Методичка

JavaScript.Графика

[Урок 1 Функции 5](#_Toc33300237)

[Создание своих функций 6](#_Toc33300238)

[Упрощение кода с помощью функций 8](#_Toc33300239)

[Использование return в условных конструкциях 9](#_Toc33300240)

[Виселица 10](#_Toc33300241)

[Практические задания 11](#_Toc33300242)

[Домашнее задание 11](#_Toc33300243)

[Урок 2 Основы CSS 12](#_Toc33300244)

[Виды таблиц стилей 12](#_Toc33300245)

[Внешняя таблица стилей 12](#_Toc33300246)

[Внутренние стили 13](#_Toc33300247)

[Встроенные стили 13](#_Toc33300248)

[Свойства CSS 13](#_Toc33300249)

[Виды селекторов 17](#_Toc33300250)

[Универсальный селектор 17](#_Toc33300251)

[Селектор элемента 17](#_Toc33300252)

[Селектор класса 18](#_Toc33300253)

[Домашнее задание 18](#_Toc33300254)

[Урок 3 Создание игры Виселица 18](#_Toc33300255)

[Взаимодействие с игроком 19](#_Toc33300256)

[Проектирование игры 19](#_Toc33300257)

[Используем псевдокод для проектирования игры 20](#_Toc33300258)

[Отображение состояния игры 20](#_Toc33300259)

[Проектируем игровой цикл 21](#_Toc33300260)

[Программируем игру 22](#_Toc33300261)

[Программируем игровой цикл 23](#_Toc33300262)

[Отображение состояния игры 23](#_Toc33300263)

[Обработка введенного ответа 23](#_Toc33300264)

[Обновление состояния игры 24](#_Toc33300265)

[Попытки 25](#_Toc33300266)

[Домашнее задание 27](#_Toc33300267)

[Урок 4 DOM-дерево 27](#_Toc33300268)

[Пример DOM 27](#_Toc33300269)

[Автоисправление 28](#_Toc33300270)

[Работа с DOM 28](#_Toc33300271)

[Поиск элементов на странице 28](#_Toc33300272)

[Урок 5 Случайные числа 29](#_Toc33300273)

[Работа над домашним заданием 29](#_Toc33300274)

[Случайные числа 29](#_Toc33300275)

[Случайные индексы 30](#_Toc33300276)

[Программа случайного выбора вариантов 31](#_Toc33300277)

[Практическая работа 32](#_Toc33300278)

[Генератор случайных дразнилок 32](#_Toc33300279)

[Домашнее задание 32](#_Toc33300280)

[Урок 6 Объекты 33](#_Toc33300281)

[Знакомство с объектами 33](#_Toc33300282)

[Добавление элементов объекта 34](#_Toc33300283)

[Добавление ключей через точку 35](#_Toc33300284)

[Undefined и Null 35](#_Toc33300285)

[Массивы объектов 35](#_Toc33300286)

[Массив друзей 36](#_Toc33300287)

[Практическая работа – Учёт долгов 37](#_Toc33300288)

[Домашнее задание 38](#_Toc33300289)

[Урок 7 Основы HTML 38](#_Toc33300290)

[Знакомство с HTML 38](#_Toc33300291)

[Основные HTML теги 40](#_Toc33300292)

[Теги заголовков 41](#_Toc33300293)

[Форматирование и работа с текстом 41](#_Toc33300294)

[Встраиваемые элементы HTML 42](#_Toc33300295)

[Практическая работа 42](#_Toc33300296)

[Домашнее задание 42](#_Toc33300297)

[Урок 8 HTML и JavaScript 42](#_Toc33300298)

[Внедрение JavaScript-кода в HTML 42](#_Toc33300299)

[Подключение стороннего скрипта 43](#_Toc33300300)

[Практическая работа – Викторина 44](#_Toc33300301)

[Домашнее задание 44](#_Toc33300302)

[Урок 9 Опять строки 44](#_Toc33300303)

[Работа со строками в циклах 44](#_Toc33300304)

[Методы split() и replace() 46](#_Toc33300305)

[Работа с регистром символов 47](#_Toc33300306)

[Практическая работа – Хакерский чат 49](#_Toc33300307)

[Домашнее задание 53](#_Toc33300308)

[Урок 10 Создание игры «Виселица» 53](#_Toc33300309)

[Взаимодействие с игроком 53](#_Toc33300310)

[Проектирование игры 54](#_Toc33300311)

[Используем псевдокод для проектирования игры 55](#_Toc33300312)

[Отображение состояния игры 55](#_Toc33300313)

[Проектируем игровой цикл 56](#_Toc33300314)

[Программируем игру 57](#_Toc33300315)

[Программируем игровой цикл 57](#_Toc33300316)

[Отображение состояния игры 58](#_Toc33300317)

[Обработка введенного ответа 58](#_Toc33300318)

[Обновление состояния игры 59](#_Toc33300319)

[Домашнее задание 60](#_Toc33300320)

[Урок 11 Доработка игры. Функции и кнопки 60](#_Toc33300321)

[Доработка игры «Виселица» 60](#_Toc33300322)

[Регистр букв 60](#_Toc33300323)

[Попытки 61](#_Toc33300324)

[Поздравление игрока 63](#_Toc33300325)

[Возможность ввести слово 63](#_Toc33300326)

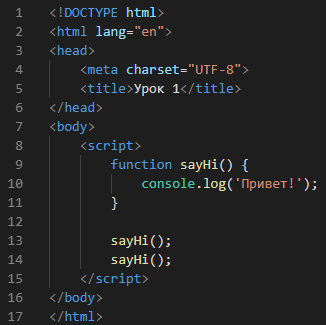
[Функции и кнопки 64](#_Toc33300327)

[Создание кнопки 64](#_Toc33300328)

[Связь кнопки и скрипта 66](#_Toc33300329)

[Домашнее задание 67](#_Toc33300330)

# Урок 1 Функции

Зачастую нам надо повторять одно и то же действие во многих  
частях программы. Например, красиво вывести сообщение необходимо при  
приветствии посетителя, при выходе посетителя с сайта, ещё  
где-нибудь. Чтобы не повторять один и тот же код во многих местах,  
придуманы функции. Функции являются основными «строительными  
блоками» программы. Пример создания функции для приветствия.

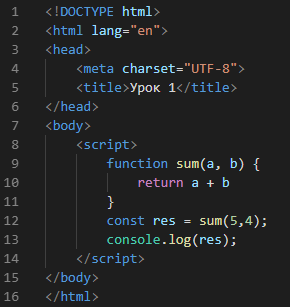
Данная функция выведет в консоль сообщение «Привет!», но что делать, если мы хотим поздороваться с кем-то лично. Для этого функции нужно знать имя человека, с которым мы хотим поздороваться. Сказать ей имя можно используя параметр или как его по-другому называют – аргумент функции. Это то значение, которое мы передаём в функцию, чтобы она могла использовать её. Параметры могут быть разные, но функция проделает со всеми ними одно и тоже действие.

Мы передаем параметр **name** в качестве аргумента функции **sayName**

Откроем нашу страничку в браузере и увидим два приветствия в консоли. Мы можем вызвать эту функцию прямо оттуда, передав ей какое-нибудь имя в качестве параметра.

## Создание своих функций

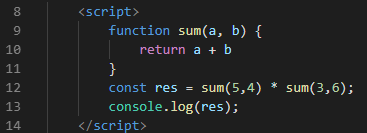
Сейчас мы рассмотрели функцию, которая просто выполняет некоторое действие, когда мы её вызываем. Но есть функции, которые что-то возвращают после своей работы.

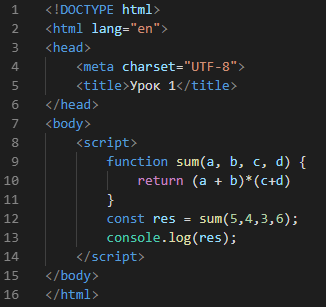
 Возвращаемое значение – это значение, которое функция выдаёт наружу, чтобы потом его можно было использовать где угодно в программе. Для того, чтобы вернуть какое-то значение нужно использовать ключевое слово **return.**

Эта функция складывает два числа и возвращает нам сумму, которую мы можем использовать дальше в нашей программе.

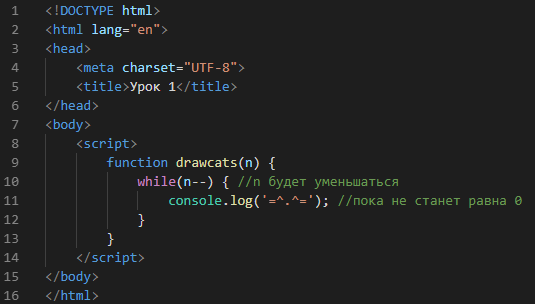
Давайте решим следующий пример, используя нашу функцию **sum**:

**(a+b)\*(c+d) = ?**

 В качестве **a, b, c, d** давайте возьмём числа **5, 4, 3, 6**

Теперь давайте перепишем нашу функцию так, чтобы мы могли передать ей четыре числа, а она вернула бы нам ответ на наш пример. Кто справится самостоятельно, получит **Подсказку 1** к домашнему заданию.

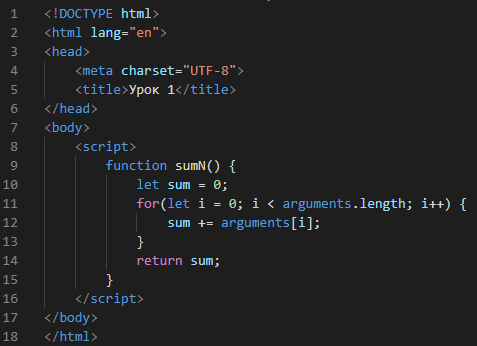
Не забываем, что мы можем вызвать эту же функцию в консоли, только с другими параметрами, и она проделает тот же результат.

 Давайте создадим функцию, которая будет рисовать котов. В качестве аргумента этой функции будет выступать число котов, которых мы хотим нарисовать.

Теперь, вызвав эту функцию в консоли, мы можем указать в скобках любое число и именно столько котов нарисует наша функция.

Самостоятельно усовершенствуйте эту функцию так, чтобы в неё можно было передать не только сколько раз мы хотим нарисовать, но и что мы хотим нарисовать.

Мы с вами уже складывали два числа, но что если нам нужно сложить больше чисел? Да, вы можете написть функцию **sum**, которая будет считать сумму трёх числе, можете написать такую же, которая будет считать сумму четырёх чисел. Но что если мы не знаем сколько чисел мы хотим сложить? Можно ли сделать такую функцию, которая будет считать сумму двух чисел, если мы передадим в неё два аргумента, трёх – если три и так далее? Да, именно такую функцию мысейчас и напишем.

 Все аргументы, которые мы передаём в функцию помещаются в некоторый массив **arguments** невидимый для нас. Но сейчас мы попробуем воспользоваться им.

Теперь мы можем вызвать функцию **sum** с любым количеством аргументов.

## Упрощение кода с помощью функций

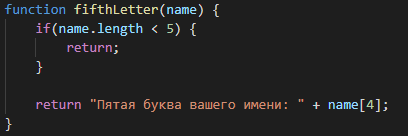
 Мы уже использовали функции Math.random и Math.floor, чтобы выбирать случайные слова из массивов. Вот так мы её записывали:

 Если поместить этот код в функцию, можно многократно вызывать его для получения случайного слова из любого массива – вместо ого чтобы вводить тот же код снова и снова. Например, давайте определим такую функцию **pickRandomWord:**

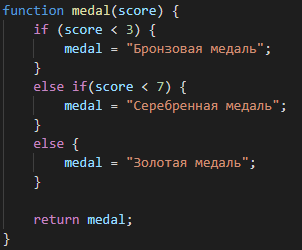
Всё что мы сделали – поместили прежний код в функцию. Теперь давайте самостоятельно создадим какой-нибудь массив **randomWords**. И с помощью, созданной нами функции, выберем из него рандомное слово.

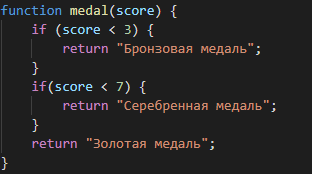
## Использование return в условных конструкциях

Как только JavaScript, выполняя код функции, встречает return, он завершает функцию, даже если после return остался ещё какой-нибудь код.

 Оператор return часто используют, чтобы выйти из функции в самом начале, если какие-нибудь из переданных аргументов имеют некорректные значения – то есть если с такими аргументами функция не сможет правильно работать. Например, следующая функция возвращает строку с информацией о пятой букве вашего имени. Но если в имени, переданном в аргументе name, меньше пяти букв, будет выполнен return, чтобы сразу же выйти из функции. При этом оператор return в конце функции (тот, что возвращает сообщение о пятой букве) так и не будет выполнен.

Раньше мы использовали много раз конструкцию **if..else if..else.** Внутри функций мы можем использовать только **if** вставляя в каждый из них return, чтобы закончить выполнение функции.

 Предположим вы пишите игру, в которой игроки награждаются медалями согласно набранным очкам. Счёту меньше трёх очков соответствует бронзовая медаль, от трёх до шести – серебренная, а от семи и выше – золотая. Так бы выглядел наш код, если бы мы использовали **if..else if..else:**

 А вот так он будет выглядеть, если мы будем использовать return:

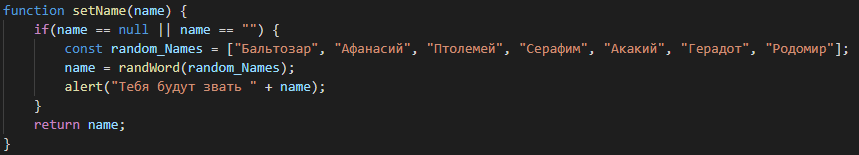
Кто напишет самостоятельно получит **Подсказку** к домашнему заданию.

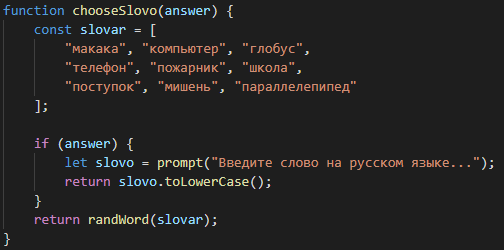
***Если у детей есть игра Виселица с предыдущего курса JS, то следовать дальше по методичке, иначе перейти к практическим заданиям и выполнить пункт со \****

## Виселица

Давайте вернёмся к нашей игре виселица и перепишем её код используя функции.

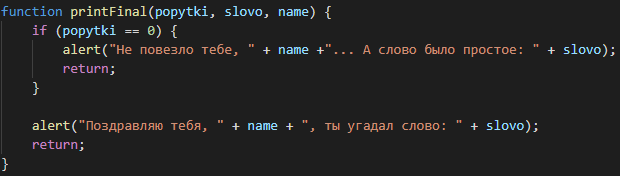
 Для начала создадим функцию **randWord,** которая будет выбирать случайное слово из переданного ей массива.

 Следующую функцию назовём **setName,** она будет задавать имя нашему игроку.

 В отдельную функцию также поместим кусок кода, в котором мы выбираем слово:

Обратите внимание, что внутри этой функции мы используем созданную нами же функцию **randWord**.

 На каждом шаге нашей игры мы выводим текущий счёт. Давайте вынесем эту строчку в отдельную функцию обновления счёта:

 И последняя функция **printFinal** будет отвечать за вывод последнего сообщения:

*Полный код в папке Урок1*

## Практические задания

1. Написать функцию-генератор рандомных дразнилок.
2. **(\*) Написать функцию, которая будет создавать вымышленных людей и выводить их ФИО.**

## Домашнее задание

**Совпадают ли массивы?**

Напишите функцию **same(massiv1, massiv2),** которая принимает два массива с числами в качестве аргументов. Она должна возвращать **true**, если эти массивы одинаковые (то есть содержат одни и те же числа в одном и том же порядке), или **false,** если массивы различаются. Убедитесь, что ваша функция работает правильно, запустив такой код:

**same([1,2,3],[4,5,6];**

**same([1,2,3],[1,2,3];**

**same([1,2,3],[3,2,1];**

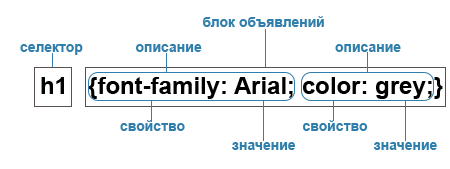
**same([1,2,3],[1,2,3,4];**

 **Подсказка 1**: вам понадобиться перебрать все значения из первого массива в цикле for и убедиться, что они совпадают со значениями из второго массива. Вы можете вернуть false прямо из тела for, если обнаружите несовпадение значения

**Подсказка 2**: вы можете сразу выйти из функции, если у массивов разная длина.

# Урок 2 Основы CSS

**CSS (Cascading Style Sheets)** — язык таблиц стилей, который позволяет прикреплять стиль (например, шрифты и цвет) к документам HTML. Каскадные таблицы стилей описывают правила оформления элементов на странице с помощью свойств и допустимых значений этих свойств. Для каждого элемента можно использовать подходящие для него свойства, остальные свойства не будут оказывать на него никакого влияния.

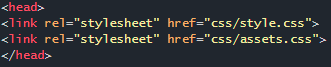
Объявление стиля состоит из двух частей: **селектора** и **объявления**. В HTML имена элементов нечувствительны к регистру, поэтому «**h1**» работает так же, как и «**H1**». Объявление состоит из двух частей: имя свойства (например, **color**) и значение свойства (**grey**). Селектор сообщает браузеру, какой именно элемент форматировать, а в блоке объявления (код в фигурных скобках) перечисляются форматирующие команды — свойства и их значения.

## Виды таблиц стилей

### Внешняя таблица стилей

**Внешняя таблица стилей** представляет собой текстовый файл с расширением **.css**, в котором находится набор CSS-стилей элементов. Файл создаётся в редакторе кода, так же как и HTML-страница. Внутри файла могут содержатся только стили, без HTML-разметки.

Внешняя таблица стилей подключается к веб-странице с помощью тега **<link>**, расположенного внутри раздела **<head></head>**. Такие стили работают для всех страниц сайта.

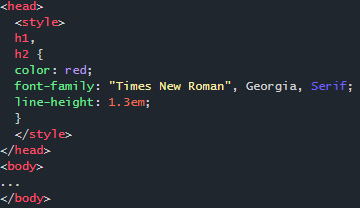
К каждой веб-странице можно присоединить несколько таблиц стилей, добавляя последовательно несколько тегов **<link>**.

**rel="stylesheet"** указывает тип ссылки (ссылка на таблицу стилей).

Атрибут **type="text/css"** не является обязательным по стандарту HTML5, поэтому его можно не указывать. Если атрибут отсутствует, по умолчанию используется значение **type="text/css"**.

Давайте создадим файл стилей, подключим его к нашему документу и поменяем цвет у некоторых элементов на нашей странице.

### Внутренние стили

**Внутренние стили** встраиваются в раздел **<head></head>** HTML-документа и определяются внутри тега **<style></style>**. Внутренние стили имеют приоритет над внешними, то есть являются более значимыми.

### Встроенные стили

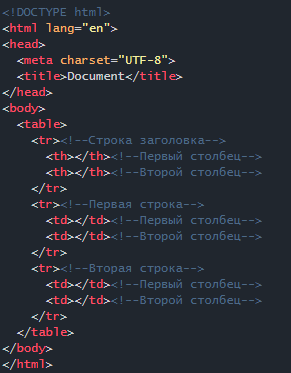
Когда мы пишем **встроенные стили**, мы пишем CSS-код в HTML-файл, непосредственно внутри тега элемента с помощью атрибута **style**:

## Свойства CSS

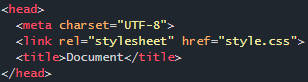
Вообще, CSS довольно элементарный формальный язык, который придумали для описания внешнего вида документов. Это говорит о том, что он довольно прост и состоит из примитивных конструкций, которые не так сложны для изучения. Самое сложное не синтаксис, не правила написания конструкций, а огромное количество CSS свойств для запоминания, которые выполняют различные задачи.

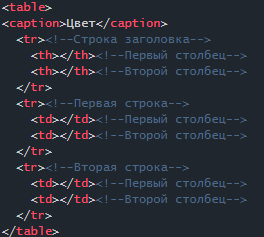
Благо, все правила англоязычные с соответствующей смысловой нагрузкой. Простой перевод на наш язык дает понятие о том, что это правило делает и наоборот - при переводе того, что мы хотим добиться определенным свойством на английский язык, велика вероятность того, что мы получим правильное свойство.

Это значительно упрощает запоминание CSS правил на интуитивном уровне. Например, если нужно задать **фоновый цвет** достаточно сделать перевод на английский, в результате чего получаем **background-color** (отдельные слова в CSS пишутся через дефис).

Создадим следующий HTML-документ, как памятку с самыми часто используемыми свойствами CSS.

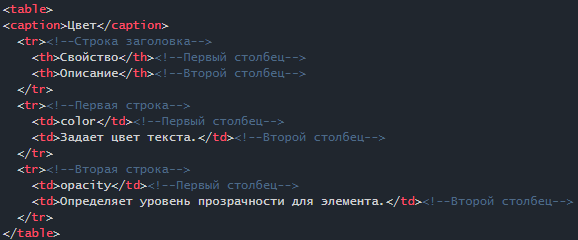
Элемент **<table>** служит контейнером для элементов, определяющих содержимое таблицы. Любая таблица состоит из строк и ячеек, которые задаются с помощью тегов **<tr>** и **<td>**. Где **<tr>** – это строка, а **<td> –** столбец. В нашем примере мы ещё будем использовать тег **<th> -** это ячейка специального форматирования, которая будет являться заголовком. Подпишем комментарии ко всем этим тегам, чтобы легче запомнить их предназначение.

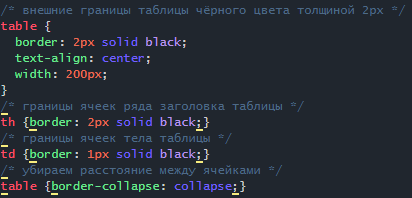
Создадим файл стилей и подключим его к HTML-документу:

В нём в дальнейшем мы будем описывать стили нашей таблицы и страницы в целом.

Чтобы добавить заголовок для нашей таблицы, добавим тег **<caption>**.

Теперь мы увидим наш заголовок, но где же сама таблица? Дело в том, что мы ещё не написали текст в ячейках таблицы и не указали стили границ для нашей таблицы. А размеры таблицы в HTML зависят от её содержимого, если мы не укажем их дополнительно.

Давайте напишем следующий текст, а именно, заголовки и свойства, с помощью которых можно управлять цветом в CSS.

Сейчас то, что мы создали, не совсем похоже на таблицу. Что бы это исправить, добавим стили в созданный заранее файл. (*после каждого блока обновляем страницу и смотрим результат*)

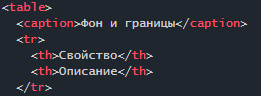
Теперь наша таблица преобразилась и можем продолжать создавать новые таблицы.

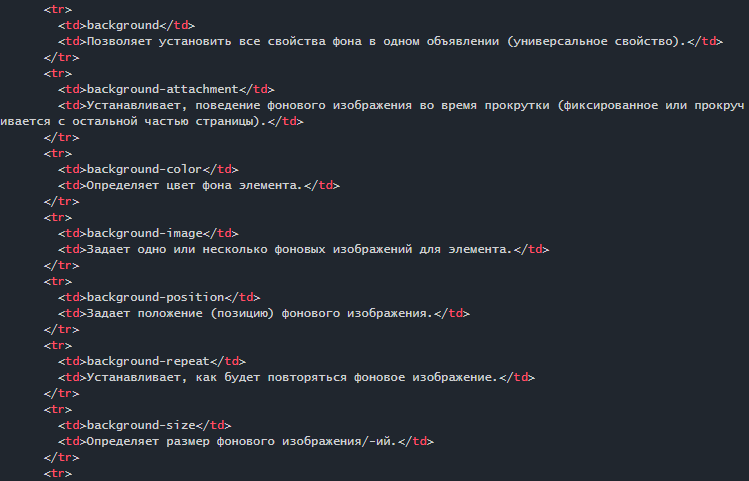
Следующая таблица будет подсказывать нам о свойствах фона и границ. Поэтому назовём её **Фон и границы**. Создадим таблицу с помощью плагина **emmet,** который из одной простой строчки создаст целую таблицу.

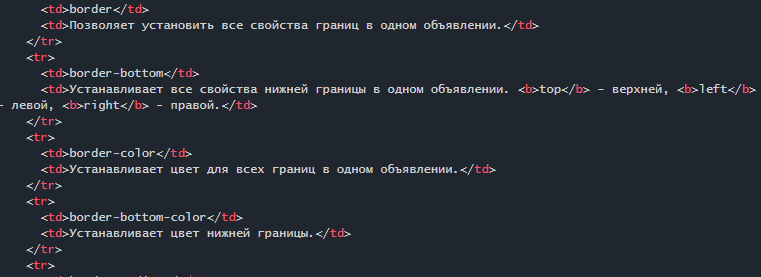
Таблица будет иметь 19 строчек + строка заголовка. Столбцов по-прежнему 2.

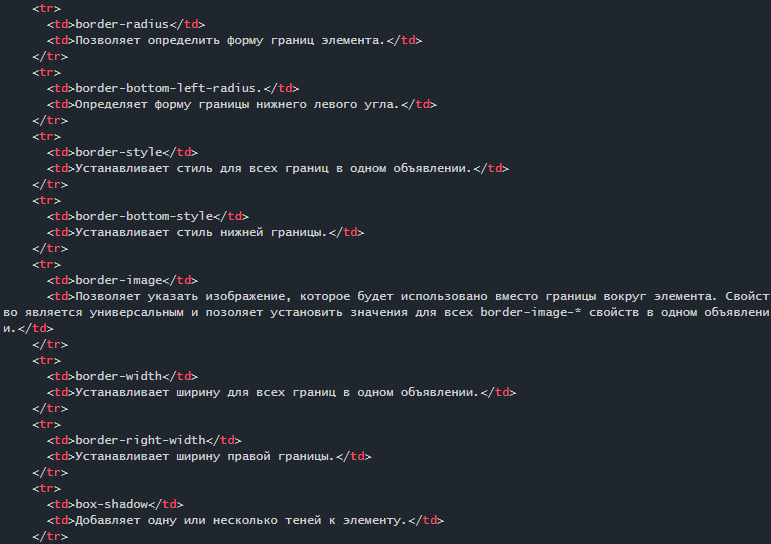
Напишем следующую строчку в HTML-документе:

(*Объяснить значение всех элементов*)

Теперь исправим теги **<td>** в первой строке на **<th>,** т.к. первая строка – это строка заголовков.

И начнём заполнять нашу таблицу:

Проверяем как выглядит наша страничка сейчас. И переходим к следующим стилям:

 Во всех свойствах, где указывается определённая сторона, вместо **bottom**(низ)можно использовать название любого другого края, как в примере с просто **border.**

Это основные свойства оформления, которые мы будем использовать для создания стилей нашей страницы. Все эти свойства можно применять к любым селекторам, но сначала давайте узнаем, то такое селекторы.

## Виды селекторов

**Селекторы** представляют структуру веб-страницы. С их помощью создаются правила для форматирования элементов веб-страницы. Селекторами могут быть элементы, их классы и идентификаторы.

Давайте разберёмся и приведём примеры селекторов. Какие вы знаете?

### Универсальный селектор

Соответствует любому HTML-элементу. Например, **\* {color: red;}** установит цветом текста красный для всех элементов сайта.

### Селектор элемента

Селекторы элементов позволяют форматировать все элементы данного типа на всех страницах сайта. Например, **h1 {font-family: Lobster, cursive;}** задаст общий стиль форматирования всех заголовков **h1**.

### Селектор класса

Селекторы класса позволяют задавать стили для одного и более элементов с одинаковым именем класса, размещенных в разных местах страницы или на разных страницах сайта. Например, для создания заголовка с классом **headline** необходимо добавить атрибут **class** со значением **headline** в открывающий тег **<caption>** и задать стиль для указанного класса. Стили, созданные с помощью класса, можно применять к другим элементам, не обязательно данного типа.

Пропишем стили для наших названий таблиц, чтобы они выделялись среди таблиц. ***Самостоятельно***

## Домашнее задание

Добавить классы ко всем ячейкам, которые входят в первый столбик и изменить стиль текста для них. То же самое сделать со вторым столбцом и со строкой заголовков. На следующем занятии будем сравнивать, чья таблица получилась лучше.

(*Картинку с описанием свойств форматирования текста скинуть всем на флешки –* ***textStyle.png****)*

# Урок 3 ****Создание игры Виселица****

***Если игра Виселица была создана детьми в предыдущем курсе JS и доделана в начале этого курса, то на данном уроке предлагается попрактиковаться в написании кода и использовании функций на примере игры-викторины – viktorina.js из папки к данному уроку. Не забудьте создать HTML-документ и кнопку запускающую основную функцию из подключенного скрипта. Иначе следуйте плану урока.***

«Виселица» — игра на угадывание слов. Один игрок выбирает слово, а второй пытается его отгадать. Например, если первый игрок загадал слово *КАПУСТА* , он изобразит семь «пустых мест», по одному на каждую букву слова:

\_ \_ \_ \_ \_ \_ \_

Второй игрок старается отгадать это слово, называя буквы. Каждый

раз, когда он угадывает букву, первый игрок заполняет пустоты, вписывая ее везде, где она встречается. Например, если второй игрок назвал букву «А», первый должен вписать все «А» для слова КАПУСТА, вот так:

\_ А \_ \_ \_ \_ А

Если второй игрок назовет букву, которой нет в слове, у него отнимается очко, а первый игрок рисует руку, ногу или другую часть тела

человечка. Если первый игрок закончит рисовать человечка раньше, чем второй угадает все буквы, второй игрок проиграл.

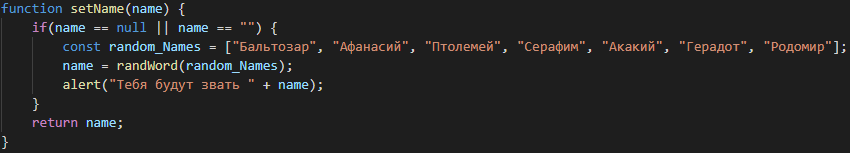
В нашем варианте «Виселицы» JavaScript будет выбирать слово, а игрок-человек — отгадывать буквы. И рисовать человечка наша программа не будет, поскольку мы пока не знаем, как это делается.

## Взаимодействие с игроком

Для этой игры нам нужно, чтобы игрок (человек) мог каким-то образом вводить в программу свои ответы. Один из способов это сделать — открывать диалоговое окно (в JavaScript оно называется *prompt* ), в котором игрок может что-нибудь напечатать.

Сначала создадим файлы проекта. Файл **index.html**. Создадим в нём основную разметку, подключим наш скрипт **script.js**.

Создайте и откройте файл **script.js,** а потом самостоятельно создайте обращение к игроку с запросом: «Введите своё имя».

После этого мы сделаем функцию, проверку на тот случай, если пользователь нажал «Отмена» или не ввёл ничего.

Функцию **randWord** создайте самостоятельно.

## Проектирование игры

Прежде чем перейти к созданию игры «Виселица», давайте подумаем

о ее структуре. Нам нужно, чтобы программа умела выполнять следующие действия:

1. Случайным образом выбирать слово.

2. Запрашивать у игрока вариант ответа (букву).

3. Завершать игру по желанию игрока.

4. Проверять, является ли введенный ответ буквой.

5. Вести учет угаданных букв.

6. Показывать игроку, сколько букв он угадал и сколько еще пред-

стоит угадать.

1. Завершать игру, если слово отгадано.

Все эти действия, кроме первого и последнего (выбор слова и завершение игры), нужно выполнять многократно, причем заранее неизвестно, сколько раз (это зависит от ответов игрока). И, как мы теперь

знаем, если требуется повторять какие-то действия, значит, в программе

нужен цикл.

Однако в нашем списке действий ничего не говорится о том, что и когда должно происходить. Чтобы выяснить этот вопрос и лучше представить себе структуру будущей программы, мы можем воспользоваться

*псевдокодом* .

### Используем псевдокод для проектирования игры

Псевдокод — удобный инструмент, который программисты часто

используют при проектировании программ. Слово «псевдокод» означает

«ненастоящий код». Хотя в псевдокоде есть циклы и условия, в целом программа описывается обычным человеческим языком. Чтобы разобраться, что это значит, давайте посмотрим на описание нашей игры в псевдокоде:

*(Записать этот псевдокод вместе с детьми в отдельный файл)*

Выбрать случайное слово

Пока слово не угадано {

Показать игроку текущее состояние игры

Запросить у игрока вариант ответа

Если игрок хочет выйти из игры {

Выйти из игры

}

Иначе Если вариант ответа — не одиночная буква {

Сообщить игроку, что он должен ввести букву

}

Иначе {

Если такая буква есть в слове {

Обновить состояние игры, подставив новую букву

}

}

}

Поздравить игрока с победой — слово угадано

Как видите, это не программный код, который может выполнить компьютер. Однако такая запись дает нам представление о структуре программы прежде, чем мы перейдем к написанию кода и выяснению мелких деталей, например, *как* именновыбирать случайное слово.

### Отображение состояния игры

Одна из строк нашего псевдокода гласит: «Показать игроку текущее

состояние игры». Для игры «Виселица» это означает подставить в слово

угаданные игроком буквы, а также показать, какие буквы осталось угадать. Как мы будем это делать? В сущности, можно хранить состояние игры тем же способом, что и в обычной «Виселице»: в виде последовательности «пустых мест», которые мы будем заполнять по мере того, как игрок угадывает буквы.

Мы сделаем это с помощью массива «пустых мест» – по одному

элементу для каждой буквы в слове. Назовем этот массив «итоговым массивом» и будем по ходу игры заполнять его угаданными буквами. А каждое из «пустых мест» представим в виде строки со знаком подчеркивания: "\_".

Сначала наш итоговый массив будет просто набором «пустых мест», количество которых равно количеству букв в загаданном слове. Например, если загадано слово «рыба», массив будет выглядеть так:

[" \_ ", " \_ ", " \_ ", " \_ "]

Если игрок угадает букву «ы», мы заменим второй элемент на «ы»:

[" \_ ", "ы", " \_ ", " \_ "]

А когда игрок угадает все буквы, массив примет вид:

["р", "ы", "б", "а"]

Также нам понадобится переменная для хранения количества букв, которые осталось угадать. Для каждого вхождения верно угаданной буквы эта переменная будет уменьшаться на 1, и, когда она примет значение 0, мы поймем, что игрок победил.

### Проектируем игровой цикл

Основная часть игры будет располагаться внутри цикла while (в нашем

псевдокоде этот цикл начинается со строки «Пока слово не угадано»).

В цикле мы будем отображать текущее состояние игры (то есть слово,

поначалу представленное одними знаками подчеркивания), запрашивать у игрока вариант ответа (и проверять, действительно ли тот ввел одиночную букву), а также обновлять итоговый массив, подставляя введенную букву, если она действительно есть в слове.

Практически все компьютерные игры организованы в виде того или

иного цикла, нередко структурно похожего на цикл нашей «Виселицы».

В целом игровой цикл выполняет следующие задачи:

1. Принимает ввод от игрока.

2. Обновляет состояние игры.

3. Показывает игроку текущее состояние игры.

Такого рода цикл применяется даже в играх, где непрерывно что-то

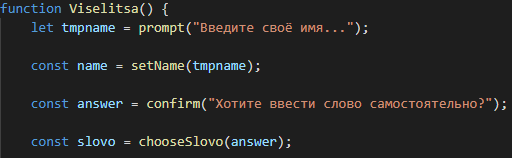
меняется, – просто он выполняется *очень* быстро. В случае нашей «Виселицы» программа запрашивает у игрока вариант ответа, обновляет

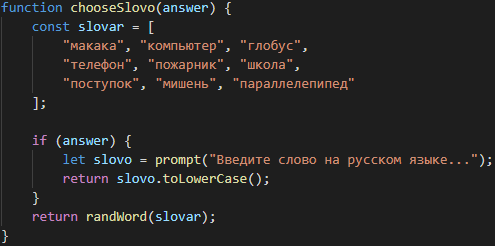
итоговый массив (если ответ верный) и отображает новое состояние итогового массива.

Если игрок угадает все буквы в слове, мы должны показать ему законченное слово, а также вывести сообщение, поздравляющее с победой.

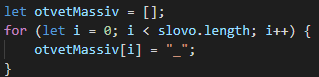
## Программируем игру

Теперь, когда у нас есть представление о структуре игры, можно переходить к написанию кода.

 Первым делом нам нужно выбрать случайное слово или дать возможность пользователю ввести его:

Реализуем это при помощи функций, по возможности самостоятельно.

Далее создадим пустой массив под названием **otvetMassiv** (итоговый

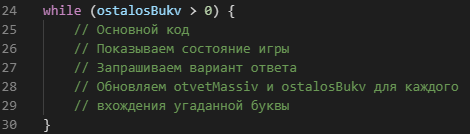
массив) и заполним его символами подчеркивания (\_), количество которых соответствует количеству букв в загаданном слове.

Когда цикл завершится, длина **otvetMassiv**будет такой же, как длина слова. Например, если было выбрано слово «макака» (в котором шесть букв),

**otvetMassiv** примет вид [" \_ ", " \_ ", " \_ ", " \_ ", " \_ ", " \_ "] (шесть знаков подчеркивания).

Создадим переменную **ostalosBukv**, приравняв ее к длине загаданного слова. Эта переменная понадобится, чтобы отслеживать количество букв, которые осталось угадать. Каждый раз, когда игрок угадает букву, мы будем уменьшать значение этой переменной: на 1 для каждого вхождения буквы в слово.

### Программируем игровой цикл

 Основа игрового цикла выглядит так:

Мы используем цикл **while**, который будет повторяться до тех пор, пока условие **ostalosBukv > 0** дает **true**. В теле цикла надо будет обновлять **ostalosBukv** для каждого правильного ответа игрока. Когда игрок угадает все буквы, **ostalosBukv** примет значение 0, и цикл завершится.

### Отображение состояния игры

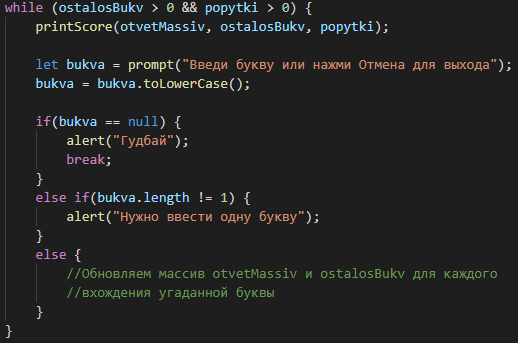
Первым делом в теле игрового цикла нужно показать игроку его имя и текущее состояние игры.

Мы делаем это, объединяя элементы **otvetMassiv** в строку с пробелом в качестве разделителя, а затем с помощью **alert**показываем эту строку.

Например, пусть загадано слово «макака» и игрок угадал буквы «м» и «а». Тогда итоговый массив примет вид: ["м", "а", " \_ ", "а", " \_ ", "а"] и **otvetMassiv.join("")** вернет строку "м а \_ а \_ а".

 Создадим функцию **printScore:**

### Обработка введенного ответа

Теперь нужно запросить у игрока ответ и убедиться, что он ввел одиночную букву.

**prompt** запрашивает у игрока ответ и сохраняет его в переменной **bukva**. Далее возможен один из четырех вариантов развития событий.

Первый вариант — если игрок нажмет кнопку «Отмена», **bukva** примет значение **null**. Этот вариант мы проверяем командой **if (bukva == null)**. Если это условие даст **true**, мы с помощью **break** выйдем из цикла.

Ключевое слово **break** можно использовать для немедленного выхода

из любого цикла, независимо от того, где именно внутри цикла это про-

исходит, или от того, выполняется ли на этот момент условие **while***.*

Второй и третий варианты — игрок не ввел ничего либо ввел

несколько букв. Если он просто нажал «ОК», ничего не вводя, в **bukva**

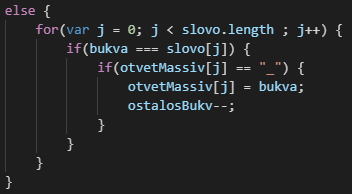
окажется пустая строка (""), а **bukva.length** вернет 0. Если же игрок ввел больше одной буквы, **bukva.length** вернет число больше 1. В следующем условии мы с помощью **else if (bukva.length !== 1)** обрабатываем эти варианты, то есть проверяем, что **bukva** содержит в точности одну букву. В противном случае мы отображаем диалог **alert**, гласящий: «Нужно ввести одну букву».

Четвертый вариант — игрок, как и положено, ввел одну букву. Тогда

мы должны обновить состояние игры в секции **else**.

### Обновление состояния игры

Если игрок ввел корректный ответ, мы должны обновить **otvetMassiv**

согласно этому ответу. Для этого добавим в тело **else** такой код:

Мы задали цикл **for** с новой переменной **j**, которая будет

менять значение от 0 до **slovo.length**, не включая само значение **slovo.**

**length**. В этом цикле мы проверяем каждую букву переменной **slovo**. Например, пусть в **slovo** находится строка "оладушек". Тогда при первом повторе цикла, когда **j** равно 0, **slovo[j]** вернет "о". При следующем повторе **slovo[j]**вернет "л", затем "а", "д","у", "ш", "е" и, наконец, "к".

Далеемы с помощью **if (slovo[j]=== bukva)** проверяем, совпадает ли текущая буква (**slovo[j]**) с ответом игрока. Если это так, то мы проверяем: «А не угадали ли мы уже эту букву?». Если эту проверку не сделать, то, если вы будете снова и снова вводить одну и ту же букву, которая есть в загаданном слове, **ostalosBukv** будет снова и снова уменьшаться.

Если на позиции совпавшей буквы до сих пор стоит «\_», то мы обновляем итоговый массив, добавляя туда букву командой **otvetMassiv[j] = bukva**. Для каждой буквы, совпадающей с ответом, мы обновляем соответствующую позицию итогового массива.

Этот код работает, поскольку переменную цикла **j** можно использовать

одновременно в качестве индекса в строке **slovo** и индекса в массиве **otvetMassiv.**

Помимо обновления **otvetMassiv** для каждого совпадения с **bukva**

требуется уменьшать **ostalosBukv** на 1. Мы делаем это в строке 40 командой **ostalosBukv**--;. Каждый раз, когда **bukva** совпадает с буквой из **slovo**, **ostalosBukv** уменьшается на 1, и, когда игрок угадает все буквы, **ostalosBukv** примет значение 0.

### Попытки

Сейчас наша «Виселица» позволяет угадывать буквы сколько угодно раз, до победного конца. Добавьте в программу переменную для учета попыток и завершайте игру, если игрок израсходовал все попытки. (Подсказка: проверяйте значение этой переменной в том же цикле **while**, где проверяется условие **ostalosBukv** **> 0**, а с помощью операции **&&** можно

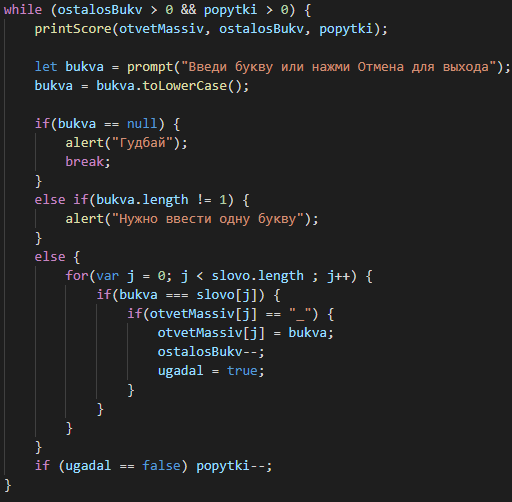
убедиться, что сразу два логических условия дают **true.**

Изначально мы присваиваем переменной **popytki** значение: длина слова – 1. Если программа загадала слово «телефон», то переменная **popytki** будет равна 6.

В цикле мы будем уменьшать переменную **popytki** каждый раз, когда игрок будет ошибаться, а в условии проверять: не стала ли она равна нулю, то есть цикл будет работать пока **popytki > 0**.

Если мы добавим такой код, мы не получим желаемого результата. Так как цикл **for** будет повторяться несколько раз, как и находящийся выше цикл **while.** А это значит, что переменная будет уменьшаться столько раз, сколько введённая буква будет отличаться от всех букв в слове.

Если программа загадала слово «телефон», то введя букву «а», запуститься цикл, в котором мы спросим: «Равна ли буква «а» букве «т»?» – нет, значит **poputki--**. На следующем произойдёт проверка с буквой «е», и мы получим такой же результат. Таким образом к последней букве, переменная **popytki**, которая изначально равнялась 6, будет равна -2.

Реализовать проверку на угаданную букву можно с помощью отдельной переменной **ugadal.** Эта переменная будет булевой и будет принимать либо значение **false** либо **true.**

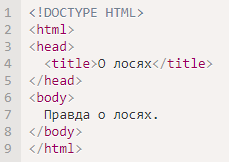
## Домашнее задание

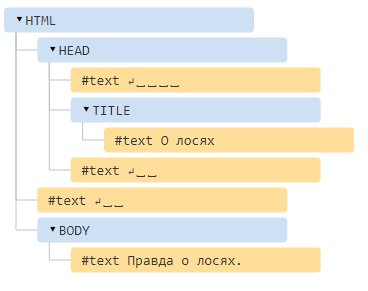
1. Добавить больше слов в словарь.
2. После окончания основного игрового цикла, поздравить игрока с победой и вывести слово полностью при помощи новой функции.
3. Стилизировать веб-страничку вместе с кнопкой.

# Урок 4 ****DOM-дерево****

Основой HTML-документа являются теги. В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом. Все эти объекты доступны при помощи JavaScript, мы можем использовать их для изменения страницы.

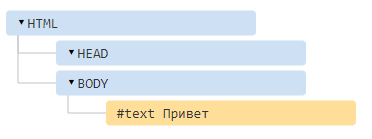
## Пример DOM

Начнём с такого, простого, документа:

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:

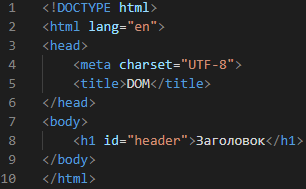
## Автоисправление

Если браузер сталкивается с некорректно написанным HTML-кодом, он автоматически корректирует его при построении DOM. Например, в начале документа всегда должен быть тег <html>. Даже если его нет в документе – он будет в дереве DOM, браузер его создаст. То же самое касается и тега <body>.

Например, если HTML-файл состоит из единственного слова "Привет", браузер обернёт его в теги <html> и <body>, добавит необходимый тег <head>, и DOM будет выглядеть так:

## Работа с DOM

### Поиск элементов на странице

HTML-элементу на странице можно присвоить уникальное имя или идентификатор, с помощью атрибута **id.** Например, у элемента h1 задан атрибут id:

Задав атрибуту id значение (в данном случае «header»), мы получаем возможность впоследствии найти этот конкретный заголовок по его id и что-нибудь с ним сделать, не затрагивая остальные элементы, даже если это другие заголовки h1. А сделать мы это сможем, потому что id – уникальное значение, которое может иметь только один элемент на веб-странице, то есть не могут быть два элемента с одинаковым id.

# Урок 5 ****Случайные числа****

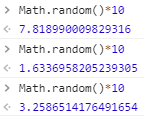
## Работа над домашним заданием

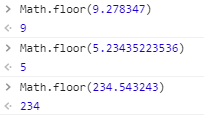
В 17-й строке расположено наше условие. Если ответ на наш вопрос – «Дом», то нам необходимо создать второй массив, который будет являться перевёрнутым первым. В 18-й строке мы создаём этот массив, а в 19-22 заполняем его элементами из первого массива. Мы забираем последний элемент из нашего изначального массива в переменную lastElement, а после добавляем этот элемент в конец второго массива.

## Случайные числа

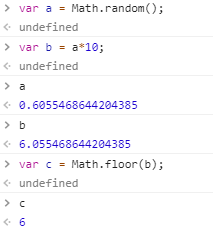
Многие из вас слышали о генераторах случайных чисел, но же такое случайное число и как его получить? На самом деле получить случайное число не так уж и легко. Человек не может сказать случайное число, потому что его выбор всегда с чем-то связан, на чём-то основан. Но JavaScript не человек, поэтому для него это не такая уж и сложная задача. Выбор случайного числа построен на разрешении некоторой математической функции. Для работы со всем, что связано с математикой у JavaScript есть библиотека Math. Чтобы получить необходимую функцию из этой библиотеки, нужно просто написать название – **Math** и через точку дописать название необходимого нам инструмента.

Первый метод из библиотеки Math, который мы рассмотрим – это метод для получения случайных чисел – **Math.random()**. Давайте разберёмся, как он работает, и посмотрим в консоли, что он возвращает при вызове. Если вызвать его несколько раз, сразу можно заметить, что он возвращает случайные числа от 0 до 1.

 Важно помнить, что Math.random() возвращает числа от 0 до 1, но не может вернуть 0 и 1. Но что, если нам нужно случайное число от 0 до 10, как быть в этом случае? На самом деле, сделать это очень легко, для этого нам нужно умножить полученное с помощью Math.random() значение на то число, до которого мы хотим выдавать случайные числа (в нашем случае 10).

 Научившись генерировать случайные числа до любого числа, мы всё равно не достигли того чего хотели, потому что наши числа не целые, и нам это не нравится. Но JavaScript и тут спешит нам на помощь. Оказывается, он умеет округлять числа с помощью метода **Math.floor()**. Этот метод отбрасывает все цифры после запятой, оставляя целое число, стоящее перед запятой.

 А поскольку мы можем объединять методы для получения необходимого нам результата, то давайте объединим два изученных метода из библиотеки Math.

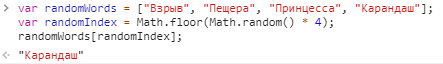
Давайте разберём этот пример. Сначала мы генерируем случайное число от 0 до 1 с помощью метода Math.random(). Потом мы умножаем полученное значение на 10, чтобы получить число от 0 до 10. И последнее наше действие – это округление полученного числа. Попробуйте сделать то же самое, но только поэтапно. Но для этого потребуется использовать переменные.

Ещё один маленький нюанс, который стоит знать при работе со случайными числами, связан с возможностью сгенерировать случайные числа в некотором диапазоне. Например, вам нужно выбрать случайное число между 10 и 20. Так как изученная нами конструкция выдаёт числа от нуля, то для решения поставленной задачи, нужно всего лишь прибавить к полученному результату недостающие число.

В первом случае мы можем получить числа меньше 10, а во втором будут числа только из диапазона 10 – 20.

## Случайные индексы

Давайте используем оба метода, чтобы получить случайный индекс. Нужно лишь умножить *Math.random()* на длину массива и затем округлить полученное число методом *Math.floor()*. Например, если в массиве четыре элемента, это можно сделать так:

При каждом запуске этот код будет возвращать случайное число от 0 до 3 (включая 0 и 3). Поскольку *Math.random()* всегда возвращает значение меньше 1, *Math.random() \* 4* никогда не вернет 4 или большее число. Используя это случайное число как индекс, можно получить случайный элемент массива:

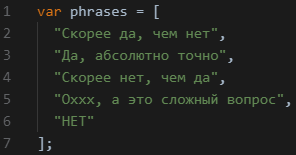
С помощью *Math.floor(Math.random() \* 4);* мы получили случайное число от 0 до 3. Сохранив это число в переменной *randomIndex*, мы использовали его как индекс для получения строки из массива *randomWords*.

В сущности, можно сделать этот код короче, избавившись от переменной *randomIndex*:

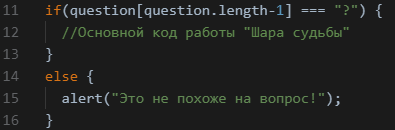
## Программа случайного выбора вариантов

Давайте попробуем применить наши знание на практике. Мы с вами создадим «Шар судьбы», который будет помогать отвечать на вопросы.

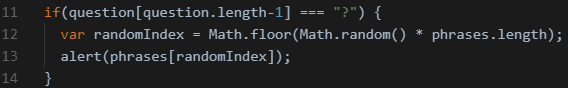
1. Спрашиваем у пользователя: «Спросите у шара, что пожелаете».
2. Проверяем правильно ли задан вопрос.
3. Выводим случайный ответ из, заранее подготовленного, массива ответов.

Первым делом создаём массив ответов.

Когда ответы готовы, нужно вывести сообщение для пользователя. Так как сообщение вопросительное, то нам нужно воспользоваться функцией prompt().

Следующий шаг – это проверка правильности полученного вопроса. Для этого нам нужно проверить, стоит ли вопросительный знак в конце предложения. И если он там отсутствует, сообщить пользователю, что он неправильно задал вопрос.

После отсеивания предложений не похожих на вопросительные, мы должны реализовать основную часть – работу «Шара судьбы».

 Для этого нам нужно создать сгенерировать случайный индекс, с помощью которого мы выберем случайный ответ из заготовленного массива. А после выведем полученный ответ пользователю.

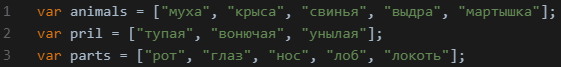
## Практическая работа

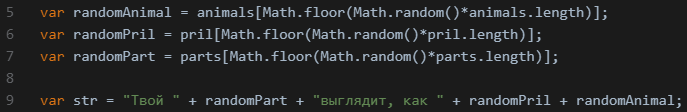
Доработать скрипт таким образом, чтоб можно было задавать вопрос шару судьбы несколько раз. Для этого нужно воспользоваться вопросом confirm() и циклом while.

*Полный скрипт в файле* **destiny.js**

## Генератор случайных дразнилок

Можно усовершенствовать код выбора вариантов, создав программу, которая при каждом запуске генерирует случайную дразнилку!

Создадим новый файл, в котором реализуем дразнилку. Для этого нам понадобиться три массива: массив животных, массив прилагательных и массив частей тела.

А после с помощью Math.random() и случайных индексов выберем из каждого массива по слову и соединим их в предложение.

А теперь самостоятельно сделайте вывод предложения.

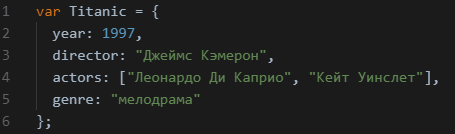
## Домашнее задание

1. Добавить в дразнилку больше слов.
2. Спрашивать имя пользователя в начале и добавлять его к предложению.
3. Усовершенствуйте генератор дразнилок, чтобы он создавал дразнилки такого типа: «У тебя [часть тела] еще более [прилагательное], чем [часть тела животного] у [животное]». Подсказка: нужно будет создать еще один массив.

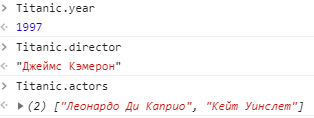
# Урок 6 ****Объекты****

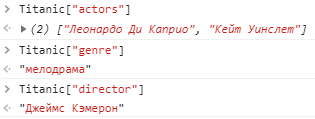
## Знакомство с объектами

В одном из прошлых уроков мы создавали массив, в котором хранили информацию о нашем классе, точнее, о каждом ученике нашего класса. Но для того, чтобы сохранить и имена, и возраст нам приходилось создавать разные массивы. А что, если нам нужно хранить сразу несколько характеристик какого-то одного объекта? Например, уже рассматриваемый нами раньше пример, с фильмами. Представьте, что нам нужно запомнить не только название фильма, но и год выпуска, жанр, имя режиссёра и список актёров. Тогда нам на помощь придут ***объекты***.

 В отличии от массивов, элементы которых соответствуют какому-то индексу, объекты представляют с помощью двух строк: первая строка – ***ключи***, а вторая ***значение***. Давайте рассмотрим на примере фильмов:

На данном фрагменте мы можем увидеть, как правильно создаются объекты (используя фигурные скобочки) и как создаются пары ключ-значение. Теперь, чтобы посмотреть режиссёра фильма Титаник или год премьеры, нужно воспользоваться одним из следующих способов.

 Первый способ – это обращение к ключу через точку.

 Второй способ – это обращение как к элементу массива:

 Теперь предположим, что вы хотите узнать, какие вообще ключи

есть у данного объекта. Для этого в JavaScript есть удобное средство —

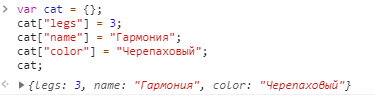
команда *Object.keys()*:

Object.keys(anyObject) возвращает массив, содержащий все ключи объекта anyObject.

## Добавление элементов объекта

Пустой объект похож на пустой массив, только вместо квадратных скобок при его создании используются фигурные:

**var object = {};**

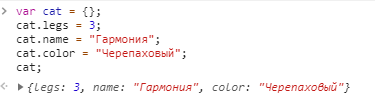
Добавлять элементы объекта можно так же, как элементы массива, — но используя строки вместо чисел:

Мы начали с пустого объекта под названием **cat**, а затем поочередно добавили к нему три пары «ключ-значение». Потом мы ввели **cat;** и браузер отобразил содержимое объекта.

В массивах элементы расположены строго один за другим: индекс 0 перед индексом 1, индекс 3 после индекса 2; однако в случае объектов неясно, как расположить элементы друг относительно друга. Должен ли

ключ color стоять перед legs или после? «Правильного» ответа на этот вопрос нет, поэтому объекты хранят свои ключи без конкретной очередности, в результате чего разные браузеры показывают ключи в разном порядке. Так что никогда не полагайтесь в своих программах на тот или иной порядок ключей.

### Добавление ключей через точку

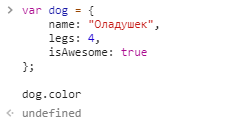
Новые ключи также можно добавлять через точечную нотацию. Давайте перепишем этим способом предыдущий пример, то есть создадим пустой объект и заполним его данными:

Если обратиться к несуществующему свойству объекта, JavaScript

вернет специальное значение *undefined*, сообщая таким образом: «здесь

ничего нет». Например:

Здесь мы определили три свойства объекта **dog**: *name*, *legs* и *isAwesome*. Свойства *color* среди них нет, поэтому **dog.color** возвращает *undefined*.



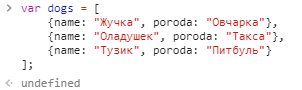
## Undefined и Null

Помимо специального значения *undefined* есть ещё одно специальное значение *null*. Давайте попробуем разобраться, что они означают и чем они отличаются.

Оба этих значения означают «пусто», но смысл этого в обоих случаях немного различается. JavaScript использует значение *undefined*, когда не может найти иного значения. Например, если, создав новую переменную, вы не присвоите ей значение с помощью оператора =, ее значением будет *undefined,* такое вы уже видели много раз.

А значение null обычно используется, чтобы явно обозначить — «тут пусто».

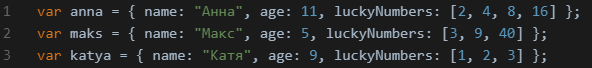
## Массивы объектов

До этого момента мы рассматривали только массивы и объекты, в которых содержатся данные простых типов, такие как числа и строки. Однако ничто не мешает сделать элементом массива или объекта другой массив или объект. Например, так может выглядеть массив с объектами, описывающими собак:

Получить сведения о первой собаке можно уже известным нам способом — указав индекс в квадратных скобках: **dinosaurs[0];**

А если нужно только имя первой собаки, достаточно указать ключ объекта в еще одних квадратных скобках, следом за индексом или воспользоваться точечной нотацией:

### Массив друзей

Давайте рассмотрим более сложный пример — массив объектов со сведениями о друзьях, где в каждый из объектов вложено по еще одному массиву. Сначала создадим объекты, а затем поместим их в массив.

Мы создали три объекта, сохранив их в переменных. У каждого из этих объектов есть по три свойства: *name, age* и *luckyNumbers*. Каждому ключу *name* соответствует строковое значение, ключу *age* — числовое, а ключу *luckyNumbers* — массив, содержащий несколько чисел.

Теперь создадим массив друзей:

Итак, в переменной friends находится массив с тремя элементами: anna, maks и katya (каждый из них является объектом). Мы можем получить любой из объектов по его индексу в массиве:

Здесь мы извлекли из массива второй объект, maks (по индексу 1). Вместо массива luckyNumbers Chrome напечатал Array(3) что означает «это массив с тремя элементами» (можно изучить содержимое этого массива с помощью Chrome)

Также мы можем получить значение, хранящееся в объекте, указав индекс объекта в квадратных скобках, поставив точку и написав соответствующий ключ:

Этот код запрашивает элемент по индексу 0 (что соответствует переменной anna), а затем — свойство этого объекта, хранящееся по ключу "name" (это "Анна"). Можно даже получить значение из массива, находящегося в объекте, который, в свою очередь, находится в массиве friends:

friends[0]

**↓**

var friends **=** [anna, maks, katya];

friends[0].luckyNumbers[0]

**↓**

{ name: "Анна", age: 11, luckyNumbers: [2, 4, 8, 16] };

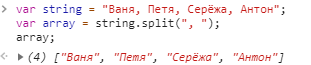
**↑**

friends[0].luckyNumbers[1]

Объекты тоже можно исследовать в консоли, узнав все необходимые свойства, элементы вложенных массивов и их длину.

## Практическая работа – Учёт долгов

Перед тем как приступить к написанию скрипта, позволяющего вести учёт долгов, познакомимся с ещё одним методом для работы со строками – **split().** Так как строки – это частный случай массивов (строка – это массив, в котором в каждой ячейке отдельный символ строки), многие методы для работы со строками и с массивами совпадают. Но иногда всё-таки бывает необходимо преобразовать строку в массив, причём, таким образом, чтоб в каждой ячейке полученного массива были не отдельные символы, а целые слова или какие-либо другие блоки текста из исходной строки.

Для этого используют метод **split("разделитель"),** который преобразует строку в массив, таким образом, что в нулевую ячейку помещается кусок строки до разделителя, в первую – до следующего вхождения разделителя, и так далее.

А теперь реализуем учёт долгов:

1. Создаём объект следующего вида:
2. Создаём массив из сформированных объектов.
3. Выводим сообщение: "Чтобы добавить должника напишите его имя и сумму, которую он должен, через пробел".
4. Добавляем нового должника используя метод split().
5. Выводим список должников.

*(дать возможность детям самим реализовать пункты 1 и 2)*

*(Пример скрипта в файле* **debtors.js***)*

## Домашнее задание

Доработать скрипт «Учёт долгов» таким образом:

1. Если во время вопроса о занесении нового должника нажать кнопку «Отмена», то мы получим ошибку: Cannot read property 'split' of nul. Так как в переменной содержащей ответ и вправду ничего нет. Чтобы избежать такой ошибки, мы добавим условие, в котором будем проверять: не равняется ли наша строка null. Если не равняется, то добавим нового должника, иначе – не добавим.
2. После вывода всех должников, вспомните тему Случайные числа и сгенерируйте случайное число, которое будет являться номером человека, которому вы простите долг.
3. Установите значение долга выбранного человека в 0 и выведите список ещё раз.
4. **(повышенной сложности)** Посчитайте общую сумму, которую вам должны.

# Урок 7 Основы HTML

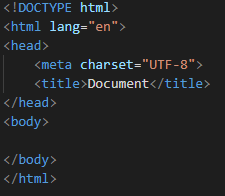
## Знакомство с HTML

Встроенная в браузер JavaScript-консоль, которой мы до сих пор пользовались, хороша, когда нужно протестировать небольшой фрагмент кода, но для создания более масштабных программ понадобится чуть более гибкое и универсальное средство — вроде веб-страницы со встроенным JavaScript-кодом. В этой главе мы как раз и научимся создавать несложные странички на языке HTML.

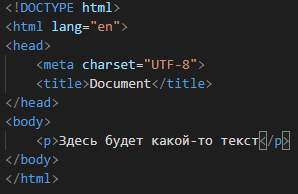
Гипертекстовый язык разметки HTML предназначен специально для создания веб-страниц. Слово гипертекстовый означает, что фрагменты текста связаны между собой гиперссылками — то есть ссылками в документе на другие объекты. А язык разметки — это способ встраивать в текст дополнительную информацию. Разметка указывает программам (таким как браузер), как отображать текст и что с ним делать.

Мы будем создавать HTML-документы в кроссплатформенном (совместимом с Windows, Mac OS и Linux) редакторе Sublime Text, с которым вы уже хорошо знакомы.

Переходим к созданию первой html-странички. Для этого создаём новый файл и сразу же сохраняем его с названием **index.html**.

Пишем в нём следующее или вводим восклицательный знак и нажимаем Tab, если установлен плагин Emmet.

Отлично, мы создали нашу первую веб-страницу! Теперь можно посмотреть на результат наших усилий – сохраняем файл index.html и открываем его в браузере.

На данный момент мы создали пустую страницу. Добавим какого-нибудь контента, например, текст.

Ещё раз зайдём в браузер и посмотрим на результат. У нас появился текст – отлично. Теперь давайте разбираться, что же мы написали и что всё это означает.

Любой HTML документ состоит из тегов. **Тег** — это специальное зарезервированное слово, заключённое в угловые скобки, например, **<body>**). Тег — основная составляющая HTML: им код начинается, им же заканчивается, внутри тегов заключается отображаемая на веб-странице информация. Записывать их рекомендуется в нижнем регистре, то есть обычными маленькими буквами: не **<HEAD>**, а **<head>**.

Большинство HTML-тегов являются парными: есть открывающий (например, **<html>**) и закрывающий тег, которого от открывающего отличает слеш (**/**) после первой угловой скобки (например, **</html>**). Всё, что находится внутри пары тегов, называется их **содержимым**.

***<Открывающий\_тег>Содержимое</Закрывающий\_тег>***

Бывают и непарные (одиночные) теги, которые называются **метками**. Их, в отличие от парных тегов, не нужно закрывать, потому что они не работают с содержимым, а выполняют какую-то функцию сами по себе. Пример одиночного тега — **<br>**. Он устанавливает перенос текста на следующую строку.

Если проводить параллели с русским языком, то можно сказать, что парные теги — это кавычки или скобки, влияющие на свойства заключённого в них текста (попробуйте не закрыть кавычки вовремя), а одиночные теги (метки) — это знаки препинания (восклицательный, вопросительный или точка).

Любой тег состоит из:

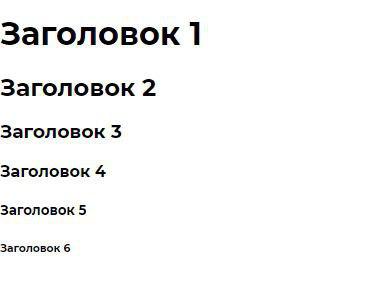
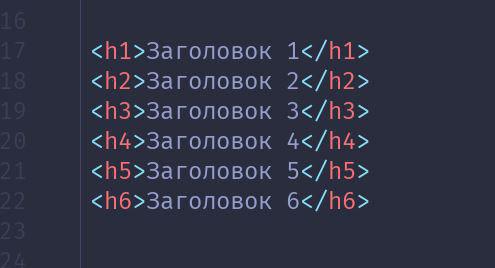
* Открывающей угловой скобки (**<**).
* Специального слова (имени тега). Например, **h1**, **p**,**b**.
* Закрывающей угловой скобки (**>**).

## Основные HTML теги

Начнём по порядку:

* **<!DOCTYPE>** - тег, который определяет тип документа. Он необходим для того, чтобы браузер понимал, как интерпретировать текущую веб-страницу. Этот тег одиночный и указывается он всегда в самом начале HTML-документа.
* **<html>** - является главным тегом всей страницы, потому что включает в себя всё её содержимое. Пишется этот тег, как правило, сразу после <!DOCTYPE>. Тег <html> парный и закрывающий тег </html> ставится в самом конце HTML-документа.
* **<head>** - тег, который служит своего рода хранилищем для других head-элементов и помогает браузеру при работе с данными. Напрямую его содержимое не отображается на веб-странице. Этот тег и его содержимое очень важны для SEO-оптимизации. Внутри этого тега могут содержаться другие теги: <base>, <basefont>, <bgsound>, <link>, <meta>, <script>, <style>, <title>.
* **<meta>** - это один из тегов, который размещается внутри <head>. Он определяет метатеги, предназначенные для хранения информации для браузера и поисковой системы. Метаданные не отображаются на странице и используются для описания страницы, ключевых слов, автора документа.
* **<title>** - заголовок документа, который отображается не на странице, а во вкладке браузера. Это обязательный элемент HTML-документа, он может встречаться только один раз во всем документе и содержится внутри <head>.
* **<body>** - «Body» с английского означает «тело». И в HTML этот тег используется как раз в этом значении. То есть этот тег определяет всё тело документа, хранит весь его контент и отображает на странице. <body> - парный тег, но использование закрывающего </body> не обязательно, хотя считается хорошим тоном.

## Теги заголовков

Это элементы HTML-документа, которые имеют огромное значение. Они помогают упорядочить всю информацию, формируют структуру документа. В теги заголовков включаются 6 тегов различного уровня важности: <h1>, <h2>, <h3>, <h4>, <h5>, <h6>.

Все теги заголовков являются парными и требуют обязательного использования закрывающих тегов.

## Форматирование и работа с текстом

* **<p>** - текстовый абзац. Блочный парный тег, пишется всегда с новой строки.
* **<br>** - тег, используемый для разрыва строки в том месте, где он стоит.
* **<blockquote>** - текст, выделенный этим тегом, представляет собой отдельный выделенный блок с отступами. Как правило, используется для выделения длинных цитат в HTML-документе.
* **<q>** - тег, используемый для выделения кратких цитат в тексте. По умолчанию содержание тега заключается в кавычки.
* **<b>** - задаёт шрифту жирное начертание. Возможно использование вместе с другими тегами, задающими параметры тексту.
* **<i>** - курсивное написание содержащегося внутри текста.
* **<small>** - уменьшает шрифт на 1 единицу. Текст в HTML имеет размер от 1 до 7, по умолчанию шрифту задается размер 3. А с помощью тега <small> текст уменьшится до 2. Допускается неоднократное использование этого тега, но размером меньше 1 текст не может быть.
* **<u>** - тег, который делает шрифт подчеркнутым.
* **<s>** - зачеркнутый шрифт.
* **<sub>** - подстрочное написание шрифта.
* **<sup>** - надстрочное написание шрифта.
* **<ol>** - создание упорядоченного нумерованного списка. Элементы списка должны начинаться с тега <li>. Обязательно использование закрывающего тега.
* **<ul>** - тоже список, только маркированный. Требования к элементам и закрытию тега как и у <ol>.
* **<a>** - этот тег делает из текста гиперссылку. Обязательно использование закрывающего тега.

## Встраиваемые элементы HTML

Добавлением и управлением аудиозаписей на веб-странице занимается тег <audio>. Путь к файлу прописывается через тег <source>. Это парный тег, который обязательно должен быть закрыт.

Суть добавления и управления видеозаписями такая же, как и с тегом <audio>. Только осуществляется это через тег <video>.

К аудио и видео в HTML можно добавлять субтитры с помощью тега <track>.

Для отображения изображений формата JPEG, PNG, GIF на веб-странице используется тег <img>.

## Практическая работа

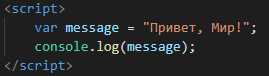
Создать свою первую страничку с использованием наибольшего количества тегов из предложенных.

## Домашнее задание

Создать своё портфолио. Пример в файле **index.html.**

# Урок 8 HTML и JavaScript

## Внедрение JavaScript-кода в HTML

Давайте создадим новую html страничку и добавим ей заголовок. Теперь, используя парный тег **<script>,** напишем следующие пару строчек:

Содержимое большинства элементов HTML отображается на страничке, однако то, что находится внутри тегов **script**, считается JavaScript-кодом и передается для выполнения интерпретатору JavaScript.

Выполнение JavaScript, встроенного в HTML, заметно отличается от запуска кода в консоли. Введенный в консоли код выполнится при первом же нажатии ENTER, после чего вы увидите результат сработавшей команды. Однако код, встроенный в веб-страницу, выполняется сразу и целиком, от верхних строк к нижним, причем в консоль ничего автоматически не выводится — если мы не попросим браузер об этом отдельно.

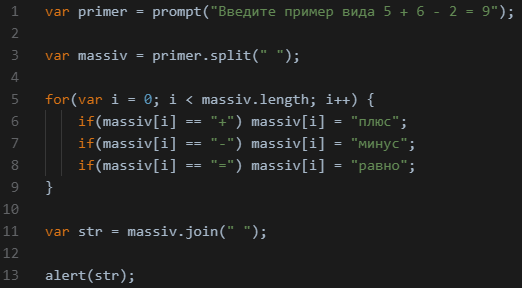
Для вывода в консоль можно воспользоваться командой console.log — это поможет следить за ходом выполнения программы. Метод console.log принимает любое значение и печатает это значение в консоли. Например, загрузив в браузер наш последний пример, вы увидите в консоли вот что (разумеется, если она открыта): ***Привет, Мир!***

## Подключение стороннего скрипта

Это был первый способ внедрения JavaScript кода. Его удобно использовать при начале изучения языка JavaScript или при работе с очень маленькими скриптами (как в нашем примере). Но в повседневной практике обычно используют второй способ внедрения.

Разместите между тегами **<head>** следующий код:

**<script src="myscript.js"></script>**

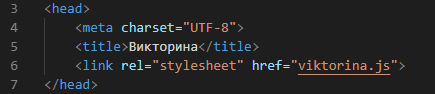
В той же папке где хранится ваша HTML-страница, создайте файл с именем **myscript.js**, в данный файл вы можете записывать любой JavaScript-код, но уже без использования тегов <script>. Например:

Рассмотрим подробнее данный пример:

В первой строчке мы выводим с помощью prompt() правила введения примеров. Введённый пример записывается в переменную primer. В третей строке, с помощью метода split(), строка, которая находится в primer, превращается в массив, где каждый элемент массива – это отдельный символ (Это делать необязательно, так как к строке можно применять те же методы, что и к массиву. Но для повторения методов split() и join() можно использовать такой вариант). В строчках 5-9, используя цикл, мы каждый элемент проверяем: не является ли он математическим знаком. Если это математический знак, то заменяем его словом. В 11-й строчке мы собираем массив в строчку, разделяя элементы пробелом. И в 13-й строке выводим полученную строчку с примером.

## Практическая работа – Викторина

Игра-викторина будет заключаться в том, что 3 участника будут отвечать на заранее составленные вопросы, набирать очки, а в конце подведутся итоги и выведется рейтинговая таблица.

Первым делом создаём HTML файл и подключаем к нему скрипт viktorina.js, который мы создадим позже. Будьте внимательны, здесь мы подключаем скрипт новым способом, используя тег **<link>**

В дальнейшем, созданный нами файл **index.html**, не пригодиться. Сохраняем его, закрываем и создаём новый файл – **viktorina.js**.

(Скрипт прописан в файле **viktorina.js** с комментариями)

## Домашнее задание

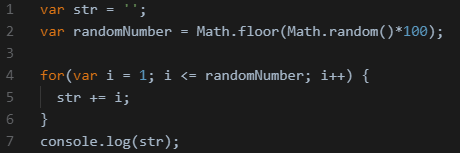
1. Добавить вопросы в викторину (не забыть поменять ограничения в циклах в виде ‘название массива’.length)
2. Добавить возможность выбрать количество игроков.
3. Закомментировать весь скрипт.

# Урок 9 Опять строки

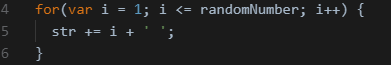
## Работа со строками в циклах

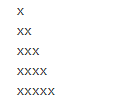
Для повторения и закрепления пройденного материала, сегодня мы опять будем работать со строками и циклами. Первое чем мы займёмся – это сформируем строку чисел от единицы до случайного числа, используя метод генерации случайных чисел.

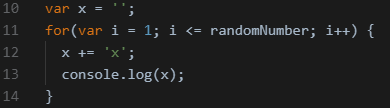
Первое что нам нужно сделать – создать пустую строчку. После – выбрать случайное число в пределах 100. И с помощью цикла добавить все числа от единицы до выбранного нами случайного в исходную строчку.



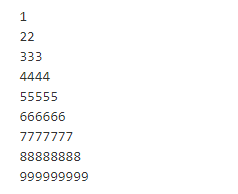
Выводить полученную строчку будем, не с помощью привычного нам alert(), а с помощью console.log(). Тогда мы сможем посмотреть результат работы нашего скрипта в браузерной консоли.

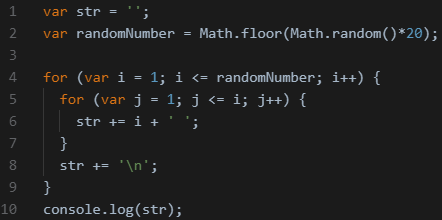
В итоге выполнения нашего скрипта, мы получили сплошную строчку чисел. Чтобы это исправить, давайте после каждого числа будем добавлять пробел.

 Теперь давайте закомментируем кусочек кода, отвечающий за вывод чисел, к нему мы позже вернёмся. Создайте самостоятельно следующую картинку.

Только количество рядов в такой пирамиде должно равняться вашему возрасту.

В данном примере вместо возраста используется всё та же переменная randomNumber, которая хранит случайно сгенерированное число.

Теперь соединим эти два цикла, чтобы получить следующую картинку.

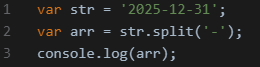
Задача аналогична предыдущим, только теперь мы будем использовать цикл внутри цикла. Чтобы пирамида не была очень большой, будем генерировать случайное число до 20.

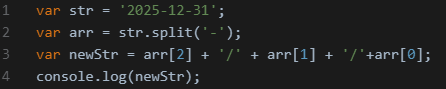
Самостоятельно переделайте этот скрипт таким образом, чтоб пирамида состояла из случайных чисел.

## Методы split() и replace()

Следующая наша задача – это перевод даты из одного формата в другой.

**Задача.** В переменной **date** лежит дата в формате **'2025-12-31'**. Преобразуйте эту дату в формат **'31/12/2025'**.

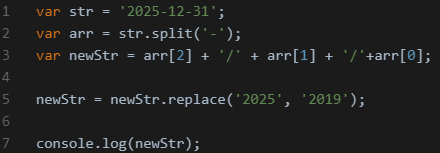
**Решение:** с помощью метода split разобьем нашу строку **'2025-12-31'** в массив по разделителю **'-'**, при этом в нулевом элементе окажется год, в первом - месяц, во втором - день:

Получим массив ['2025', '12', '31']. Теперь, обращаясь к разным элементам массива по их индексам, сформируем нужную нам строку:

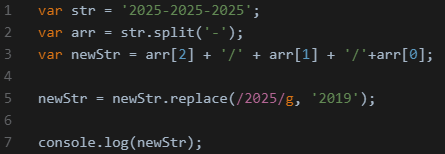
Теперь познакомимся с новым методом **replace().**

Метод **replace** осуществляет поиск и **замену** частей строки. **Первым** параметром принимается подстрока, *которую заменяем*, а **вторым** – подстрока, *на которую заменяем*.

**строка\_в\_к-й\_заменяем.replace(что заменяем, на что заменяем);**

Заменим получившейся дате **2025** год на **2019:**

К сожалению, данный метод заменяет только первое совпадение. Если мы поменяем нашу исходную строку на **'2025-2025-2025',** то изменится только первое совпадение.

Чтобы функция заменила все совпадения, следует воспользоваться так называемым **глобальным поиском**:

## Работа с регистром символов

Метод **toUpperCase** производит преобразование строки в **верхний регистр** (из маленьких букв делает большие). При этом возвращается новая строка, а исходная строка не меняется.

**Новая\_строка = Старая\_строка.toUpperCase();**

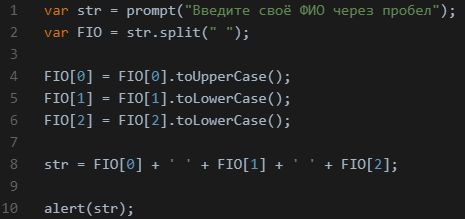
Попробуйте использовать этот метод на простых примерах. Создайте какую-нибудь строчку и преобразуйте её в верхний регистр.

Метод **toLowerCase** производит преобразование строки в **нижний регистр** (из больших букв делает маленькие). При этом также возвращается новая строка, а исходная строка не меняется.

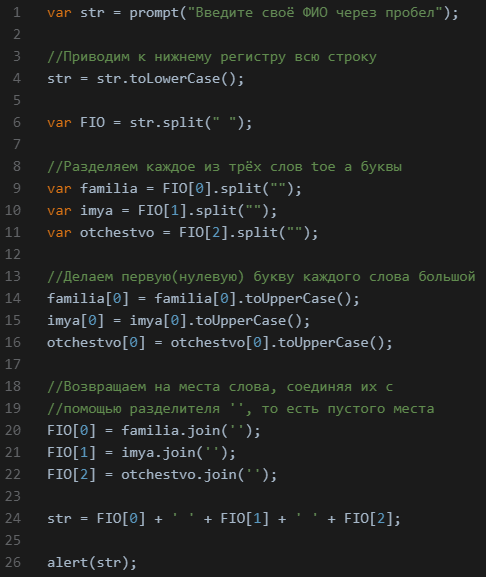
**Новая\_строка = Старая\_строка.toLowerCase();**

**Задача.** Пользователь вводит Фамилию, Имя и Отчество через пробел. Вывести Фамилию большими буквами, а Имя и Отчество маленькими.

**Решение. (дать детям возможность решить самостоятельно)**

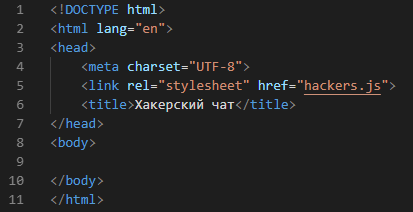
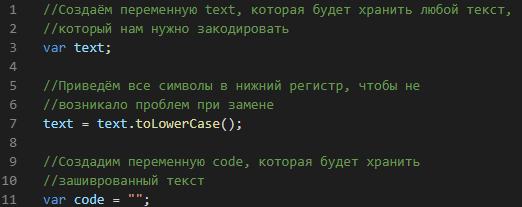
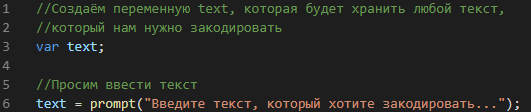
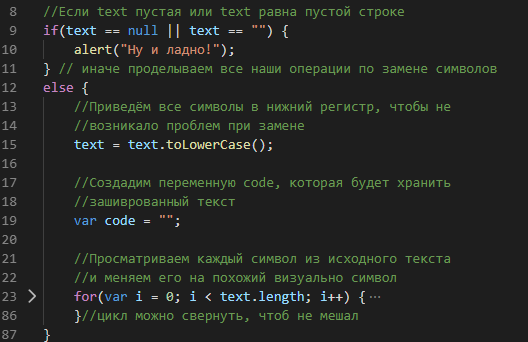
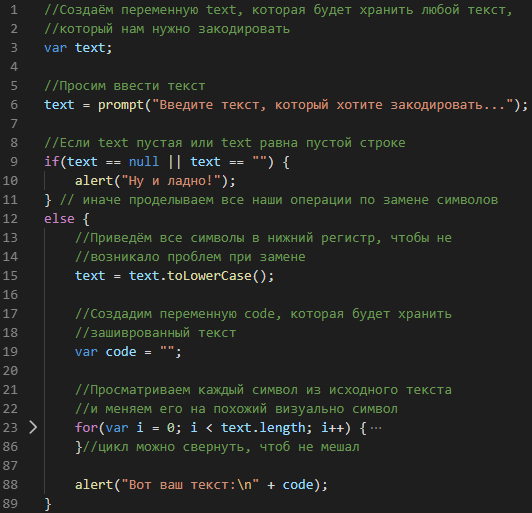


Самостоятельно переделать задачу таким образом, чтоб все буквы кроме первой (в каждом слове) были маленькие, а первые буквы – большими.

В качестве подсказки можно сказать, что для этого нужно использовать методы **split()** и **join()**.

## Практическая работа – Хакерский чат

С помощью изученных ранее циклов, а также различных методов работы со строками, мы создадим чат, в котором каждое сообщение будет кодироваться. Так как существует множество скриптов для поиска сообщений, по ключевым словам, наша кодировка будет необычной и, разумеется, защищённой от таких поисков. При этом она не будет предполагать наличия у собеседников ключей или специальных программ для дешифровки. Она итак будет понятна всем собеседникам.

1. Создаём файлы hackers.html и hackers.js.
2. Создаём основную разметку в файле hackers.html и подключаем в него hackers.js.
3. Создаём переменные, которые будут хранить начальный текст и зашифрованный
4. Создаём правила замены символов, в которых указываем «какой символ на какой менять». (Дать детям возможность самостоятельно придумать символы для замены)
5. Предложим пользователю ввести текст, который он хочет закодировать, а результат запишем в переменную **text.**
6. Добавим проверку текста, на тот случай, если пользователь нажал «Отмена» при вводе текста или ничего не ввёл.
7. Выводим закодированный текст

*(Полный скрипт в* **hackers.js***)*

## Домашнее задание

1. Спрашивать имя в самом начале и выводить его вместе с закодированным текстом в конце.
2. Проверять, вдруг пользователь не ввёл имя или ввёл пустое, тогда давать ему имя «Аноним».
3. Если в исходном сообщении есть ваше имя (в любом регистре), то заменить его на «Повелитель».

# Урок 10 Создание игры «Виселица»

«Виселица» — игра на угадывание слов. Один игрок выбирает слово, а второй пытается его отгадать. Например, если первый игрок загадал слово *КАПУСТА* , он изобразит семь «пустых мест», по одному на каждую букву слова:

\_ \_ \_ \_ \_ \_ \_

Второй игрок старается отгадать это слово, называя буквы. Каждый

раз, когда он угадывает букву, первый игрок заполняет пустоты, вписывая ее везде, где она встречается. Например, если второй игрок назвал букву «А», первый должен вписать все «А» для слова КАПУСТА, вот так:

\_ А \_ \_ \_ \_ А

Если второй игрок назовет букву, которой нет в слове, у него отнимается очко, а первый игрок рисует руку, ногу или другую часть тела

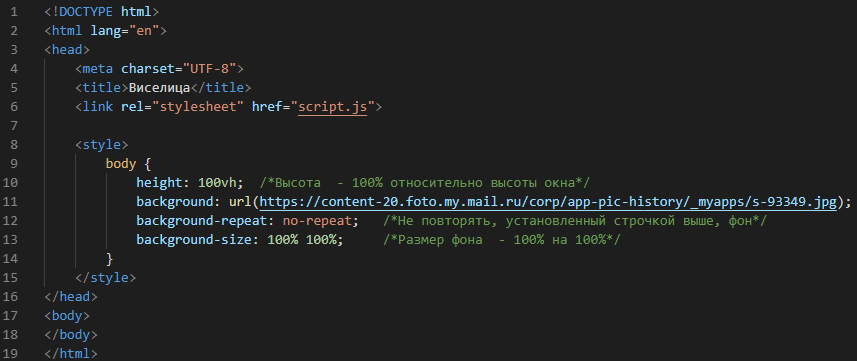
человечка. Если первый игрок закончит рисовать человечка раньше, чем второй угадает все буквы, второй игрок проиграл.

В нашем варианте «Виселицы» JavaScript будет выбирать слово, а игрок-человек — отгадывать буквы. И рисовать человечка наша программа не будет, поскольку мы пока не знаем, как это делается.

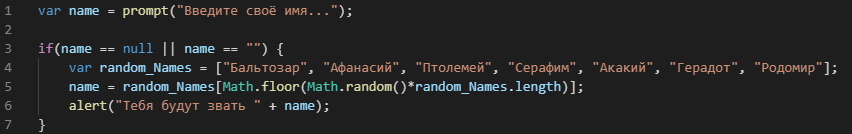
## Взаимодействие с игроком

Для этой игры нам нужно, чтобы игрок (человек) мог каким-то образом вводить в программу свои ответы. Один из способов это сделать — открывать диалоговое окно (в JavaScript оно называется *prompt* ), в котором игрок может что-нибудь напечатать.

Сначала создадим файлы проекта. Файл **index.html**. Создадим в нём основную разметку, подключим наш скрипт **script.js** и добавим немного стилей, а именно, поставим картинку на задний фон.



Создайте и откройте файл **script.js,** а потом самостоятельно создайте обращение к игроку с запросом: «Введите своё имя».

После этого, как и в прошлых проектах, мы сделаем проверку на тот случай, если пользователь нажал «Отмена» или не ввёл ничего.

## Проектирование игры

Прежде чем перейти к созданию игры «Виселица», давайте подумаем

о ее структуре. Нам нужно, чтобы программа умела выполнять следующие действия:

1. Случайным образом выбирать слово.

2. Запрашивать у игрока вариант ответа (букву).

3. Завершать игру по желанию игрока.

4. Проверять, является ли введенный ответ буквой.

5. Вести учет угаданных букв.

6. Показывать игроку, сколько букв он угадал и сколько еще пред-

стоит угадать.

1. Завершать игру, если слово отгадано.

Все эти действия, кроме первого и последнего (выбор слова и завершение игры), нужно выполнять многократно, причем заранее неизвестно, сколько раз (это зависит от ответов игрока). И, как мы теперь

знаем, если требуется повторять какие-то действия, значит, в программе

нужен цикл.

Однако в нашем списке действий ничего не говорится о том, что и когда должно происходить. Чтобы выяснить этот вопрос и лучше представить себе структуру будущей программы, мы можем воспользоваться

*псевдокодом* .

### Используем псевдокод для проектирования игры

Псевдокод — удобный инструмент, который программисты часто

используют при проектировании программ. Слово «псевдокод» означает

«ненастоящий код». Хотя в псевдокоде есть циклы и условия, в целом программа описывается обычным человеческим языком. Чтобы разобраться, что это значит, давайте посмотрим на описание нашей игры в псевдокоде:

*(Записать этот псевдокод вместе с детьми в отдельный файл)*

Выбрать случайное слово

Пока слово не угадано {

Показать игроку текущее состояние игры

Запросить у игрока вариант ответа

Если игрок хочет выйти из игры {

Выйти из игры

}

Иначе Если вариант ответа — не одиночная буква {

Сообщить игроку, что он должен ввести букву

}

Иначе {

Если такая буква есть в слове {

Обновить состояние игры, подставив новую букву

}

}

}

Поздравить игрока с победой — слово угадано

Как видите, это не программный код, который может выполнить компьютер. Однако такая запись дает нам представление о структуре программы прежде, чем мы перейдем к написанию кода и выяснению мелких деталей, например, *как* именновыбирать случайное слово.

### Отображение состояния игры

Одна из строк нашего псевдокода гласит: «Показать игроку текущее

состояние игры». Для игры «Виселица» это означает подставить в слово

угаданные игроком буквы, а также показать, какие буквы осталось угадать. Как мы будем это делать? В сущности, можно хранить состояние игры тем же способом, что и в обычной «Виселице»: в виде последовательности «пустых мест», которые мы будем заполнять по мере того, как игрок угадывает буквы.

Мы сделаем это с помощью массива «пустых мест» – по одному

элементу для каждой буквы в слове. Назовем этот массив «итоговым массивом» и будем по ходу игры заполнять его угаданными буквами. А каждое из «пустых мест» представим в виде строки со знаком подчеркивания: "\_".

Сначала наш итоговый массив будет просто набором «пустых мест», количество которых равно количеству букв в загаданном слове. Например, если загадано слово «рыба», массив будет выглядеть так:

[" \_ ", " \_ ", " \_ ", " \_ "]

Если игрок угадает букву «ы», мы заменим второй элемент на «ы»:

[" \_ ", "ы", " \_ ", " \_ "]

А когда игрок угадает все буквы, массив примет вид:

["р", "ы", "б", "а"]

Также нам понадобится переменная для хранения количества букв, которые осталось угадать. Для каждого вхождения верно угаданной буквы эта переменная будет уменьшаться на 1, и, когда она примет значение 0, мы поймем, что игрок победил.

### Проектируем игровой цикл

Основная часть игры будет располагаться внутри цикла while (в нашем

псевдокоде этот цикл начинается со строки «Пока слово не угадано»).

В цикле мы будем отображать текущее состояние игры (то есть слово,

поначалу представленное одними знаками подчеркивания), запрашивать у игрока вариант ответа (и проверять, действительно ли тот ввел одиночную букву), а также обновлять итоговый массив, подставляя введенную букву, если она действительно есть в слове.

Практически все компьютерные игры организованы в виде того или

иного цикла, нередко структурно похожего на цикл нашей «Виселицы».

В целом игровой цикл выполняет следующие задачи:

1. Принимает ввод от игрока.

2. Обновляет состояние игры.

3. Показывает игроку текущее состояние игры.

Такого рода цикл применяется даже в играх, где непрерывно что-то

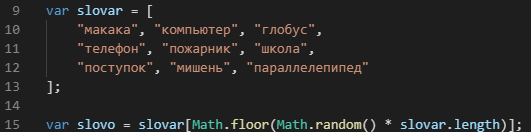
меняется, – просто он выполняется *очень* быстро. В случае нашей «Виселицы» программа запрашивает у игрока вариант ответа, обновляет

итоговый массив (если ответ верный) и отображает новое состояние итогового массива.

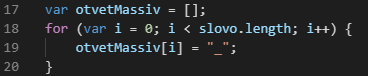
Если игрок угадает все буквы в слове, мы должны показать ему законченное слово, а также вывести сообщение, поздравляющее с победой.

## Программируем игру

Теперь, когда у нас есть представление о структуре игры, можно переходить к написанию кода.

 Первым делом нам нужно выбрать случайное слово:

Далее создадим пустой массив под названием **otvetMassiv** (итоговый

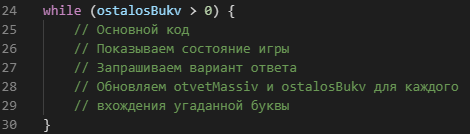
массив) и заполним его символами подчеркивания (\_), количество которых соответствует количеству букв в загаданном слове.

Когда цикл завершится, длина **otvetMassiv**будет такой же, как длина слова. Например, если было выбрано слово «макака» (в котором шесть букв),

**otvetMassiv** примет вид [" \_ ", " \_ ", " \_ ", " \_ ", " \_ ", " \_ "] (шесть знаков подчеркивания).

Наконец, создадим переменную **ostalosBukv**, приравняв ее к длине загаданного слова. Эта переменная понадобится, чтобы отслеживать количество букв, которые осталось угадать. Каждый раз, когда игрок угадает букву, мы будем уменьшать значение этой переменной: на 1 для каждого вхождения буквы в слово.

### Программируем игровой цикл

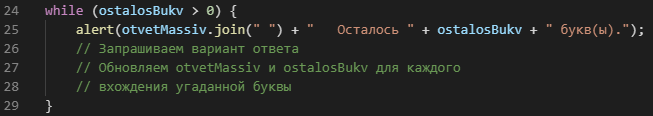
 Основа игрового цикла выглядит так:

Мы используем цикл **while**, который будет повторяться до тех пор, пока условие **ostalosBukv > 0** дает **true**. В теле цикла надо будет обновлять **ostalosBukv** для каждого правильного ответа игрока. Когда игрок угадает все буквы, **ostalosBukv** примет значение 0, и цикл завершится.

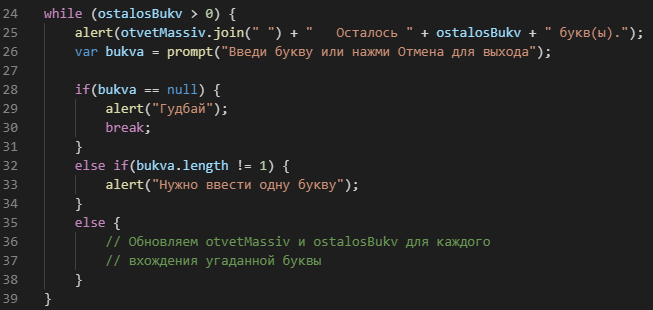
### Отображение состояния игры

Первым делом в теле игрового цикла нужно показать игроку его имя и текущее состояние игры.

Мы делаем это, объединяя элементы **otvetMassiv** в строку с пробелом в качестве разделителя, а затем с помощью **alert**показываем эту строку.

Например, пусть загадано слово «макака» и игрок угадал буквы «м» и «а». Тогда итоговый массив примет вид: ["м", "а", " \_ ", "а", " \_ ", "а"] и **otvetMassiv.join("")** вернет строку "м а \_ а \_ а".

### Обработка введенного ответа

Теперь нужно запросить у игрока ответ и убедиться, что он ввел одиночную букву.

В строке 26 **prompt** запрашивает у игрока ответ и сохраняет его в переменной **bukva**. Далее возможен один из четырех вариантов развития событий.

Первый вариант — если игрок нажмет кнопку «Отмена», **bukva** примет значение **bukva**. Этот вариант мы проверяем в строке 28 командой

**if (bukva == null)**. Если это условие даст **true**, мы с помощью **break** выйдем из цикла.

Ключевое слово **break** можно использовать для немедленного выхода

из любого цикла, независимо от того, где именно внутри цикла это про-

исходит, или от того, выполняется ли на этот момент условие **while***.*

Второй и третий варианты — игрок не ввел ничего либо ввел

несколько букв. Если он просто нажал «ОК», ничего не вводя, в **bukva**

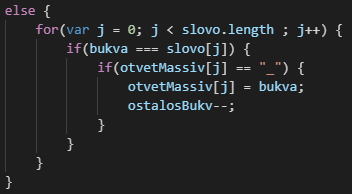
окажется пустая строка (""), а **bukva.length** вернет 0. Если же игрок ввел больше одной буквы, **bukva.length** вернет число больше 1. В строке 32 мы с помощью **else if (bukva.length !== 1)** обрабатываем эти варианты, то есть проверяем, что **bukva** содержит в точности одну букву. В противном случае мы отображаем диалог **alert**, гласящий: «Нужно ввести одну букву».

Четвертый вариант — игрок, как и положено, ввел одну букву. Тогда

мы должны обновить состояние игры — это происходит в строке 35, в секции **else**.

### Обновление состояния игры

Если игрок ввел корректный ответ, мы должны обновить **otvetMassiv**

согласно этому ответу. Для этого добавим в тело **else** такой код:

В строке 36мы задали цикл **for** с новой переменной **j**, которая будет

менять значение от 0 до **slovo.length**, не включая само значение **slovo.**

**length**. (Мы назвали переменную **j**, поскольку имя i уже использовано в предыдущем цикле **for**.) В этом цикле мы проверяем каждую

букву переменной **slovo**. Например, пусть в **slovo** находится строка

"оладушек". Тогда при первом повторе цикла, когда **j** равно 0, **slovo[j]**

вернет "о". При следующем повторе **slovo[j]**вернет "л", затем "а", "д",

"у", "ш", "е" и, наконец, "к".

В строке 38мы с помощью **if (slovo[j]=== bukva)** проверяем, совпадает ли текущая буква (**slovo[j]**) с ответом игрока. Если это так, то мы проверяем: «А не угадали ли мы уже эту букву?». Если эту проверку не сделать, то, если вы будете снова и снова вводить одну и ту же букву, которая есть в загаданном слове, **ostalosBukv** будет снова и снова уменьшаться.

Если на позиции совпавшей буквы до сих пор стоит «\_», то мы обновляем итоговый массив, добавляя туда букву командой **otvetMassiv[j] = bukva**. Для каждой буквы, совпадающей с ответом, мы обновляем соответствующую позицию итогового массива.

Этот код работает, поскольку переменную цикла **j** можно использовать

одновременно в качестве индекса в строке **slovo** и индекса в массиве **otvetMassiv.**

Помимо обновления **otvetMassiv** для каждого совпадения с **bukva**

требуется уменьшать **ostalosBukv** на 1. Мы делаем это в строке 40 командой **ostalosBukv**--;. Каждый раз, когда **bukva** совпадает с буквой из **slovo**, **ostalosBukv** уменьшается на 1, и, когда игрок угадает все буквы, **ostalosBukv** примет значение 0.

## Домашнее задание

1. Добавить больше слов в словарь.
2. После окончания основного игрового цикла, поздравить игрока с победой и вывести слово полностью.

# Урок 11 Доработка игры. Функции и кнопки

## Доработка игры «Виселица»

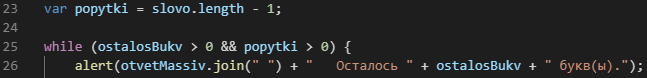
### Регистр букв

Давайте подумаем, какие ещё недочёты остались в нашей игре. Если хорошо потестировать её, то можно заметить, что, если игрок введет заглавную букву, она не совпадет с такой же строчной буквой в загаданном слове. Эту проблему можно решить, преобразовав введенную букву в строчный регистр. (Подсказка: для этого подойдет метод **toLowerCase**.)

*Попробуйте сделать это самостоятельно*

### Попытки

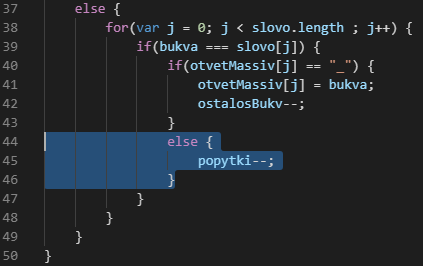
Сейчас наша «Виселица» позволяет угадывать буквы сколько угодно раз, до победного конца. Добавьте в программу переменную для учета попыток и завершайте игру, если игрок израсходовал все попытки. (Подсказка: проверяйте значение этой переменной в том же цикле **while**, где проверяется условие **ostalosBukv** **> 0**, а с помощью операции **&&** можно

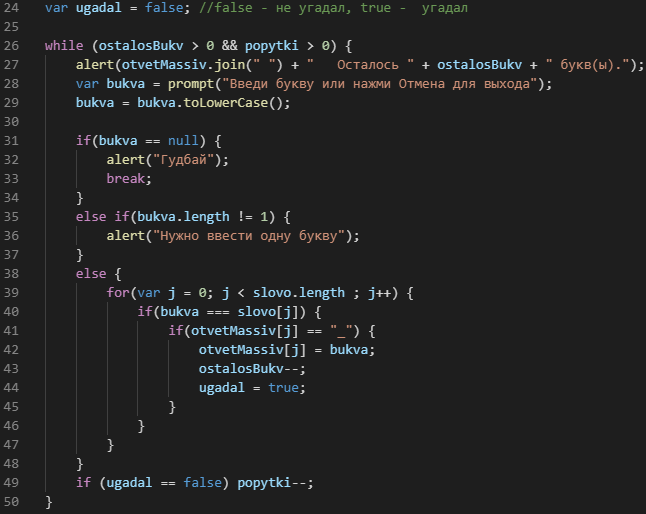
убедиться, что сразу два логических условия дают **true.**

Изначально мы присваиваем переменной **popytki** значение: длина слова – 1. Если программа загадала слово «телефон», то переменная **popytki** будет равна 6.

В цикле мы будем уменьшать переменную **popytki** каждый раз, когда игрок будет ошибаться, а в условии проверять: не стала ли она равна нулю, то есть цикл будет работать пока **popytki > 0**.

Если мы добавим такой код, мы не получим желаемого результата. Так как цикл **for** будет повторяться несколько раз, как и находящийся выше цикл **while.** А это значит, что переменная будет уменьшаться столько раз, сколько введённая буква будет отличаться от всех букв в слове.

Если программа загадала слово «телефон», то введя букву «а», запуститься цикл, в котором мы спросим: «Равна ли буква «а» букве «т»?» – нет, значит **poputki--**. На следующем произойдёт проверка с буквой «е», и мы получим такой же результат. Таким образом к последней букве, переменная **popytki**, которая изначально равнялась 6, будет равна -2.

Реализовать проверку на угаданную букву можно с помощью отдельной переменной **ugadal.** Эта переменная будет булевой и будет принимать либо значение **false** либо **true.**

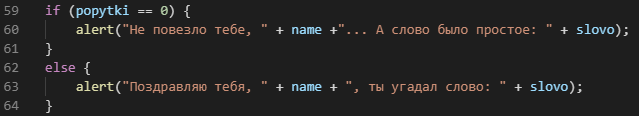
В строке 24 мы создаём переменную **ugadal** и присваиваем ей значение **false,** так как буква ещё не угадана. В строке 44 мы меняем значение этой переменной на **true**, так как в это условие мы попадаем, когда буква совпала. Если мы попадём в это условие ещё раз (в слове несколько одинаковых букв), то ничего не изменится – переменная **ugadal** останется со значением **true**.

В строке 49 мы проверяем совпала ли введённая нами буква хотя бы с одной буквой в загаданном слове (если бы совпала **ugadal** было бы равно **true**). Если не совпала, то вычитаем попытки.

Самостоятельно добавьте информацию о количестве оставшихся попыток в сообщение, отвечающее за отображение состояния.

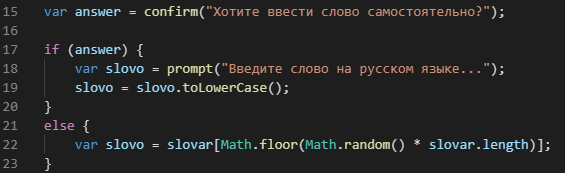
### Поздравление игрока

Дома вы сделали поздравление игрока, но поздравляем мы его и тогда, когда он выиграл (**ostalosBukv == 0**), и тогда, когда он проиграл (**popytki == 0**). Таковы условия нашего цикла.

Чтобы правильно выводить последнее сообщение, мы должны узнать, что у нас всё-таки закончилось: неразгаданные буквы или попытки.

### Возможность ввести слово

Чтобы наша игра была более интересной, мы добавим возможность второму игроку ввести своё слово, чтоб первый угадывал. Для этого нам нужно будет задать вопрос: «Хотите ввести слово самостоятельно?».

Делать мы это будем с помощью функции **confirm,** которая возвращает **true,** когда пользователь нажал на кнопку «ОК», и **false**, когда нажали «Отмена». А так как условный оператор **if** выполняется только, когда выражение в его скобках равно **true**, то мы можем проверить ответ, который нам вернёт **confirm** и, если он «правда», то мы дадим возможность пользователю ввести слово, иначе – выберем его из словаря, как мы делали раньше.

## Функции и кнопки

Функции — это механизм для многократного использования частей кода. Они позволяют запускать один и тот же код из разных мест программы

без необходимости его копировать. Кроме того, если вы «спрячете» сложные фрагменты кода внутри функций, вам будет легче сосредоточиться на проектировании программы — так вы будете налаживать взаимодействие между функциями, а не барахтаться в мелких деталях, из которых состоит код этих фрагментов.

Организация кода в виде небольших, легко контролируемых частей позволяет видеть общую картину и думать о строении программы на более высоком уровне.

Функции очень удобны, когда нужно многократно выполнять в программе некие расчеты или другие действия. Мы уже пользовались готовыми функциями, такими как **Math.random**, **Math.floor**, **alert**, **prompt**

и **confirm**. А теперь мы научимся создавать функции самостоятельно.

Только мы не просто будем создавать функции, мы будем привязывать их к кнопкам. Так мы сможем избежать существующей у нас проблемы. Ведь, когда мы открываем файл HTML, к которому подключен наш скрипт, скрипт тут же запускается.

### Создание кнопки

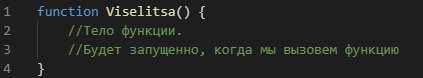
Добавим в файл **index.html**, в котором записана разметка страницы с игрой «Виселица» кнопку, с помощью которой мы будем запускать наш скрипт, когда захотим.

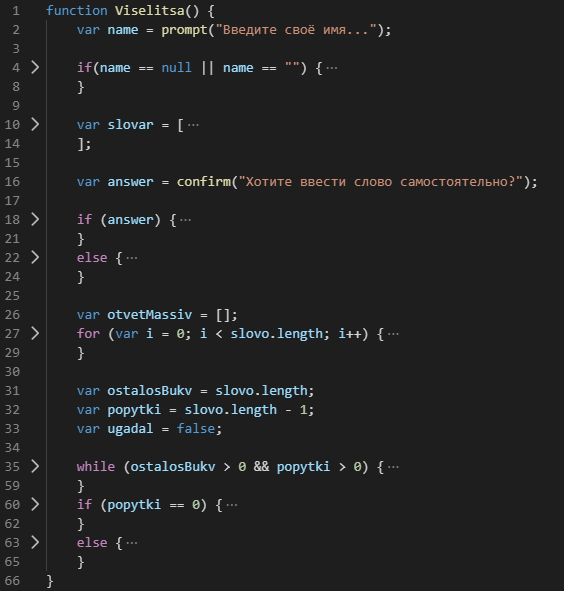
Тег **<button>** создает на веб-странице кнопки, на которых можно размещать любые элементы HTML, в том числе изображения. Используя стили можно определить вид кнопки путем изменения шрифта, цвета фона, размеров и других параметров.

Когда мы добавили кнопку на нашу страницу, открываем её в браузере и убеждаемся, что сама по себе кнопка ничего не делает.

Давайте зададим кнопке нужные размеры и расположим её в правильном месте.

### Связь кнопки и скрипта

Как мы уже сказали, чтобы запускать скрипт с помощью кнопки, мы должны создать в скрипте функцию. В нашем случае весь скрипт будет являться функцией.

Теперь вставим весь наш скрипт между фигурными скобочками, только что созданной, функции.

Теперь нам осталось только указать кнопке, что именно нужно вызывать, когда по ней щёлкнут. Для этого бы будем использовать событие **onclick**. Оно возникает при щелчке левой кнопкой мыши на элементе, к которому добавлен атрибут **onclick.**

## Домашнее задание

Добавить во все предыдущие проекты кнопку запуска скрипта, по аналогии с «Виселицей».