BaseFunctionsJS_RU

Вступление

В этом гайде вы сможете получить основные зания по JavaScript. Причиной написания гайда была сложность и запутаность которую я заметил в другий гайдах и руководствах. Безусловно я не эксперт в JavaScript, но при этом имею какой-то опыт и думаю, что могу помочь этими знаниями другим :)

Полезные ссылки

Так как этот гайд достаточно субъективный я не уверен что захвачу все темы и не уверен что у нас не появяться вопросы после прочтения, по-этому в этом блоке прикреплю хорошие ссылки который можно использовать в таких случаях.

В первую очередь вы можете написать мне на почты и я попробую вам помочь:

boyarkin.gleb@gmail.com;

source.boar@gmail.com

Так же почты подублированы у меня в профиле.

Кроме меня вы можете воспользоваться:

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

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference

Эти источники гарантирую что являються проверенными, сам по ним учился, но как по мне многие вещи там расписаны слишком сложными словами.

Коротко про JavaScript

JavaScript (JS) — достаточно молодой и известный динамический язык программирования. Основной отраслью применения являеться в первую очередь Web разработка, а так же разработка приложений.

Синтаксис

Синтаксис JS в большинстве своём явяеться C-подобным, но со своими мелкими отличиями, с которыми мы будем сталкиваться по ходу.

Если проходиться по основным правилам синтаксиса то:

- `;` – символ окончания строки
- `{}` – сиволы для открытия и закрытия блоков кода
- `camelCase` – стиль именования переменный и функций
- `PascalCase` – стиль именования классов
- `//` – символ для комментария отдной строки
- `/* */` – символы для комментирования блока кода

Мелкая важная заметка

`consoe.log(x);` — функция для вывода какого либо `x` в консоль. Конкретнее про это будет дальше, но это нужно для понимания ближайшего участка кода

Переменные

Как я думаю вам известно, или станет известо сейчас, по факту весь язык программирования держиться на переменных и операциях над ними.

В JS переменная — это указатель на область или ячейку памяти в которой хранится какое то зничени. Факт про память пока можно просто положить в голову, к нему мы прийдем потом.

Для объявления переменных используються ключевые слова `var`, `let` и `const`.

Пример объявления:

```
var a1 = "Hello";
let a2 = 1;
const A3 = true;
```

При именовании переменных, как указано было выше, используеться стиль camelCase.

Что ж в него входит?

camelCase:

- первая буква – нижнего регистра
- каждая первая буква следуещего слова – верхнего регистра

Так же если конкретно про JS то в нем название переменной может начинать с символа `_` или буквы любого регистра, последующие символы уже могут быть такие же плюс цифры(0-9). Так же стоит уточнить что JS чувствителен к регистру, то есть в нем `a` и `A` это разные имена.

Разница между `var`, `let` и `const`

- `var` – способ объявления переменной с контекстной областью видимости, не требует значения при объявлении, по умолчанию `undefined`, переменная может быть переобъявлена. Данный способ объявления считается устаревшим, и рекомендуется вместо него использовать `let`
- `let` – способ объявления переменной с блочной областью видимости, не требует значения при объявлении, по умолчанию `undefined`, переменная не может быть переобъявлена.
- `const` – способ объявления переменной с блочной областью видимости, но при этом доступ в дальнейшем к переменной после объявления только для чтения, из этого исходит что значение при объявлении является обязательным. Так же константые переменные принято именовать буквами верхнего регистра, но это не являеться обязательным.
- `Без ключевого слова` – способ объявления переменной с глоабльной областью видимости

А теперь простыми словами

Область видимости — участок программы на котором ваша переменная, функция, класс будут видны другим объектам программы.

Глобальная область видимости — областью видимости является вся программа

Блочная область видимости – область видимости в пределах блока программы

Блок программы – функция, класс, вся программа

Контекстная область видимости — область видимости в зависимости от контекста объявления: при объявлении в функии область видимости будет эта функция, при объявлении во всей программе область видимости будет вся программа итд.

Примеры

```
var a1 = 1;
let a2 = 2;
const A3 = 3;
a4 = 4;

function f(){
    var f1 = 1;
    let f2 = 2;
    const F3 = 3;
    f4 = 4;
}

f();

function g(){
    console.log(a1); // выведет
    console.log(a2); // выведет
    console.log(A3); // выведет
    console.log(a4); // выведет
    console.log(f1); // не выведет
    console.log(f2); // не выведет
    console.log(F3); // не выведет
    console.log(f4); // выведет
}

g();

function h(){
    var a1 = 0; // сработает
    let a2 = 0; // сработает "но"
    const A3 = 0; // сработает "но"
    a4 = 0; // сработает
    f4 = 0; // сработает
}

h();

var a1 = 0; // сработает
```

```
let a2 = 0; // не сработает
const A3 = 0; // не сработает
a4 = 0; // сработает
```

А теперь разберем что происходит:

1. Переменные `a1`, `a2`, `A3`, `a4` объявлены на уровне программы, а `f1`, `f2`, `F3`, `f4` на уровне функции.
2. Что будет с `var`:
    1. Из-за контекстной типизации `a1` станет глобальной, а `f1` локальной переменными, из-за чего в функции `g` `a1` будет видна, а `f1` нет
    2. В функции `h` или в самой программе переменную `a1` можно будет объявить заново
3. Что будет с `let`:
    1. Из-за блочной типизации `a2` будет видна всей программе, а `f2` только функции в которой объявлена, из-за чего в функции `g` `a2` будет видна, а `f2` нет
    2. В функции `h` переменную `a2` можно будет объявить заново, "но" теперь `a2` для всего блока программы будет новой и старая не будет доступна
    3. На уровне блока, на котором была задана `a2`, переобъявить её уже будет нельзя
4. Что будет с `const`:
    1. Из-за блочной типизации `A3` будет видна всей программе, а `F3` только функции в которой объявлена, из-за чего в функции `g` `A3` будет видна, а `F3` нет
    2. В функции `h` переменную `A3` можно будет объявить заново, "но" теперь `A3` для всего блока программы будет новой и старая не будет доступна
    3. На уровне блока, на котором была задана `A3`, переобъявить её уже будет нельзя
    4. Изменить значение никакой из переменных этого типа будет невозможно
5. Что будет с `без ключевого слова`:
    1. Обе переменные станут глобальными из-за глобальной области видимости
    2. Переобъявление не отличаеться от переназначения так что технически её можно будет изменять как угодно в каком угодно блоке программы.

Почему `var` считается устаревшей?

Как мы увидели выше: переменные объявленые через `var` могут переобъявляться из-за чего могут возникать массы ошибок и несостыковок в коде. Что б решить эту проблему был создан `let` который в свою очередь такими проблемами не страдает.

Типы данных

В JS есть 7 примитивных типо данных.

Тип данных — типы того, какого типа значение может хранить переменная.

Примитивы (примитивные типы данных) — типы данных которые не являються объектами и не имеют методов.

Простыми словами: самые простые системные типы данных.

Примитивы в JS:

- `Boolean` – логический тип данных, хранит 2 вида значений: `true` или `false`
- `Number` – числовой тип данных, хранит числа в диапазоне приблизительно от *-2*10^307* до *2*10^307*, в случаее передачи значения меньше будет записіваться `-Infinity`, а больше – `Infinity`
- `String` – строчный тип данных, хранит строку текста почти неограниченого размера
- `null` – ключевое слово, которое означает что нулевое или "пустое" значение
- `undefined` – ключевое слово, которое означает что переменная не хранит никакого значение, по умолчанию присваивается всем переменных при объявлении без значение, за исключением `const`

Типизация

Типизация JS – это динамическая, слабая, неявная.

А теперь попробуем понять:

- Динамическая – значит что переменная не привязана к единому типу при объявлении, переменная может хранить разные типы по ходу использования программы
- Слабая – значит что при проделование действий с переменной одного типа действий другого типа не будет вызывать ошибку. *Известный пример*: `1 + "2" == "12"`, то есть что число попытались просумировать со строкой, но это не вызвало оишбку и сработало
- Неявная – значит что при объявлении не нужно "явно" указывать тип переменной. *Например как это в C++ с явной типизацией:* `string a = "a"`, когда в JS хватит `let a = "a"`, без указание типа

Статья, где эта тема раскрыта глубже: https://tproger.ru/explain/tipizacija-jazykov-programmirovanija-razbiraemsja-v-osnovah/

Что же нам дают эти зания?

Из типизации JS мы можем извлечь те факты, что:

- Нам не нужно следить за типами на уровне программы
- Нам не нужно объявлять переменную указывая её тип
- Мы можем для одной переменной присваивать значения разных типов

Расмотрим конкретно разные типы

Boolean

Это логический тип данных, который используется в местах, где нужно в переменной хранить 2 варианта значения: ДА (`true`) или НЕТ (`false`)

Объявления переменных этого типа могут выглядить так:

```
let t = true;
let f = false;
```

Используются они, в основном, в условных конструкциях, которые мы расмотрим позже

Number

Числовой тип данных. Может хранить разные числа: положительные, отрицательные, ноль, целые, не целые...

Объявления переменных этого типа могут выглядить так:

```
let n = 2;
let p = 3.14;
let m = -4;
let o = 0;
```

Так же для преобразование переменных другого типа в `Number` можно использовать функции `ParseInt(string)` , `ParseFloat(string)` или просто созданием объекта `Number()`

String

Сточный тип данных, хранит в себе текст.

Объявления переменных этого типа могут выглядить так:

```
let txt1 = "Hello";
let txt2 = 'World';
let txt3 = `!`;
```

Разницы между кавычками нету (за исключением ``, но про это не сейчас), главное что б они открывали и закрывали строку одновременно.

Так же для преобразование переменных другого типа в `String` можно использовать функции `String()` или `.toString()`

Так же для строк существует очень большое количество методов, перечень которых и инструкции к ним можно най по ссылке: https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String

Операторы

Операторы присваивания

`=` – присваивание

`+=` – присваивание со сложением

`-=` – присваивание с вычитанием

`*=` – присваивание с умножением

`/=` – присваивание с делением

`%=` – присваивание с делением с остатком

`**=` – присваивание с возведением в степень

`++` – инкремент, тоже самое что `+= 1`

`--` – декремент, тоже самое что `-= 1`

`<<=` – присваивание с левым сдвигом

`>>=` – присваивание с правым сдвигом

`>>>=` – присваивание с беззнаковым сдвигом вправо

`&=` – присваивание с побитовым `И`

`|=` – присваивание с побитовым `ИЛИ`

`^=` – присваивание с побитовым исключающим `ИЛИ`

Логические операторы

`==` – равенство

`===` – равенство с проверкой типов

`!=` – неравенство

`!==` – неравенство с проверкой типов

`>` – больше

`>=` – больше или равно

`<` – меньше

`<=` – меньше или равно

`&&` – логическое `И`, то есть оба условия в конструкции `(условие 1)&&(условие 2)` доожны быть `true`

`||` – логическое `ИЛИ`, то есть хотя б одно условие в конструкции `(условие 1)||(условие 2)` доожно быть `true`

Арифметические операторы

`+` – плюс

`-` – минус

`*` – умножение

`/` – деление

`%` – деление с остатком

`**` – возведение в степень

Побитовые операторы

`&` – побитовое `И`

`|` – побитовое `ИЛИ`

`^` – побитовое исключающее `ИЛИ`

`~` – побитовое `НЕ`

`<<` – сдвиг влево

`>>` – сдвиг вправо

`>>>` – сдвиг вправо с заполнением нулями

Условные конструкции

Переменные это конечно хорошо, но надо ж как-то с ними работать и вот условные конструкции это один из способов прописывания логики программы.

if...else

Самая известная и часто используемая конструкция.

В `if` передаёться условие, если условие истинное то выполняеться то, что в блоке `if` в ином случае то, что в блоке `else`

Синтаксис выглядит примерно так:

```
if(/*условние: или логическое выражение или переменная типа Boolean*/){
    // действие если условие == true
}
else{
    // действие если условие == false
}
```

Так же в эту конструкцию можно добавить `else if`

Пример применения:

```
if(/*условние 1: или логическое выражение или переменная типа Boolean*/){
    // действие если условие 1 == true
}
else if(/*условние 2: или логическое выражение или переменная типа Boolean*/){
    // действие если условие 1 == false, но условие 2 == true
}
...
else{
    // действие если условие 1 == false и действие 2 == false
}
```

Таких конструкций `else if` можно добавлять любое количество раз

switch..case...default

Так же условная конструкция которая укорачивает код который возможно написать через `if...else if...else`

`switch...case...default` отличается тем, что мы в switch передаём какою либо переменную, а потом конструкция сравнивает эту переменную с каждым значением `case` и если никакого совпадения с `case` не нашлось выполняеться блок `default`

Синтаксис выглядит примерно так:

```
switch(/*переменная которую сравниваем*/){
    case /*значение 1*/:
        // действие если совпало значение 1
        break;
    case /*значение 2*/:
        // действие если совпало значение 2
        break;
    ...
    default:
        // действие если никакое значение не совпало
        break;
}
```

`break` являеться обязательной командой в конце блоков, что б сравнение переменных не пошло дальше

try...case...finally

Условный блок, при котором условием выполнения будет наличие или отсутвие ошибки в блоке кода `try`

Синтаксис выглядит примерно так:

```
try{
    // блок кода который будет воспроизводить на поиск ошибки
}
catch(/*объект ошибки типа Error*/){
    // действие если ошибка
}
finally{
    // действие после выполнения блока, которое не будет воспроизводить на поиск
ошибки
}
```

Подробнее про объекты Error:
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Error

Циклы

while

Цикл при котором действие выполняеться пока условие истинно

Синтаксис выглядит примерно так:

```
while(/*условие: или логическое выражение или переменная типа Boolean*/){
    // действие если условие true
}
```

do...while

Можно сказать разновидность цикла `while`, только в `while` сначала проверяется условие а потом действие, а в `do...while` наоборот — сначала действие, а потом проверка

Синтаксис выглядит примерно так:

```
do{
    // действие если условие true
} while(/*условие: или логическое выражение или переменная типа Boolean*/)
```

for

Вид цикла с действием которое выполняеться до начала обхода, условием выполнения и действием которое выполняеться в конце каждого прохода

Синтаксис выглядит примерно так:

```
for(/*действие в начале*/; /*условие выполнения*/; /*действие в конце каждого прохода*/){
    // действие пока условие true
}
```

Ключевые слова `break` и `continue`

`break` используеться для досрочного выхода из цикла, то есть завершение работы цикла вне зависимости от условия

Пример использования:

```
for(let i = 0; i < 10; i ++){
    if(i % 3 == 0){ // если число делится на 3 без остатка
        break;
    }
    consoe.log(i);
}
```

То есть цикл выполнялся пока не наткнулся на первое число которое делится на 3

`continue` используеться для досрочного прекращения прохода цикла и перехода на следующий проход, то есть используется когда вам надо что б все после его вызова не выполнялось в это проходе, а выполнялось сразу сделующий проход

```
for(let i = 0; i < 10; i ++){
    if(i % 3 == 0){ // если число делится на 3 без остатка
        continue;
```

```
    }
    consoe.log(i);
}
```

То есть для чисел которые деляться на 3 не выполнялся `console.log()` , а сразу выполнялся следующий прогон цикла

Практические задания на данных знаниях

Для лучшего закрепления предлогаю выполнить несколько простых задач, которые проверят, насколько вы поняли этот уровень материала, все знания которые могут быть нужны для выполнения задач есть выше.

Все решения задач будут в папке "Задачи" и будут пронумерованы, но не рекомендую вам ими пользваться пока у вас самих не получится

1. Допишите в начало кода такие объявления переменных, что б остальной код заработал как написано в коментарии. Не все места для написания кода обязаны быть заполнены

   ```
   // запишите ваши объявления сюда
   function func1(){
       // или сюда
   }

   function func2(){
       try{
           A1 = 0;
       }
       catch(e){
           console.log("Нельзя изменять переменную A1");
       }
       console.log(a2);
   }

   func1();
   func2();
   ```

   (Должны произойти все выводы в консоль)

2. Напишите цикл который выведет в консоль все числа от 1 до 100 включительно которые деляться на 5 или на 7 (`n % /*число*/ == 0` – условие делимости `n` на какое либо число)

3. Напишите блок кода который будет в случае выполнения действия `throw ""` выводилась фраза "Произошла ошибка" и после этого выводилась фраза "Ошибки обработаны"

Фукнции

Функции — ещё один важный инструмент из программирования. Если простыми словами это объединение какого либо участка кода для его дальнейшего вызова. В JS функции это объект и с ними можно работать как с объектом, то есть записывать в переменные, передавать как аргумент, писать методы и так далее.

Каждая функция в любом языке программирования имеет:

- Имя
- Список аргументов (параметров) – переменные, значения которых задаёться при вызове функции. Объявляються они в `()` через запятую. Так же при вызове их значения указываются при вызове так же в `()`. В блоке кода нашей функции они отыгрывают роль обычных переменных с блочной областью видимости, только не нуждаються в объявлении, так как объявляються при вызове функции
- Блок кода который будет выполняться при вызове. Записываеться в `{}`

Синтаксис выглядит примерно так:

```
function /*имя функции*/(/*аргумент1*/, /*аргумент2*/.../*аргументN*/){
    // блок испольняемого кода
}
```

`function` – ключевое слово которое означает объявление функции

Для вызова функции достаточно написать её имя и `()` в которых можно перечислить аргументы если они были объявлены

Пример функции:

```
function f(a1, a2, a3){
    console.log(a1);
    console.log(a2);
    console.log(a3);
}

f(1, 2, 3);
```

Фукнции и процедуры

Если отходить от темы JS, где синтаксической разницы между этими понятиями по факту нету, то это очень важный вопрос.

В чем же разница?

Функция — блок кода, который результате своего выполнения возвращет какое либо занчение.

Процедура — функция, которая не возвращает значения

За возвращение значения в JS отвечает ключевое слово `return`

Давайте расмотрим разницу на примерах:

```
function func(a1, a2, a3, a4){ // функция
    console.log(a1);
    console.log(a2);
    console.log(a3);
    return a4;
}

function proc(b1, b2, b3, b4){ // процедура
```

```
    console.log(b1);
    console.log(b2);
    console.log(b3);
}

let a = func(1, 2, 3, 4) // == 4(a4)
let b = proc(1, 2, 3, 4) // == undefined
```

Что же случилось?

При объявлении **функции** мы использовали `return` по-этому вызов функции можно будет присвоить переменной и он будет иметь значение которое мы укзали в `return`

А при объявленнии **процедуры** `return` мы не использовали и-за чего наш вызов не имеет значения, то есть `undefined`

Стрелочные функции и присваивание функции переменной

Так как функция в JS — это объект, то её можно присвоить переменной.

То есть объявить например так:

```
let f = function(a1, a2, a3){
    console.log(a1);
    console.log(a2);
    console.log(a3);
}
```

Такой способ не является не особо коректным с логической точки зрения, но при этом возможным.

Для упрощения записи приведённой выше были придуманы так званые стрелочные функции

Синтаксис выглядит примерно так:

```
let /*имя функции*/ = (/*аргумент1*/, /*аргумент2*/.../*аргументN*/) => {
    // блок испольняемого кода
}
```

Есть несколько важных аспектов:

- Если в блоке кода всего одна команда `{}` не обязательны
- Если в блоке кода всего одна команда то она автоматически будет возвращаться через блок `return` даже не указывая его
- Если аргумент всего один то `()` не обязательны

```
let func = a1 => a1;
let a = func(2); // == 2(a1)
```

То есть `a1` будет автоматически возвращаться и `()` не обязательны так как аргумент всего один

Упорядочиные типы данных

Переменные — это конечно хорошо, но иногда приходиться хранить много каких либо данных объеденив их в какую то одну структуру. Для таких вещей придуманы массивы и словари.

Массивы

Массивы — упорядоченый набор значений, которые можно получить обращаясь по индексу.

Массив в JS это объект, так что он обладает всеми свойствами объекта.

Так же рамотрим несколько способов его объявления:

```
let arr = new Array(); // пустой массив через объект
let arr = new Array(/*элемент 1*/, /*элемент 2*/, ... /*элемент N*/); // через создание объекта, для заполнения
let arr = new Array(/*длина массива*/); // через создание объекта, для создание с длиной по умолчанию
let arr = []; // пустой массив явным образом
let arr = [/*элемент 1*/, /*элемент 2*/, ... /*элемент N*/]; // просто явным образом
```

Для лучшего понимания воспользуемся картинкой выше. Как видно на ней массив — это грубо говоря набор "ячеек" со значениями к которым параллельно прикриплены индексы по которым можно получить значение которому они соответствуют.

Для получения значения или присваивания по индексу достаточно после имени массива в `[]` указать индекс. Для получения длины массива можно вызвать свойство `.length`.

```
let a = arr[0]; // получение
arr[0] = 1; // присваивание
let len = arr.length; // получения длины
```

!!!Индесация в языках программирования начинается с 0!!!

Плохая и хорошая особенность JS:

Если в JS обратиться по необъявленому индексу то при получении значения будет возвращено `undefined`, а при присваивании будет создан индекс с этим номером.

Пример:

```
let arr = [1, 2, 3];
let a = arr[4]; // undefined
arr[4] = 4; // arr == [1, 2, 3, 4]
```

Такое использование являеться не особо правильном с логической точки зрения, но оно присутствует и на самом деле появляеться из-за того, что массив в JS это объект.

Основные методы массивов

- `.concat()` – объеденение массивов

  ```
  let arr1 = [1, 2, 3];
  let arr2 = [4, 5, 6];
  let arr3 = arr1.concat(arr2); // == [1, 2, 3, 4, 5, 6]
  ```

- `.join(/*разделитель*/)` – объеденяет массив в строку и вставляет между элементами разделитель который мы укажем в аргументе

  ```
  let arr = [1, 2, 3];
  let s = arr.join("-"); // == "1-2-3"
  ```

- `.push()` – добавление значения в конец массива

  ```
  let arr = [1, 2, 3];
  arr.push(4);
  // arr == [1, 2, 3, 4]
  ```

- `.pop()` – удаляет последний элемент массива

  ```
  let arr = [1, 2, 3];
  arr.pop();
  // arr == [1, 2]
  ```

- `.unshift()` – добавляет значения в начало массива

  ```
  let arr = [1, 2, 3];
  arr.unshift(0);
  // arr == [0, 1, 2, 3]
  ```

- `.shift()` – удаляет первый элемент массива

  ```
  let arr = [1, 2, 3];
  arr.shift();
  // arr == [2, 3]
  ```

- `.indexOf()` – возвращает первый индекс элемента в массиве

```
let arr = [1, 2, 2, 3];
let i = arr.indexOf(2); // 1
```

- `.lastIndexOf()` – возвращает последний индекс элемента в массиве

```
let arr = [1, 2, 2, 3];
let i = arr.lastIndexOf(2); // 2
```

- `.slice(/*с*/, /*до*/)` – создание среза

```
let arr1 = [1, 2, 3, 4, 5, 6];
let arr2 = arr1.slice(1, 4); // [2, 3, 4]
```

- `.splice(/*индекс*/, /*сколько элементов удалить*/, /*элементы которые надо вставить*/)` – удаляет какое либо количество элементов из массива по индексу и на их место вставляет элементы которые перечислены потом

```
let arr = [1, 2, 3, 4, 5, 6];
arr.splice(1, 2, 7, 8);
// arr == [1, 7, 8, 4, 5, 6]
```

- `.reverse()` – переставляет элементы в обратном порядке

```
let arr = [1, 2, 3];
arr.reverse();
// arr == [3, 2, 1]
```

Про остальные можно узнать по ссылке:
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array

Перебор массива

for

```
for(let i = 0; i < arr.length; i ++){
    // элемент массива arr[i]
}
```

for...in

```
for(let i in arr){
    // элемент массива arr[i]
}
```

for...of

```
for(let a of arr){
    // элемент массива a
}
```

.forEach()

```
arr.forEach(a => {
    // элемент массива a
});
```

Словари

Иногда возникают ситуации, когда надо что б индексами были не цифры, а что-то своё. Для такого были созданы словари.

Словари – это массив с самоуказаными индексами.

Синтаксис объявления

```
let dic = {
    "/*название индекса 1*/": /*занчение 1*/,
    "/*название индекса 2*/": /*занчение 2*/,
    ...
    "/*название индекса N*/": /*занчение N*/,};
```

Обращение и присваивание такое же, только индексы не обязательно цифры:

```
let dic = {"name": "Sam", "age": 17}
let d = dic["name"] // "Sam"
dic["age"] = 18;
```

Работа со строками

После того, как мы ознакомились с массивами можно поговорить про строки.

По факту: **строка** – это массив символов. К её элементам можно обращаться как в массиве, но нельзя изменять как элементы в массиве.

```
let s = "abc";
let c = s[0]; // "a"
s[0] = "d"; // невозможно
```

Методы строк

- `.charCodeAt()` – получение кода элемента

```
let c = "a".charCodeAt(); // 97
```

- `String.fromCharCode()` – получение элемента по коду

```
let c = String.fromCharCode(97); // "a"
```

- `.indexOf()` – возвращает первый индекс символа в строке

```
let s = "abca";
let i = s.indexOf("a"); // 0
```

- `.lastIndexOf()` – возвращает последний индекс символа в строке

```
let s = "abca";
let i = s.lastIndexOf("a"); // 3
```

- `.slice(/*с*/, /*до*/)` – создание среза

```
let s1 = "abcdef";
let s2 = s1.slice(1, 4); // "bcd"
```

- `.split(/*разделитель*/)` – разбивает строку на массив строк по разделителю

```
let s = "a-b-c";
let arr = s.split("-"); // ["a", "b", "c"]
```

- `.toLowerCase()` – возвращает строку в нижнем регистре

```
let s1 = "AbC";
let s2 = s1.toLowerCase(); // "abc"
```

- `.toUpperCase()` – возвращает строку в нижнем регистре

```
let s1 = "AbC";
let s2 = s1.toUpperCase(); // "ABC"
```

- `.repeat()` – повторяет строку количество раз, которое указано в аргументе

```
let s1 = "abc";
let s2 = s1.repeat(3); // "abcabcabc"
```

- `.trim()` – удаляет пробелы в начале и конце строки

```
let s1 = " abc ";
let s2 = s1.trim(); // "abc"
```

Практические задания на данных знаниях

4. Придумайте как перевернуть строку в обратном порядке
5. Найдите способ строку `"abcdef"` превратить в `"cdefgh"`
6. Напишите функцию которая будет возвращать строку созданую по правилу `camelCase` (разделитель должен передаваться вторым аргументом)

   Пример:

   `hello world` -> `helloWorld`
7. Создайте функцию для заполнения массива словарей, которые содержат информацию про человека

   Структура:

   ```
   let people = []; // основной массив
   /* структура словаря
   {
       "firstName": "",
       "secondName": "",
       "age": "",
       "country": "",
       "city": ""
   };
   */
   ```
8. *Придумайте как можно отсортировать массив

"*" – сложная задача

Всё!

В принцепе это всё! Я дал вам всё что хотел. Надеюсь этот гайд помог вам.

Так же у меня есть гайд посложнее с более сложными темами, но он больше просто сборник решений, но кому интересно можете почитать: https://github.com/s0urce18/AdditionalFunctions

Всем спасибо! Надеюсь ещё увидимся :)