекты. Вполне вероятно, что, привыкнув к ним, Вы уже не захотите снова писать процедурный код!

Объекты в JavaScript

В JS объекты занимают ключевую позицию в построении сложного кода. Это может быть управление моделью HTML-документа, организация хранения данных, упаковка библиотек JavaScript. Тема объектов пронизывает язык практически насквозь.

Главный секрет объектов JavaScript состоит в том, что они представляют собой коллекцию свойств. В качестве примера мы рассмотрим бы автомобиль. Он обладает следующими свойствами:

- Марка;
- Модель;
- Цвет;
- Количество пассажиров;
- Мощность двигателя;
- Год изготовления.

Разумеется, полный список свойств реального автомобиля не ограничивается вышеперечисленными пунктами. Но в программе зачастую все свойства не нужны, а требуются только некоторые из них. Поэтому давайте переведем то, что мы написали выше на JavaScript.

Для того, чтобы создать объект, нужно присвоить наши свойства некой переменной, чтобы впоследствии иметь возможность манипулировать объектом. Для этого после стандартного объявления переменной и символа «=» необходимо открыть фигурные скобки. Внутри них перечисляются все необходимые нам свойства.

```
const car = {
```

```
    make: "Audi",
    model: "A5",
    year: 2023,
    color: "red",
    passengers: 2,
    power: 249
};
```

Только что мы создали самый настоящий JS-объект с набором свойств. Мы тут же присвоили его переменной, делая возможным обращение к объекту в дальнейшем. С этого момента у нас есть возможность обращаться к нашему объекту, читать значения его свойств, редактировать их, а также добавлять новые свойства или удалить их.

Итак, при создании объекта:

1. Объявление объектов происходит за счет ключевого слова **const**
2. Заключайте определение объекта в фигурные скобки.
3. Имя свойства отделяется от значения двоеточием.
4. Имя свойства может быть произвольной строкой.
5. Объект не может содержать два свойства с одинаковыми именами.
6. Пары «имя/значение» свойств разделяются запятыми.

Преимущества использования объектов

Итак, мы начали работать с объектами, уходя тем самым от процедурного подхода к программированию. Теперь решаемая задача будет рассматриваться не как набор переменных, условий, циклов и функций. В объектно-ориентированном программировании (ООП) решение задачи происходит в контексте объектов. Они обладают неким состоянием (значения переменных у конкретного объекта) и поведением (функции, которые доступны отдельно взятому объекту).

Какие преимущества приносит такой подход? Мы начинаем мыслить не в искусственно выдуманных сущностях переменных и функций, а переходим на реальный и более высокий уровень представления сущностей. Ведь в повседневной жизни мы как раз и реализуем концепцию объектов.

Допустим, при приготовлении кофе Вы не бежите выращивать кофейное дерево, собирать зерна и так далее. Вы берете объект Кофемашина и выполняете у него метод Сварить кофе (Арабика). Более подробно концепция ООП разбирается на курсе JavaScript Level 2. Пока нам достаточно нашего уровня знаний для применения объектов в JS.

Объекты позволяют скрывают сложную структуру данных, давая разработчикам возможность работать на высоком уровне кода и не тратить время на технические нюансы. В примере с кофе-машиной Вам не нужно знать, как происходит нагрев воды или подача её под давлением на перемолотый кофе.

Свойства объектов

Таким образом, у нас есть объект с упакованными внутри него свойствами. Для того, чтобы начать работать со свойством объекта, нужно указать имя объекта, поставить точку, после чего указать имя свойства. Такой синтаксис (с точечной записью), выглядит так

```
const car = {
   make: "Audi",
   model: "A5",
   year: 2023,
   color: "red",
   passengers: 2,
   power: 249
};

console.log(car.model);
car.power = 350;
```

В любой момент после инициации объекта его можно дополнить новыми свойствами. Для этого нужно указать новое свойство и присвоить ему значение

```
car.odometer = 100;
```

Для удаления свойств необходимо применить ключевое слово delete. Оно уничтожает не только значение свойства, но и его само. При попытке обращения к удалённому свойству результат будет равен undefined.

```
delete car.odometer;
```

Сам delete возвращает true при успешном удалении свойства. false вернётся только в случае, когда свойство не было удалено (например, если свойство входит в защищенный объект, принадлежащий браузеру).

Можно создать объект без свойств или же объект с сотнями свойств. Ограничений тут нет.

Ключевое отличие примитивных типов от объектов

Ключевое различие между примитивами и объектами — способ их хранения в памяти компьютера. Все примитивные значения сохраняются в переменной напрямую. Присваивая одну примитивную переменную другой, мы копируем значение этой переменной в новую переменную, и они остаются независимыми.

Объекты могут быть очень большими (например, большой массив или большой объект), и хранить их значения в каждой переменной не рационально, поэтому используется принцип хранения объектов.

Хранение объектов

В отличие от простых переменных (строк, чисел), которые мы непосредственно помещаем в переменную, объекты хранятся несколько иначе. Когда мы присваиваем переменной объект, то, по сути, мы помещаем в неё ссылку на объект. Т.е. переменная будет содержать не сам объект, а указатель на его область в памяти. И в JS мы не знаем, как именно выглядит этот указатель, но, так или иначе, он указывает на наш объект.

В случае процедурного подхода аргументы передаются функциям по значению. Т.е. мы создаём копию текущей переменной, что-то с этой копией делаем, но исходная переменная при этом не меняется. Те же правила работают и для объектов, но ведут они себя несколько иначе. Поскольку в переменной хранится ссылка на объект, а не сам объект, при рассмотрении передачи по значению в параметре передается копия ссылки, которая, по сути, указывает на исходный объект. Таким образом, в отличие от примитивов, при изменении свойства объекта в функции изменяется свойство исходного объекта. Следовательно, все изменения, вносимые в объект внутри функции, продолжат действовать и после завершения функции.

```
const car = {
   make: "Audi",
   model: "A5",
   year: 2023,
   color: "red",
   passengers: 2,
   power: 249
};

function haveRoadTrip(myCar, distance) {
   myCar.odometer += distance;
}
```

```
haveRoadTrip(car, 150);
console.log(car.odometer);
```

Проблема нежелательного изменения данных в объекте встречается очень часто, так как объекты всегда хранятся в переменных и передаются ссылками. Поэтому, прежде чем передавать объект в функцию или присвоить его другой переменной, нужно подумать, есть ли опасность изменить исходный объект. Если она есть, стоит сделать его полное клонирование. Об этом вы узнаете на следующих уроках.

Всё вышесказанное накладывает отпечаток на операции сравнения переменных примитивного и объектного типов.

Так, при попытке проверить на равенство две переменные, которые ссылаются на один и тот же объект, всегда будет возвращаться правда. А при попытке проверить на равенство две переменные, ссылающиеся на разные объекты, хоть и абсолютно одинаковые по набору данных, будет возвращаться ложь, так как ссылки на эти объекты не равны.

Методы объектов

Помимо свойств, которые описывают состояние объекта, в JS у объектов есть методы, определяющие поведение. Объекты активны и могут выполнять различные операции. Автомобиль не стоит на месте – он передвигается, включает фары и т.д. Таким образом, объект car также должен уметь совершать эти действия.

Метод можно добавить непосредственно в объект. Это делается следующим образом.

```
const car = {
   make: "Audi",
   model: "A5",
   year: 2023,
   color: "red",
   passengers: 2,
   power: 249,
   odometer: 0,
   startEngine: function() {
       console.log("Engine started");
   }
};
```

Как видите, объявление метода представляет собой простое присвоение функции свойству. При этом мы не указываем имя функции, а ставим ключевое слово function, после которого уже описываем само поведение метода. Имя метода совпадает с именем свойства.

При вызове метода startEngine используется точечная запись, только вместо свойства мы пишем имя функции и круглые скобки, внутри которых при необходимости можем перечислять аргументы.

```
car.startEngine();
```

Пока мы оперируем простым выводом строк в консоль. Но давайте усложним наш объект и добавим к нему функцию haveRoadTrip. Она изменится – теперь нам не надо передавать в неё объект автомобиля. Если рассудить логически, то машина никуда не поедет с заглушенным двигателем. Поэтому нам понадобятся:

- Булевое свойство для хранения состояния двигателя (запущен или нет).
- Условная проверка в методе haveRoadTrip, которая удостоверяется, что двигатель запущен, прежде чем вы сможете вести машину.

```
const car = {
   make: "Audi",
   model: "A5",
   year: 2023,
   color: "red",
   passengers: 2,
   power: 249,
   odometer: 0,
   engineIsStarted: false,
   startEngine: function() {
       this.engineIsStarted = true;
   },
   stopEngine: function() {
       this.engineIsStarted = false;
   },
   haveRoadTrip: function(distance) {
       if (this.engineIsStarted) {
           this.odometer += distance;
       } else {
           alert("Сначала запустите двигатель!");
       }
```

```
    }
};
```

Чтобы запустить двигатель, мы могли бы убрать метод **startEngine**, а вместо него использовать простое изменение значения свойства. Но если подумать, то мы поймём, что запуск двигателя может быть сложнее. Что, если мы нарастим туда проверку заряда аккумулятора или снятие с сигнализации? Каждый раз вызвать такой код будет неудобно. Лучше создать единый метод, который знает весь технический процесс запуска двигателя.

Вы, наверняка, заметили, что в коде появилось странное слово **this**. Что же оно обозначает?

В наших методах переменные **engineIsStarted** и **odometer** не являются локальными или глобальными. Они являются свойством объекта **car**. Именно для них в JavaScript существует ключевое слово **this**: оно обозначает текущий объект, с которым мы работаем.

Можно представлять this в качестве переменной, которая указывает на объект, метод которого был только что вызван. Таким образом, если вызвать метод **startEngine** объекта car, то **this** будет указывать на объект car. При вызове любого метода вы можете быть уверены, что **this** в теле метода будет указывать на объект, метод которого был вызван.

Таким образом мы видим, что:

- Поведение влияет на состояние. Т.е. внутри методов объекта производится изменение значений его свойств.
- Состояние влияет на поведение. Не совсем очевидная истина, но посмотрите на проверку на заведенный двигатель. В зависимости от значения true/false метод будет вести себя по-разному.

Перебор значений

```
for (const key in object) {
   console.log(key + ": " + object[key]);
}
```

Цикл **for in** перебирает свойства объекта, которые последовательно присваиваются переменной key.

Также Вы уже могли заметить, что мы применили альтернативный способ обращения к свойствам объекта: запись с квадратными скобками. Она эквивалентна известной нам точечной записи, делает то же самое, но обладает чуть большей гибкостью. Пример

```
const car = {
   make: "Audi",
   model: "A5",
   year: 2023,
   color: "red",
   passengers: 2,
   power: 249,
   odometer: 0,
};

console.log('Все ключи объекта car');
for (const key in car) {
   console.log(key);
}
console.log('Все значения объекта car');
for (const key in car) {
   console.log(car[key]);
}
```

Преобразование объекта в массив

Очень часто бывает что мы получаем для работы объект, но нам необходимо работать с его данными как с массивом, поэтому начинающие программисты на JavaScript часто пытаются применить метод map массива к объекту:

```
const object = {
   1: 'Ivanov',
   2: 'Petrov',
};

const students = object.map((student) => `student: ${student}`); //
VM223:6 Uncaught TypeError: object.map is not a function at < anonymous >:
6: 25
```

При этом код возвращает ошибку, т.к. у объектов нет метода map(), поэтому мы должны преобразовать объект в массив, и уже потом с ним работать. Давайте рассмотрим методы, позволяющие это сделать:

Object.keys

К примеру наша группа студентов была заведена в программе ранее как объект, где ключами являются фамилии студентов, и мы хотим получить массив фамилий этих студентов. Тут нам поможет метод **Object.keys(<исходный объект>)**.

Данный метод позволяет получить из объекта все его ключи первого уровня и положить их в массив. Например у нас есть объект студентов, в котором ключи - это фамилии студентов, а значения более подробная информация, а мы хотим получить только список фамилий студентов:

```
const group1 = {
   "Ivanov": {
       practicePlace: "ldu-1",
       practiceTime: 56
   },
   "Petrov": {
       practicePlace: "kamaz",
       practiceTime: 120
   },
   "Sidorov": {
       practicePlace: "ldu-1",
       practiceTime: 148
   },
   "Belkin": {
       practicePlace: "GosDZU",
       practiceTime: 20
   },
   "Avdeev": {
       practicePlace: "LPK",
       practiceTime: 160
   }
}

const group1Students = Object.keys(group1);
console.log(group1Students); // ["Ivanov", "Petrov", "Sidorov", "Belkin",
"Avdeev"]
```

Object.values

Но чаще мы все таки хотим иметь дело со значениями в объектах, например мы хотим вывести данные студентов в виде таблицы, для получения их в качестве массива нам поможет метод **Object.values(<исходный объект>)**.

Данный метод позволяет получить значения из объекта в виде массива.

```
const car = {
   make: "Audi",
   model: "A5",
   year: 2023,
   color: "red",
   passengers: 2,
   power: 249,
   odometer: 0,
};

console.log(Object.values(car)); //  ['Audi', 'A5', 2023, 'red', 2, 249,
0]
```

Object.entries

А порой нам нужны и ключи объекта и их значения, например мы хотим вывести таблицу с фамилиями и данными о студентах, вот тут нам поможет метод **Object.entries(<исходный объект>)**.

Данный метод позволяет получить из объекта как ключи, так и значения в виде одного массива массивов. Т.е. В результате работы этого метода на выходе мы получаем массив, содержащий массивы из двух элементов: ключ и его значение. Это очень удобный метод, давайте улучшим нашу таблицу отработанной практики, добавив туда колонку с фамилией студента:

```
const car = {
   make: "Audi",
   model: "A5",
   year: 2023,
   color: "red",
```

```
    passengers: 2,
    power: 249,
    odometer: 0,
};

console.log(Object.entries(car));
// 0: (2) ['make', 'Audi']
// 1: (2) ['model', 'A5']
// 2: (2) ['year', 2023]
// 3: (2) ['color', 'red']
// 4: (2) ['passengers', 2]
// 5: (2) ['power', 249]
// 6: (2) ['odometer', 0]
```

Мы могли бы итерировать объект с помощью for...in, без необходимости конвертировать их в массив, но есть одно большое отличие этих методов - цикл for...in итерирует все свойства объекта, даже из прототипов, а не только собственные, а в массив попадают только собственные свойства объекта. Смотрите подробное описание этих методов на MDN: Object.keys, Object.values, Object.entries.

Глобальный объект window

В любой среде исполнения JavaScript всегда есть глобальный объект:

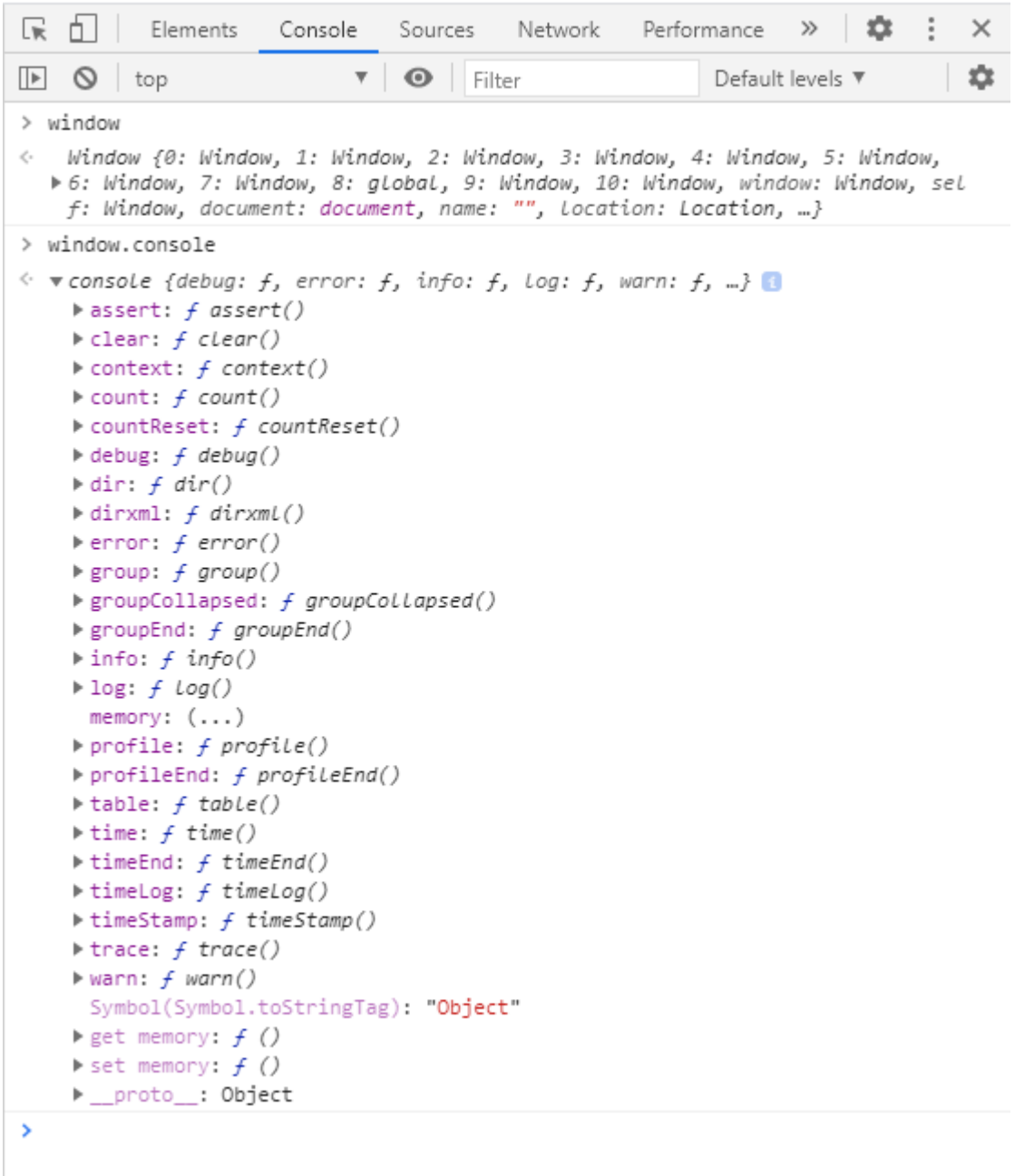
- объект window в браузере;
- global в Node.js;
- WorkerGlobalScope в воркере.

Это не часть языка как таковая, но мы всегда можем обращаться к его свойствам и методам. Глобальный объект создаётся движком JavaScript, чтобы в нём хранить все глобальные функции, необходимые для взаимодействия со средой, в которой этот движок запущен. Например, в браузере объект `window` содержит `alert`, `console.log`, `promt` и многие другие полезные функции для работы с DOM.

Чтобы увидеть его в действии, откройте консоль браузера и введите в ней `window`. Консоль будет вам подсказывать, что есть в этом объекте. Если вы поставите точку, консоль начнёт подсказывать, какие свойства и методы есть в этом объекте. Начнём вводить `console` и после ввода нажмём на клавишу «Ввод»:

```
window.console
```

После этого мы можем видеть, что объект `window` содержит объект `console`, который имеет метод `log()` и другие методы, например `info`, `count`, `error`, `warn`, `debug`, о которых вы можете прочитать в статье на MDN.



Все методы и свойства объекта `window` для удобства также доступны напрямую из консоли или из кода. Нам не нужно каждый раз писать `window.console.log()`, будет достаточно просто `console.log()`.

Работа с объектами и функции высшего порядка

Функции высшего порядка

Студент это не просто фамилия в массиве, часто вместе с фамилией нам необходимо сохранять разную другую информации о каждом студенте, например его имя, возраст, успеваемость и т.п. Тогда в массиве удобно хранить объекты. А для работы с таким массивом уже недостаточно простых методов добавления, извлечения и поиска элементов. Нам нужны методы позволяющие проходить по массиву и собирать из него данные, либо их преобразовывать. Такие методы являются функциями высшего порядка - это функции, которые в качестве аргумента могут принимать другую функцию, которую они будут вызывать в результате своих вычислений. Функцию-аргумент часто называют функцией обратного вызова (callBack). Таке функциями высшего порядка называют функции, которые возвращают другую функцию (привет замыкания). Такие функции дают удобный способ обрабатывать элементы массивов в циклах, передавая в метод свою функцию обратного вызова. Давайте посмотрим на такие функции, которые упрощают нам работу с массивами в JavaScript.

К примеру у нас есть массив студентов, с информацией о количестве отработанных часов практики, и нам нужно построить отчет в виде таблицы, в которой вывести фамилии студентов и информацию прошли ли они практику или нет. Метод для формирования нового массива на основе исходного называется **map**.

map

Метод **map** является одним из самых используемых при работе с массивом. Он позволяет проитерировать весь массив, и создать на основе него новый массив. **Map** (от английского карта) - данные метод позволяет сделать как бы карту соответствия исходного массива и нового. Метод принимает аргумент-функцию, в которую при работе передаются аргументы: текущий элемент массива, его индекс, и полный массив. Функция аргумент должна сделать необходимые вычисления и вернуть новый элемент, из которых будет построен новый массив. Давайте посмотрим на примере.

```
const studentsPracticeTime = [
   {
       firstName: "Ivanov",
       practiceTime: 56
```

```
    },
    {
        firstName: "Petrov",
        practiceTime: 120
    },
    {
        firstName: "Sidorov",
        practiceTime: 148
    },
    {
        firstName: "Belkin",
        practiceTime: 20
    },
    {
        firstName: "Avdeev",
        practiceTime: 160
    }
];

// Мы хотим вывести в таблицу студентов и информацию для каждого, прошел
ли он практику уже. Практика будет считаться пройденной, если студент
отработал 120 часов или больше.

const dataForTable = studentsPracticeTime.map((student) => {
    if (student.practiceTime < 120) {
        return { // Мы возвращаем новый объект, более удобный для вывода.
            Student: student.firstName,
            Practice: "Not passed"
        };
    } else {
        return {
            Student: student.firstName,
            Practice: "Passed"
        };
    }
});

console.table(dataForTable); // В консоль можно выводить разными
```

```
способами, если использовать метод table, и передать туда массив или
объект, этот метод выводит данные в виде таблицы.
```

(index)	Student	Practice
0	"Ivanov"	"Not passed"
1	"Petrov"	"Passed"
2	"Sidorov"	"Passed"
3	"Belkin"	"Not passed"
4	"Avdeev"	"Passed"

В итоге выполнения данного алгоритма, мы получили новый массив с измененными данными.

Полное описание метода смотрите в MDN.

filter

Порой нам нужно отфильтровать список студентов, чтобы получить только тех, кто уже прошел практику, для этого можно использовать метод **filter**.

Метод **filter** используется для фильтрации элементов массива по какому-нибудь правилу, которое задаете вы сами. Это очень полезный метод, который позволяет исключить из исходного массива лишние элементы и получить новый массив. Метод **filter** также, как и метод **map** получает в качестве аргумента функцию обратного вызова, с теми же аргументами, только эта функция должна вернуть **false** если элемент должен быть исключен, или **true**, чтобы текущий элемент попал в новый, возвращаемый функцией **filter** массив. Давайте возьмём предыдущий пример, и отфильтруем массив студентов проходящих практику, оставив только тех, кто уже прошел практику.

```
const studentsPracticeTime = [{
        firstName: "Ivanov",
        practiceTime: 56
    },
    {
        firstName: "Petrov",
        practiceTime: 120
    },
    {
```

```
        firstName: "Sidorov",
        practiceTime: 148
    },
    {
        firstName: "Belkin",
        practiceTime: 20
    },
    {
        firstName: "Avdeev",
        practiceTime: 160
    }
];

// Мы хотим отфильтровать массив студентов, оставив в новом массиве только тех, кто уже прошел практику. Практика будет считаться пройденной, если студент отработал 120 часов или больше.

const studentsPassedPractice = studentsPracticeTime.filter((student) => {
    if (student.practiceTime < 120) return false
    return true
});

console.log(studentsPassedPractice); // Получили новый массив, в котором только те студенты, кто уже прошел практику.
// [ {
//     "firstName": "Petrov",
//     "practiceTime": 120
// }, {
//     "firstName": "Sidorov",
//     "practiceTime": 148
// }, {
//     "firstName": "Avdeev",
//     "practiceTime": 160
// }
// ]
```

Полное описание метода смотрите в MDN.

reduce

Порой деканату нужно собрать статистику для отчетности, например сколько всего часов практики наработали студенты группы. Для этого удобно использовать метод **reduce**.

Метод **reduce** не так просто понять с первого раза. Этот метод еще называют свёрткой, т.к. он проходится по всему массиву и позволяет собрать и обработать его значение в новую форму, например в одно значение (допустим сумму значений всех элементов). Чтобы лучше всего понять этот метод, давайте сначала посмотрим на алгоритм без использования **reduce**, а потом оптимизируем наш алгоритм - применим метод **reduce**.

Без **reduce**:

```
const studentsPracticeTime = [{
        firstName: "Ivanov",
        practiceTime: 56
    },
    {
        firstName: "Petrov",
        practiceTime: 120
    },
    {
        firstName: "Sidorov",
        practiceTime: 148
    },
    {
        firstName: "Belkin",
        practiceTime: 20
    },
    {
        firstName: "Avdeev",
        practiceTime: 160
    }
];

// Посчитаем сколько всего часов практики отработали студенты.
let totalTime = 0; // Объявим переменную для хранения суммы всех часов.

for (let index = 0; index < studentsPracticeTime.length; index++) {
    totalTime = totalTime + studentsPracticeTime[index].practiceTime;
}
```

```
console.log(totalTime); // 504
```

А теперь давайте применим метод **reduce** для этой же цели. Метод **reduce** принимает два аргумента, первый это функция обратного вызова с 4 аргументами (обычно используются только два и реже три из них): аккумулятор, текущий элемент массива, индекс этого элемента и весь массив, функция обратного вызова должна вернуть обновленное значение аккумулятора, которое будет передано в следующую итерацию; второй - первоначальное значение аккумулятора.

С использованием **reduce**:

```
const studentsPracticeTime = [{
        firstName: "Ivanov",
        practiceTime: 56
    },
    {
        firstName: "Petrov",
        practiceTime: 120
    },
    {
        firstName: "Sidorov",
        practiceTime: 148
    },
    {
        firstName: "Belkin",
        practiceTime: 20
    },
    {
        firstName: "Avdeev",
        practiceTime: 160
    }
];

// Посчитаем сколько всего часов практики отработали студенты.
const totalTime = studentsPracticeTime.reduce((acc, student) => { //
Первое значение - это функция обратного вызова, которая будет получать
значение аккумулятора (acc) при каждой итерации; и текущий элемент массива
(student).
    return acc + student.practiceTime;
```

```
}, 0); // Второй аргумент - это первоначальное значение аккумулятора - 0.

console.log(totalTime); // 504
const studentsPracticeTime = [
{
   firstName: "Ivanov",
   practiceTime: 56
}, {
   firstName: "Petrov",
   practiceTime: 120
}, {
   firstName: "Sidorov",
   practiceTime: 148
}, {
   firstName: "Belkin",
   practiceTime: 20
}, {
   firstName: "Avdeev",
   practiceTime: 160
}
];

// Посчитаем сколько всего часов практики отработали студенты.
const totalTime = studentsPracticeTime.reduce((acc, student) => { //
Первое значение - это функция обратного вызова, которая будет получать
значение аккумулятора (acc) при каждой итерации; и текущий элемент массива
(student).
   return acc + student.practiceTime;
}, 0); // Второй аргумент - это первоначальное значение аккумулятора - 0.

console.log(totalTime); // 504
```

some

Когда деканат спрашивает, есть ли среди вашей группы студенты, когда уже прошли практику, мы можем это выяснить использовав метод **some**.

Метод **some** используется, когда нам нужно проверить, что в массиве есть хоть один подходящий нам элемент. Например мы хотим проверить, есть ли среди всех студентов хоть кто-то, кто прошел практику. Метод **some** принимает функцию обратного вызова, которая вызывается для каждого элемента массива с аргументами: текущий элемент массива, его индекс, и весь исходный массив; данная функция должна проверить необходимое условие и вернуть **true**, если элемент подходит и **false** в противном случае. Как только будет найден первый подходящий элемент выполнение метода **some** прекращается и он возвращает **true**, в противном случае метод пройдет по всем элементам массива и вернет **false**. Если исходный массив пустой, то метод сразу же вернет **false**.

```
const studentsPracticeTime = [
   {
     firstName: "Ivanov",
     practiceTime: 56
   },
   {
     firstName: "Petrov",
     practiceTime: 120
   },
   {
     firstName: "Sidorov",
     practiceTime: 148
   },
   {
     firstName: "Belkin",
     practiceTime: 20
   },
   {
     firstName: "Avdeev",
     practiceTime: 160
   }
 ];
  // Проверим, есть ли хоть один студент, который прошел практику.
 const isSomebodyPassedPractice = studentsPracticeTime.some((student) => {
   console.log(student.firstName); // Добавим вывод студента, чтобы
посмотреть сколько элементов массива будет проитерированно.
   return student.practiceTime >= 120;
 });
  // "Ivanov"
```

```
 // "Petrov" - итерации остановились на втором элементе массива, т.к. он удовлетворяет нашему условию и метод some дальше итерации не выполняет.
 console.log(isSomebodyPassedPractice); // true - среди студентов есть те, кто прошел практику.
```

Метод **some** очень удобен для проверок, когда размер массива очень большой, т.к. Позволяет остановиться сразу же как только будет найден подходящий элемент, не тратя зря ресурсы.

Полное описание метода смотрите в MDN.

Посмотрите также обратный метод every, который работает как some только наоборот, для того чтобы получить **true** все элементы массива должны удовлетворять условию, метод мгновенно останавливается и возвращает **false**, как только находится первый элемент, который не удовлетворяет условию. Полное описание смотрите на MDN.

find

Когда в нашем массиве `studentsPracticeTime` лежат объекты, а мы хотим найти элемент массива по фамилии студента, но не знаем сколько часов практики он уже отработал, мы не можем применить метод **`indexOf`**, т.к. в нашем массиве объекты, а как мы знаем объекты сравниваются по ссылкам, соответственно для поиска в массиве объектов мы должны иметь ссылку на конкретный объект, вряд ли у нас такая будет, если мы ищем элемент. Нам поможет метод **find** - он позволяет найти элемент в массиве по заданному условию. В отличия от метода **indexOf**, метод **find** позволяет искать не точное совпадение элемента.

Чтобы найти нужный нам элемент, нужно в метод find передать функцию обратного вызова, которая как и для многих других принимает 3 аргумента: текущий элемент, индекс, и весь исходный массив и должна вернуть **true**, если элемент нам подходит, иначе вернуть **false**. Метод **find** вернет значение первого найденного элемента в массиве или **undefined** в противном случае.

```
const studentsPracticeTime = [
   {
     firstName: "Ivanov",
     practiceTime: 56
   },
   {
     firstName: "Petrov",
     practiceTime: 120
   },
   {
     firstName: "Sidorov",
     practiceTime: 148
```

```
    },
    {
      firstName: "Belkin",
      practiceTime: 20
    },
    {
      firstName: "Avdeev",
      practiceTime: 160
    }
 ];
  // Мы хотим найти студента Belkin и посмотреть сколько времени он
отработал на практике.
 const studentBelkin = studentsPracticeTime.find((student) => {
   return student.firstName === "Belkin";
 });
 console.log(studentBelkin.practiceTime); // 20
```

Полное описание метода смотрите в MDN.

Другие методы для массива

Есть еще несколько интересных и полезных методов для работы с массивами, все они работают схожим образом. Предлагаем ознакомиться с ними самостоятельно в документации MDN.

Деструктуризация

Порой нам необходимо часто обращаться к определенным элементам массива или ключам объекта в нашем алгоритме, и чтобы каждый раз не писать имя массива/объекта, с путем до нужного нам элемента, мы можем сохранить данные такого элемента в отдельную переменную. Новый стандарт ES2015 позволил легко получать данные из массивов и объектов, сохраняя их при этом в новые переменные деструктуризируя массив или объект.. Давайте посмотрим как мы могли скопировать данные из массива или объекта в отдельные переменные раньше:

```
// Сбор данных из объекта.
const student = {
   firstName: "Ivan",
   lastName: "Petrov",
   age: 21,
```

```
};

const firstName = student.firstName; // мы объявляем отдельно переменную,
под каждый нужный нам параметр.
const lastName = student.lastName;
const age = student.age;

// Сбор данных из массива.
const students = ["Ivanov", "Petrov", "Belkin"];
const student1 = students[0];
const student2 = students[1];
const student3 = students[2];
```

Не очень удобно. Деструктуризация дала нам гораздо более простой и удобный синтаксис:

```
// Сбор данных из объекта.
const student = {
   firstName: "Ivan",
   lastName: "Petrov",
   age: 21,
};

const { firstName, lastName, age } = student; // Деструктуризация -  мы
объявляем все переменные в фигурных скобках, название должно совпадать с
нужным нам параметром.

// Сбор данных из массива.
const students = ["Ivanov", "Petrov", "Belkin"];
const [student1, student2, student3] = students; // Деструктуризация - Тут
мы указываем имена переменных в квадратных скобках, и в них по порядку
будут записаны элементы массива.
```

Теперь мы можем объявлять все нужные нам переменные в одном месте и получать данные из объектов или массивов одной строкой.

Деструктуризация также позволяет задавать значения по умолчанию для переменных, на случай если в объекте не окажется такого ключа, который мы запрашиваем, или в массиве в элементе окажется undefined значение. Также при деструктуризации объекта можно переименовать переменные в которые будут сохраняться ключи, если мы например не хотим или не можем использовать переменную с таким же именем как и ключ:

```
// Сбор данных из объекта.
const student = {
   firstName: "Ivan",
   lastName: "Petrov",
   age: 21,
};

// Деструктурируем значение ключа firstName в переменную studentName. И зададим для возраста значение по умолчанию, равное 20.
const { firstName: studentName, lastName, age = 20 } = student;

// Сбор данных из массива.
const students = ["Ivanov", "Petrov", "Belkin"];
const [student1, student2, student3] = students; // Тут мы указываем имена переменных в квадратных скобках, и в них по порядку будут записаны элементы массива.
```

Дополнительные материалы

1. Документация о массивах на MDN - https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array
2. Массив: перебирающие методы - https://learn.javascript.ru/array-iteration
3. Что такое чистые функции в JavaScript - https://habr.com/ru/post/437512/

Используемые источники

1. https://learn.javascript.ru/
2. https://learn.javascript.ru/destructuring
3. https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array
4. https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Working_with_Objects