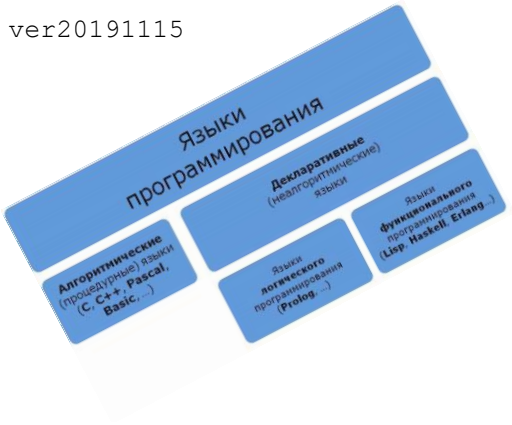
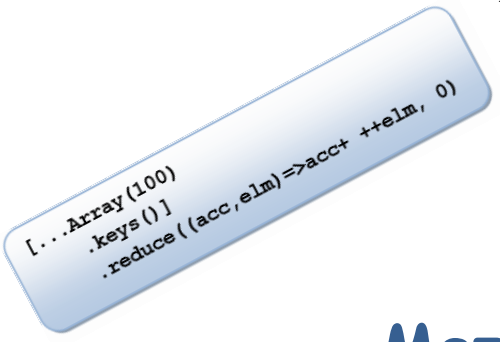




Методы функционального программирования в JavaScript

Содержание

Тема 1. Функциональный стиль обработки массивов

Массив - это упорядоченная коллекция значений. Значения в массиве называются элементами, и каждый элемент характеризуется числовой позицией в массиве, которая называется индексом. Массивы в языке JavaScript являются нетипизированными: элементы массива могут иметь любой тип, причем разные элементы одного и того же массива могут иметь разные типы. Элементы массива могут даже быть объектами или другими массивами, что позволяет создавать сложные структуры данных, такие как массивы объектов и массивы массивов.

Отсчет индексов массивов в языке JavaScript начинается с нуля и для них используются 32-битные целые числа - первый элемент массива имеет индекс 0. Массивы в JavaScript являются динамическими: они могут увеличиваться и уменьшаться в размерах по мере необходимости; нет необходимости объявлять фиксированные размеры массивов при их создании или повторно распределять память при изменении их размеров. Массивы в языке JavaScript являются объектами и у них есть свои методы.

Создание массивов

Проще всего создать массив с помощью литерала, который представляет собой простой список разделенных запятыми элементов массива в квадратных скобках. Значения в литерале массива не обязательно должны быть константами - это могут быть любые выражения, в том числе и литералы объектов:

```
var empty = []; // пустой массив
var numbers = [2, 3, 5, 7, 11]; // массив с элементами
var base = 1024;
var table = [base, base+1, base+2, base+3]; // с переменными
var arrObj = [[1,{x:1, y:2}], [2, {x:3, y:4}]]; // с объектами
```

Синтаксис литералов массивов позволяет вставлять необязательную завершающую запятую, т.е. литерал [,,] соответствует массиву с двумя элементами, а не с тремя.

```
var misc = [ 1.1, true, "a", ]; // длина массива 4
```

Другой способ создания массива состоит в вызове конструктора Array(). Вызвать конструктор можно тремя разными способами:

Вызвать конструктор без аргументов:

```
var arr = new Array();
```

В этом случае будет создан пустой массив, эквивалентный литералу [].

Вызвать конструктор с единственным числовым аргументом, определяющим длину массива:

```
var arr = new Array(10);
```

В этом случае будет создан пустой массив указанной длины. Такая форма вызова конструктора Array() может использоваться для предварительного распределения памяти под массив, если заранее известно количество его элементов. Обратите внимание, что при этом в массиве не сохраняется никаких значений.

Явно указать в вызове конструктора значения первых двух или более элементов массива или один нечисловой элемент:

```
var arr = new Array(5, 4, 3, 2, 1, "тест");
```

В этом случае аргументы конструктора становятся значениями элементов нового массива. Использование литералов массивов практически всегда проще, чем подобное применение конструктора Array().

Чтение и запись элементов массива

Доступ к элементам массива осуществляется с помощью оператора []. Слева от скобок должна присутствовать ссылка на массив. Внутри скобок должно находиться произвольное выражение, возвращающее неотрицательное целое значение. Этот синтаксис пригоден как для чтения, так и для записи значения элемента массива. Следовательно, допустимы все приведенные далее JavaScript-инструкции:

```
     // Создать массив с одним элементом
var arr = ["world"];
     // Прочитать элемент 0
```

```
var value = arr[0];
     // Записать значение в элемент 1
arr[1] = 3.14;
     // Записать значение в элемент 2
i = 2; arr[i] = 3;
     // Записать значение в элемент 3
arr[i + 1] = 'привет';
     // Прочитать элементы 0, записать значение в элемент 3
arr[arr[i]] = arr[0];
```

Так как массивы являются специализированной разновидностью объектов, то квадратные скобки, используемые для доступа к элементам массива, действуют точно так же, как квадратные скобки, используемые для доступа к свойствам объекта. Интерпретатор JavaScript преобразует указанные в скобках числовые индексы в строки - индекс 1 превращается в строку "1" - а затем использует строки как имена свойств.

В преобразовании числовых индексов в строки нет ничего особенного: то же самое можно проделывать с обычными объектами:

```
var object = {}; // Создать объект
object = { // Индексировать целыми числами
    1: "one",
    2: "two"
}
object[3] = "three"; // добавить свойство

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

Особенность массивов состоит в том, что при использовании имен свойств, которые являются неотрицательными целыми числами, массивы автоматически определяют значение свойства length. Например, создадим пустой массив arr, затем присвоим значения его некоторым элементам массива. В результате этих операций значение свойства length массива изменится и приобретёт значение, равное последнему индексу массива + 1:

```
arr = [];
arr[1] = 1;
arr[3] = 3;
console.log(arr.length); // => 4
```

```
console.log(arr[2]); // => undefined
```

Следует четко отличать индексы в массиве от имен свойств объектов. Все индексы являются именами свойств, но только свойства с именами, представленными целыми числами являются индексами. Все массивы являются объектами, и вы можете добавлять к ним свойства с любыми именами. Однако если вы затрагиваете свойства, которые являются индексами массива, массивы реагируют на это, обновляя значение свойства length при необходимости.

Обратите внимание, что в качестве индексов массивов допускается использовать отрицательные и не целые числа. В этом случае числа преобразуются в строки, которые используются как имена свойств.

Примеры:

```
let object = [];
object[0] = 0;
object["one"] = 1;
object["two"] = 2;

console.log( object.length );

Вывод:
1
```

```
arr = [];
arr["22"] = 22;
arr["2"] = 2;
arr[1] = 1;
arr[3] = 3;
console.log(arr[0]);
arr.forEach(element => {
    console.log(element);
});

Вывод:
Undefined
1
2
3
22
```

Это функциональный подход - рассмотрен в главе про функции.

Добавление и удаление элементов массива

Вы уже видели, что существует простой способ добавить элементы в массив, который состоит в том, чтобы присвоить значения элементам под новым индексом. Для добавления одного или более элементов в конец массива можно также использовать метод push():

```
var arr = []; // Создать пустой массив
arr.push('zero'); // Добавить значение в конец
arr.push('one',2); // Добавить еще два значения
```

Добавить элемент в конец массива можно также, присвоив значение элементу arr[arr.length]. Для вставки элемента в начало массива можно использовать метод unshift(), при этом существующие элементы в массиве смещаются в позиции с более высокими индексами.

Удалять элементы массива можно с помощью оператора delete, как обычные свойства объектов:

```
var arr = [1,2,'three'];
delete arr[2];
2 in arr; // false, индекс 2 в массиве не определен
arr.length; // 3 - оператор delete не изменяет свойство length
```

Да, очевидно, что проверить наличие элемента в массиве можно оператором in: **`element in array`**, который возвращает значение логического типа (true/false).

Удаление элемента напоминает присваивание значения undefined этому элементу, так как применение оператора delete к элементу массива не изменяет значение свойства length и не сдвигает вниз элементы с более высокими индексами, чтобы заполнить пустоту, оставшуюся после удаления элемента.

Кроме того имеется возможность удалять элементы в конце массива простым присваиванием нового, уменьшенного значения свойству length. Массивы имеют метод pop() (противоположный методу push()), который уменьшает длину массива на 1 и возвращает значение удаленного элемента. Также имеется метод shift() (противоположный методу unshift()), который удаляет элемент в начале массива. В отличие от оператора delete, метод shift() сдвигает все элементы вниз на позицию ниже их текущих индексов.

Наконец существует многоцелевой метод splice(), позволяющий вставлять, удалять и замещать элементы массивов. Он изменяет значение свойства length и сдвигает элементы массива с более низкими или высокими индексами по мере необходимости. Данный метод

рассмотрим позже при рассмотрении всех методов, присущих массивам.

Многомерные массивы

JavaScript не поддерживает «настоящие» многомерные массивы, но позволяет неплохо имитировать их при помощи конструкции «массив массивов». Для доступа к элементу данных в массиве массивов достаточно дважды использовать оператор [].

Например, предположим, что переменная matrix - это массив массивов чисел. Каждый элемент matrix[x] - это массив чисел. Для доступа к определенному числу в массиве можно использовать выражение matrix[x][y]. Ниже приводится конкретный пример, где двумерный массив используется в качестве таблицы умножения:

```
    // Создать многомерный массив
var table = new Array(10); // В таблице 10 строк
for(var i = 0; i < table.length; i++)
     table[i] = new Array(10); // В каждой строке 10 столбцов
     // Инициализировать массив и вывести на консоль
for(let row = 0; row < table.length; row++) {
     str = '';
     for(let col = 0; col < table[row].length; col++) {
          table[row][col] = (row+1)*(col+1);
          str += table[row][col] + '\t';
     }
     console.log(str + '\n');
}
```

Методы класса Array

Стандарт ECMAScript определяет множество удобных функций для работы с массивами, которые доступны как методы массива-объекта, доступ к которым открывается через точку.

Методы split() и join()

Метод split() разбивает строку на массив строк по указанному в качестве аргумента делителю и возвращает массив строк:

```
str = "Мама мыла раму";
console.log(str.split(' ')); // делим по пробелу
[ 'Мама', 'мыла', 'раму' ]
```

Делителем может быть и пустой символ:

```
bin = "1101";
console.log(bin.split(''));
[ '1', '1', '0', '1' ]
```

Метод join() преобразует все элементы массива в строки, объединяет их и возвращает получившуюся строку. В необязательном аргументе методу можно передать строку, которая будет использоваться для отделения элементов в строке результата. Если строка-разделитель не указана, используется запятая. Например, следующий фрагмент дает в результате строку "1,2,3":

```
var arr = [1, 2, 3];
arr.join(); // '1,2,3'
arr.join("-"); // '1-2-3'
let str = arr.join(""); // '123'
console.log(arr); // => [1, 2, 3]
console.log(str); // => '123'
```

Метод reverse()

Метод reverse() меняет порядок следования элементов в массиве на обратный и возвращает переупорядоченный массив. Перестановка выполняется непосредственно в исходном массиве, т.е. этот метод не создает новый массив с переупорядоченными элементами, а переупорядочивает их в уже существующем массиве. Например, следующий фрагмент, где используются методы reverse() и join(), дает в результате строку "3,2,1":

```
var arr = [1,2,3];
arr.reverse().join(); // "3,2,1"
```

Метод sort()

Метод sort() сортирует элементы в исходном массиве и возвращает отсортированный массив. Если метод sort() вызывается без ар-

гументов, сортировка выполняется в алфавитном порядке (для сравнения элементы временно преобразуются в строки, если это необходимо). Неопределенные элементы переносятся в конец массива.

Для сортировки в каком-либо ином порядке, отличном от алфавитного, методу sort() можно передать функцию сравнения в качестве аргумента. Эта функция устанавливает, какой из двух ее аргументов должен следовать раньше в отсортированном списке. Если первый аргумент должен предшествовать второму, функция сравнения должна возвращать отрицательное число. Если первый аргумент должен следовать за вторым в отсортированном массиве, то функция должна возвращать число больше нуля. А если два значения эквивалентны (т.е. порядок их следования не важен), функция сравнения должна возвращать 0:

```
var arr = [33, 4, 1111, 222];
arr.sort(); // Алфавитный порядок: 1111, 222, 33, 4

// Далее числовой порядок сортировки: 4, 33, 222, 1111
var arr = [3, 30, 100, 99];
function check(a, b) {
    return b-a; // Возвращает значение <0, 0 или >0
}
let checkArr = function ab(a, b) {
    return b-a;
}
// Сортируем в обратном направлении, от большего к меньшему
let newArr1 = arr.sort(function(a,b) {return b-a});
let newArr2 = arr.sort((a,b)=>b-a);
let newArr3 = arr.sort((a,b)=>check(a, b));
let newArr4 = arr.sort((a,b)=>checkArr(a, b));
console.log(newArr1);
console.log(newArr2);
console.log(newArr3);
console.log(newArr4);
Вывод:
[ 100, 99, 30, 3 ]
[ 100, 99, 30, 3 ]
[ 100, 99, 30, 3 ]
[ 100, 99, 30, 3 ]
```

Это функциональный подход - рассмотрен в главе про функции.

В примере показаны несколько способов организации сортировки, основанных на стиле функционального программирования. Подробнее про передачу в функцию в качестве аргумента другой функции можно посмотреть в разделе «Функции». Обратите внимание, насколько удобно использовать в этом фрагменте неименованную функцию. Функция сравнения используется только здесь, поэтому нет необходимости давать ей имя. Такие функции называются анонимными (*подробнее про анонимные функции см. в разделе про функции*).

Метод concat()

Метод Array.concat() создает и возвращает новый массив, содержащий элементы исходного массива, для которого был вызван метод concat(), и значения всех аргументов, переданных методу concat(). Если какой-либо из этих аргументов сам является массивом, его элементы добавляются в возвращаемый массив. Следует, однако, отметить, что рекурсивного превращения массива из массивов в одномерный массив не происходит. Метод concat() не изменяет исходный массив. Ниже приводится несколько примеров:

```
var arr = [1,2,3];
arr.concat(4, 5);         // Вернет [1,2,3,4,5]
arr.concat([4,5]);        // Вернет [1,2,3,4,5]
arr.concat([4,5],[6,7])   // Вернет [1,2,3,4,5,6,7]
arr.concat(4, [5,[6,7]]) // Вернет  [1,2,3,4,5,[6,7]]
```

Метод slice()

Метод Array.slice() возвращает фрагмент указанного массива (часть, срез массива). Два аргумента метода определяют начало и конец возвращаемого фрагмента. Возвращаемый массив содержит элемент, номер которого указан в первом аргументе, плюс все последующие элементы, вплоть до элемента, номер которого указан во втором аргументе (но не включая последний элемент).

Если указан только один аргумент, возвращаемый массив содержит все элементы от начальной позиции до конца массива. Если ка-

кой-либо из аргументов имеет отрицательное значение, он определяет номер элемента относительно конца массива. Так, аргументу -1 соответствует последний элемент массива, а аргументу -3 - третий элемент массива с конца. Вот несколько примеров:

```
var arr = [1,2,3,4,5];
arr.slice(0,3);     // Вернет [1,2,3]
arr.slice(3);       // Вернет [4,5]
arr.slice(1,-1);    // Вернет [2,3,4]
arr.slice(-3,-2);   // Вернет [3]
```

Метод splice()

Метод Array.splice() - это универсальный метод, выполняющий вставку или удаление элементов массива. В отличие от методов slice() и concat(), метод splice() изменяет исходный массив, относительно которого он был вызван. Обратите внимание, что методы splice() и slice() имеют очень похожие имена, но выполняют совершенно разные операции.

Метод splice() может удалять элементы из массива, вставлять новые элементы или выполнять обе операции одновременно. Элементы массива при необходимости смещаются, чтобы после вставки или удаления образовывалась непрерывная последовательность.

Первый аргумент метода splice() определяет позицию в массиве, начиная с которой будет выполняться вставка и/или удаление. Второй аргумент определяет количество элементов, которые должны быть удалены (вырезаны) из массива. Если второй аргумент опущен, удаляются все элементы массива от указанного до конца массива. Метод splice() возвращает массив удаленных элементов или (если ни один из элементов не был удален) пустой массив.

Первые два аргумента метода splice() определяют элементы массива, подлежащие удалению. За этими аргументами может следовать любое количество дополнительных аргументов, определяющих элементы, которые будут вставлены в массив, начиная с позиции, указанной в первом аргументе.

```
var arr = [1,2,3,4,5,6,7,8];
```

```
arr.splice(4);           // Вернет [5,6,7,8], arr = [1,2,3,4]
arr.splice(1,2);         // Вернет [2,3], arr = [1,4]
arr.splice(1,1);         // Вернет [4]; arr = [1]
arr = [1,2,3,4,5];
arr.splice(2,0,'a','b'); // Вернет []; arr = [1,2,'a','b',3,4,5]
```

Методы push() и pop()

Методы push() и pop() позволяют работать с массивами как со стеками. Метод push() добавляет один или несколько новых элементов в конец массива и возвращает его новую длину. Метод pop() выполняет обратную операцию - удаляет последний элемент массива, уменьшает длину массива и возвращает удаленное им значение. Обратите внимание, что оба эти метода изменяют исходный массив, а не создают его модифицированную копию.

Методы unshift() и shift()

Методы unshift() и shift() ведут себя почти так же, как push() и pop(), за исключением того, что они вставляют и удаляют элементы в начале массива, а не в конце. Метод unshift() смещает существующие элементы в сторону больших индексов для освобождения места, добавляет элемент или элементы в начало массива и возвращает новую длину массива. Метод shift() удаляет и возвращает первый элемент массива, смещая все последующие элементы на одну позицию вниз, чтобы занять место, освободившееся в начале массива.

Методы include, every и some.

Метод include() проверяет наличие элемента в массиве:

```
var array1 = [1, 2, 3];
console.log(array1.includes(2)); // output: true
var pets = ['cat', 'dog', 'bat'];
console.log(pets.includes('cat')); // output: true
console.log(pets.includes('at'));  // output: false
```

Метод every проверяет все элементы массива на соответствие одному условию, а метод some - возвращает истину, если хотя бы

один из элементов массива будет соответствовать заданному условию.

```
function isEven(num) {
    return num%2==0;
}
console.log(array1.every(isEven));
    // false, не все удовлетворяют условию
console.log(array1.some(isEven));
    // true, есть хоть одно подходящее
```

Методы map, reduce, filter.

Использование функциональных методов позволит сделать ваш код гораздо чище, читабельнее и удобнее в обслуживании. Безусловно, читабельность и обслуживаемость кода не должны снижать производительность, если этого требует ситуация. Современные браузеры, до сих пор, эффективнее выполняют более громоздкие традиционные конструкции, например, циклы. Применение функциональных методов позволит извлечь больше преимуществ из улучшений JS-движка, по мере того, как браузеры будут оптимизироваться для их использования.

Маппинг — фундаментальная технология в функциональном программировании. Она применяется для оперирования всеми элементами массива с целью создания другого массива той же длины, но с преобразованным содержимым (по аналогии с циклом forEach).

Чтобы было понятнее, давайте рассмотрим простой пример. Допустим, у вас есть массив слов, и вам нужно преобразовать его в массив, содержащий длины всех слов исходного массива.

Вероятно, вы уже знаете, как выполнить описанную задачу с помощью цикла for. Например, так:

```
var animals = ["cat","dog","fish"];
var lengths = [];
var item;
for (let count=0; count<animals.length; count++) {
    item = animals[count];
    lengths.push(item.length);
}
console.log(lengths); // output: [3, 3, 4]
```

Декларативный стиль функционального программирования позволяет нам избавиться от перечисления того, как именно мы собираемся решать задачу, ограничившись лишь указанием, что мы хотим получить с помощью **метода map**:

```
var animals = ["cat","dog","fish"];
var lengths = animals.map(function(animal) {
     return animal.length;
});
console.log(lengths); // [3, 3, 4]
```

или даже так:

```
var animals = ["cat","dog","fish"];
var lengths = animals.map(animal=>animal.length);
console.log(lengths); // [3, 3, 4]
```

Итак, **метод map** работает с массивом, возвращает массив такого же размера, как и исходный, применяет последовательно ко всем элементам исходного массива функцию, которая определена в методе map как аргумент.

Обратите внимание, что при таком подходе:

– код получается гораздо короче;

– объявляется меньше переменных, а, следовательно, мы создаём меньше шума в пространстве имён, снижая вероятность возникновения коллизий, если другая часть того же кода использует переменные с теми же именами;

– нет объявленных переменных, которые в ходе цикла должны менять свои значения.

Ещё одно преимущество этого подхода заключается в том, что мы можем сделать его гибче, разделив именованную функцию. При этом код станет чище. Анонимные встраиваемые функции (как в примере выше с обычной - `function(animal) { return animal.length; }` или стрелочной функцией `animal=>animal.length`) затрудняют повторное использование кода и могут выглядеть неопрятно. Можно было бы определить именованную функцию getLength() и использовать её следующим образом:

```
var animals = ["cat","dog","fish"];
function getLength(word) {
    return word.length;
```

```
}
console.log(animals.map(getLength)); //[3, 3, 4]
```

Оцените, насколько лаконичнее стал выглядеть код.

Аналогичным образом можно использовать **метод filter**, который как и метод map работает с массивом, но возвращает массив такого же или меньшего размера в зависимости от указанного условия отбора.

Пусть поставлена задача: сформировать массив длин коротких слов. Таковыми будем считать те, которые имеют длину менее четырёх. Тогда, с точки зрения функционального программирования, нужно продекларировать, что в качестве результата нам нужен:

а) массив длин слов

б) со значением меньше 4,

то есть последовательно применить функции map и filter:

```
var animals = ["cat","dog","fish"];
console.log( animals
    .map( animal=>animal.length )
    .filter( len=>len<4 )
); // [ 3, 3 ]
```

Обратите внимание, что, для повышения удобства чтения и редактирования кода, его оформляют ступенчато (выделяя структуру функции по аналогии со структурными операторами) с выделением конкатенируемых частей составной функции. Наращивание количества составных частей формально не ограничено, например, чтобы узнать сколько таких коротких слов в массиве можно использовать такой код:

```
console.log( ["cat","dog","fish"]
        .map( animal=>animal.length )
        .filter( len=>len<4 )
        .length
);
```

Метод reduce аналогичен методу map, то есть обрабатывает массив, последовательно применяя определенную функцию к элементам массива, за исключением того, что в качестве результата возвращается одно значение, а не целый массив. Например, вам нужно получить в виде числа сумму длин всех слов в массиве animals:

```
let animals = ["cat","dog","fish"];
let total = 0;
let item;
for (let count=0, loops=animals.length; count<loops; count++) {
    item = animals[count];
    total += item.length;
}
console.log(total); // 10
```

После описания начального массива, мы создаём переменную total для подсчёта суммы, и присваиваем ей ноль. Также создаём переменную item, в которой, по мере выполнения цикла for, сохраняется результат каждой итерации над массивом animals. В качестве счётчика циклов используется переменная count, а loops применяется для оптимизации итераций. Запускаем цикл for, итерируем все слова в массиве animals, присваивая значение каждого из них переменной item, и прибавляем длины слов к нарастающему итогу.

Однако, с помощью метода reduce можно это сделать гораздо проще:

```
var animals = ["cat","dog","fish"];
var total = animals.reduce(function(sum, word) {
  return sum + word.length;
}, 0);
console.log(total);
```

или так:

```
let animals = ["cat","dog","fish"];
let total = animals.reduce((sum,word)=>sum+word.length, 0);
console.log(total);
```

Здесь мы определяем новую переменную total и присваиваем ей значение результата, полученного после применения reduce к массиву animals с использованием двух параметров: анонимной встраиваемой функции (`sum,word)=>sum+word.length`) и нарастающего итога с начальным значением ноль. Метод reduce берёт каждый элемент массива, применяет к нему функцию и добавляет получаемый результат к нарастающему итогу, который затем передаётся в следующую итерацию. Подставляемая функция получает два параметра: нарастающий итог и текущее обрабатываемое слово из массива.

Возможно, описанный подход выглядит слишком сложным. Это следствие интегрированного определения встраиваемой функции в вызываемом методе reduce. Давайте зададим именованную функцию вместо анонимной встраиваемой:

```
var animals = ["cat","dog","fish"];

var addLength = function(sum, word) {
    return sum + word.length;
};

var total = animals.reduce(addLength, 0);

console.log(total);
```

Получается немного длиннее, но, в данном случае, становится понятнее, что происходит с методом reduce - он получает два параметра: функцию, которая применяется к каждому элементу массива, и начальное значение нарастающего итога.

Несколько примеров реализаций обработки массивов.

Вы сможете детальнее разобраться в правилах применения перечисленных методов, если проанализируете несколько примеров программ для обработки массивов, приведённых далее.

Генерация массива в функциональном стиле и разные способы вывода массива в консоль:

```
let arr = '0'        // взять элемент
    .repeat(10)      // повторить, сделать строку
    .split('')       // разбить посимвольно на массив
    .map( (elm)=>    // маппинг массива
        elm=Math.floor((Math.random()*100))
    );
console.log(arr);
console.log(...arr); // оператор расширения - функция применяется
                     // последовательно к каждому элементу массива
console.log(arr.join(',')); // сборка массива в строку
```

Вариант вывода для этой программы:

```
[ 91, 24, 84, 45, 6, 74, 55, 18, 89, 93 ]
91 24 84 45 6 74 55 18 89 93
91,24,84,45,6,74,55,18,89,93
```

Найти сумму элементов массива.

```
const array1 = [1, 2, 3, 4];
const reducer =
     (accumulator, currentValue) => accumulator + currentValue;

console.log(array1.reduce(reducer)); // output: 10

console.log(array1.reduce(reducer, 5)); // 5 + 1+2+3+4

[0,1,2,3,4].reduce(function(accumulator, currentValue, index, array)
{
    return accumulator + currentValue;
}, 10); // = 20
```

Найти сумму элементов массива, если элементы хранятся как строки или как числа.

```
function sumArrayStr(values) {
     return values.map(Number).reduce(
          (preValue,currenrValue) => preValue + currenrValue, 0
     );
}
function sumArray(values) {
     return values.reduce(
          (preValue,currenrValue) => preValue + currenrValue, 0
     );
}
console.log(sumArrayStr(['2','5','10']));
console.log(sumArray([2,5,10]));
```

Методы forEach, sort и slice.

```
var arr = ["Яблоко", "Апельсин", "Груша"];
arr.forEach(function(item, i, arr) {
          console.log(i + " - " + item);
     });
arr.sort();
console.log(arr);
console.log(arr.slice(-1)); // вырезать часть массива
```

Метод filter.

```
var newArr = arr.filter(function(name) {
          return name.length < 6;
     });
console.log(newArr);

var nameLengths = arr.map(function(name) {
          return name.length;
     });
console.log(nameLengths);
```

Метод reduce.

```
var arrNumb = [1, 2, 3, 4, 5];
// для каждого элемента массива запустить функцию,
// промежуточный результат передавать первым аргументом далее
var result = arrNumb.reduce
     (
          function(sum, current) {
               return sum + current;
          },
          0
     );
console.log(result); // 15
```

Метод map.

```
var arrCase = arr.map(function(name) {
     return name.toLowerCase();
});
document.write(arrCase + "<br>"); // 15

var arrCaseLast = arr.map(function(name) {
     return name.slice(0,-1) + name.slice(-1).toUpperCase();
});
document.write(arrCaseLast + "<br>"); // 15
```

Некоторые способы генерации массива.

```
num = 10;
console.log(
    [...Array(10).keys()]
);
console.log(
    [...Array(10).keys()].map(_=>0)
);
console.log(
    Array
    .apply(null, {length: num})
    .map(Number.call, Number)
);
console.log(
    Array
    .apply({}, Array(num))
    .map((_, i) => i++)
);
console.log(
    Array
    .apply({}, Array(num))
    .map((_, i) => ++i)
    .reduce((a, b) => a + b)
);
```

Вывод:

```
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
55
```

Сравнение императивного и функционального подходов.

Рекурсивная функция.

```
// сумма чисел от 1 до 100
function summaRec(num) {
     return ! num ? num : num + summaRec(--num);
}
console.log(summaRec(100)); // output: 5050
```

Не всегда функциональный подход делает программу короче или читабельней. Оцените решение этой же задачи функциональным стилем:

```
arr = [...Array(100).keys()]
    .map((elm,index)=>elm=index+1)
    .reduce((acc,elm)=>acc+elm);
console.log(arr); // 5050
```

или даже так:

```
console.log([...Array(100).keys()]
     .reduce((acc,elm)=>acc+ ++elm, 0)
); // 5050
```

2.1. Функции

Функции являются основными структурными элементами программного кода в функциональном программировании. Они являются фундаментом, на котором можно построить модульную, стабильную, «прозрачную», легко изменяемую и тестируемую систему. Прежде всего, подход строится на композиции элементарных функций, сочетанием которых можно создавать более сложные функции.

Функция — это любая именованная часть программного кода, включающего вычисляемое выражение. Функции могут возвращать вызывающему коду вычисляемое или неопределенное (undefined в функции типа void, функция без команды return) значение.

```
function getA(x) {
    return 2 * x;
}
function getB(x) {
    console.log(2 * x);
}

console.log(getA(5));
console.log(getB(5));
```

Вывод данной программы:

```
10
10
undefined
```

Так как функциональное программирование, во многом, соответствует принципам математики, то функции имеют смысл лишь в том случае, если они выдают полезный, а не пустой (null) или неопределенный (undefined) результат. В противном случае предполагается, что они модифицируют внешние данные и вызывают побочные эффекты.

Здесь следует явно выразить различие между структурными операторами, как функциями, не выдающими никаких значений (например, оператор if else), и операциями/выражениями, как функциями, возвращающими значения (например, условная тернарная операция _?_:_). Процедурное или императивное (*от император - повелитель*) программирование состоит, главным образом, из упорядоченных последовательностей операторов, а функциональное программирование — полностью или почти полностью из выражений (операций, что-либо возвращающих).

Как правило, в универсальном языке, к каковым и относится JS, затруднительно (да, и не имеет смысла) написать программу, полностью отражающую парадигму функционального программирования. К языкам функционального программирования относят: Lisp (Scheme), Erlang, Haskell, F#. Теоретической базой для описания и вычисления функций в таких языках стало лямбда-исчисление, которое базируется на двух фундаментальных принципах: аппликация (лат. applicatio —присоединение) и абстракция (лат. abstractio —отделение). Можно привести аналоги подобного подхода в универсальных языках программирования. Например, в C# используются лямбда-выражения в виде анонимных функций, как особого вида функций, которые объявляются в месте их использования и не получают уникального идентификатора для доступа к ним с другой точки программы: (x,y) => x+y. Здесь круглые скобки являются вызовом функции без имени с передачей аргументов, а литерал => интерпретируется как команда return для возвращения вычисляемого значения из функции. Функции, написанные в таком формате, часто называют стрелочными (Arrow Functions) из-за литерала =>. Более того, сами функциональные языки могут быть, в зависимости от точности соблюдения обозначенных ранее принципов, «чистыми» (Haskell) и «нечистыми» (F#).

Одной из основных концепций функционального программирования является использование функций высшего порядка, которые могут в качестве аргументов передавать и возвращать не только переменные, но функции:

```
// стандарт ES5
var f = function (x) {
    return x + 3;
};
var g = function (func, x) {
    return func(x) * func(x);
};
console.log(g(f, 7));

// стандарт ES6
let t = x => x + 3;
let q = (func, x) => func(x) * func(x);
console.log(q(t, 7));

Вывод для данной программы:
100
100
```

К дополнительным возможностям функций высшего порядка в JS стоит отнести поддержку присваивания их переменным и сохранения их в структурах данных. Правильно построенная («чистая») функция:

- должна принимать аргументы,
- должна возвращать значение,
- не использует глобальные переменные,
- не использует методы объектов «со стороны».

Можно сделать вывод, что определение «чистой функции» в JS тождественно понятию функции в математике: y = f(x).

Изучите примеры функций:

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

```
function step(a, b) {
   return Math.pow(a,b); // не чистая (сторонний метод)
}
```

```
function getDate() {
   return Date.now(); // не чистая
}
```

```
let structStudent = {}; // объект
function addName(name) { // не чистая
   structStudent.name = name; // свойство объекта
}
addName("Uny");
document.write(structStudent.name);
```

```
// Пример передачи функции как параметра в другую функцию
var timer;
var interval = 500;
var maxRow = 20, maxCol = 100, row = 0;
var symbols = "_|";

function move() { // сначала декларация функции
    let str = "";
    for (let col=0; col<maxCol; col++) {
        let pos = Math.floor(Math.random()*symbols.length);
        str += symbols[pos];
    }
    console.log(str);
    if (++row>maxRow) clearInterval(timer);
}
console.clear();
timer = setInterval(move, interval); // затем передача функции
```

```
// вызов функции через аргумент другой функции
function addition(a, b) {
    return a+b;
};
function callFunction(func, ...args) {
    return func(...args); // … - это оператор расширения
}
console.log( callFunction(addition, 2, 8) );
```

Анонимные функции

Анонимные функции (Named Function Expression) используют в функциональном программировании в ситуациях, когда функция применяется один раз в рамках некоторого выражения. Тогда и не нужна именованная функция к которой можно было бы обращаться из любого места программы по имени.

Рассмотрим пример:

```
[1, 2, 3, 4].map(function (elm) { return elm*10; })
```

Вывод:

```
[ 10, 20, 30, 40 ]
```

В данном случае в качестве аргумента функции map мы передаём функцию без идентификатора. Передаваемая функция последовательно применяется к каждому из элементов исходного массива, увеличивая значения элементов в 10 раз. Если же у нас есть ранее написанная именованная функция, увеличивающая аргумент в 10 раз, то можно использовать и её:

```
function multiply(elm) { return elm*10; }
console.log([1, 2, 3, 4].map(multiply));
```

Вывод:

```
[ 10, 20, 30, 40 ]
```

Стрелочные функции

Стрелочные функции (лямбда-выражения) также являются анонимными, но имеют сокращённый синтаксис записи. Рассмотрите программу, решающую ту же задачу, но с использованием стрелочной функции:

```
console.log([1, 2, 3, 4].map(elm=>elm*10));
```

Код программы выглядит несколько короче, лишён избыточных символов и слов и, после некоторой привычки написания в таком стиле, ощущается более читабельным.

Традиционный оператор ветвления ничего не возвращает, поэтому его нельзя вставить в другую функцию, например, в

console.log(). Однако применение функционального подхода позволяет создать и использовать такую функцию, которая будет оперировать другими функциями в зависимости от условия динамически:

```
// динамический выбор функции для исполнения
function ifElse(condition, arg, funcTrue, funcFalse) {
    if (condition) {
        return funcTrue(arg);
    }
    else {
        return funcFalse(arg);
    }
}

a = arg => arg + " - even"; // используем лямбда-выражение
b = arg => arg + " - odd"; // используем лямбда-выражение

x = 18;
console.log( ifElse(x%2==0, x, a, b) );
```

Внутреннее имя функции.

Функцию можно рассматривать как объект, тогда ссылку на неё можно присвоить переменной и передавать. При таком копировании внутреннее имя функции не видно снаружи, но может использоваться самой функцией. При таком подходе могут возникать непредвиденные ситуации при обращении с функцией, с объектом-функцией, со ссылками на функцию, с обращением к внутреннему имени функции.

Возьмём функцию, вычисляющую факториал числа:

```
function f(n) {
    return n? n*f(n-1): 1;
};
alert( f(5) ); // ответ 120
```

Имя функции f видно и её можно использовать напрямую.

Попробуем перенести её в другую переменную g:

```
function f(n) {
    return n? n*f(n-1): 1;
};
var g = f;
```

```
f = null;
alert( g(5) ); // ошибка при выполнении
```

Ошибка возникла потому, что при таком присваивании - `g = f` в переменную g передаётся ссылка на объект f, а не сам объект - функция f, которой, к моменту использования, уже нет: f = null. Для того, чтобы функция всегда надёжно работала, объявим её как объект:

```
var f = function factorial(n) { // f – это объект
    return n? n*factorial (n-1): 1;
};
var g = f; // копируется сам объект
f = null;
alert( g(5) ); // функция работает
```

Уточним понятие внутреннего имени функции. Если именованную функцию присваивать в переменную, то её имя принято называть внутренним и оно недоступно напрямую из программы. Сравните несколько примеров реализаций:

```
// обычная функция
function f(n) {
    return n? n*f(n-1): 1;
};
console.log( "f)", f(5) ); // имя f доступно, её можно вызвать

console.log(f.name); // отобразит имя
console.log(typeof f); // отобразит тип function
console.log(f.toString()); // отобразит код

n = f;
f = null;
// console.log( "n)", n(5) ); // недопустимый вызов
                // так как ссылка на функцию удалена

Вывод данной программы
f) 120
f
function
function f(n) {
    return n? n*f(n-1): 1;
}
```

```
// функция-объект
a = function f(n) { // внутреннее имя f
    return n? n*f(n-1): 1;
};

// console.log( "#", f(5) ); // функция f недоступна

console.log("a)", a(5)); // ответ 120
b = a;
a = null; // удалили a
console.log("b)", b(5)); // ответ 120

// console.log(a.name) // недопустимый вызов

console.log(b.name); // отобразит внутреннее имя
console.log(typeof b); // отобразит тип function
console.log(b.toString()); // отобразит код

Вывод данной программы:
a) 120
b) 120
f
function
function f(n) { // внутреннее имя f
    return n? n*f(n-1): 1;
}
```

Переход от императивного к функциональному стилю.

Мы рассмотрели различные аспекты применения функций в JS. Основываясь на полученных знаниях можно проанализировать переход от императивного к функциональному программирвоанию на простых примерах.

Пример 1.

Дан массив, нужно все его элементы увеличить в 10 раз.

```
let arr = [0, 1, 2, 3, 4];
// задача: все элементы увеличить в 10 раз

// традиционный подход - императивный
for (let i=0; i<arr.length; i++) {
    arr[i] = 10*arr[i];
}

console.log(arr);
```

```
Вывод: [ 0, 10, 20, 30, 40 ]
```

```
let arr = [0, 1, 2, 3, 4];

// функциональный подход - сделаем специальную функцию
function m(arg) {
    return 10*arg;
}

// и функцию для работы с элементами массива
function forEach(arr, fn) {
    let newArr = [];
    for (let i=0; i<arr.length; i++) {
        newArr[i] = fn(arr[i]);
    }
    return newArr;
}

console.log(forEach(arr, m));

Вывод: [ 0, 10, 20, 30, 40 ]
```

```
let arr = [0, 1, 2, 3, 4];

// функциональный подход - возможности JS
m = arg => 10 * arg; // стрелочная функция
console.log( arr.map(m) ); // через имя объекта-функции
console.log( arr.map(elm=>10*elm) ); // с анонимной функцией
Вывод: [ 0, 10, 20, 30, 40 ]
```

Как видите, функциональный подход даёт более «чистый», короткий, читабельный код.

Пример 2.

Дан массив, нужно все его элементы увеличить в 10 раз и затем выбрать только большие 20.

```
let arr = [0, 1, 2, 3, 4];
// задача: все элементы увеличить в 10 раз
// и затем выбрать только большие 20

// традиционный подход - императивный
for (let i = 0; i < arr.length; i++) {
    arr[i] = 10*arr[i];
}
let newArr = [];
for (let i = 0; i < arr.length; i++) {
    if (arr[i] > 20) {
        newArr.push(arr[i]);
    }
```

```
}
console.log(newArr);
```

```
let arr = [0, 1, 2, 3, 4];
// функциональный подход – требует предварительной подготовки
// сделаем специальную функцию
function m(arg) {
    return 10*arg;
}

function forEach(arr, fn) { // изменить все элементы массива
    let newArr = [];
    for (let i=0; i<arr.length; i++) {
        newArr[i] = fn(arr[i]);
    }
    return newArr;
}

check = function (arg) {
    return arg>20;
}

function filterArr(arr, filter) { // отфильтровать элементы
    let newArr = [];
    for (let i = 0; i < arr.length; i++) {
        if (filter(arr[i])) {
            newArr.push(arr[i]);
        }
    }
    return newArr;
}

// после подготовки
// сама реализация короткая
console.log(forEach(arr, m));
console.log(filterArr( forEach(arr, m), check));

Вывод:
[ 0, 10, 20, 30, 40 ]
[ 30, 40 ]
```

```
let arr = [0, 1, 2, 3, 4];
// функциональный подход - возможности JS
console.log( arr.map(elm=>10*elm).filter(elm=>elm>20) );

Вывод:
[ 30, 40 ]
```

Пример 3.

Дан массив, нужно все его элементы увеличить в 10 раз и затем выбрать только большие 20 и найти их сумму.

По аналогии можете самостоятельно проделать этапы перехода от императивного стиля к функциональному для данной задачи. Я же приведу итоговый лаконичный результат:

```
console.log(
    arr
        .map(elm => 10 * elm)
        .filter(elm => elm > 20)
        .reduce((acc,elm)=>acc+elm, 0)
);

Вывод:
70
```

Как видите, при усложнении задачи обработки данных, можно частные функции собирать в одну сложную.

Итак, базовые возможности функционального подхода мы рассмотрели и можем сделать обобщающий вывод, что для функционального программирования характерно наличие следующих возможностей:

- сохранять функцию в переменной;
- передавать функцию в качестве аргумента другой функции;
- возвращать функцию в результате вызова функции;
- составлять комбинацию из функций.

Перечисленные возможности подходят для обработки потока данных, ускоряют процесс кодирования и внесения изменений, сокращают объем программы и делают её «читабельней».

2.2. Объекты.

В традиционном объектно-ориентированном программировании сначала декларируется класс с описанием его свойств и методов и потом, на основе этого описания, создаются экземпляры класса - объекты. В JS, в отличие от традиционных объектно-ориентированных языков программирования, объект создаётся простым присваиванием в переменную набора свойств. Количество свойств не является фиксированным - их можно менять, добавлять или удалять в процессе работы программы. Значением свойства в JS может быть и функция, которую можно назвать методом объекта.

Итак, в JS объект - это некоторая абстрактная сущность, которая объединяет в себе набор переменных (свойства) и функций (методы). Объект в JS структурно устроен как ассоциативный массив, то есть в нём можно хранить некоторое количество значений разного типа и доступ к ним осуществляется не по порядковым номерам, а по ключам. То есть свойства и методы объекта, подобно именованным функциям и переменным, обладают и идентификатором и значением. В объекте такой идентификатор называется ключом. Ключи используются для доступа к значениям, которые с ними ассоциированы. Значением свойства может быть число, строка, логическое значение или другой объект. Значением метода может быть только функция.

Рассмотрим несколько способов инициализации (инстанцирования) объектов в JS и доступа к их свойствам и методам.

Традиционный подход состоит в фиксированном задании объекта, когда его содержимое, то есть свойства и методы описываются сразу все через запятую и единовременно списком присваиваются в одну переменную:

```
// способ 1 работы с объектом
    var obj = {
        x: 10,
        y: 21,
        getS: function() {
```

```
            return this.x*this.y;
        }
    }
```

Для некоторых случаев объект можно формировать динамически в необходимый момент работы программы, добавляя свойства и методы:

```
// способ 2 работы с объектом
    var obj = new Object(); // создать пустой объект
    obj = {
        x: 10 // добавить свойство
    }
    obj.y = 20; // добавить свойство
    obj.getS = function() { // добавить метод
        return this.x*this.y;
    }
```

Если имена свойств и/или методов не известны заранее, но могут быть получены во время работы программы, то имеет смысл использовать их как ключи ассоциативного массива:

```
// способ 3 работы с объектом
    var obj = new Object(); // создать пустой объект
    obj["X"] = 10; // добавить свойство
    obj["Y"] = 22; // добавить свойство
    obj["getS"] = function() { // добавить метод
        return this.X*this.Y;
    }

console.log(obj.getS()); // вызвать метод объекта
```

Так как в JS нет возможности предварительно декларировать класс, то и в явном виде не существует конструкторов класса. Однако можно организовать конструктор объекта через обычную функцию, включающую в себя блок операторов, работающих через ключевое слово this. This является ссылкой на текущее имя объекта в тексте программы как на идентификатор, в который ранее было осуществлено присвоение объекта:

```
// способ 4 работы с объектом
function Student(_name, _age) { // функция - конструктор объекта
```

```
    this.name1 = _name.split(' ')[0];
    this.name2 = _name.split(' ')[1];
    this.age = _age || 16; // _age или по умолчанию
    this.printAge = function () {
        console.log("Мой возраст =", this.age);
    }
    this.setAge = function (age) {
        this.age = age;
    }
    this.getName = function () {
        return this.name2;
    }
}
man = new Student("Петров Вася"); // вызов конструктора
man.printAge(); // использование методов объекта
man.setAge(22);
man.printAge();
console.log(man.getName());
```

Вывод:

```
Мой возраст = 16
Мой возраст = 22
Вася
```

В данном варианте использования объекта прямо при его создании можно передавать некоторые параметры как аргументы функции `Student("Петров Вася")`, которые инициализируют значения свойств объекта. Обратите внимание на вывод и смену значений по ходу работы данной программы. Если мы, с помощью команды new, в конструктор передаём не все необходимые для корректной работы объекта аргументы, то их можно назначить значениями по умолчанию внутри самого объекта: `this.age = _age || 16`. В данном случае переменная _age не определена и имеет значение undefined, поэтому срабатывает логическое ИЛИ, что приводит к выбору и присвоению в this.age значения 16 (это значение по умолчанию).

Передавать, хранить и обрабатывать данные сгруппированные в объекты или даже массивы объектов довольно удобно, поэтому в web-программировании очень часто используется так называемый

JSON-формат представления данных (от англ. JavaScript Object Notation). В самом языке программирования есть специальные методы, которые позволяют объект собирать в строку (метод `stringify`) и обратно – из строки формировать объект (метод `parse`):

```
var event = {
    title: "Конференция",
    date: "сегодня",
    count: 120 // кол-во участников
};

var str = JSON.stringify(event); // из объекта в строку
console.log( str );
     // {"title":"Конференция","date":"сегодня","count":120}

// Обратное преобразование
newEvent = JSON.parse(str); // создаём объект
for (const key in newEvent) { // для всех членов объекта
    if (typeof(newEvent[key])!='number') // если не число
        console.log(key,":",newEvent[key]);
}
```

Вывод:

```
title : Конференция
date : сегодня
```

Приведу наглядный пример применения методов функционального программирования к массиву объектов. Пусть даны три объекта – это люди (свойства: имя, возраст, пол), данные поместили в массив arr. Нужно по определённым правилам обработать элементы массива и вывести результат на экран: выбрать людей мужского пола, с сортировкой по алфавиту, с сортировкой по возрасту в обратном порядке, средний возраст совершеннолетних:

```
men1 = {
    name: "Petrov",
    age: 14,
    sex: true
}
men2 = {
    name: "Bratko",
    age: 22,
```

```
    sex: false
}
men3 = {
    name: "Mask",
    age: 28,
    sex: true
}
arr = [men1, men2, men3]; // массив объектов

// только мужского пола - в столбик
arr.filter(elm=>elm.sex).map(elm=>console.log(elm.name));

// только мужского пола - в строку
console.log(arr
    .filter(elm=>elm.sex)
    .reduce((acc,elm)=>acc.name+" "+elm.name)
);  // reduce можете заменить на join

// только мужского пола - в строку - с сортировкой по алфавиту
console.log(arr
    .filter(elm=>elm.sex)
    .sort((a,b)=>(a.name>b.name?1:-1))
    .reduce((acc,elm)=>acc.name+", "+elm.name)
);  // reduce можете заменить на join

// только мужского пола - отсортировать по возрасту по убыванию
arr
    .filter(elm=>elm.sex)
    .sort((a,b)=>b.age-a.age)
    .map(elm=>console.log(elm.name));

// средний возраст совершеннолетних
console.log(arr
    .filter((elm)=>elm.age>17)
    .reduce((acc,elm)=>acc.age+elm.age)
    /
    arr.filter((elm)=>elm.age>17).length
);
```

Работать с данными, сгруппированными в массив объектов удобно, когда необходимо обработать табличную информацию. Представьте таблицу как массив строк, а строку как массив элементов строки - тогда можно из каждой строки сделать объект с полями,

составленными из элементов строки, а сами объекты поместить в массив.

Подробный пример по использованию объектов для работы со структурированными текстовыми файлами будет представлен в разделе, посвященном работе с файловой системой в Node.js.

Ещё несколько примеров работы с объектами:

```
                                                    // пример работы с объектами
let obj = {
    // поля
    arrFib: [],
    count: 0,
    // методы
    addElm(num) {
        this.arrFib.push(num);
        this.count++;
    },
    getElm(index) {
        return this.arrFib[index];
    }
}
obj.addElm(100);
obj.addElm(200);
for (let i=0; i<obj.count; i++) {
    console.log( obj.getElm(i) );
}
Вывод:
100
200
```

```
                                                    // пример работы с объектом
let obj = {
    arr: [],
    addElm(num) {
        this.arr = this.arr.concat(num);
        return this.arr;
    }
}
console.log( obj.addElm(1) );
console.log( obj.addElm(2) );
console.log( obj.addElm(3) );
Вывод:
[ 1 ]
[ 1, 2 ]
[ 1, 2, 3 ]
```

```
                                  // как перебирать свойства объекта
let obj = {
  name: "Петров",
  age: 18
};

// in для перебора свойств в объекте
for (let property in obj) {
  console.log(property);
}

for (let property in obj) {
  if (typeof(obj[property])=="string")
    console.log(property);
}

for (let property in obj) {
  if (typeof(obj[property])=="number") {
    obj[property] += 1;
    console.log(obj[property]);
  }
}

Вывод данной программы:
name
age
name
19
```

Итак, в JS при работе с объектами не декларируют сначала классы и не строят цепочки наследования, однако, чтобы эмулировать наследование в JS можно воспользоваться механизмом прототипирования.

Наследование в JS

Если некоторые свойства и методы повторяются в объектах разного вида, то их общие характеристики можно вынести в отдельный объект и наследовать их от него. Это позволяет оптимизировать программу, исключив дублирование программного кода, и гибче работать со структурами данных, комбинируя их функционал.

Однако в JS нет декларации классов и, соответственно нельзя описать наследование в классах, и уже на их основе создавать экземпляры - объекты с требуемым функционалом. Свойства и методы объектов в JS можно передавать с помощью механизма прототипирования, подразумевающего копирование структуры (свойств и методов) от одного объекта (прототипа) непосредственно к другому объекту. Связующим звеном между объектами выступает специальное свойство __proto__. Если объект имеет такое свойство, определяющее ссылку на другой объект, то свойства обоих объектов доступны через имя первого. Такое наследование называется прототипным и оно может быть только прямым – от объекта-прототипа к объекту-потомку, но не наоборот. Следует учитывать, что, при обращении к некоторому свойству объекта-потомка, оно сначала ищется в самом объекте и, если свойство отсутствует в нём, тогда уже свойство ищется в прототипе:

```
animal = {
    age: 16 // у живого есть возраст
}
human = {
    name: "Иванов Иван Иванович", // у человека есть имя
}
human.__proto__ = animal;
console.log(human.age+"\t"+human.name);
```

Вывод для данной программы:

```
16      Иванов Иван Иванович
```

Прототипирование не может обеспечить множественное наследование - когда объект-потомок наследует одновременно от нескольких прототипов:

```
animal = {
    age: 16 // у живого есть возраст
}
human = {
    name: "Иванов Иван Иванович", // у человека есть имя
    getName: function (pos) {
        return this.name.split(' ')[pos];
    }
}
student = {
    group: "ПИб-1"
}

student.__proto__ = animal;
student.__proto__ = human;

console.log(
    student.getName(1)+"\t"+student.age+"\t"+student.group
);
```

Вывод для данной программы:

```
Иван      undefined      ПИб-1
```

Однако, при прототипировании объекты можно организовать в цепочки, когда сам объект-прототип является потомком от другого прототипа.

```
animal = {
    age: 16 // у живого есть возраст
}
human = {
    name: "Иванов Иван Иванович", // у человека есть имя
    getName: function (pos) {
        return this.name.split(' ')[pos];
    }
}
student = {
    group: "ПИб-1" // для студента определена группа
}

human.__proto__ = animal;
student.__proto__ = human;
```

```
console.log(
     student.getName(1)+"\t"+student.age+"\t"+student.group
);
```

Вывод для данной программы:

```
Иван    16      ПИб-1
```

Прототипирование работает не только для статичных объектов, но и при использовании функций - конструкторов объектов:

```
function Animal (_age) { // конструктор объекта
    this.age = _age; // у живого есть возраст
}
function Human(_name) { // конструктор объекта
    this.name = _name, // у человека есть имя
    this.getInit = function () {
        return this.name.split(' ')[0]+" "+
            this.name.split(' ')[1][0]+"."+
            this.name.split(' ')[1][0]+"."; }
}
student = { // статичный объект
    group: "ПИб-1"
}

animal = new Animal(16); // создаём объект
human = new Human("Иванов Иван Иванович"); // создаём объект

human.__proto__ = animal; // прототипирование
student.__proto__ = human; // прототипирование

for (key in student) { // перечисляем все свойства объекта
    console.log(key, '\t', student[key]);
}

for (key in student) { // перечисляем только родные свойства объекта
    if (student.hasOwnProperty(key))
        console.log('->', key, '\t', student[key]);
}

console.log(student.getInit()+"\t"+student.age);
```

Вывод данной программы:

```
group    ПИб-1
name     Иванов Иван Иванович
getInit          function () {
        return this.name.split(' ')[0]+" "+
```

```
            this.name.split(' ')[1][0]+"."+
            this.name.split(' ')[1][0]+"."; }
age      16
-> group          ПИб-1
Иванов И.И.      16
```

Summary

В JavaScript есть встроенное «наследование» непосредственно между объектами при помощи специального свойства __proto__.

При установке свойства obj2.__proto__ = obj1 говорят, что объект obj1 будет прототипом для obj2.

Если при обращении к свойству объекта-потомка окажется, что в нём нет искомого идентификатора, то свойство ищется в прототипе.

Из прототипа свойства можно только прочитать, но операции удаления свойства delete obj.prop или присвоения к свойству obj.prop = совершаются непосредственно над элементами самого объекта-потомка и не затрагивают объект-прототип.

Несколько прототипов одному объекту присвоить нельзя, но можно организовать объекты в цепочку, когда первый объект ссылается на второй при помощи прототипирования, второй ссылается на третий, и так далее.

2.3. Модули

Одним из удобных способов структурирования программ является модульное программирование, подразумевающее хранение текста программы в нескольких отдельных файлах. В модуль можно вынести константы, часто используемые функции, коллекции, объекты (классы).

Из явных преимуществ такого подхода можно указать следующие:

– одну программу могут создавать несколько программистов одновременно;

– функции модуля можно использовать сразу в нескольких разных программах;

- при изменении функционала, не нужно переустанавливать всё, а только избранные изменённые модули.

Итак, модуль состоит, прежде всего, из функций, но не все из них, как правило, доступны для использования во внешней, по отношению к модулю, программе. Разработчик модуля имеет возможность часть функций, тех, которые используются только как вспомогательные для работы основных функций модуля, скрыть. Открытая часть модуля представляет собой, так называемый API (application programming interface) - это набор готовых констант, функций, классов, структур, процедур, предоставляемых модулем, сервисом, веб-приложением либо операционной системой для применения во внешних программных продуктах.

От API зависит функциональность, предоставляемая модулем или программным продуктом, и при этом пользователю можно не задумываться, как именно это удобство реализовано в коде. Если мы будем рассматривать некий модуль, как "чёрный ящик", то API представляет собой некоторое количество тумблеров, кнопок, ручек настройки, которые может крутить и нажимать пользователь данного

"чёрного ящика", чтобы получить необходимый результат, не вдаваясь в подробности, как именно это реализовано.

Оформление модуля в node.js и порядок работы с его функциями рассмотрим на примере программы генерации массива случайных целых чисел и вывода на консоль суммы его чётных и нечётных элементов.

Пример оформления модуля m_00.js:

```
var maxDefault = 10; // по умолчанию
function getArrRnd(count, max) {
    maxValue = max || maxDefault;
    // maxValue определяем динамически
    return '0' // взять символ
        .repeat(count) // повторить count раз
        .split('') // разделить на массив
        .map((elm)=>elm=Math.floor((Math.random()*maxValue)));
        // сделать элементы массива случайными
}
function getArrAdd(arr) { // сумма элементов массива
    return arr.reduce((acc,elm)=>acc+elm);
}
module.exports.getArrRnd = getArrRnd; // доступ открываем
module.exports.getAdd = getArrAdd; // имя можно подменить
global.maxLocal = maxDefault; // доступна во всех частях программы
```

и программы p_00.js, работающей с данным модулем:

```
m = require("./m_00.js");
console.clear();

arr = m.getArrRnd(10,100); // 10 чисел от 0 до 100
//let arr = m.getArrRnd(10); // второй аргумент можно
// не указывать, тогда будет назначено значение
// определённое в модуле по умолчанию = 10
arrEven = arr.filter((elm)=>elm%2==0);
arrOdd = arr.filter((elm)=>elm%2!=0);
```

```
str = arr.join(' ') +
          '\n' + arrEven.join(' ') +
          '\n' + arrOdd.join(' ');
console.log(str);
str = m.getAdd(arrEven)+"+"+
      m.getAdd(arrOdd)+"="+
      m.getAdd(arr);

console.log(str);
console.log(maxLocal);
```

Обратите внимание, что в модуле присутствуют строки кода: **module.exports**, которые предназначены для того, чтобы сделать указанные в них методы публичными, то есть доступными извне модуля (сторонними программами).

В самой программе создаётся объект для работы с публичными функциями модуля: **m = require("./m_00.js")**.

В дальнейшем через него можно будет использовать функции модуля: `arr = m.getArrRnd(10,100)`.

Разберём логику работы функции `getArrRnd` по шагам.

Функция предназначена для генерации массива случайных целых чисел в диапазоне от 0 до указанного пользователем.

Функция принимает два аргумента `(count, max)` – это количество элементов в генерируемом массиве и верхняя граница генерируемого целого числа. Если пользователь не указал max, тогда значение принимается равным величине, определённой в модуле по умолчанию `maxDefault`. Этим значением, впоследствии, можно будет воспользоваться и в других частях программы, так как в модуле данную переменную определили как глобальную.

Далее рассмотрим детально, что именно функция возвращает:

```
'0'.repeat(count)
    .split('')
    .map( (elm)=>
                elm=Math.floor((Math.random()*max))
    );
```

На первом шаге составной функции за основу берётся символ '0', из которого формируется строка длинной count. Далее строка разбивается на массив отдельных символов методом split, затем, с помощью метода map перебираются все элементы массива и к ним применяется стрелочная функция, которая к каждому элементу массива присваивает сгенерированное случайное целое число в диапазоне (0,max].

Тема 3. Работа с файловой системой в Node.js

Для работы с файловой системой в Node.js необходимо подключить модуль с соответствующими функциями чтения/записи (readFileSync/writeFileSync), доступ к которым будет возможен через созданный объект:

```
fs = require('fs'); // создаём объект

fileNameIn = "input.txt"; // имя входного файла
     // входной текстовый файл создайте заранее
fileNameOut = "output.txt"; // имя выходного файла

text = fs.readFileSync(fileNameIn, 'utf8'); // всё содержимое
     // файла загружаем в одну строковую переменную
fs.writeFileSync(fileNameOut, text); // выводим в другой файл
```

или можно вывести всё содержимое на экран:

```
console.log(text);
```

Разберём способы работы с файлами на примерах. Пусть есть конфигурационный файл для некоторой служебной программы - "setup.cfg":

```
настройки оповещения системы
[alarm:15]
// влияние на потоки
[thread:101]
// сортировка
[sort:buble]
// фильтры
[ifelse:boolean]
```

Необходимо написать программу, которая считывает содержимое конфигурационного файла, парсит его, выбирая только атрибуты и их значения, и выводит их на экран в формате строки через запятую в отсортированном виде:

```
alarm:15, ifelse:Boolean, sort:buble, thread:101
```

В функциональном стиле данная программа не будет хранить результаты промежуточных вычислений в переменных и уложится в одну строку из сочетания функций:

```
console.log( // вывести на экран
```

```
    require("fs") // объект для работы с файлами
        .readFileSync("setup.cfg", "utf-8") // читаем файл
        .split("\r\n") // разбиваем на массив из строк файла
        .filter(line=>line[0]=='[') // берём только с атрибутами
        .map(line=>line.slice(1,line.length-1)) // без скобок
        .sort() // отсортируем
        .join(", ") // элементы массива в строку
);
```

Вывод:

```
alarm:15, ifelse:Boolean, sort:buble, thread:101
```

Если потребуется написать программу, которая принимает название конфигурационного файла из консоли в качестве аргумента, то в программу следует внести некоторые изменения (не включая проверку на корректность ввода пользователем):

```
fileIni = process.argv[2]==undefined? "setup.cfg": process.argv[2];
console.log(
    require("fs")
        .readFileSync(fileIni, "utf-8")
        .split("\r\n")
        .filter(line=>line[0]=='[')
        .map(line=>line.slice(1,line.length-1))
        .sort()
        .join(", ")
);
```

Сохранение имени входного файла в промежуточной переменной fileIni не обязательно, но внесено тут для наглядности. Программа анализирует - был ли установлен при вводе пользователем аргумент (имя конфигурационного файла), если нет, то тогда устанавливается имя по умолчанию (`"setup.cfg"`). Для этой программы (пусть её имя будет – "readToArray.js") корректный вызов из консоли должен выглядеть так:

```
node readToArray setup.cfg
```

Теперь рассмотрим более сложный пример. Нужно будет распарсить файл "football.txt" с результатами российского футбольного чемпионата и представить результаты в определённом формате.

Для начала рассмотрим формат входного файла – в первых двух строках заключена служебная информация (источник и названия полей таблицы), а все последующие строки содержат необходимые данные по командам:

```
Источник: https://www.sport-express.ru/football/L/russia/premier/2017-2018/
место     Команда   И    В    Н    П    М    О
1    Локомотив 30   18   6    6    41 - 21   60
2    ЦСКА 30   17   7    6    49 - 23   58
3    Спартак   30   16   8    6    51 - 32   56
4    Краснодар 30   16   6    8    46 - 30   54
5    Зенит     30   14   11   5    46 - 21   53
6    Уфа  30   11   10   9    34 - 30   43
7    Арсенал Т 30   12   6    12   35 - 41   42
8    Динамо    30   10   10   10   29 - 30   40
9    Ахмат     30   10   9    11   30 - 34   39
10   Рубин     30   9    11   10   32 - 25   38
11   Ростов    30   9    10   11   27 - 28   37
12   Урал 30   8    13   9    31 - 32   37
13   Амкар     30   9    8    13   20 - 30   35
14   Анжи 30   6    6    18   31 - 55   24
15   Тосно     30   6    6    18   23 - 54   24
16   СКА-Хабаровск  30   2    7    21   16 - 55   13
```

Названия поле обозначают столбцы:

```
Место команды в чемпионате
Название команды
Количество игр
Количество выигрышей
Количество ничьих
Количество проигрышей
Забитые и пропущенные мячи
Набранные очки
```

Все данные по одной команде написаны в одной строке через символ табуляции. Столбец – забитые и пропущенные мячи состоит из: числа забитых мячей, затем пробел, затем дефис, снова пробел, число пропущенных мячей.

Исследуйте текст программы:

```
var fs = require('fs'); // объект для работы с файлами
let fileNameIn = "football.txt"; // входной файл

let text = fs.readFileSync(fileNameIn, 'utf8'); // читаем файл
let lines = // тут будет массив строк файла
     text // весь текст
          .split('\r\n') // разбить на строки
          .slice(2); // срез массива без служебных строк

let properties = // тут имена столбцов/полей
     ["place","name","game","wins","n","p","balls","points"];
let arrObj = []; // создать массив для хранения объектов
for (const line of lines) { // перебрать строки
    let newObj = {}; // новый объект
    for (let i=0; i<properties.length; i++) {
        newObj[properties[i]] = line.split('\t')[i];
    } // добавляем значения по полям
    arrObj.push(newObj); // добавить объект
} // сформировать массив объектов из строк

arrObj.forEach( // можно сделать через map
    obj=>
        print(obj.place, obj.balls.split(' ')[2], obj.name)
); // для всех элементов массива объектов

function print(...arg) { // пакует аргументы в массив
    console.log(arg.join('\t')); // объединяет в строку
} // печать массива через TAB
```

Продукционная экспертная система

Традиционно под экспертной системой понимают компьютерную программу, предназначенную для выдачи ответов по ограниченному ряду вопросов в рамках определенной предметной области знаний. Работа экспертной системы основана на модели знаний, как правило, продукционной или фреймовой. Структурно экспертная система может быть представлена тремя самостоятельными блоками:

– базой знаний,
– машиной вывода,
– интерфейсом пользователя.

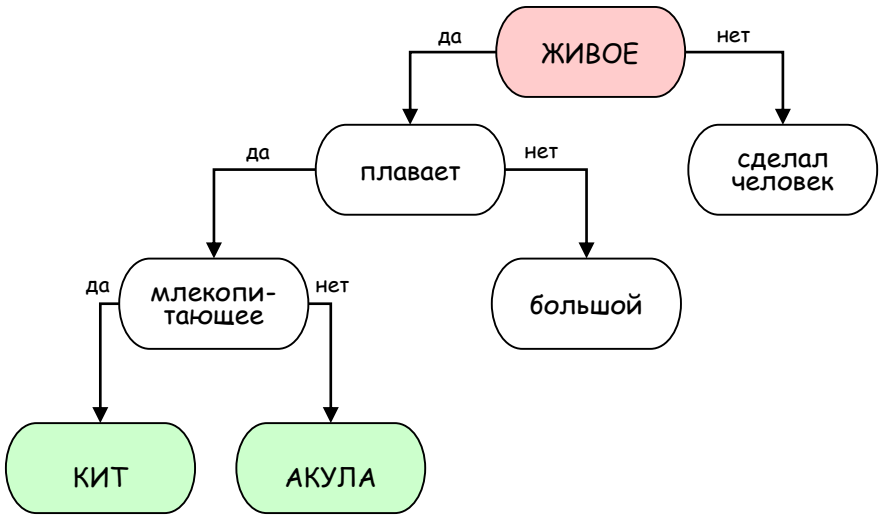
Наименее технологичным является этап создания базы знаний ввиду сложности формализации знаний. Знания являются слабо структурированной или вовсе неформализованной информацией. Для того чтобы представить знания в продукционной форме, знания нужно сначала хотя бы частично формализовать. На этом этапе часто приходится идти на компромисс, в чём-то упрощая и снижая достоверность модели предметной области. В нашем случае, для простоты рассмотрения, будем использовать модель знаний с бинарным деревом решений. Напомню, что дерево решений представляет собой структуру, состоящую из корневого узла, иерархии ветвей с узлами и листьями. Корневой узел один – в нём первый вопрос о предметной области – с него начинается поиск решения. Если дерево бинарное, то в каждом узле может быть только развилка на два пути (две ветви или ветвь с листком или два листка). Ветвь подразумевает переход к следующему вопросу (узлу), а листок понимается как конечный узел – ответ. Если из одной ветви может быть более чем два выхода, то такое дерево называют мультидеревом.

Дерево решений может быть представлено в виде структурированного файла, например, текстового с разметкой. Далее приведён пример такого файла для бинарного дерева решений (в каждом узле

только два ответа: ДА/НЕТ) о некоторых объектах живой и неживой природы:

```
живое
    плавает
        млекопитающее
            _кит
            _акула
        большой
            косолапый
                _медведь
                длинная шея
                    _жираф
                    _слон
            _крыса
    сделал человек
        есть колеса
            два колеса
                есть двигатель
                    _мотоцикл
                    _велосипед
                _автомобиль
            работает, если включить в розетку
                _телевизор
                _кубик Рубика
        _камень
```

Данная модель знаний интерпретирует бинарное дерево, в корне которого располагается первый и ключевой вопрос, делящий дерево на два поддерева, в свою очередь являющиеся двоичными деревьями поиска:



Модель знаний можно хранить в любом формате (XML, Excel, csv, json, txt), но в нашем случае, чтобы обойтись малыми средствами и иметь возможность простого редактирования базы данных ограничимся пока обычным текстовым файлом. Чтобы заменить графические обозначения на текстовые аналоги оговорим некоторые условности:

– первая строка текстового файла содержит корень дерева и находится на нулевом уровне погружения в дерево;

– дерево бинарное, поэтому у вершины не может быть более двух ветвей;

– каждая ветвь оканчивается вершиной одного из двух видов: узел или листок;

– у узла могут быть ветви, у листка – нет (это и есть искомое решение);

– каждая новая ветвь добавляет уровень погружения, для которого в текстовом файле применяется специальный символ, например, табуляция (каждый новый символ табуляции смещает строку вправо, обозначая погружение на следующий уровень);

– для обозначения листка (конечного узла перебора) в текстовом файле применяется специальный символ, например, символ подчёркивания '_'.

Итак, архитектура программы Экспертная система будет представлена тремя файлами: База знаний, Машина вывода, Интерфейс пользователя. Базу знаний мы сформируем в виде структурированного текстового файла. Машину вывода мы оформим в виде модуля с объектом, который содержит все необходимые поля и методы. Интерфейс пользователя сделаем в виде обычного консольного приложения, где пользователь сможет вводить с клавиатуры ответы на диагностические вопросы программы.

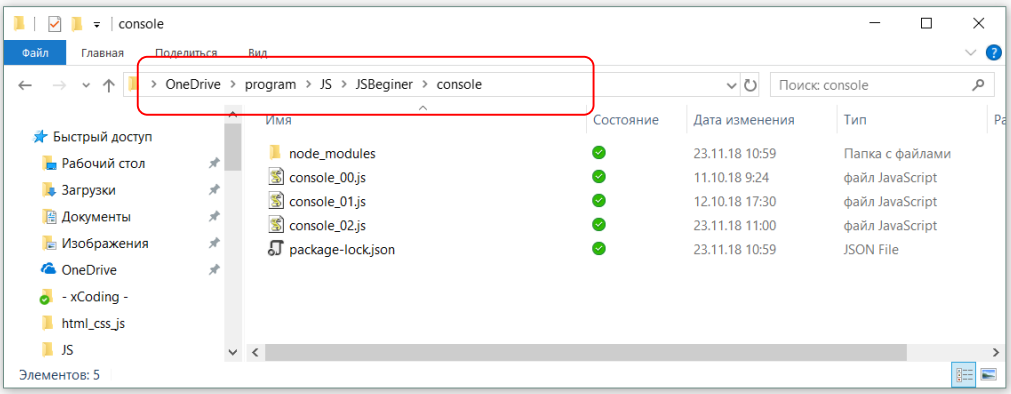
Есть некоторая сложность в организации диалога с пользователем в консоли – начнём с разрешения этой проблемы.

Работа в консоли на Node.js

Как сделать так, чтобы можно было вводить данные с клавиатуры в консоли? Дело в том, что, если мы пишем программу для браузера, то там мы можем ввести данные в программу либо через диалог prompt, либо через объект input – и эти возможности присутствуют в языке JS по умолчанию. Если же мы программируем под Node.js, а это, прежде всего, серверная платформа для исполнения, то там, по умолчанию, нет возможности брать от пользователя данные из консоли, так как не для этого она создавалась. Но мы имеем возможность подключить дополнительный модуль с необходимыми функциями.

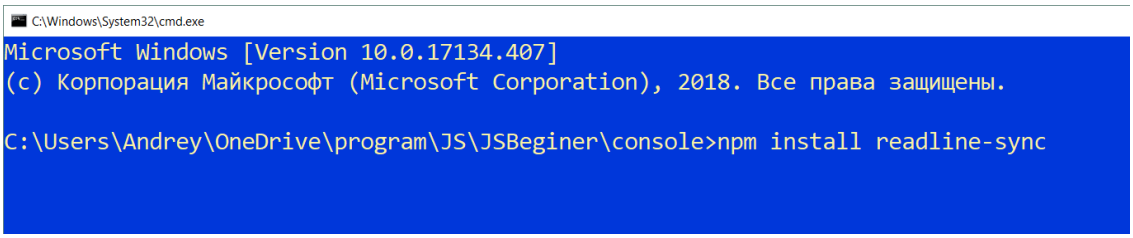
Первое, что нужно сделать: установить дополнительный модуль `'readline-sync'`, Его можно установить, как через терминал Visual Studio Code для текущей папки, так и через терминал Windows

Для запуска терминала Windows откройте проводник, перейдите в папку проекта, поставьте курсор в адресную строку, всё там сотрите и наберите команду cmd, нажмите Enter.

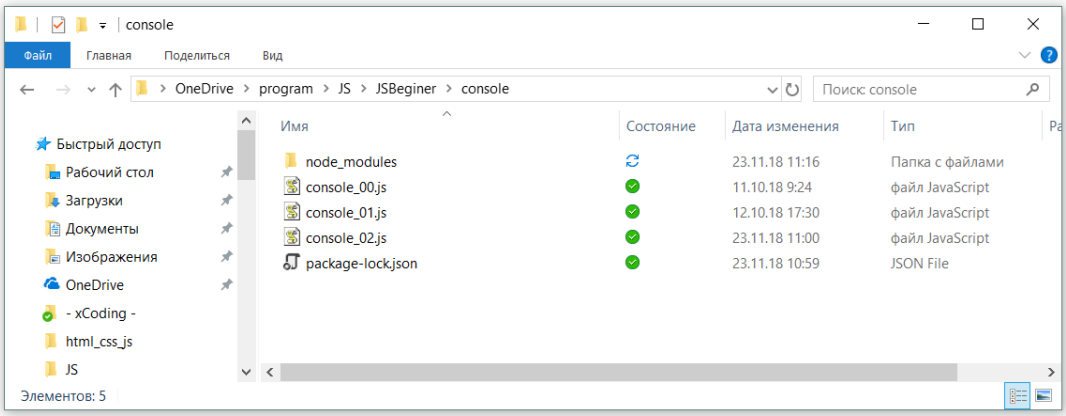


В каком бы терминале вы не работали, чтобы установить в папку с проектом новый пакет нужно в консоли набрать и запустить на исполнение следующую инструкцию:

```
npm install readline-sync
```



Первое, что тут указано: npm (Node.js Package Manager) — менеджер пакетов, входящий в состав Node.js - это просто программа, которая хранится в папке с установленным Node.js и отвечает за установку дополнительных модулей. Мы ей отдаём распоряжение установить определённый пакет. Сама установка занимает несколько секунд. После установки пакета (обязательно это делать именно в текущей папке проекта) у вас на жёстком диске, там, где ваша редактируемая программа, появится новая директория с установленным модулем (node_modules).



Загляните в неё, там вы найдёте директорию под модуль readline-sync. Если, в дальнейшем, вы в свой проект будете добавлять ещё сторонние модули, то все они будут аккумулироваться в папке node_modules. Внимание, при переносе проекта на другой компьютер с собой нужно брать также и папку с установленными дополнительными модулями (node_modules) - все они указаны в файле package-lock.json, который лежит также в текущей папке и не является обязательным для переноса.

После установки модуля readline-sync можно использовать новые возможности в нашей программе. Приведу пример диалога с пользователем в консоли:

```
// установить пакет: npm install readline-sync -S
readln = require('readline-sync'); // подключаем модуль
console.log("- введите целое число");
answer = Number(readln.question()); // читаем с консоли
result = answer%2==0? "чётное": "нечётное";
console.log("это число " + result);
```

База знаний

Создайте входной файл "db.txt" с базой данных дерева решений (первая строка в файле - служебная):

```
Бинарное дерево – поиск объекта
живое
     плавает
          _слон
          _кит
     сделал человек
          на колесах
               4 колеса
                    _автомобиль
                    _мотоцикл
               _телевизор
          _камень
```

Программу – Экспертную систему, можно написать в обычном стиле (шаблон #1 – ознакомьтесь с ним, но делать экспертную систему рекомендую в формате шаблона #2), когда алгоритм и данные находятся в одном файле и размазаны по коду программы, а, лишь некоторые функции, вынесены в отдельный модуль:

```
// это модуль utils.js с функцией чтения Базы Знаний
function getLines(nameFile) {
    let fs = require('fs');
    let text = fs.readFileSync(nameFile, 'utf8');
    let lines = text.split('\r\n'); // конец строки
    return lines;
}
module.exports.getLines = getLines;
  // после module.exports.__ можно записать иное имя,
  // которое будут использовать снаружи этого модуля

// это текст программы – Экспертная система
var utils = require("./utils.js"); // подключаем модуль
var lines = utils.getLines('db.txt'); // получаем строки
var readLine = require('readline-sync');
  // создаём объект для чтения с консоли
  // нужно установить пакет: npm install readline-sync –S
  // модуль 'readline-sync' позволяет читать с консоли ввод

console.clear(); // очистка консоли
var posLine = 0; // номер строки в Базе Данных
var posTab = 0; // глубина погружения по дереву решений
var tab = '\t'; // символ уровня погружения
var stop = ' '; // символ окончания перебора

    // далее цикл работы экспертной системы
do {
    console.log("-это " +
        lines[posLine].substring(posTab) + "? (Yes=1,No=2)");
    let answer = parseInt(readLine.question());
    posLine++; posTab++;
    if (answer>1) {
        do {
            posLine++;
```

```
        } while (lines[posLine].lastIndexOf(tab)!=posTab-1);
    }
} while (lines[posLine].substring(posTab)[0]!=stop);
    // вывод результата поиска
console.log(lines[posLine].substring(posTab+1));
```

Целесообразнее программу разделить на интерфейсную и аналитическую части (шаблон #2). Аналитическую часть вынесем в отдельный модуль, который можно будет подключать и в другие программы. В модуле будет располагаться одна из важнейших структурных частей Экспертной системы – машина вывода. Разместим все методы машины вывода внутри одного объекта. В качестве минимального необходимого содержимого для поддержания работы экспертной системы с бинарным деревом определим:

- **поля, для хранения:**

  массива строк из текстового файла (это вся БД),

  текущей позиции по строкам в БД,

  позиции по уровню погружения в дерево,

  символа отступа по уровню,

  символа обозначения листка дерева.

- **методы:**

  читать строки файла с деревом решений,

  выдать следующий вопрос для вывода,

  выдать итоговый ответ,

  найти следующий узел, в ответ на выбор пользователя (да/нет),

  проверить, не достигнут ли конец перебора.

Это текст интерфейсной части программы:

```
read = require("readline-sync"); // объект чтения с консоли
module = require("./utils.js"); // подключаем наш модуль
es = new module.esbObj(); // создаём объект ЭС
es.getLines("db.txt"); // получаем строки из файла БД

do { // цикл поиска -> вопрос, ответ, новая строка
    console.log("-", es.getQuestion(), "? (Yes=1,No=0)");
    answer = parseInt(read.question());
    es.setNewPos(answer);
} while (es.checkNotEnd()); // пока не конец поиска

console.log("это", es.getResult()); // вывести результат
```

Далее текст модуля с конструктором объекта:

```
function esbObj(_tab, _end) { // конструктор объекта
    // поля объекта
    this.tab = _tab || '\t'; // назначим символ уровня погружения
    this.end = _end || '_'; // назначим символ окончания перебора
    this.posLine = 0; // начальная позиция по строкам БД
    this.posTab = 0; // начальная позиция погружения в дерево
    this.lines = []; // строки файла
    // методы объекта
    this.getLines = function (fileName) { // читаем строки дерева
        fs = require('fs');
        this.lines = fs
            .readFileSync(fileName, 'utf8')
            .split('\r\n') // из файла в массив строк
            .slice(1); // пропускаем строку заголовка
    };
    this.getQuestion = function () {
        return this.lines[this.posLine].substring(this.posTab);
    };
    this.getResult = function () {
        return
          this.lines[this.posLine].substring(this.posTab+1);
    };
    this.setNewPos = function(answer) {
        this.posTab++; // погрузиться на следующий уровень
        this.posLine++; // сменить линию, если ответ ДА
        if (answer==0) {
            do { // если ответ НЕТ
                this.posLine++;
            } while (this.lines[this.posLine]
                    .lastIndexOf(this.tab)!=this.posTab-1);
        }
    };
    this.checkNotEnd = function() {
        return this.lines[this.posLine]
               .substring(this.posTab)[0] != this.end;
    };
}

module.exports.esbObj = esbObj; // сделаем объект видимым снаружи
```

Приведённые примеры программ работают с бинарным деревом решений (ответы: Да/Нет), но их можно преобразовать для моделирования работы «мультидерева» решений, а также для разработки Экспертной системы способной пополнять дерево решений в ходе диалога с пользователем. Например, если в ходе диалога по приведённому ранее бинарному дереву («Живое» - нет, «Сделал человек» - нет) программа выходит на решение «Камень», но пользователь указывает программе, что это решение неверно, то можно дополнить интерфейс Экспертной системы модулем дополнения нового ответа и вопроса, отделяющего данный новый ответ с предыдущим. В частности, пользователю предоставляется право ввести новое слово, например, «планета» и определяющий вопрос – «Объект большой?». Соответственно, после такого расширения бинарного дерева Экспертная система после серии вопросов («Живое» - нет, «Сделал человек» - нет), прежде, чем дать заключительный ответ, задаст ещё один вопрос: «Объект большой» с возможными вариантами исходов: планета/камень.

Возможные доработки:

– добавить функции для поддержки мультидерева;

– добавить в интерфейс пользователя функционал по добавлению ветвей и узлов в дерево поиска – редактор Экспертной системы.

Фреймовая экспертная система

Фрейм (англ. frame — «каркас» или «рамка») — способ представления знаний в искусственном интеллекте. Первоначально термин «фрейм» ввёл Марвин Минский в 70-е годы XX века для обозначения структуры знаний для восприятия пространственных сцен. Фрейм — это модель абстрактного образа, минимально возможное описание сущности какого-либо объекта, явления, события, ситуации, процесса.

Фреймы используются в системах искусственного интеллекта (например, в экспертных системах) как одна из распространенных форм представления знаний.

Фрейм отличает наличие определённой структуры. Фрейм состоит из имени и отдельных единиц, называемых слотами. Он имеет однородную структуру:

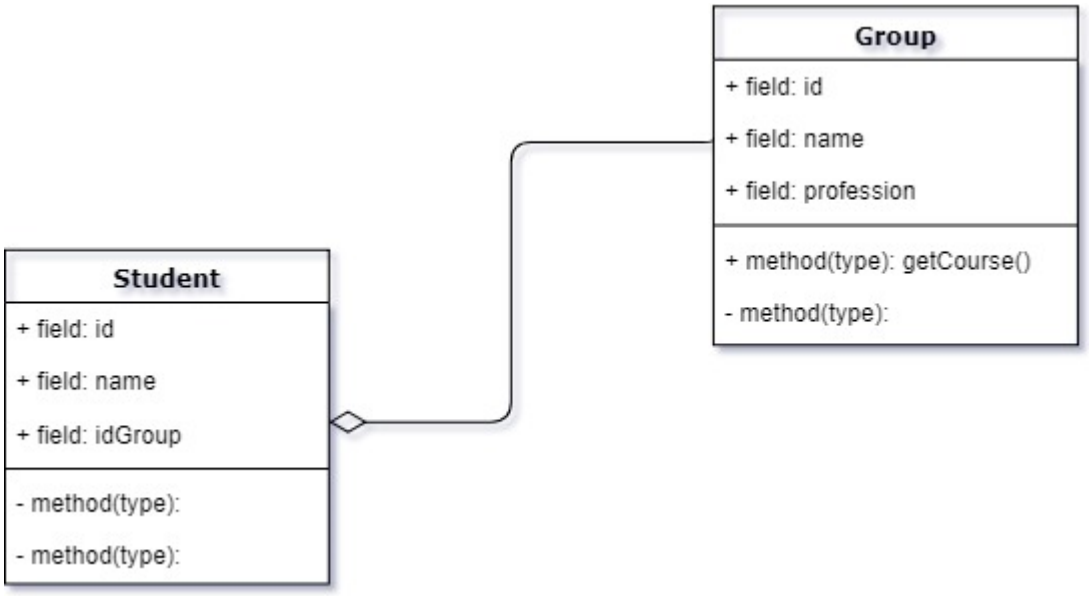
```
ИМЯ ФРЕЙМА
Имя 1-го слота: значение 1-го слота
Имя 2-го слота: значение 2-го слота
……
Имя N-го слота: значение N-го слота
```

В качестве значения слота может выступать ссылка на другой фрейм. Таким образом фреймы объединяются в сеть. Свойства фреймов наследуются сверху вниз, то есть от вышестоящих к нижестоящим через так называемые АКО-связи (от англ. A Kind Of — «разновидность»). Слот с именем АКО указывает на имя фрейма более высокого уровня иерархии.

Незаполненный фрейм называется протофреймом, а заполненный — экзофреймом. Роль протофрейма, как оболочки в экзофрейме, весьма важна. Эта оболочка позволяет осуществлять процедуру внутренней интерпретации, благодаря которой данные в памяти системы не безлики, а имеют вполне определенный, известный системе смысл.

Обратите внимание, что с точки зрения программирования протофрейм можно представить в виде класса с полями разного типа данных и методами для обработки этих данных, а экзофрейм в виде экземпляра класса – объекта с заполненными значениями полей. Совокупность однотипных фреймов можно хранить в структуре типа массив, список или стек.

Итак, данные в фреймовой модели знаний можно представить в табличном виде, а совокупность связных по некоторым полям таблиц представляет собой наглядную форму хранения знаний, как в реляционной базе данных. Пусть у нас есть некоторое множество студентов, учащихся в разных группах:



Разработаем программу, основанную на фреймовой модели представления знаний. Табличные данные разместим в текстовых файлах (поля разделены символом табуляции):

frameStudents.txt			**frameGroups.txt**		
id	nameStud	idGroup	id	nameGroup	prof
1	Иванов	1	1	ИСб-3	1
2	Петров	1	2	ИСб-2	1
3	Шамова	3	3	ПИб-1	2
4	Юдина	5	4	ИСбу-1	1
5	Дымов	4	5	ПИнб-1	3
6	Миков	4	6	ИСб-1	1
7	Кличко	4			
8	Берия	6			

Программа будет считывать данные из таблиц построчно разбивая их на объекты, а сами объекты добавлять в массивы объектов. Соответственно, массив объектов является интерпретацией таблицы.

Первая часть программы осуществляет чтение текстовых файлов и формирование массивов с объектами, а вторая часть - содержит примеры запросов к базе данных. Во фреймовой экспертной системе, в связи со стабильностью структуры данных, запросы могут быть однотипными, а сам код достаточно лаконичным и простым.

Шаблон для запроса может состоять всего из двух методов: выбрать данные по условию и вывести их на экран, поэтому, с точки зрения функционального программирования, запрос можно составить из имени массива с данными и методов filter и map:

```
arrName
    .filter(<условие отбора>)
    .map(obj=>console.log(obj));
```

Для наглядности в предлагаемом варианте программы условия отбора будем выносить в отдельные функции.

Так как первая строка в файлах с данными служебная, то её, при считывании и обработке, будем пропускать методом slice.

Первая часть программы - чтение данных:

```
fs = require("fs"); // подключаем модуль чтения работы с файлами

// формируем массив объектов «Студенты»
text = fs.readFileSync("frameStudents.txt", "utf-8");
lines = text.split("\r\n").slice(1);
arrStudents = []; // массив объектов «Студенты»
          // конструктор объекта «Студент»
function Student(_id, _name, _idGroup) {
    this.id = _id,
    this.name = _name,
    this.idGroup = _idGroup
}
lines.map( function (line) {
    temp = line.split('\t');
    arrStudents.push(new Student(temp[0],temp[1],temp[2]));
});
console.log(arrStudents);
```

```
// формируем массив объектов «Группы»
text = fs.readFileSync("frameGroups.txt", "utf-8");
lines = text.split("\r\n").slice(1);
arrGroups = [];// массив объектов «Группы»
          // конструктор объекта «Группа»
function Group(_id, _name, _idProf) {
    this.id = _id,
    this.name = _name,
    this.idProf = _idProf,
    this.getCourse = function () { // добавляем метод
        pos = this.name.indexOf('-') + 1;
        return +this.name[pos];
    }
}
lines.map( function (line) {
    temp = line.split('\t');
    arrGroups.push(new Group(temp[0],temp[1],temp[2]));
});
console.log(arrGroups);
```

Вторая часть программы - запросы к базе данных:

```
// выбрать студентов из группы ИСб-3
find = "ИСб-3";
function checkGroup(st, find) { // найти id группы по её названию
    return st.idGroup==arrGroups.filter(gr=>gr.name==find)[0].id;
}
arrStudents
    .filter(stud=>checkGroup(stud, find))
    .map(stud=>console.log(stud.name));

// выбрать первокурсников
course = 1;
function checkCourse(st, cs) {
    selectGroup = arrGroups.filter(gr=>gr.id==st.idGroup)[0];
    return selectGroup.getCourse()==cs;
}
arrStudents
    .filter(stud=>checkCourse(stud, course))
    .map(stud=>console.log(stud.name));
```

Обратите внимание на то, что сами запросы к базе данных действительно выглядят очень однотипно и легки для восприятия и написания:

```
arrStudents
    .filter(stud=>checkXXX())
    .map(stud=>console.log(stud));
```

Таким образом, основной труд и мастерство программиста в реализации запросов в фреймовой экспертной системе заключается в формировании условий отбора: `checkXXX()`. Лучшим решением для них является функциональный подход… Следует отметить, что это дополнительное условие отбора является вынужденной мерой, обусловленной ограничениями прототипного наследования в JS (подробнее смотрите в подразделе Объекты и наследование).

Итак, экспертные системы, основанные на фреймах, являются объектно-ориентированными. Каждый фрейм соответствует некоторому типу объектов предметной области, поэтому фрейм может быть представлен в виде списка свойств, а, если использовать средства базы данных, то в виде записи. Совокупность фреймов, моделирующая какую-либо предметную область, представляет собой иерархическую структуру, в которую фреймы собираются с помощью связей. На верхнем уровне иерархии находится фрейм, содержащий наиболее общую информацию, истинную для всех остальных фреймов. Фреймы обладают способностью наследовать значения характеристик своих родителей, находящихся на более высоком уровне иерархии. Эти значения могут передаваться по умолчанию фреймам, находящимся ниже них в иерархии.

К сожалению, в JS есть возможность организовать наследование, но только прототипное и не множественное. Для приведённого выше примера это возможно реализовать, связав студента и группу, но для более сложной и разнообразной модели знаний средства прототипного наследования языка JS не подходят. Для полноценной реализации фреймовой модели знаний более подойдёт такой язык как C#.

Тема 5. Регулярные выражения

Под регулярными выражениями понимают шаблоны для поиска определённых комбинаций символов. Шаблоны строятся по формальным правилам и позволяют осуществить поиск любой сложности в строке.

В JS для строкового объекта определены методы, существенно облегчающие поиск в строках за счёт использования регулярных выражений:

- split,
- replace,
- match,
- search.

Сначала мы рассмотрим порядок применения заданных методов по ходу уточняя некоторые особенности оформления регулярных выражений, но без уточнения деталей и всех возможностей их использования. После рассмотрения простейших примеров будут приведены справочные данные об особенностях построения шаблонов регулярных выражений и правилах их использования.

Метод split принимает строку, но возвращает массив – он делит строку по заданному символу или последовательности символов на массив подстрок. Однако не всегда исходная строка имеет чётко установленные правила деления или эти правила зависят от операционной системы или пользователь оформил строку с нарушениями эти правил. Рассмотрим примеры.

Пример 1.

Дан текстовый файл input.txt:

1

2

3

4

5

Нужно написать программу чтения всех строк этого файла и вывод полученных значений через пробел в одну строку:

```
let f = require('fs');

let delim0 = "\r\n";
let delim1 = /\r\n|\n/;
let delim2 = /\r?\n/;
let delim3 = /\r{0,1}\n/;

let lines = f
    .readFileSync('input.txt', 'utf-8')
    .split(delim2);

console.log(
    lines
        .map(line=>String(line))
        .join(' ')
);
```

Регулярные выражения, в отличие от строк, формируются не кавычками, а прямым слешем (delim1, delim2, delim3).

Если для метода split мы выберем делителем delim0, то эта программа будет корректно работать только в операционной системе Windows (для Linux и MacOS следует использовать только `"\n"`).

Используя регулярные выражения, мы можем сделать программу более универсальной. Например, delim1 определяет, что сепаратором может служить пара символов `\r\n` или один символ `\n`. Вертикальная черта `|` в регулярных выражениях означает логическое ИЛИ. А в delim2 используется подход с указанием сколько раз может встречаться символ `\r?` – тут вопросительный знак указывает, что символ `\r` может встречаться 0 или 1 раз. Аналогичным образом работает и delim3, в котором в явном виде указано сколько раз может встречаться искомый символ (0 или 1 раз).

Как видите, использование регулярный выражений может упростить написание программы и сократить объём кода.

Пример 2.

Пусть дана строка, в которой через запятую хранится полезная информация, например, e-mail адреса клиентов магазина. Попробуем поработать с данной строкой и разбить её на отдельные составляющие по символу «,»:

```
str = "a@reg.reg,2000,tt@59.ru,2001,2002,2003@google.com";
div = ",";
arr = str.split(div);
console.log(arr);
```

Вывод:

```
[ 'a@reg.reg',
  '2000',
  'tt@59.ru',
  '2001',
  '2002',
  '2003@google.com' ]
```

Пока используем метод split традиционным образом, однако, если пользователь, при подготовке исходной строки, допустил нарушения шаблона, в частности, иногда вместо запятой использовал символ «;», тогда будут получены некорректные результаты:

- для такой строки:

```
str = "a@reg.reg;2000,tt@59.ru,2001,2002;2003@google.com";
```

- вывод такой:

```
[ 'a@reg.reg;2000', 'tt@59.ru', '2001', '2002;2003@google.com' ]
```

Как видите некоторые элементы массива - `'a@reg.reg;2000'` и `'2002;2003@google.com'` включают сразу по две подстроки. С помощью регулярных выражений можно внести дополнительное условие для разбиения исходной строки: делить не только по символу «,», но и по символу «;»:

```
str = "a@reg.reg;2000,tt@59.ru,2001,2002;2003@google.com";
div = /[,;]/; // это и есть регулярное выражение
arr = str.split(div);
console.log(arr);
```

Вывод:

```
[ 'a@reg.reg',
  '2000',
  'tt@59.ru',
  '2001',
```

```
  '2002',
  '2003@google.com' ]
```

Обратите внимание, что регулярное выражение `/[,;]/` похоже на обычную строку символов, но выделяется не кавычками, а слешами. Всё, что находится между слешами является шаблоном для поиска в строке. В данной программе мы указали, что ищем набор символов (обозначается прямоугольными скобками), представленный множеством символов «,» и «;». Таким образом метод split срабатывает при нахождении любого из указанных символов.

Аналогичного результата можно достичь для данной задачи при использовании **метода replace**:

```
str = "a@reg.reg;2000,tt@59.ru,2001,2002;2003@google.com";
div = ",";
rep = /;/;
arr = str
          .replace(rep,',')
          .split(div);
console.log(arr);
```

Вывод:

```
[ 'a@reg.reg',
  '2000',
  'tt@59.ru',
  '2001',
  '2002;2003@google.com' ] // тут ошибка
```

Однако, здесь мы видим, что метод замены сработал только при первом нахождении подходящего под шаблон сочетания. Чтобы указанный в регулярном выражении шаблон действовал на всю строку следует установить флаг поиска ***g*** (global), он ставится сразу после закрывающего слеша:

```
str = "a@reg.reg;2000,tt@59.ru,2001,2002;2003@google.com";
div = ",";
rep = /;/g;
arr = str
          .replace(rep,',')
          .split(div);
console.log(arr);
```

Вывод:

```
[ 'a@reg.reg',
  '2000',
  'tt@59.ru',
  '2001',
```

```
  '2002';
  '2003@google.com' ] // теперь нет ошибки
```

Хочу обратить ваше внимание, что объект регулярного выражения можно сформировать конструктором: Смените вот эту строку: `rep = /;/g;` На эту: `rep = new RegExp(';','g');` Первый аргумент конструктора это сам шаблон, а второй - это флаги поиска. Второй аргумент (флаг) необязателен.	

Но, если во входной строке между интересующими нас данными будут стоять дополнительные пробелы (один или более), то мы опять получим искажённые результаты:

```
str = "a@reg.reg;  2000,   tt@59.ru,2001,2002;2003@google.com";
div = ",";
rep = /;/g;
arr = str.replace(rep,',').split(div);
console.log(arr);
```

Вывод:

```
[ 'a@reg.reg',
  '  2000', // тут ошибка
  '   tt@59.ru', // тут ошибка
  '2001',
  '2002';
  '2003@google.com' ]
```

Есть несколько вариантов преодоления данной проблемы, но, продолжая предыдущее решение, можно предложить следующий вариант:

```
str = "a@reg.reg;  2000,   tt@59.ru,2001,2002;2003@google.com";
div = ",";
rep = /;\s*|,\s*/g;
arr = str.replace(rep,',').split(div);
console.log(arr);
```

Вывод:

```
[ 'a@reg.reg',
  '2000', // тут исправили ошибку
  'tt@59.ru', // тут исправили ошибку
  '2001',
  '2002';
  '2003@google.com' ]
```

Тут используется логическое ИЛИ (символ «|»), который предлагает выбор или этот шаблон **;\s*** или этот шаблон **,\s***. Сам шаблон обозначает наличие символа, например, «;» и следующего за ним пробела (указано «\s»), причём пробел может встречаться любое количество раз - от нуля до бесконечности (указано символом звёздочка).

Такого же эффекта можно достичь, если заменить любое количество пробелов на пустой символ (нет символа), а затем разделить строку по символу «;» или «,»:

```
str = "a@reg.reg;  2000,   tt@59.ru,2001,2002;2003@google.com";
div = /,|;/;
rep = /\s*/g;
arr = str.replace(rep,'').split(div);
console.log(arr);
```

Однако, мы так и не решили поставленную задачу – отобрать только e-mail`ы.

Давайте попробуем найти позицию вхождения e-mail`а во входную анализируемую строку, для чего будем использовать новый шаблон:

```
str = "a@reg.reg;  2000,   tt@59.ru,2001,2002;2003@google.com";
div = /,|;/;
rep = /\s*/g;
smb = "[a-z0-9_-]+";
email = new RegExp(smb + "@" + smb + "\." + smb);
console.log(str.search(email));
```

Вывод:

```
0
```

Всё верно, с самого начала строки находятся символы соответствующие нашему шаблону: любое количество (большее нуля) разрешённых символов, потом символ «@», затем снова разрешённые символы, потом точка и опять разрешённые символы. Заметим, что это очень упрощенная версия шаблона с рядом ошибок – приводится только для демонстрации возможностей минимальными средствами. Символ «+», в отличие от звёздочки, обозначает, что указанные слева от него символы могут встречаться любое количество раз - от одного раза до бесконечности. Если же мы хотим уточнить, что

символов должно быть в определённом диапазоне, то можно подкорректировать шаблон так

```
smb = "[a-z0-9_-]{2,}"; // от двух до бесконечности
```

или так:

```
smb = "[a-z0-9_-]{2,6}"; // от двух до шести
```

После внесения такого изменения из списка найденных значений будет исключена строка `a@reg.reg`, так как в ней есть позиция длиной менее двух символов.

Так как **метод search** возвращает значение большее -1, если строка найдена, то это можно использовать для отбора только удовлетворяющих условию строк – e-mail`ов:

```
str = "a@reg.reg;  2000,   tt@59.ru,2001,2002;2003@google.com";
div = /,|;/;
rep = /\s*/g;
smb = "[a-z0-9_-]{2,}";
email = new RegExp(smb + "@" + smb + "\." + smb);
arr = str.replace(rep,'').split(div);
for (elm of arr) {
    if (elm.search(email)>-1)
        console.log(elm);
}
```

Вывод:

```
tt@59.ru
2003@google.com
```

С учётом использования подходов функционального программирования можем переписать данный код так:

```
str = "a@reg.reg;  2000,   tt@59.ru,2001,2002;2003@google.com";
div = /,|;/;
rep = /\s*/g;
smb = "[a-z0-9_-]{2,}";
email = new RegExp(smb + "@" + smb + "\." + smb);
arr = str
    .replace(rep,'')
    .split(div)
    .filter(elm=>elm.search(email)>-1)
    .map(elm=>console.log(elm));
```

Вывод:

```
tt@59.ru
2003@google.com
```

Однако, всю эту конструкцию (replace+split+filter) поиска можно реализовать одним **методом match**:

```
str = "a@reg.reg;  2000,   tt@59.ru,2001,2002;2003@google.com";
smb = "[a-z0-9_-]{2,}";
```

```
email = new RegExp(smb + "@" + smb + "\." + smb, "g");
arr = str
    .match(email)
    .map(elm=>console.log(elm));
```
Вывод:
```
tt@59.ru
2003@google.com
```

Метод match возвращает массив подстрок, отобранных из исходной строки по шаблону регулярного выражения. Не забудьте добавить флаг глобального поиска, чтобы массив не ограничился только первым найденным совпадением.

Теперь, опираясь на полученные знания, решим одну практическую задачку. Предположим нам нужно провести парсинг интернет-странички и выбрать из неё только ссылки на pdf-документы. Html-документ, как правило, достаточно объёмный, поэтому я здесь приведу только его небольшой отрезок:
```
<li>
      <a href=https://pcoding.ru/ref/184.txt target=_blank>184.txt</a>
</li>
<li>
      <a href=https://pcoding.ru/ref/185.txt target=_blank>185.txt</a>
<li>
      <a href=https://pcoding.ru/pdf/CSharpHelp.pdf>CSharpHelp.pdf</a>
</li>
<li>
      <a href=https://pcoding.ru/pdf/ES_CSharp.pdf>ES_CSharp.pdf</a>
</li>
```

Сохраним сам html-документ в файле «input.html», откроем его в нашей программе и выведем на экран для предварительного контроля работоспособности программы:
```
fileNameIn = "input.html"; // имя входного файла
// файл загружаем в строковую переменную
text = require('fs').readFileSync(fileNameIn, 'utf8');
console.log(text);
```

Апробируйте работоспособность данной программы, если всё в порядке, то можно приступать к решению поставленной задачи -

сформируем регулярное выражение для отбора только ссылок на pdf-документы:

```
fileNameIn = "input.html";
text = require('fs').readFileSync(fileNameIn, 'utf8');
pdf = /https.+?\.pdf/g;

arr = text
    .match(pdf)
    .map(elm=>console.log(elm));
```

Вывод:

```
https://pcoding.ru/pdf/CSharpHelp.pdf
https://pcoding.ru/pdf/ES_CSharp.pdf
```

В шаблоне указано, что нужна строка, которая начинается с https, далее идёт любой символ (обозначено точкой) в количестве более нуля (обозначено плюсом), далее стоит вопросительный знак – это квантификатор, который включает режим «ленивого поиска», обозначающий, что нужно искать до ближайшей встречи следующего за квантификатором требования в шаблоне. Уточню, что по умолчанию в регулярных выражениях включён режим «жадного поиска», который выдал бы нам излишне длинные строки:

```
https://pcoding.ru/pdf/CSharpHelp.pdf>CSharpHelp.pdf
https://pcoding.ru/pdf/ES_CSharp.pdf>ES_CSharp.pdf
```

а нужны такие:

```
https://pcoding.ru/pdf/CSharpHelp.pdf
https://pcoding.ru/pdf/ES_CSharp.pdf
```

При «жадном поиске» в решение включается подстрока максимально длинная, которая возможна до наступления следующей подходящей по шаблону подстроки.

Следующее за квантификатором «?» требование в нашем шаблоне – это сочетание символов точки и pdf: \.pdf. Именно «\.» в отличие от просто точки обозначает наличие символа точки в строке. Просто точка – обозначает наличие любого символа. Это было завершающее требование шаблона, так как далее стоит закрывающий слеш. И не забывайте ставить флаг глобального поиска g.

Аналогичного результата можно достичь, если в паттерне регулярного выражения применить способ поиска с упреждением. Напомню, что для группировки символов в один блок используются обычные круглые скобки. Если нам нужно выделить подстроку, но не включать некоторые символы, то можно использовать такой шаблон: (?=>):

```
pdf = /https.+pdf(?=>)/g;
```

Используйте данное регулярное выражение в предыдущей программе. Вы должны получить эквивалентный результат. В шаблоне указано, что нужно из исходной строки выбрать такие подстроки, в которых сначала идут символы «https», потом любое количество символов, потом символы «pdf» и затем символ «>», но завершающее требование мы взяли в скобки и исключили его из итогового результата (?=>).

Если, по каким-то причинам, в оформлении имени файла символы расширения написаны заглавными буквами (не pdf, а PDF), то можно преодолеть данную проблему добавлением флага *i* (ignore) игнорирования регистра символов:

```
pdf = /https.+pdf(?=>)/gi;
```

Внесите соответствующие изменения во входной файл и в текст программы и апробируйте её работоспособность.

Как вы уже поняли, регулярные выражения, как инструмент имеет довольно широкие возможности. Описывать в рамках данного учебного пособия все особенности и правила формирования и использования паттернов регулярных выражений не имеет смысла. Рекомендую краткое перечисление всех возможностей посмотреть в MDN web docs:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions

В данной таблице приведён выборочный справочный материал по регулярным выражениям:

^	Соответствует началу ввода. Если установлен флаг многострочности, также производит сопоставление непосредственно после переноса строки. Например, /^A/ не соответствует 'A' в "an A", но соответствует 'A' в "An E". Этот символ имеет другое значение при появлении в начале шаблона набора символов. Например, /[^a-z\s]/ соответствует 'I' в "I have 3 sisters".
$	Соответствует концу ввода. Если установлен флаг многострочности, также сопоставляется содержимому до переноса строки. Например, /t$/ не соответствует 't' в строке "eater", но соответствует строке "eat".
*****	Соответствует предыдущему символу повторенному 0 или более раз. Эквивалентно {0,}. Например, /bo*/ соответствует 'boooo' в "A ghost booooed" и 'b' в "A bird warbled", но не в "A goat grunted".
+	Соответствует предыдущему символу повторенному 1 или более раз. Эквивалентно {1,}. Например, /a+/ соответствует 'a' в "candy" и всем символам 'a' в "caaaaaaandy".
?	Соответствует предыдущему символу повторенному 0 или 1 раз. Эквивалентно {0,1}.
.	Десятичная точка соответствует любому символу кроме переноса строки. Например, /.n/ соответствует 'an' и 'on' в "nay, an apple is on the tree", но не 'nay'.
x|y	Соответствует либо 'x' либо 'y'. Например, /green|red/ соответствует 'green' в "green apple" и 'red' в "red apple."
\s	Соответствует одиночному символу пустого пространства, включая пробел, табуляция, прогон страницы, перевод строки. Перенос строки занимает два символа \r\n – этому сочетанию соответствует шаблон \s\s.

Лабораторная работа 1.

Обработка массивов в функциональном стиле.

Задания для самостоятельного исполнения.

Задания выполнить в функциональном стиле.

Базовые задания

1. Сформировать из исходной строки "Есть ли жизнь на Марсе" массив слов arrWords.
2. Сформировать из arrWords массив длин элементов arrLength.
3. Сформировать из arrWords массив коротких слов (короче 3-х букв).
4. Отсортировать arrWords по убыванию по длине строки, не используя reverse.
5. Сформировать из отсортированного массива предыдущего пункта строку.
6. Найти самое длинное слово в arrWords.
7. Напишите программу, которая на первом этапе генерирует массив из десяти целых неотрицательных чисел от 0 до 100 и выводит их на экран. На втором этапе - ищет и выводит на экран разницу между суммой чётных значений элементов массива и нечётных.

   Уточнения:

   - пункты 2-6 не сохранять в дополнительную переменную, а сразу печатать;
   - для п.1 использовать split, для п.5 использовать join;
   - для п.4 и п.6 использовать настраиваемый метод sort.

Бонусные задания

7. Написать функцию перевода из двоичной в десятичную систему счисления.
8. Написать функцию проверки фразы на палиндром.

   Примеры фраз, которые должны быть признаны палиндромами:

   «Аргентина манит негра!»

   «А роза упала на лапу Азора.»

Лабораторная работа 2.

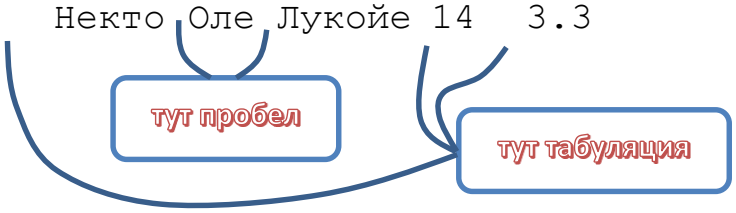
Работа с файловой системой.

Задания для самостоятельного исполнения.

Задания выполнить в функциональном стиле.

Пусть дан текстовый файл:

```
№   Фамилия Имя Отчество Возраст Средний балл
1   Абрамович Петя Иванович  24  4.5
2   Кокорин Петро Петрович   18  3.8
3   Саакашвили Адольф Семёнович  33  4.1
4   Папов Коля Олич  19  4.4
5   Некто Оле Лукойе 14  3.3
```



Первая строка служебная и не участвует в обработке информации, но присутствует в файле и нужна только для наглядности представления информации. Во время чтения данных из файла учтите это и первую строку не включайте в массив для обработки данных. Разделителем для столбцов Номер, «Фамилия Имя Отчество», Возраст, «Средний балл» является символ **табуляции**, а Фамилия Имя Отчество разделены **пробелом**.

Во время выполнения заданий данные из файла сначала считывайте в **массив объектов**, потом уже ведите обработку этого массива в разных заданиях по-разному.

Самостоятельно создайте файл с указанными параметрами структуры и выполните для него в функциональном стиле следующие **задания**:

1. Вывести в столбик на экран Средний балл и, через табуляцию, Фамилии студентов отсортированные по среднему баллу по возрастанию:

```
3.3     Некто
3.8     Кокорин
4.1     Саакашвили
4.4     Папов
4.5     Абрамович
```

2. Вывести на экран в столбик Возраст и, через табуляцию, Фамилии совершеннолетних студентов:

```
24      Абрамович
18      Кокорин
33      Саакашвили
19      Папов
```

3. Вывести на экран в столбик Средний балл и, через пробел, Фамилии студентов, у которых Средний балл выше, чем средний балл в группе:

```
4.5 Абрамович
4.1 Саакашвили
4.4 Папов
```

4. Вывести на экран в столбик Возраст и, через пробел, Имя Отчество студентов, у которых Средний балл выше, чем средний балл в группе, отсортировав их по возрасту:

```
19 Коля Олич
24 Петя Иванович
33 Адольф Семёнович
```

5. Распарсить строки входного файла и вывести в файл «output.txt» информацию в виде:

```
Абрамович П.И.
Кокорин П.П.
Саакашвили А.С.
Папов К.О.
Некто О.Л.
```

Лабораторная работа 3.

Разработка продукционной Экспертной системы.

Задания для самостоятельного исполнения.

1. Разработайте бинарное дерево решений по интересующей вас предметной области с объёмом листочков не менее 16. Базу знаний разместите в структурированный текстовый файл (где отступы слева обозначают глубину погружения в дерево решений). Консольную программу «Экспертная система» выполните с использованием объектно-ориентированного подхода (**шаблон #2**) с размещением конструктора объекта в отдельном модуле.

2. Разработайте мультидерево решений по интересующей вас предметной области с объёмом листочков не менее 16. Базу знаний разместите в структурированный текстовый файл (где отступы слева обозначают глубину погружения в дерево решений). Консольную программу «Экспертная система» выполните с использованием объектно-ориентированного подхода (**шаблон #2**) с размещением конструктора объекта в отдельном модуле.

Бонусное задание

3. Разработайте интернет-страничку для работы с Экспертной системой (html+css+js). В js файлах разместите «Машину вывода» (это конструктор объекта с методами обработки запросов в Экспертной системе) и «Базу знаний» (это модуль js с хранимым массивом json-строк).

Внимание: при достижении листочка дерева программа должна останавливать диалог и не должна уходить дальше по вопросам и ответам.

Лабораторная работа 4.

Структуры данных для Экспертной системы.

Задания для самостоятельного исполнения.

1. Разработайте конвертер для перевода «Базы знаний» из формата структурированного текстового файла в массив объектов с ссылками с сохранением массива объектов в текстовый файл с json-строками. Внесите изменение в ранее разработанную консольную Экспертную систему, так чтобы База знаний считывалась из файла с json-строками.

2. Разработайте конвертер для перевода «Базы знаний» из формата структурированного текстового файла в словарь с ссылками. Внесите изменение в ранее разработанную консольную Экспертную систему, так чтобы машина вывода экспертной системы работала с этим словарём.

Бонусное задание

3. Дополните интерфейс консольной программы «Экспертная система» диалогом с пользователем для подключения нового объекта («листочек» и узел с вопросом) в дерево решений (любое: бинарное дерево или мультидерево – по выбору).

Например, изначально в дереве решений

живое

слон

камень

есть только вопрос: живое? и есть ответы (ДА/НЕТ) – это Слон и Камень. Пользователь на вопрос Живое? Ответил ДА, на что Экспертная система пишет Слон, и предлагает подтвердить или опровергнуть Yes/No. Если пользователь подтверждает, то работа ЭС заканчивается, если же нет, то ЭС предлагает пользователю внести дополнение, то есть вместо слона появится новый узел с новым ветвлением, например, так:

живое

маленькое

мышь

слон

камень

То есть пользователь сам формулирует дополнительный вопрос и положительный исход на него, а отрицательный уже есть в базе знаний. Экспертная система вносит новые знания в базу знаний. Естественно, что это удобнее сделать, если вы будете хранить данные в словаре, так как достаточно просто будет поменять одну ссылку в элементе «живое» и добавить один элемент в словарь – «маленькое».

Лабораторная работа 5.

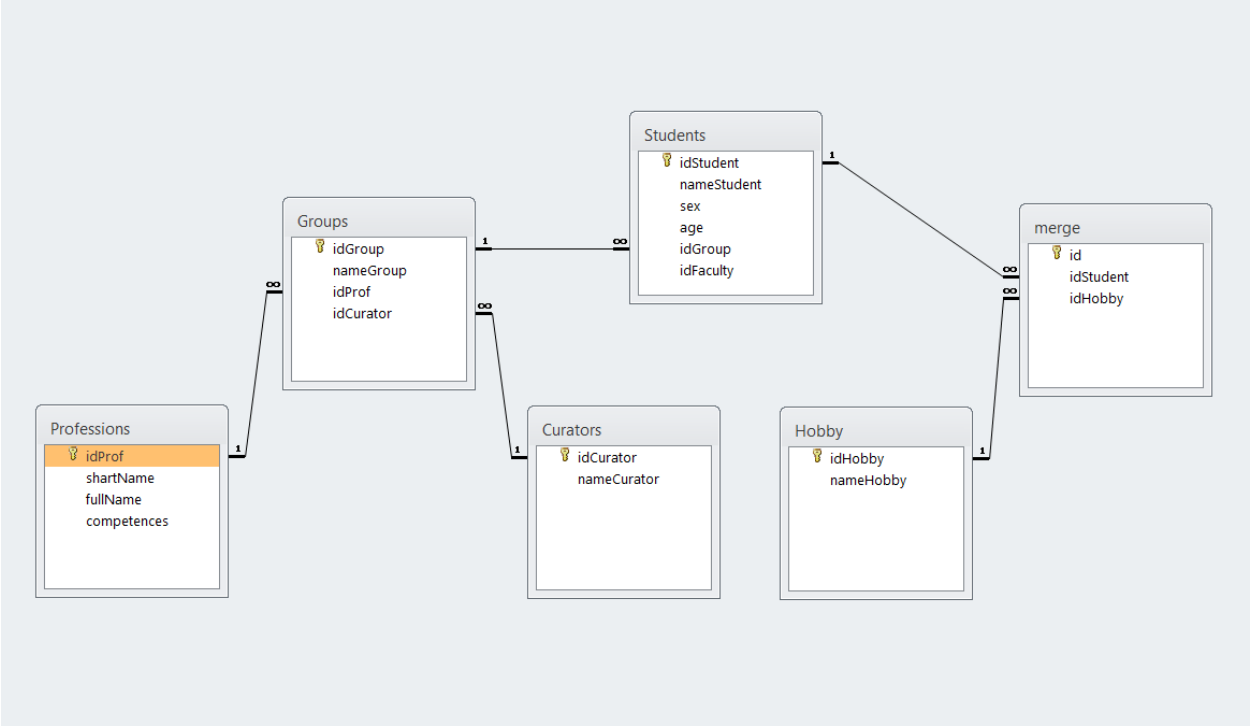
Фреймовая Экспертная система.

Задания для самостоятельного исполнения.

1. Разработайте таблицы с данными для фреймовой экспертной системы в табличном редакторе (Excel, Google-таблицы или LibreOffice-таблицы). Можно взять уже готовые данные из документа:

https://qps.ru/CVhxi

- там несколько таблиц, которые могут поддерживать реляционную базу данных с соответствующими связями:



Сохраните данные из таблиц - students, groups, curators, hobby, merge - в соответствующие csv-файлы в папку с программой для дальнейшей работы в node.js. В данном проекте работе мы будем работать без СУБД и только с указанными несвязными таблицами, а все запросы к таблицам будут осуществляться посредством функций, написанных на node.js.

2. Разработайте консольное приложение «Фреймовая экспертная система», которое работает с Базой знаний из csv-файлов.

Напишите функции, реализующие запросы SELECT, INSERT, UPDATE, DELETE:

1. Вывести на экран все данные из указанной таблицы – имя таблицы вводит пользователь.
2. Добавить нового студента/хобби/куратора.
3. Изменить возраст указанного по фамилии студента.
4. Удалить куратора из списка по фамилии.
5. Найти всех студентов, указанного по фамилии куратора.
6. Выбрать всех студентов, указанного возраста/пола/группы.
7. Найти куратора по фамилии студента.
8. Сменить куратора у группы (на вход подаётся имя группы и имя нового куратора).

После работы с запросами типа INSERT, UPDATE, DELETE следует обновлять данные в соответствующих файлах…

Лабораторная работа 6.

Регулярные выражения.

Во всех заданиях нужно применять регулярные выражения, совмещая решение с функциональным стилем программирования, например, при необходимости использовать методы map, split и т п. Не стоит использовать операторы цикла for, while.

В качестве входного файла дан фрагмент html-документа – refers.html:

```
<div>Это таблица с сайта https://pCoding.ru</div>
<table align=center width=500>
    <tr height=100>
        <td width=50%>
            <a href=https://pcoding.ru/Manual_JS.txt target=_blank>
                Справочник по JS
            </a>
        </td>
        <td bgcolor=#FF0000>
            <a href=https://answer.com/answer_PHP.DOC target=_blank>
                Введение в web-программирование
            </a>
        </td>
    </tr>
    <tr height=100>
        <td bgcolor=#ff0>
            <a href=https://pCoding.ru/CSharp.doc>ООП на C#</a>
        </td>
        <td bgcolor=red>
            <a href=https://pcoding.ru/MySQL.docx>Основы MySQL</a>
        </td>
    </tr>
</table>
```

Задания для самостоятельного исполнения:

1. Найти количество ссылок на файлы с расширением txt.
2. Найти количество ссылок на файлы с расширением doc, docx, DOC, DOCX.
3. Найти количество ссылок на файлы с сайта pCoding (это те, у которых начало `https://pcoding.ru/`).
4. Вывести все ссылки, которые открываются в новом окне (это те, у которых `target=_blank`).
5. Сменить цвет красных ячеек таблицы на зелёный. Все изменения сохранить в файл task5.html – проверьте результат.
6. Выбрать из входного файла все ссылки на учебники (это те, что с расширением doc, docx, DOC, DOCX). Вывести в файл task6.txt информацию в таком формате - два столбца разделённые символом TAB с соответствующим содержанием:

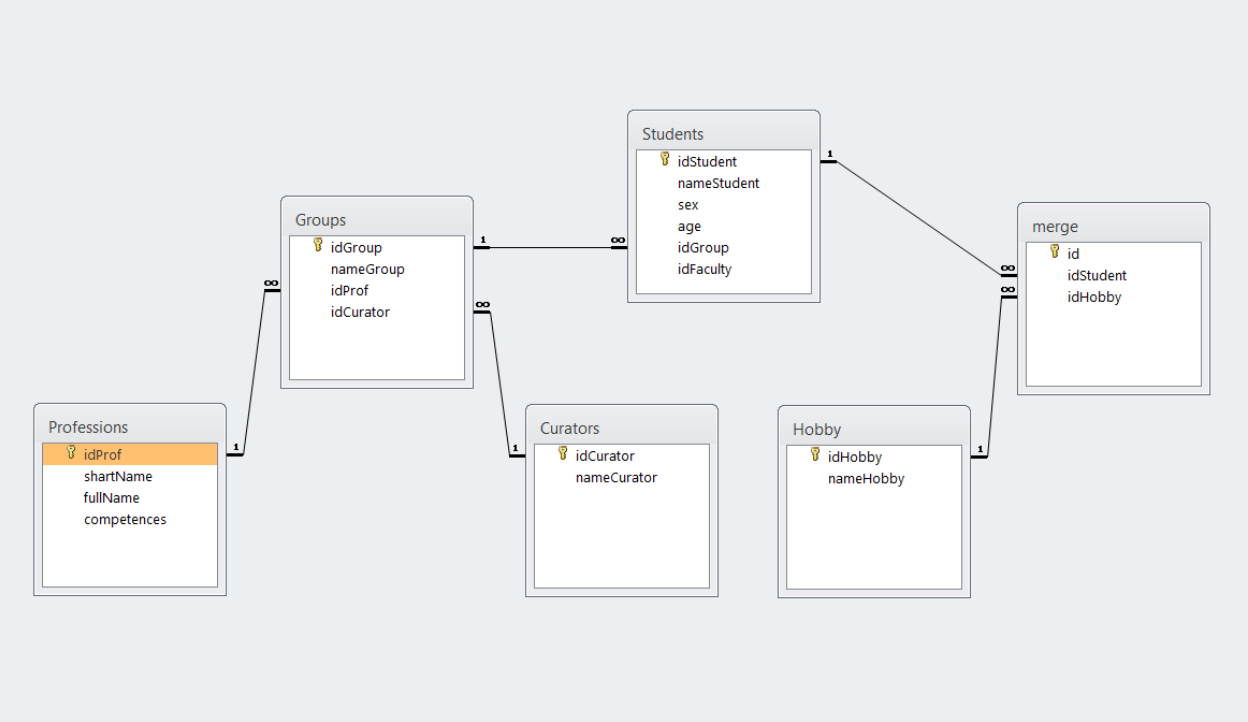
`Название_учебника  Ссылка_на_учебник`

Лабораторная работа 7.

Библиотека lodash.

В это работе во всех заданиях нужно
применять функционал из библиотеки **lodash**...

Для выполнения заданий нам потребуются таблицы – возьмите их из документа: https://qps.ru/CVhxi - там несколько таблиц, которые могут поддерживать реляционную базу данных с соответствующими связями:



Сохраните данные из таблиц - students, groups, curators, hobby, merge - в соответствующие csv-файлы в папку с программой для дальнейшей работы в node.js.

В данном проекте работе мы будем работать без СУБД и только с указанными несвязными таблицами, а все запросы к таблицам будут осуществляться посредством **функций**, написанных на **node.js**.

1. Напишите функцию подсчёта среднего возраста среди всех студентов.
2. Напишите функцию подсчёта среднего возраста среди студентов **в указанной группе**.

3. Напишите функцию выбора совершеннолетних студентов.
4. Напишите функцию выбора совершеннолетних студентов **из указанной группы**.
5. Напишите функцию, которая будет выводить на экран **для указанного куратора** в отсортированном порядке студентов курируемой группы. Направление сортировки должно быть одним из параметров функции, то есть пользователь функции должен иметь возможность выбора направления сортировки.
6. Напишите функцию, которая будет выводить на экран **для указанной группы** в отсортированном порядке два столбца:
   фамилия студента (по возрастанию), хобби (по убыванию)
7. Напишите функцию, которая будет выводить на экран в отсортированном порядке два столбца:
   название группы (по возрастанию), фамилия студента (по убыванию возраста)
8. Напишите функцию, которая будет выводить на экран **всех студентов для указанного пользователем хобби**.