

2018

Справочник по JS

Базовые методы и шаблоны программ

Описаны базовые методы языка программирования JavaScript, структурные операторы, способы работы с массивами и строками, функции и организация модулей, архитектура программы (html+css+js).

Справочник предназначен для начинающих и забывчивых.

Практикум по JS

Задачи по темам

Обновления по адресу:

<https://pcoding.ru/pdf/jsManual.pdf>



Andrey Belyakov
pCoding.ru
21.12.2018



Содержание

0	Шаблоны программ <ul style="list-style-type: none">- диалоги- арифметические операции- ветвления и циклы- массивы и строки- функции и модули- архитектура программы	3
1	Самое начало Переменные и области видимости Структурные операторы	тут
2	Массивы	тут
3	Функции и модули	тут
4	Работа в консоли на Node.js	тут
5	Работа с файлами в Node.js	тут
6	Генерация html-документа	тут
7	Событийное программирование	тут
8	Программирование микроконтроллеров	тут
9	darkNet и настройки сайта	тут
A	Анимация. События таймера	...
B	jQuery	...
10	Практикум <ul style="list-style-type: none">0. Разработка своего сайта1. Ветвления2. Циклы3. Массивы4. Файлы5. Генерация html-документа6. Событийное программирование7. Программирование микроконтроллеров8. darkNet9. ИТ-Хакатон	тут-1 тут-2 тут-3 тут-4 тут-5 тут-6 тут-7 тут-8 тут-9



Первая программа и диалоги

Организация HTML-документа с программой на JS

```
<html>
<head>
  <meta charset="utf-8">
  <title>Заголовок</title>
</head>
<body>
  <script>
    document.write("Первая строка" + "<br>");
    document.write("Вторая строка");
  </script>
</body>
</html>
```

Парный тег script и простейший диалог

```
<script>
  x = prompt("Введите имя");
  alert("Привет - " + x);
</script>
```

Подтвердите действие

Введите имя

OK Отмена

Диалог со значением по умолчанию и приведение типа данных

```
<script>
  x = Number(prompt("Введите число", -2018));
  alert("Ответ = " + Math.abs(x));
</script>
```

Подтвердите действие

Введите число

OK Отмена

Диалог confirm (ДА/НЕТ) и случайное число

```
<script>
  if (confirm("Хотите сгенерировать случайное число ?")) {
    x = Math.floor(Math.random() * 100);
    alert("Число = " + x);
  }
  else{
    alert("Ну пока ...")
  }
</script>
```

Подтвердите действие

Хотите сгенерировать случайное число ?

OK Отмена

Разные виды циклов

```
function func_02(n){
    sum = 0
    for (i=1; i<=n; i=i+1){
        sum = sum + i;
    }
    return sum;
}
```

```
function func_03(n){
    sum = 0
    for (i=1; i<=n; i++){
        sum += i;
    }
    return sum;
}
```

```
function func_04(n){
    sum = 0; i=1;
    while (i<=n) {
        sum += i;
        i++;
    }
    return sum;
}
```

```
function func_05(n){
    sum = 0; i=1;
    do {
        sum += i;
        i++;
    } while (i<=n);
    return sum;
}
```

Цикл со сложной шапкой

```
var max = 7;
for (let i = 0, j=9; (i<max)&&(j>0); i++, j--) {
    document.write(i, " - ", j, "<br>");
}
```

Цикл со сложной шапкой

```
str = "JavaScript.com is a resource for the JavaScript community.";
for (i = 0, len=str.length; i<len; i++) {
    console.log(str[i]);
}
```

Многоальтернативный выбор по значению switch. Из Арабской в Римскую

```
function func_06(n){ // подаём арабскую цифру
    switch (n) {
        case 1:
            otv="I";
            break;
        case 4:
            otv="IV";
            break;
        case 8:
            otv="VIII";
            break;
        case 9:
            otv="IX";
            break;
        default:
            otv="data error";
    }
    return otv;
}
```

```
<script>
  var age = +prompt("Ваш возраст ?");
  switch (true) {
    case (age<18):
      alert("кандидат в призывники");
      break;
    case (age<28):
      alert("призывник");
      break;
    default:
      alert("уже свободен");
  }
</script>
```

Использование **тернарной операции** (может возвращать результаты)

```
<script>
  function factorial_rec(n) { // вычисление факториала числа
    return n==1? 1: n*factorial_rec(n-1);
  }
  document.write( factorial(5) );
/*
Тернарная условная операция (от лат. ternarius – «тройной», так как используются три позиции
между символами "?" и ":" – «условие?если истинно:если ложно») – операция, возвращающая свой
второй или третий операнд в зависимости от значения логического выражения, заданного первым опе-
рандом.
*/
</script>
```

Округление до указанной точности и перенос в диалоговом окне

```
<script>
  x = Math.random()*1000;
  y = 3;
  alert("x="+x+"\nx="+x.toFixed(y));
  //округление до 3-го разряда
</script>
```

Подтвердите действие

x=396.7945798981374

x=396.795

OK

Объект по умолчанию – оператор with

```
<script>
  r = 2; // радиус круга
  with (Math) {
    s = PI*pow(r, 2); // Math.PI * Math.pow(r, 2)
  }
  document.write(s);</script>
```

Вывести символ по его коду

```
<script>
    for (let num=48; num<58; num++) {
        smb = String.fromCharCode(num);
        document.write(smb, "<br>");
    }
</script>
```

Вывести код символа

```
<script>
    for (let i=0; i<str.length; i++) {
        code = str.charCodeAt(i);
        // code = str[i].charCodeAt(); // вариант
        document.write(str[i], " - ", code, "<br>");
    }
</script>
```

Вывести код символа. Пример работы с элементами оформления.

```
<html>

<body>
    <p>Check ASCII code</p>
    <button onclick="myFunction()">Click me</button>
    <p>
        Enter any value:
        <input type="text" id="idString"> </br>
    </p>
    <p id="result" style="color:red;"></p>
    <script>
        function myFunction() {
            str = document.getElementById("idString");
            title = "ASCII Code is - ";
            temp = str.value.charCodeAt(0);
            document.getElementById("result").innerHTML = title + temp;
        }
    </script>
</body>

</html>
```

Организация работы с функциями

```
<script>
    function factorial(n) { // вычисление факториала числа
        let result = 1;
        for (let i=n; i>0; i--) {
            result *= i;
        }
        return result;
    }
    document.write( factorial(5) );
</script>
```

Организация рекурсивной функции

```
<script>
    function factorial_rec(n) { // вычисление факториала числа
        if (n==1)
            return 1;
        else
            return n*factorial_rec(n-1);
        }
    document.write( factorial(5) );
</script>
```

Организация работы с модулями *.js

```
<html>
<head>
    <title>Зароловок</title>
    <meta charset=utf-8>
    <script src="my.js"></script> // подключение модуля
</head>
<body>
    <script>
        func_01(); // вызов функции из модуля
    </script>
</body>
</html>
```

Содержимое модуля my.js

```
var diap = 9;

function func_01(){
    if (confirm("Хотите сгенерировать случайное число ?")) {
        x = Math.floor(Math.random() * diap);
        alert("Число = " + x)
    }
    else { alert("Ну пока ..."); }
}
```




Переменные

Когда мы работаем с числами или строками – мы работаем с данными. В процессе их обработки начальные, промежуточные и конечные результаты где-то нужно хранить. Хранятся данные программы, конечно, в ячейках оперативной памяти, однако, язык программирования позволяет обращаться к этим ячейкам не по их адресу, а по имени (идентификатору). Итак, переменная – это именованная область памяти. Область памяти может занимать несколько ячеек памяти, в зависимости от типа хранимых данных.

Изучите пример программы:

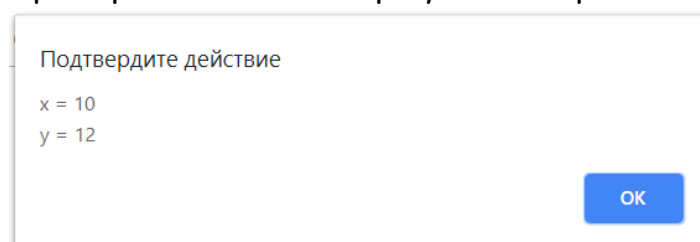
```
<script>
  x = 10; // объявили переменную и присвоили значение
  y = 10 + 2; // объявили переменную и присвоили выражение
  alert("x = " + x + "\n" + "y = " + y); // вывели ответ
</script>
```

В программе есть комментарии, они следуют после литерала «два слеша» и не выполняются интерпретатором js, а только хранятся в коде для удобства работы программиста. Если программа тривиальная, то комментарии не имеют смысла.

Запустите редактор Visual Studio Code, наберите текст программы, сохраните в специально подготовленной папке. Так выглядит окно редактора VS Code с текстом программы:



Перейдите в проводник, кликните дважды по файлу с программой, запустится браузер, как исполнитель назначенный по умолчанию в операционной системе и отобразит примерно такое окно с результатом работы программы:



Можно декларировать переменные без присвоения им значений. Такой подход чреват непредвиденными ошибками, так как в процессе выполнения дальнейшего алгоритма может случиться так, что к моменту использования переменной она ещё не приобретёт значения:

```
<script>
  var radius; // радиус круга
  /*
    тут предполагалось дать значение радиусу
  */
  S = Math.PI*Math.pow(radius,2); // площадь круга
  result = "S = " + String(S);
  document.write(result);
</script>
```

Обратите внимание: JavaScript регистрозависимый язык программирования, то есть **Math.Pi** и **Math.PI** считаются разными командами.

Апробируйте эту программу - вы получите такой результат:

S = NaN

где NaN (англ. Not-a-Number, «не число»).

Внесём изменения в эту программу:

```
<script>
  var radius; // радиус круга
  radius = Number(prompt("Введите радиус круга"));
  S = Math.PI*Math.pow(radius,2); // площадь круга
  result = "S = " + String(S);
  document.write(result);
</script>
```

Запустите программу и введите радиус круга, равный 1, вы получите такой результат:

S = 3.141592653589793

Если не нужна такая точность, то без искажения значения в самой переменной, можно вывести ответ на экран с ограничением количества знаков после запятой таким образом:

```
result = "S = " + S.toFixed(3);
```

Внесите соответствующие изменения в код и апробируйте новый вариант программы, получите такой результат:

S = 3.142

Структурные операторы

Операторы языка программирования можно разделить на два класса: простые и сложные. Простые операторы называют простыми, так как они не содержат внутри себя других операторов. Сложные операторы чаще называют структурными, так как они представляют собой специальным образом организованную структуру, в которую по определенным правилам могут включаться простые операторы. К простым операторам относят операторы ввода и вывода информации, присваивания, а также пустой оператор (может быть обозначен отдельно стоящим терминальным символом «;»).

1. Составной оператор (блок).

Наверное, чаще всего из структурных операторов в текстах программ можно встретить составной оператор, представляющий собой операторные скобки (блок), то есть, по сути, скобки для объединения других операторов. В качестве открывающей скобки используется символ "{", а в качестве закрывающей "}". Между скобками можно расположить произвольное число простых и/или структурных операторов, при этом составной оператор будет восприниматься как один цельный оператор. Именно для этого он и предназначен, то есть он обычно используется в том месте программного кода, где по правилам языка может стоять только один оператор, а для реализации алгоритма требуется использование нескольких:

```
// если условие истинно, то сделай блок операторов
{
    console.log("Первая строка")
    console.log("Вторая строка"); console.log("Третья строка");
}
```

Тут используется команда вывода на экран консоли, если же вы собираетесь выводить в око браузера, то поменяйте console.log() на document.write().

Синтаксис языка обязывает ставить символ «;» только в тех случаях, когда нужно операторы (команды) размещать на одной строке кода последовательно. Если же операторы располагаются на разных строчках, то терминальный символ необязателен, однако, для сохранения стиля рекомендуется ставить «;» во всех случаях.

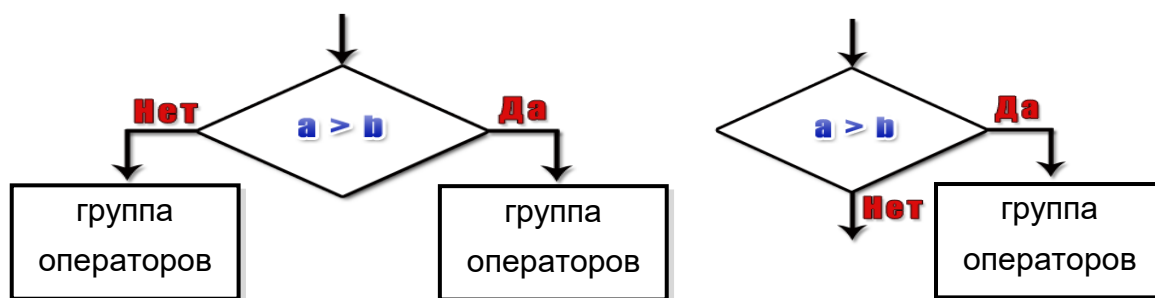
```
{
    console.log("Первая строка");
    console.log("Вторая строка");
    console.log("Третья строка");
}
```

Составной оператор обычно используется в составе других структурных операторов (ветвления, цикла, ...).

2. Оператор ветвления.

Оператор ветвления (другое название - условный оператор) обеспечивает выполнение или невыполнение некоторого другого оператора (в том числе и составного) в зависимости от проверяемого условия. Иными словами, оператор ветвления разделяет поток действий алгоритма на две ветви. Условный оператор позволяет выполнять нелинейные алгоритмы, такие в которых предусмотрено прерывание линейной последовательности действий в зависимости от обрабатываемых данных и выполнение одной из предусмотренных программистом ветвей алгоритма.

Конструкция условного оператора позволяет организовать два варианта его использования: с **двумя** нагруженными ветвями алгоритма и с **одной**.



Вариант 1 (с двумя ветвями):

```
if (условие) {  
    // группа операторов №1  
    // эти операторы выполняются,  
    // если условие истинно  
}  
else {  
    // группа операторов №2  
    // эти операторы выполняются,  
    // если условие ложно  
};
```

Этот вариант можно интерпретировать так - если условие истинно, то выполняй первую группу операторов, иначе выполняй вторую группу операторов. Обратите внимание, что условие обязательно нужно брать в круглые скобки. Если блок операторов состоит из одной команды (одного оператора), то её можно и не брать в фигурные скобки, но лучше их ставить всегда для сохранения чистоты стиля.

Апробируйте пример кода с оператором ветвления:

```
// полное ветвление
num = Math.random();
if (num<0.5) {
    console.log(num, "меньше 0.5");
}
else {
    console.log(num, "больше или равен 0.5");
}
```

Эта программа генерирует число в диапазоне [0; 1), проверяет условие и, в соответствии с указанным условием, выдаёт результат.

Как видите, в программе есть часть кода, которая повторяется - это вывод на экран консоли. Здесь повторяющаяся часть кода невелика, но в случае дублирования значительных частей кода имеет смысл заняться оптимизацией программы. Рассмотрим один из вариантов совершенствования - условную тернарную операцию - её синтаксис такой - **___? ___: ___** :

Условие? Что делать, если условие истинно: Что делать, если условие ложно

Напишем программу проверки кратности одного числа другому. Напомню, что числа кратны, если делятся без остатка, то есть 15 делится на 5 нацело (остаток от деления равен нулю, а 17 делится на пять с остатком два. Есть арифметическая операция, которая вычисляет остаток от такого деления - **%**.

Оцените программу:

```
<script>
    a = Number(prompt("Введите число a"));
    b = Number(prompt("Введите число b"));
    result = a%b==0? "кратно": "не кратно";
    document.write(result);
</script>
```

Функция `prompt()` обеспечивает диалог с пользователем, результат её выполнения возвращается в программу в виде строки, мы его вставляем в функцию `Number()`, чтобы привести к числовому типу данных и полученный результат размещаем в переменной. Так делаем дважды для обеих переменных (a, b). Потом, используя условную тернарную операцию определяем кратность чисел. Двойной знак равно используется для сравнения переменных, значений. Если левая часть от `==` равна правой, то возвращается истина (`true`), иначе - ложь (`false`). Если нужно искать именно не равные величины, то можно использовать операцию сравнения

не равно, пишется так: `!=`. В завершении программы выводим результат в окно браузера.

Часто арифметическую операцию вычисления остатка от целочисленного деления `%` используют для проверки чётности числа, так как при делении числа на 2 нацело, могут получиться только два варианта остатка: ноль или единица. Если число чётное, то оно делится на два нацело и остаток от деления равен нулю, а иначе остаток равен единице – других вариантов нет.

Иногда возникает необходимость не просто округлить число при выводе результата на экран как мы делали ранее (`toFixed()`), а именно получить результат в виде целого числа. Для этих целей можно использовать такие функции как `floor` и `ceil`:

```
<script>
  a = 17;
  b = 5;
  result = 17/5;
  document.write(result + "<br>");
  document.write(Math.floor(result) + "<br>");
  document.write(Math.ceil(result) + "<br>");
</script>
```

Оцените результаты работы данной программы, сделайте выводы, учитывая, что `floor` можно перевести как пол, а `ceil` – как потолок. Данные функции можно использовать для получения целого из не целого ☺, например, необходимо получить случайное целое число от 1 до 10:

```
<script>
  result = Math.ceil(Math.random()*10);
  document.write(result);
</script>
```

Для проверки этой программы можете запустить её несколько раз, обновляя результат на экране нажатием клавиши F5 в браузере.

► Самостоятельно разработайте программу `p_01.js`, которая генерирует натуральное (целое положительное) число в диапазоне `[0; 100)` и проверяет его на чётность/нечётность (используйте операцию вычисления остатка `%`), результат выводится на экран. Прямоугольная скоба означает, что число входит в диапазон, а круглая, что не входит. ◀

3. Многоальтернативное ветвление

Как вы уже поняли, структурный оператор `if` или тернарная условная операция `_?:_` могут разделить единый поток действий на два пути. Если по условиям задачи нужно делить на большее количество путей, тогда приходится комбинировать несколько условных операторов. Однако, в ряде случаев, удобнее использовать оператор многоальтернативного ветвления `switch`:

```
<script>
    num = parseInt(prompt("Введите число"));
    switch (num) {
        case 1:
            result = "один";
            break;
        case 2:
            result = "два";
            break;
        case 3:
            result = "три";
            break;
        default:
            result = "error data";
            break;
    }
    document.write(result);
</script>
```

Как вы уже заметили, при работе оператора `switch` происходит буквальное сопоставление параметра с вариантами значений. А как поступать, если нужно сравнивать параметр с диапазоном значений. Один из вариантов использования оператора `switch` для этих целей состоит в том, чтобы искать истинность проверки условия вхождения параметра в диапазон:

```
<script>
    num = parseInt(prompt("Введите число месяца"));
    switch (true) {
        case num<=10:
            result = "первая триада";
            break;
        case num<=20:
            result = "вторая триада";
            break;
        case num<=31:
            result = "последняя триада";
```

```

        break;
    default:
        result = "ошибка ввода данных";
    }
    document.write(result);
</script>

```

У вас уже созрел резонный вопрос: зачем нужен оператор `break`? Он используется для настройки функциональности оператора `switch` - можно сделать так, чтобы пути реализации сочетались логическим «ИЛИ» или «исключающим ИЛИ». В первом случае варианты решений могут быть выбраны так: или тот или другой или оба вместе (тогда `break` писать не нужно), во втором случае - ЛИБО тот, ЛИБО другой, но никак не вместе (тогда `break` нужны). Давайте в предыдущий пример внесём небольшие изменения - закомментируем операторы `break` и добавим вывод в каждый из путей реализации:

```

<script>
    num = parseInt(prompt("Введите число месяца"));
    switch (true) {
        case num<=10:
            result = "первая триада";
            document.write(result + "<br>");
            // break;
        case num<=20:
            result = "вторая триада";
            document.write(result + "<br>");
            // break;
        case num<=31:
            result = "последняя триада";
            document.write(result + "<br>");
            // break;
        default:
            result = "ошибка ввода данных";
            document.write(result + "<br>");
    }
</script>

```

Апробируйте данную программу, испытайте её на числах 7, 17, 27, 37 - вы увидите, что без оператора `break` в вывод будут включаться сразу несколько путей. Итак, при проверке диапазонов значений оператор `break` имеет особое значение.

4. Операторы цикла.

Когда нужно проверять условия для нескольких элементов структуры данных (например, для массива) или, когда нужно делать однотипные действия фиксированное или неопределённое количество раз, используют операторы цикла: `for`, `while`, `do while`. В принципе, они взаимозаменяемы, но в разных случаях удобнее выбрать и использовать какой-то один из них. Например, оператор `for` обычно используют, когда количество повторов predetermined заранее. Рассмотрим простейший пример - **вывести на экран цифры от 0 до 9**:

```
<script>
    for (i=0; i<10; i++) { // это шапка цикла
        document.write(i + " "); // это тело цикла
    }
</script>
```

Итак, оператор цикла `for`, его ещё называют параметрическим циклом, содержит две части: шапку цикла и тело цикла. В шапке цикла должны быть определены три позиции: инициализации переменной-счётчика (откуда начать), условие продолжения перебора (докуда идти), с каким шагом идти. В теле цикла, собственно, описываются действия, которые нужно повторять заданное количество раз.

Эту же задачу можно реализовать и двумя другими операторами цикла: циклом с предусловием - `while` и циклом с постусловием - `do while`:

while	do while
<pre><script> i = 0; while (i<10) { document.write(i+" "); i++; } </script></pre>	<pre><script> i = 0; do { document.write(i+" "); i++; } while (i<10); </script></pre>

Как видите, для реализации всех видов цикла нужно использовать инициализацию переменной-счётчика, проверку окончания перебора и определения шага изменения переменной-счётчика.

Для модификации работы цикла можно использовать операторы **`continue`** и **`break`**. Первый - принудительно пропускает текущую итерацию цикла, переходя на следующий шаг, второй - пропускает все итерации, выходя из цикла совсем и досрочно (то есть, не выполняя оставшиеся итерации).

Рассмотрим примеры.

Пусть требуется вывести на экран из диапазона чисел от 0 до 9 только нечётные:

```
<script>
    for (i=0; i<10; i++) {
        if (i%2==0) continue;
        document.write(i + " ");
    }
</script>
```

Эта программа на экран выведет такую последовательность: 1 3 5 7 9.

Пусть есть ряд чисел от 1 до 100 - они означают веса предметов. Есть контейнер, вместимость которого ограничена 30 килограммами. Какое максимально количество предметов можно взять, чтобы не превысить вместимость контейнера?

```
<script>
    max = 30; // ограничения на вместимость контейнера
    count = 0; // количество предметов
    summ = 0; // суммарная масса предметов
    for (i=1; i<=100; i++) {
        if (summ+i>max)
            break; // выходим, если предмет i уже лишний
        count++; // берём следующий предмет
        summ += i; // добавляем массу предмета
    }
    document.write(count);
</script>
```



До сих пор в алгоритмах мы рассматривали одиночные переменные, но в программах, ориентированных на решение практических задач, данные, как правило, хранят в определённых структурах, например, в массивах данных. Массив – это проиндексированная (то есть с порядковыми номерами) последовательность элементов.

Например, строка это такой одномерный массив символов и, если в переменной `str` хранится строка `str = "JavaScript"`, то к каждому отдельному элементу строки можно обратиться по их порядковым номерам: `str[0] == 'J'`, `str[1] == 'a'`.

Массив – это не обязательно только одномерная структура, в частности, таблица умножения, это такой двумерный массив, заполненный целыми числами по определённому правилу. Чтобы обратиться к определённому элементу из массива, нужно сначала написать имя массива, а, затем в скобках (иногда используют круглые скобки, иногда – прямоугольные) номер его позиции в массиве, например, так: `m[5]`. Если этот массив одномерный, то количество индексов – один, если двумерный, то два индекса (например, `m[1, 3]`) и т.д. Примером одномерного массива может служить алфавит, в котором под определённым порядковым номером расположен определённый символ. Во многих языках программирования (и в JS, в частности, так же) в массивах индексация по умолчанию начинается с нуля, то есть первый элемент массива имеет порядковый номер равный нулю, второй элемент – 1 и т.д.

Итак, массив – это структура для хранения и обработки упорядоченного набора элементов. В JS элементы массива могут быть разного типа данных (во многих других языках программирования это не так). Так как архитектурно массивы в JS устроены как коллекции, то хранят они просто пары «порядковый номер – элемент».

Можно объявить пустой массив и потом его заполнять:

```
var arr = [];
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
document.write( arr );
```

Можно объявить массив сразу с элементами и потом их все вывести на экран:

```
var arr = ["Мама", "мыла", "раму"];
for (let i=0; i<arr.length; i++) {
    document.write( arr[i], " " );
}
```

Можно объявить массив через вызов конструктора массива new Array:

```
var arr = new Array("Мама", "мыла", "раму");
for (let i=0; i<arr.length; i++) {
    document.write( arr[i], " " );
}
```

Можно задать длину массива, но не заполнять элементами:

```
var arr = new Array(3);
arr[0] = "Мама";
arr[2] = "раму";
document.write( arr[0], "<br>" );
document.write( arr[1], "<br>" );
document.write( arr[2] );
```

Результат:

```
Мама
undefined
раму
```

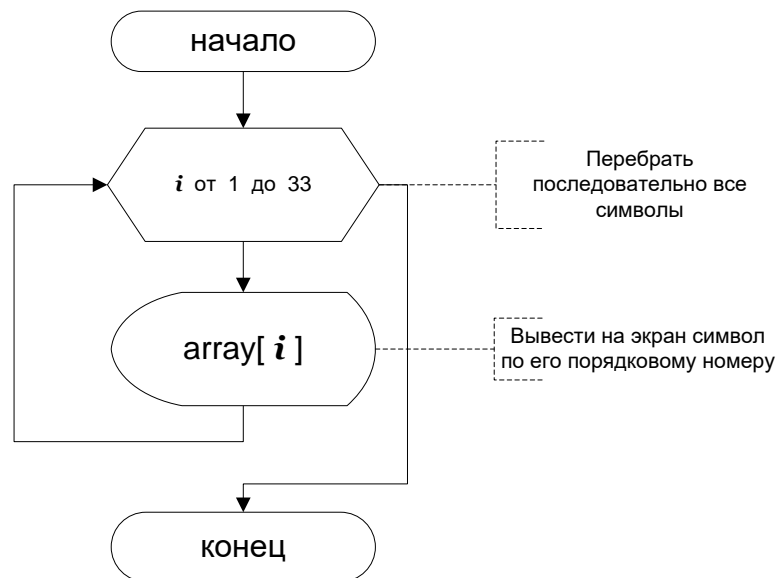
При этом свойство `length` хранит не количество заполненных элементов массива, а индекс последнего элемента плюс 1.

Можно создавать и работать с многомерными массивами, понимая их как массивы массивов:

```
var table = [
    [1, 2, 3],
    [4, 5, 6, 7],
    [8, 9]
];
for (let row=0; row<table.length; row++) {
    for (let col=0; col<table[row].length; col++){
        document.write( table[row][col], " " );
    }
    document.write( "<br>" );
}
```

В некоторых случаях можно или требуется изменить нижнюю границу массива и начать нумерацию не с нуля, а, например, с единицы.

Пусть у нас для хранения алфавита русского языка назначена переменная `array`, индексация начинается с единицы, тогда элемент `array[3]` будет содержать символ «в». Разработаем алгоритм вывода на экран символов русского языка:



В этом алгоритме присутствует только блок параметрического цикла, где в шапке указано сколько делать повторов, а тело цикла содержит одно единственное действие – вывод на экран.

Для примера рассмотрим немного более трудоёмкую задачу: нужно разработать алгоритм и программу генерации массива из `count` (`count` задаёт пользователь) чисел (в диапазоне (0,10]) с последующим подсчётом суммы элементов массива больших 5. Можно сказать, что словесное описание алгоритма содержится в самой задаче. Оформим решение в двух шагах:

- на первом шаге пользователь вводит количество элементов массива, программа генерирует числа и заполняет ими массив;
- на втором шаге программа перечисляет все элементы, но для суммирования выбирает лишь те из них, которые соответствуют условию задачи, и суммирует их.

Для разработки алгоритма будем использовать специальный редактор Блок-схем-алгоритмов AFCE (Algorithm Flowchart Editor). В левом окне редактора располагаются инструменты (блоки), по центру – окно редактирования блок-схемы, справа – исходный код программы и окно помощи (в нижней части) (см. рис.).

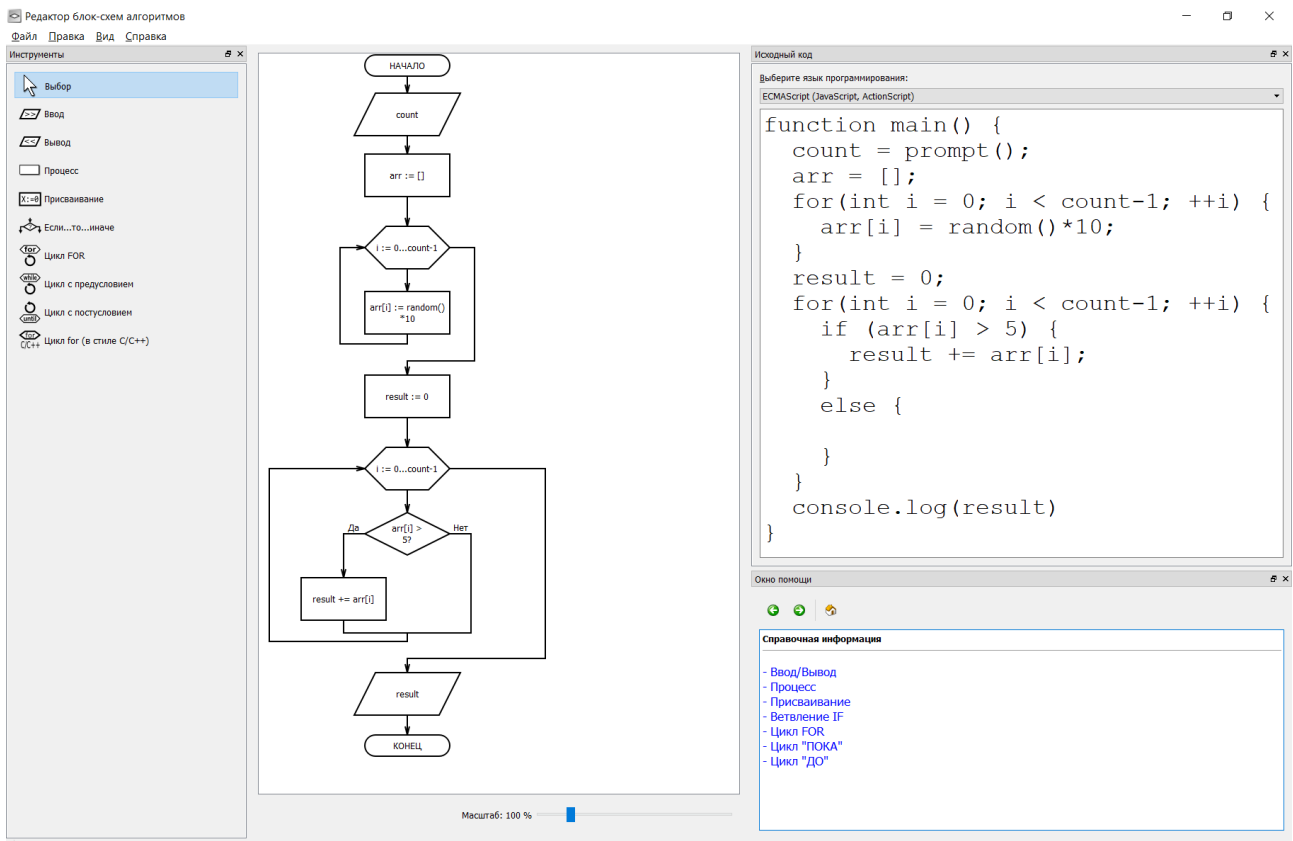


Рис. Окна редактора Algorithm Flowchart Editor.

Программный код в правом окне генерируется автоматически, вы имеете возможность выбрать язык программирования в выпадающем списке.

Окна кода и блок-схемы масштабируются. Разработанный алгоритм можно сохранить как во внутреннем формате редактора для последующего редактирования, так и в виде графического файла через главное меню программы. Для того чтобы установить блок в схему алгоритма достаточно просто выделить его мышкой в левом окне и затем кликнуть в необходимую позицию в окне блок-схемы.

После разработки алгоритма сгенерированный код программы можно скопировать из окна кода, перенести в редактор, которым вы

пользуетесь для отладки программ, и продолжить работу с программой уже без данного редактора. Однако, не всегда удаётся сгенерировать автоматически полностью корректный код. Иногда среда генерации AFCE просто не учитывает необходимый программисту контекст использования, поэтому имеет смысл доработать программу.

Посмотрите доработки кода по обсуждаемой задаче и попробуйте самостоятельно разобраться во внесённых изменениях, при необходимости воспользуйтесь помощью интернета.

```
<!--
  Программа генерации массива
  из count чисел в диапазоне [0,10) // 10 не входит
  с последующим подсчётом
  суммы элементов массива больших 5.
-->
<script>
  var count = +prompt(); // вводим кол-во элементов
  var arr = []; // инициализируем массив
  for(let i = 0; i < count; ++i) {
    arr[i] = Math.random()*10; // заполняем элементы
  }
  var result = 0; // инициализируем переменную
  for(let i = 0; i < count; ++i) {
    if (arr[i] > 5) { // проверяем условие
      result += arr[i]; // суммируем
    }
  }
  document.write(result.toFixed(2)); // выводим результат
</script>
```

Массив можно использовать как очередь или стек. Эти структуры имеют дополнительные методы работы: можно добавлять или удалять элементы массива с его концов. Очередь – это такая структура данных, которая работает по принципу «первым пришел – первым уйдёшь» (FIFO). Стек – это такая структура данных, которая работает по принципу «первым пришёл – последним уйдёшь» (FILO).

Операции с концом массива:

- `arr.push(элемент1, элемент2...);` // добавляет элементы в конец
- `var elem = arr.pop();` // удаляет и возвращает последний элемент

Операции с началом массива (перенумеровывают все элементы, поэтому работают медленно):

- `arr.unshift(элемент1, элемент2...);` // добавляет элементы в начало
- `var elem = arr.shift();` // удаляет и возвращает первый элемент

Пример 1 работы с массивом

```
<script>
  /*
    программа определяет сколько раз число n встречается в массиве
  */
  arr = [0, 1, 2, 5, 34, 5, 5];
  n = 5; count = 0;
  for (i=0; i<arr.length; i++) {
    if (n==arr[i])
      count++;
  }
  document.write( count );
  document.write("<br>", arr);
</script>
</html>
```

Пример 2 работы с массивом

```
// это modul.js:
function rnd(min, max) {
  return min+Math.floor( (Math.random() * (max-min+1)) );
}
// это файл html
<html>

<head>
  <meta charset="utf-8">
  <title>ПИ6у-2018</title>
  <script src="modul.js"> </script>
</head>

<body bgcolor=#ffc>
  <font color=#040 size=32 face="Courier New">
    <script>
      /*
        программа генерирует десять случайных целых двузначных чисел
        в заданном диапазоне, заполняет ими массив и выводит на экран
        в обратном порядке только нечётные значения
      */

      arr = [];
      count = 10;      min = 10; max = 99;
      for (let i=0; i<count; i++) {
        arr.push( rnd(min, max) );
      }
      document.write(arr, "<br>");
      i = count;
      while (i>0) {
        i--;
        if (arr[i]%2 != 0) {
          document.write(arr[i], i>0? ",": "");
        }
      }
    </script>
  </font>
</body>

</html>
```


Обратите внимание на эту строчку:

```
document.write(arr[i], i>0? ",": "");
```

Метод `write()` выводит на экран *i*-й элемент массива и затем одно из двух значений: либо символ запятой, либо пустой символ. Это достигается использованием тернарной условной операции `_?_:`. Напоминаю, что в первой части этой операции ставится условие и, при его истинности, выполняется то, что написано во второй части, а при ложности – то, что написано в третьей части.



Функция – это именованная часть программного кода. Функция содержит код, к которому неоднократно обращаются или который имеет логически самостоятельное значение.

Напишем функцию, которая ищет площадь круга и выводит ответ с заданной точностью:

```
<meta charset="utf-8">
<script>
    function squareCircle(r, n) {
        return (Math.PI*r**2).toFixed(n);
    }
    radius = +prompt("Введите радиус");
    count = +prompt("Введите количество знаков после запятой");
    document.write(squareCircle(radius, count));
</script>
```

Как видите, здесь последние три строчки программы относятся к интерфейсу пользователя, а первые строчки содержат функцию, в которую и вынесена аналитическая часть программы.

Если функция универсальна и может использоваться в разных программах, то имеет смысл её вынести в отдельный файл – модуль. Модульное программирование также подразумевает, что отдельные части программы могут быть написаны разными программистами в разное время. Декомпозиция большой программы на отдельные части позволяет легче искать ошибки и вносить изменения в код, а, при загрузке интернет-страницы, ускоряет общий процесс открытия сайта, так как отдельные файлы загружаются параллельно.

Итак, модуль – это отдельный файл, в который вынесены значимые функции. Декомпозируем предыдущую программу на две составные части: интерфейсная часть и модуль с аналитикой:

```
<meta charset="utf-8">
<script src="modul.js"></script>
<script>
    radius = +prompt("Введите радиус");
    count = +prompt("Введите количество знаков после запятой");
    document.write(squareCircle(radius, count));
</script>
```

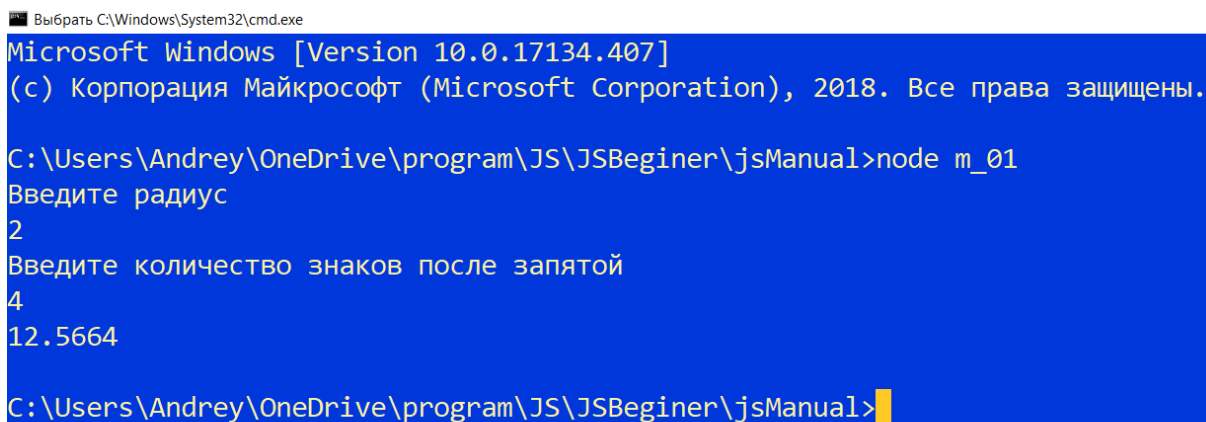
Это содержимое модуля `modul.js`:

```
function squareCircle(r, n) {  
    return (Math.PI*r**2).toFixed(n);  
}
```

Обратите внимание, что в модуль выносят только те части программы, которые непосредственно не занимаются диалогом с пользователем. Прежде всего это важно, чтобы модуль был универсальным и мог быть использован как в браузерной версии программы (см. выше), так и в консольной версии:

```
// это сама программа  
utils = require("./modul"); // подключили модуль с функцией  
read = require("readline-sync");  
// подключили модуль для чтения с клавиатуры  
  
console.log("Введите радиус");  
radius = +read.question();  
console.log("Введите количество знаков после запятой");  
count = +read.question();  
  
console.log(utils.squareCircle(radius, count));  
  
// это модуль к ней - modul.js:  
function squareCircle(r, n) {  
    return (Math.PI*r**2).toFixed(n);  
}  
module.exports.squareCircle = squareCircle;
```

Вот так она работает:



```
Выбрать C:\Windows\System32\cmd.exe  
Microsoft Windows [Version 10.0.17134.407]  
(c) Корпорация Майкрософт (Microsoft Corporation), 2018. Все права защищены.  
  
C:\Users\Andrey\OneDrive\program\JS\JSBeginer\jsManual>node m_01  
Введите радиус  
2  
Введите количество знаков после запятой  
4  
12.5664  
  
C:\Users\Andrey\OneDrive\program\JS\JSBeginer\jsManual>
```

Подробности работы в консоли на Node.js читайте в следующей главе.

Дополнительный материал.

При подключении модулей js (скриптов) к html-документу можно использовать атрибуты, как и в других тегах html.

Атрибут **async**

Скрипт выполняется полностью асинхронно с загрузкой контента страницы. То есть, при обнаружении `<script async src="...">` браузер не останавливает обработку страницы, а спокойно работает дальше. Когда скрипт будет загружен – он выполнится.

```
<script src="1.js" async></script>
<script src="2.js" async></script>
```

Атрибут **defer**

Скрипт выполняется асинхронно, не заставляет ждать страницу, но, в отличие от `async`, браузер гарантирует, что относительный порядок скриптов с `defer` будет сохранён. То есть, в таком коде (с `async`) первым сработает тот скрипт, который раньше загрузится: А в таком коде (с `defer`) первым сработает всегда `1.js`, а скрипт `2.js`, даже если загрузился раньше, будет его ждать. Атрибут `defer` используют в тех случаях, когда второй скрипт `2.js` зависит от первого `1.js`, к примеру — использует что-то, описанное первым скриптом.

```
<script src="1.js" defer></script>
<script src="2.js" defer></script>
```

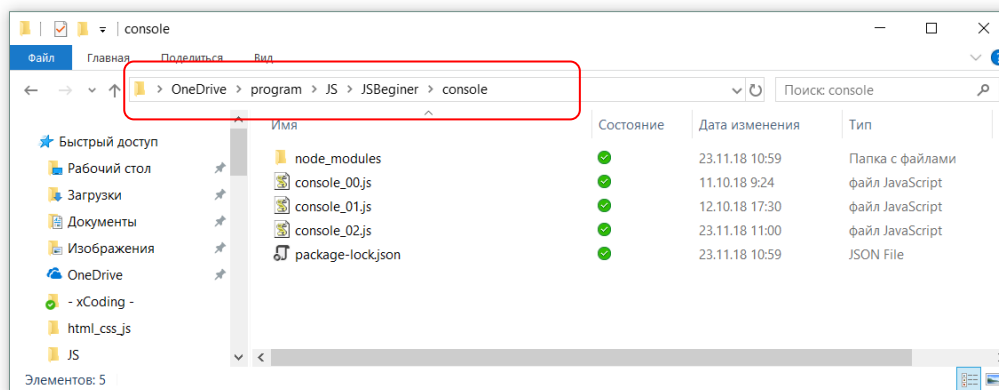
Атрибуты `async/defer` работают только в том случае, если назначены на внешние скрипты, т.е. имеющие `src`. При попытке назначить их на обычные скрипты `<script>...</script>`, они будут проигнорированы.



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

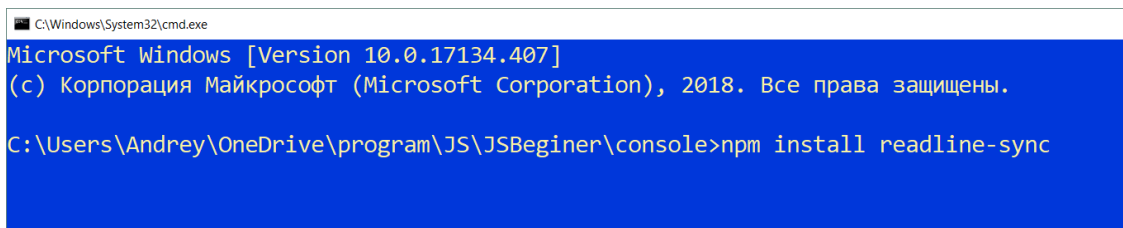
Первое, что нужно сделать: установить дополнительный модуль `'readline-sync'`, Его можно установить, как через терминал Visual Studio Code для текущей папки, так и через терминал Windows

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

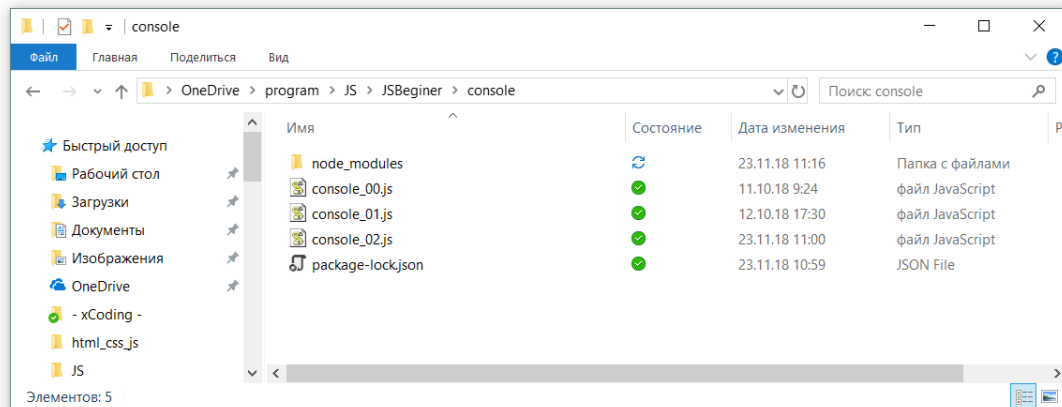


В каком бы терминале вы не работали, чтобы установить в папку с проектом новый пакет нужно в консоли набрать и запустить на исполнение следующую инструкцию:

```
npm install readline-sync
```



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



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

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

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



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

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

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

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

или можно вывести всё содержимое на экран:

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

Разберём способы работы с файлами на примерах.

Пусть есть текстовый файл input.txt:

```
1 2 3 4 5
66 -23 15
999 0 -999
```

Узнаем сколько строк в этом тестовом файле:

```
fs = require("fs");
text = fs.readFileSync("input.txt", "utf-8");
lines = text.split("\r\n");
count = lines.length;
console.log(count);
```

Рассмотрим пример посложнее - нужно вывести на экран суммы чисел из файла построчно:

```
fs = require("fs");
text = fs.readFileSync("input.txt", "utf-8");
lines = text.split("\r\n"); // разбить файл на строки

for (line of lines) { // перебираем все строки
    arr = line.split(" "); // разделим строку по пробелу
    summa = 0; // сумма в текущей строке
    for (elm of arr) { // вычисляем сумму элементов массива
        summa += Number(elm);
    }
    console.log(summa); // выводим очередную сумму на экран
}
```

Предположим есть текстовый файл input.txt:

Мама
Мыла
Раму

Нужно все строки этого файла перевести в нижний регистр и вывести в одну строку:

```
fs = require("fs");
text = fs.readFileSync("input.txt", "utf-8");
lines = text.split("\r\n");
result = "";

for (i=0; i<lines.length; i++) {
    result += lines[i].toLowerCase() + " ";
}

fs.writeFileSync("output.txt", result);
```

Можно использовать усовершенствованный вариант этой же программы:

```
fs = require("fs");
text = fs.readFileSync("input.txt", "utf-8");
lines = text.split("\r\n");
result = lines.join(" ").toLowerCase();
fs.writeFileSync("output.txt", result);
```

Обратите внимание, если метод split() разбивает строку на массив по обозначенному сеператору, то метод join() выполняет противоположное действие: массив объединяет в строку через обозначенный сеператор.

Решим обратную задачу: в файле input.txt есть строка со словами, нужно все слова вывести в файл output.txt построчно (в столбик) и строки пронумеровать.

Файл input.txt:
мама мыла раму
Файл output.txt:
1 мама
2 мыла
3 раму

Проанализируйте программу, решающую данную задачу:

```
fs = require("fs");
```



```

text = fs.readFileSync("input.txt", "utf-8");
firstLine = text.split("\r\n")[0]; // взять первую строку
lines = firstLine.split(" ");
result = "";
for (i=0; i<lines.length; i++) {
    result += String(i+1) + " " + lines[i] + "\r\n";
}
fs.writeFileSync("output.txt", result);

```

Обратите внимание, что в этой программе каждая строчка формируется индивидуально в цикле и добавляется в общую строковую переменную `result`, которую мы в конце и выводим в файл. Строка `"\r\n"` состоит из двух символов `return caret` (возврат каретки) и `new line` (новая строка), она нужна для организации переносов строк. Можно эти же символы получить через их коды в таблице символов:

```

fs = require("fs");
text = fs.readFileSync("input.txt", "utf-8");
firstLine = text.split("\r\n")[0]; // взять первую строку
lines = firstLine.split(" ");

newLine = String.fromCharCode(13) + String.fromCharCode(10);

result = "";
for (i=0; i<lines.length; i++) {
    result += String(i+1) + " " + lines[i] + newLine;
}
fs.writeFileSync("output.txt", result);

```

Например, если нужно вывести данные через символ табуляции, то можно написать две реализации программы:

```
result += String(i+1) + "\t" + lines[i] + newLine;
```

или так:

```
result += String(i+1) + String.fromCharCode(9) + lines[i] + newLine;
```

В обоих случаях результат будет такой:

Файл `output.txt`:

```

1   мама
2   мыла
3   раму

```

Метод `String.fromCharCode()` является универсальным, так как через него можно вывести любой символ, в отличие от использования

управляющей конструкции "\n", предназначенной для вывода спец-символов. Однако и это ограничение можно преодолеть, если после обратного слеша писать код символа в шестнадцатеричном формате:

```
result += String(i+1) + "\u0020" + lines[i] + newLine;
```

В данном случае строка "\u0020" задаёт шестнадцатеричное число, которое в переводе в десятичную систему счисления равно 32, а под этим номером в таблице символов расположен пробел.

Part #6 - Генерация html-документа



Видео лекции

Генерация документов здесь рассматривается только как пример автоматизации обработки данных. В дальнейшем вы можете использовать данный подход для повышения интерактивности сайта и для автоматического формирования новых документов по заданному формату как в этом так и в других языках программирования (php, Python, C#).

Сначала попробуем генерировать не весь, а часть html-документа, например, выведем на экран браузера площадь круга:

```
<html>
  <head>
    <script>
      function squareCircle(r, n) {
        return (Math.PI*r**2);
      }
    </script>
  </head>
  <body>
    <script>
      document.write(squareCircle(2));
    </script>
  </body>
</html>
```

Можно добавить интерактивности в эту программу, если добавить диалог с пользователем:

```
<html>
  <head>
    <script>
      function squareCircle(r, n) {
        return (Math.PI*r**2);
      }
    </script>
  </head>
  <body>
    <script>
      radius = parseInt(prompt("Введите радиус"));
      document.write(squareCircle(2));
    </script>
  </body>
</html>
```

Итак, содержимое страницы можно формировать динамически. Давайте решим задачу посложнее - построим таблицу умножения размерностью указанной пользователем:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Таблица умножения</title>
    <script>
      function getTable(count) {
        str = "<table>";
        for (row=1; row<=count; row++) {
          str += "<tr>";
          for (col=1; col<=count; col++) {
            str += "<td>" + String(row*col) + "</td>";
          }
          str += "</tr>";
        }
        str += "</table>";
        return str;
      }
    </script>
  </head>
  <body>
    <script>
      count = +prompt("Введите размерность таблицы");
      document.write(getTable(count));
    </script>
  </body>
</html>
```

Апробируйте данную реализацию - таблица умножения будет построена, но у неё будет невзрачный вид 😊

Желательно определить стили оформления непосредственно в тегах html-документа или в отдельном файле - таблица стилей, назовём его **style.css**:

```
body {
  background-color: antiquewhite;
}
table {
  font-family: 'Courier New';
  font-size: 24px;
  margin: auto;
}
td {
  width: 36px;
  text-align: center;
}
```

и подключим в заголовочной части html-документа такой строкой:

```
<link rel="stylesheet" href="style.css">
```

После описанных манипуляций мы получим приемлемый вариант:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

Очевидно, что иные параметры изображения (рамки, фон ячеек, цвет шрифта и др.) также можно настроить...

Давайте выделим подсветкой фона главную диагональ таблицы, однако, редактируемый файл становится все «тяжелее» и «тяжелее». Целесообразно, так же, как и стили, сам программный код функций вынести в отдельный файл, например, в modul.js:

```
function getTable(count) {  
    str = "<table>";  
    for (row=1; row<=count; row++) {  
        str += "<tr>";  
        for (col=1; col<=count; col++) {  
            str += "<td bgcolor=" + getColor(row, col) + ">";  
            str += String(row*col) + "</td>";  
        }  
        str += "</tr>";  
    }  
    str += "</table>";  
    return str;  
}  
  
function getColor(row) {  
    return row==col? "yellow": "white";  
}
```

Результат работы программы будет выглядеть примерно так:

1	2	3	4	5	6	7
2	4	6	8	10	12	14
3	6	9	12	15	18	21
4	8	12	16	20	24	28
5	10	15	20	25	30	35
6	12	18	24	30	36	42
7	14	21	28	35	42	49

Чтобы функции модуля были доступны в теле основной программы, нужно в заголовочную часть html-документа вставить строку:

```
<script src="modul.js"></script>
```

Итоговый вид html-документа:

```
<html>  
  <head>  
    <meta charset="utf-8">
```

```
<title>Таблица умножения</title>
<link rel="stylesheet" href="style.css">
<script src="modul.js"></script>
</head>
<body>
  <script>
    count = +prompt("Введите размерность таблицы");
    document.write(getTable(count));
  </script>
</body>
</html>
```

В итоговом html-документе остались только элементы интерфейса - ввод данных и вывод на экран результата, а само оформление и бизнес-логика вынесены в отдельные файлы: `modul.js` и `style.css`.



Видео лекции

Событийное программирование означает привязку предварительно декларированных функций к событиям происходящим в окне браузера: клик мышкой, движение мышкой, ввод символов с клавиатуры, закрытие окна браузера и т.п. Момент запуска функции не задаётся в программе, а определяется наступлением события. Программист в коде программы прописывает какую функцию запускать при наступлении какого события.

Возможны **три способа** организации обработки событий:

- 1) через добавление кода js прямо в теги html-документа;
- 2) через определение ссылок на события в js коде;
- 3) через создание листа событий в js коде.

Тут просто примеры, детали смотрите в видеоуроке...

В **способе первом** мы прямо в тегах элемента указываем событие и какую функцию запускать при наступлении указанного события. Функция должны быть предварительно описана. В функции мы получаем сам элемент по его id, полученный элемент сохраняем в переменную и потом работаем с содержимым этого элемента через переменную. Для проверки работоспособности кликайте по INC.

Способ 1.

```
<html>
  <head>
    <script>
      function inc() {
        elmResult = document.getElementById("result");
        num = Number(elmResult.innerHTML);
        elmResult.innerHTML = ++num;
      }
    </script>
  </head>
  <body>
    <div onclick="inc()">INC</div>
    <div id="result">0</div>
  </body>
</html>
```

В способе втором аналогичным образом организована функция для работы с содержимым элемента, только обработчик события присваивается не в теге html элемента, а непосредственно в js-коде. Обратите внимание, что js-код должен располагаться позже html-кода с элементами, чтобы js программа могла «увидеть» элементы. Второй способ предпочтительнее, он позволяет разделить работу между верстальщиком html-страницы и кодером.

Способ 2.

```
<html>
  <head>
  </head>
  <body>
    <div id="inc">INC</div>
    <div id="result">0</div>
    <script>
      function inc() {
        elmResult = document.getElementById("result");
        num = Number(elmResult.innerHTML);
        elmResult.innerHTML = ++num;
      }
      elmInc = document.getElementById("inc");
      elmInc.onclick = inc;
    </script>
  </body>
</html>
```

В способе третьем используют так называемый «слушатель событий» он позволяет не только в удобной форме назначить на один элемент несколько обработчиков событий, но и указать порядок обработки событий в иерархии («матрёшке») html-документа

Способ 3.

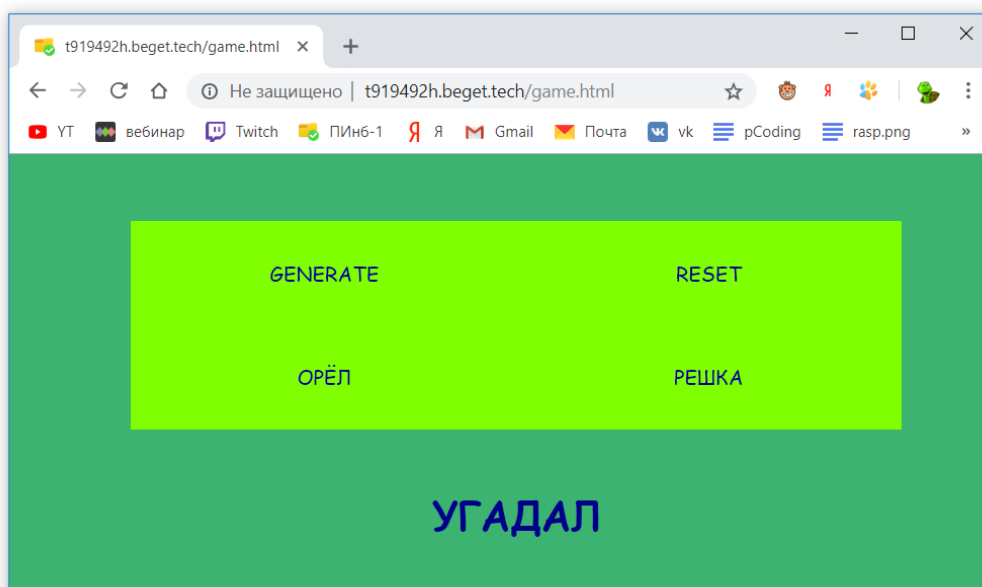
```
<html>
  <head>
  </head>
  <body>
    <div id="inc">INC</div>
    <div id="result">0</div>
    <script>
      function inc() {
        elmResult = document.getElementById("result");
        num = Number(elmResult.innerHTML);
        elmResult.innerHTML = ++num;
      }
    </script>
  </body>
</html>
```



```
elmInc = document.getElementById("inc");  
elmInc.addEventListener('click', inc, false);  
</script>  
</body>  
</html>
```

Если третий параметр в «слушателе» определён как `false`, то выбирается порядок обработки, который называется «всплытием событий». Это означает, что для всех элементов, к которому приписан обработчик события, сначала событие обрабатывается на самом внутреннем элементе, потом на том, в котором он располагается, потом на следующем ... и так с самого маленького элемента постепенно происходит всплытие вплоть до самого глобального элемента `document`. Если же третий параметр определить как `true`, тогда будет задействован обратный порядок инициации событий от самого внешнего элемента, к самому внутреннему. Такой порядок называется «захватом событий».

Вы можете использовать разные события, не только клик мышкой, для расширения функционала приложения и удобства работы пользователя. Так же вы можете немного приукрасить внешний вид вашего приложения за счёт использования стилей.



Вы можете посмотреть как организована эта игра по угадыванию стороны брошенной монетки (орёл/решка) на моём сайте-шаблоне: <http://t919492h.beget.tech/>

Далее три таблицы содержат файлы html, js и css для этой игры.

game.html

```
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <br>
    <table>
      <tr>
        <td><div id="gen">GENERATE</div></td>
        <td><div id="res">RESET</div></td>
      </tr>
      <tr>
        <td>
          <div id="var0">ОПЁЛ</div>
        </td>
        <td>
          <div id="var1">ПЕШКА</div>
        </td>
      </tr>
    </table>
    <br>
    <div id="result" align="center">_____</div>
    <script src="game.js"></script>
  </body>
</html>
```

game.js

```
var xxx = 0; // загаданное число
var elm = document.getElementById("result");
function getRnd() {
  return Math.floor(Math.random() * 2);
}
function setNewRnd() {
  xxx = getRnd();
  elm.innerHTML = "ЗАГАДАНО";
}
function reset() {
  elm.innerHTML = "0";
}
function check(src) {
  let human = src.target.innerHTML == "ОПЁЛ" ? 0 : 1;
  elm.innerHTML = human==xxx? "УГАДАЛ": "ПРОИГРАЛ";
}
with (document) {
  getElementById("gen").onclick = setNewRnd;
  getElementById("res").onclick = reset;
  getElementById("var0").onclick = check;
  getElementById("var1").onclick = check;
}
```

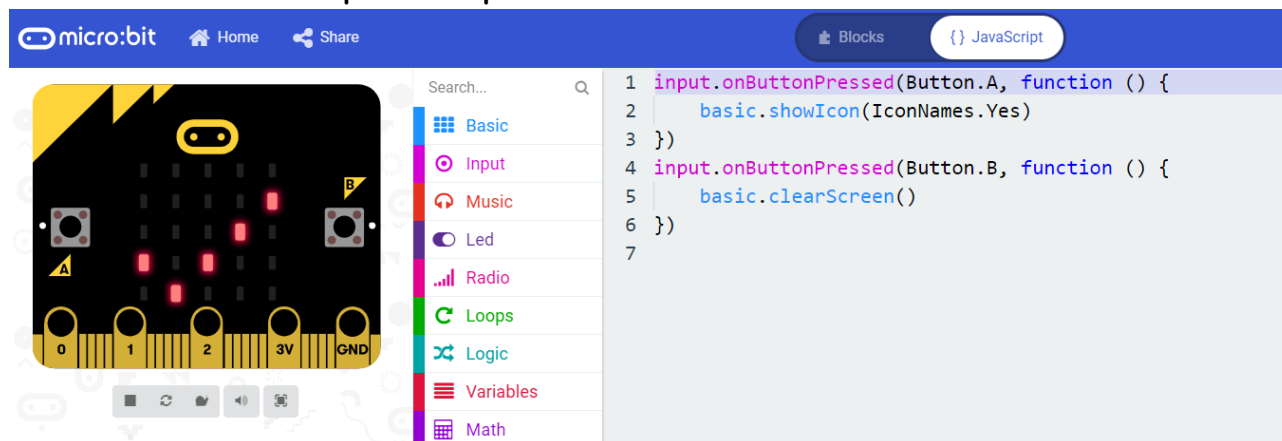
```
body {
    background-color: #3CB371;
    font-family: 'Comic Sans MS';
    font-size: 24pt;
    color: #000088;
    font-weight: 600;
    margin-left: 30px;
}
a:link, a:visited {
    text-decoration: none;
    font-family: 'Courier New';
    font-size: 20pt;
    color: #FFEEEE;
    font-weight: 600;
    margin-left: 30px;
}
a:hover {
    color: yellow;
    font-size: 21pt;
    margin-left: 50px;
}
p {
    font-family: 'Courier New';
    font-size: 22pt;
    color: #000088;
}
.send {
    width: 12em;
    height: 2em;
    font-family: 'Comic Sans MS';
    font-size: 22pt;
    color: #000088;
    text-align: center;
    font-weight: bold;
}
table {
    background-color: chartreuse;
    margin: auto;
}
tr {
    height: 80;
}
td {
    width: 300;
    text-align: center;
}
```



Видео лекции

Будем кодить тут: <https://makecode.microbit.org/>

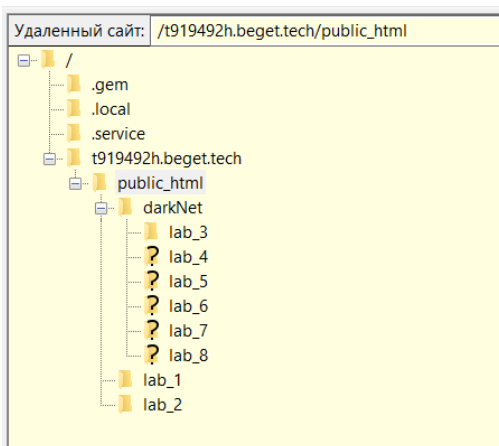
Так выглядит редактор:



Part #9 - НАСТРОЙКИ МОЕГО САЙТА



Это папки моего сайта:



Все файлы моего сайта вы можете скопировать прямо с сайта:

<http://t919492h.beget.tech/>

Вы не сможете скопировать только файл darkNet.php, а в нём программа для проверки пароля при входе.

Это файлы в публичной папке:

Имя файла	Размер	Тип файла	Последнее...	Права	Владеле...
..					
darkNet		Папка с файлами	30.11.18 22:...	drwx-----	9643 601
lab_1		Папка с файлами	23.11.18 20:...	drwx-----	9643 601
lab_2		Папка с файлами	30.11.18 23:...	drwx-----	9643 601
darkNet.php	1 551	Файл "PHP"	30.11.18 23:...	-rwxr-xr-x	9643 601
favicon.ico	67 646	Файл "ICO"	23.11.18 20:...	-rw-----	9643 601
index.html	651	Chrome HTML Document	30.11.18 22:...	-rwx-----	9643 601
lr_1.html	401	Chrome HTML Document	30.11.18 23:...	-rwx-----	9643 601
lr_2.html	400	Chrome HTML Document	30.11.18 23:...	-rwx-----	9643 601
modul.js	276	файл JavaScript	30.11.18 22:...	-rw-----	9643 601
style.css	613	CSS-документ	30.11.18 23:...	-rw-----	9643 601

7 файлов и 3 каталога. Общий размер: 71 538 байт

Это файлы в папке darkNet:

Имя файла	Размер	Тип файла	Последнее...	Права	Владеле...
..					
lab_3		Папка с файлами	30.11.18 19:...	drwx-----	9643 601
lab_4		Папка с файлами	30.11.18 19:...	drwx-----	9643 601
lab_5		Папка с файлами	30.11.18 19:...	drwx-----	9643 601
lab_6		Папка с файлами	30.11.18 19:...	drwx-----	9643 601
lab_7		Папка с файлами	30.11.18 19:...	drwx-----	9643 601
lab_8		Папка с файлами	30.11.18 19:...	drwx-----	9643 601
dark.html	471	Chrome HTML Document	30.11.18 22:...	-rw-----	9643 601
lr_3.html	485	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601
lr_4.html	458	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601
lr_5.html	458	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601
lr_6.html	458	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601
lr_7.html	458	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601
lr_8.html	458	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601

7 файлов и 6 каталогов. Общий размер: 3 246 байт

Содержимое любой папки с лабораторками однотипно:

Имя файла	Размер	Тип файла	Последнее...	Права	Владеле...
..					
task_1.html	334	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601
task_2.html	570	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601
task_3.html	215	Chrome HTML Document	30.11.18 19:...	-rw-----	9643 601

3 файла. Общий размер: 1 119 байт



Способ 1 – вход под паролем через js

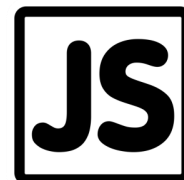
Тут всё достаточно просто. Если вы понимаете немного в тегах html.

Вот так можно сделать ссылочку на страничку с кодом:

```
<a href=darkNet.html>Войти в darkNet через js</a>
```

А вот так просто можно оформить саму страницу входа через пароль:

```
<html>
<head>
  <meta charset="utf-8">
  <title>Вход через JS</title>
</head>
<body>
  <script>
    password = prompt("Введите пароль");
    document.location = "darkNet/" + password + ".html";
  </script>
</body>
</html>
```



Как видите тут пароль, по сути, является именем страницы, на которую мы будем переходить. Если папку, в которой лежит скрытая часть сайта ещё можно увидеть через «просмотреть код страницы», то вот адрес страницы уже не увидеть. И другие страницы в папке darkNet сделайте с каким-нибудь «префиксом», чтобы посторонний не смог их предугадать. То есть, если вы и сделаете вход под паролем, а имена папок и файлов будут интуитивно понятны, типа такого: «[http://t919492h.beget.tech/darkNet/lr_3.html](#)», то на них можно будет перейти, просто набрав в адресной строке браузера. Добавьте свой префикс к именам, например: «[http://t919492h.beget.tech/darkNet/**zer0**_lr_3.html](#)».

Способ 2 – вход под паролем через php

Небольшая справка. Язык php (первоначально Personal Home Page Tools) довольно старенький (1995г.) и несложный для начального обучения – скриптовый, как и JS, но возни больше с настройками из-за его серверной локализации – он работает на стороне сервера, отдавая клиенту только результаты своей работы... Php поддерживается подавляющим большинством хостинг-провайдеров и является одним из лидеров среди языков, применяющихся для создания динамических веб-сайтов, обеспечивающий работу с файлами и сетевыми базами данных.



Программу на php, конечно, можно запустить и у себя на компьютере через браузер. Это возможно, если вы заблаговременно установили специальный сервер

локально на своём компьютере, имитируя хостинг в интернете... Так делают, когда редактируют код программы на php, но мы с вами не будем сейчас изучать php - это будет на 3-4 курсах...

Чтобы вам запустить программу php для входа по паролю (которую уже скопировали или ещё скопируете у меня) и испытать её - нужно загрузить её на свой сайт - там есть уже развёрнутая платформа для php...

А для самых продвинутых и любопытных - читать тут: <http://www.denwer.ru/>

Денвер включает в себя кроссплатформенные продукты: сервер Apache, язык программирования PHP, сетевую базу данных MySQL.

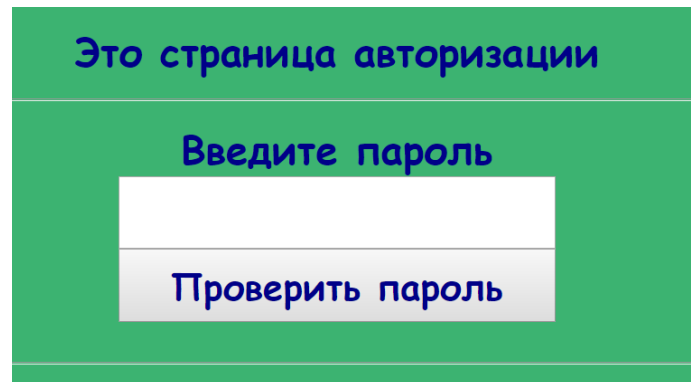
Итак, программу для входа по паролю на php вы можете взять у меня, но не с учебного сайта-шаблона, а тут из учебника (или из файла 185.txt/183.txt - лучше из текстового файла, так как не будет искажений в символах) - **darkNet.php**:

```
<?php
    if (isset($_POST['btn'])) {
        $pass = '666'; // тут задайте свой пароль
        if ($_POST['password'] == $pass) {
            header('Location: darkNet/dark.html');
            exit;
        }
        else {
            echo '<div align=center>Пароль не подходит</div>';
        }
    }
?>
<html>
    <head>
        <title>darkNet</title>
        <meta charset=utf-8>
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <center>
            <br>Это страница авторизации
            <hr>Введите пароль
            <form method="POST">
                <input class="send" type="Password" name=password><br>
                <input class="send" type="Submit" name=btn value="Проверить">
            </form>
        </center>
        <hr>
    </body>
</html>
```

Как видите в файле две части: сначала (красным цветом) идёт код на php - он проверяет введённый пароль и, если правильный, то передаёт управление на страницу в папке (darkNet/darkNet.html - имя папки и имя страницы поменяйте, чтобы никто не подобрал, не волнуйтесь - их никто не увидит внутри кода, даже когда

данная страница будет загружена в браузер клиента. Через "просмотр кода страницы" можно увидеть только вторую часть, а именно, html-код.

Чтобы объекты `input` (поле для ввода пароля и кнопка для отправки его на проверку) смотрелись приемлемо:



Это страница авторизации

Введите пароль

Проверить пароль

я добавил к ним в теги **атрибут** `class="send"` и добавил его описание в таблицу стилей - в файл `style.css`:

```
.send {  
    width: 12em;  
    height: 2em;  
    font-family: 'Comic Sans MS';  
    font-size: 22pt;  
    color: #000088;  
    text-align: center;  
    font-weight: bold;  
}
```




Содержание

1	<u>Диалоги и ветвления</u>	
2	<u>Циклы</u>	
3	<u>Массивы</u>	
4	<u>Файлы</u>	
5	<u>Генерация html-документа</u>	
6	<u>Событийное программирование</u>	
7	<u>Программирование микроконтроллеров</u>	
8	<u>darkNet</u>	
9	<u>ИТ-Хакатон</u>	



Диалоги и ветвления.

Вывод в документ: `document.write`.
Диалоговые окна: `alert`, `confirm`, `prompt`.

Операторы ветвления: `if then`, `switch`.

Тернарная условная операция: `<_?_:>`.

Остаток от целочисленного деления: `%`.

Округление до заданной точности: `.toFixed()`.

Константы из стандартной библиотеки: `Math.PI`.

Читаем Методичку <https://pcoding.ru/pdf/algo.pdf>

Смотрим плейлист «Азбука программирования» с сайта <https://pcoding.ru/>

Задания для самостоятельного исполнения.

1. Пользователь вводит число. Программа определяет его чётность/нечётность и выводит ответ в виде слова: «чётное» или «нечётное».
2. Пользователь вводит два целых числа. Программа определяет: кратно ли большее из них по отношению к меньшему. Например, ввели два числа 11 и 33 – ответ: «33 кратно 11».
3. Пользователь последовательно вводит два целых числа. Программа выбирает большее из двух.
4. Пусть даны три числа. Программа выбирает меньшее из трёх.
5. Пользователь задаёт диапазон – это два целых числа от Min до Max. Программа генерирует случайное целое число в заданном диапазоне.
6. Пусть даны длины трёх отрезков. Программа определяет можно ли из них построить треугольник.
7. Программа демонстрирует пользователю последовательно два диалоговых окна `prompt`, где предлагается ввести значение радиуса круга и количество знаков после запятой в ответах. В каждом из окон также установлено значение по умолчанию, например, для радиуса круга – 1, для количества знаков – 3. Программа в ответ выводит в окно браузера два ответа в столбик: длина окружности и площадь круга с заданной точностью.
8. Пользователь вводит коэффициенты (a, b, c) квадратного уравнения. Программа ищет корни уравнения и выводит на экран.
9. Пользователь вводит арабскую цифру (1..9), программа в ответ выводит римскую (I..IX). *Используйте оператор switch.*
10. Пользователь вводит возраст молодого человека. Программа в ответ выводит его категорию («молодой», «призывник», «дембель»). Используется правило, что призывником считается человек в возрасте от 18 до 27 лет включительно. *Используйте оператор switch.*



Циклы.

Задания для самостоятельного исполнения.

1. Пользователь вводит последовательно два целых числа: valueBegin и valueEnd (диапазон от -100 до +100). Программа выводит на экран в столбик все числа от valueBegin до valueEnd.

Отметим, что valueBegin и valueEnd могут быть в любом соотношении: равны друг другу, больше или меньше.

Сделать программу с использованием трёх циклов: for, while, do while - можно в одной программе последовательно написать три цикла.

2. Пользователь вводит двоичное целое число, программа переводит его из двоичной в десятичную систему счисления.
3. Пользователь вводит десятичное целое число, программа переводит его из двоичной в десятичную систему счисления.
4. Разработайте программу вычисления квадратного корня Z числа X методом Герона:

$$Z_n = (Z_{n-1} + X/Z_{n-1})/2, \text{ где } Z_0 = 1.$$

Алгоритм выполняется за n шагов. N - определяется пользователем. Последовательно, с каждым шагом Z приближается к истинному значению корня числа. Чем больше повторений цикла - n , тем точнее будет вычислен ответ. В программе сделать так: пользователь вводит значения X и n , а программа вычисляет приближённое значение корня Z и выводит его на экран.

Шаги работы алгоритма для числа $X = 9$:

$$Z_0 = 1$$

$$Z_1 = (Z_0 + X/Z_0)/2 = (1 + 9/1)/2 = 5$$

$$Z_2 = (Z_1 + X/Z_1)/2 = (5 + 9/5)/2 = 3.4$$

$$Z_3 = (Z_2 + X/Z_2)/2 = (3.4 + 9/3.4)/2 = 3.02$$

5. Разработайте программу вычисления квадратного корня Z числа X методом Герона с заданной точностью q .

В программе сделать так: пользователь вводит значения X и q , а программа вычисляет приближённое значение корня Z и выводит его на экран. Точность q , например, 0,01, обозначает, что предыдущее значение Z_{n-1} и текущее значение Z_n отличаются друг от друга не более, чем на 0,01. Количество шагов алгоритма заранее не известно и его не надо задавать. Программа последовательно, шаг за шагом, сама приблизится к заданной точности вычисления Z .



Массивы.

Задания для самостоятельного исполнения.

Уровни сложности задач: #1, #2, #3 – простые, #4, #5, #6, #7 – средней сложности, #8 – сложная.

1. Разработайте программу генерации массива случайных целых чисел.
Пользователь вводит в консоли с клавиатуры количество элементов count и диапазон для генерации чисел, например, begin=-10 и end=10. Правая граница не входит в диапазон для генерации. Программа заполняет массив и выводит его элементы на экран в одну строку через пробел, например, так:
9 7 7 0 -4 2 -7 -8 -1
2. Разработайте программу для проверки наличия/отсутствия определённого элемента в массиве целых чисел arr.
Сначала массив из 10-ти элементов заполнить случайными целыми числами (как в программе #1). Затем пользователь вводит число, а программа проверяет, есть ли оно в массиве.
3. Разработайте программу, которая ищет разницу между максимальным и минимальным элементами массива (из программы #1).
4. Разработайте программу циклического сдвига элементов массива вправо.
При циклическом сдвиге вправо все элементы массива сдвигаются на одну позицию вправо, а самый крайний справа уходит на позицию самого левого.
5. Разработайте программу, которая из двух массивов делает один содержащий только неповторяющиеся элементы.
Сначала сгенерируйте два массива arrA и arrB (как в программе #1). Массивы могут содержать разное количество элементов. Некоторые элементы в массивах могут быть одинаковые.
6. Создайте двумерный массив и заполните числами как указано на рисунке. Разработайте программу, которая ищет сумму элементов главной (слева-сверху вправо-вниз) и второстепенной диагоналей этой матрицы.

1	2	3
8	9	4
7	6	5
7. Разработайте программу для транспонирования матрицы – двумерного массива, заполненного ранее (task #6).
Транспонирование – это отражение элементов относительно главной диагонали матрицы.
8. Напишите программу, которая сама заполняет двумерный массив «по спирали», как в задаче #6:
Пользователь вводит число в диапазоне от 1 до 9, например, 3, а программа заполняет массив и выводит его на экран:



Файлы.

Задания для самостоятельного исполнения.

Уровни сложности задач: #1, #2, #3 – простые, #4, #5, #6 – средней сложности, #7, #8 – сложные.

Для лабораторной работы используйте текстовый файл input.txt (в каждой строке числа, разделённые пробелом):	<div>1 2 3 4 5 6 7 8 9 10 11 2 13 14 15 16 17 18 19 20</div>
--	--

1. Программа читает текстовый файл `input.txt` и выводит его на экран консоли без изменений.
2. Программа читает текстовый файл `input.txt` и выводит на экран консоли только нечётные строки файла.
3. Программа читает текстовый файл `input.txt` и выводит его на экран консоли в обратном порядке (от последней строки к первой).
4. Программа читает текстовый файл `input.txt`, заменяет в нём символы пробел на символы табуляции и сохраняет в выходном файле `output.txt`:

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

5. Программа читает текстовый файл `output.txt`, пользователь вводит номер колонки, например, 3, а программа считает сумму чисел в указанной колонке и выводит её на экран.
6. Это задание из книги "ООП в C++". Р. Лафоре. стр. 139. Напишите код, который строит пирамиду из символов "X". На вход подаётся одно число - высота пирамиды, на выходе - сама пирамида. Например, если пользователь ввёл 4, то будет построена такая симметричная пирамида высотой в 4 символа:

```
  X
 XXX
XXXXX
XXXXXXXX
```

7. В первом текстовом файле (`1.txt`) хранится список фамилий. Во втором текстовом файле (`2.txt`) хранится список фамилий. Оба файла создайте сами. Некоторые из фамилий повторяются (одинаковые в обоих файлах). Программа формирует третий файл (`3.txt`), в который попадают только те фамилии, которые встречаются в обоих файлах.
8. В первом текстовом файле (`1.txt`) хранится список фамилий. Во втором текстовом файле (`2.txt`) хранится список фамилий. Некоторые из фамилий повторяются (одинаковые в обоих файлах). Программа формирует третий файл (`3.txt`), в который попадают только уникальные фамилии, то есть те, которые встречаются только в одном из файлов.



Генерация html-документа.

Задания для самостоятельного исполнения.

Видео лекции

Задачи для браузерной реализации

1. Напишите программу, которая спрашивает у пользователя размеры таблицы - количество строк и количество столбцов (например, 3 и 7) и в ответ строит таблицу умножения по заданным параметрам 3x7. Для организации диалога используйте `prompt()`.
2. Напишите программу, которая спрашивает у пользователя размерность таблицы умножения (например, если пользователь ввёл 4, то строится таблица умножения 4x4) и в ответ строит таблицу умножения с выделенной строкой сверху и выделенным столбцом слева, которые отличаются по цвету и содержат номера столбцов и строк, соответственно.

1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16

Цвет отдельной ячейки можно задавать атрибутом тега `td`, например:

```
<td bgcolor=green>4</td>
```

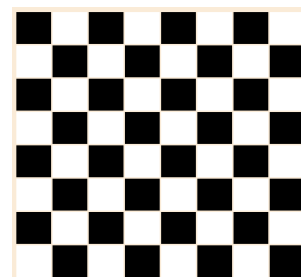
3. Напишите программу, которая строит поверхность шахматной доски (размерность задаёт пользователь):

Высоту строки можно задавать атрибутом тега `tr`:

```
<tr height=35> </tr>
```

Ширину ячейки можно задать так:

```
<td width=25> </td>
```



Задачи для реализации в Node.js

4. Напишите программу, как в задании №1, только для Node.js. Пусть пользователь вводит размеры таблицы в командной строке терминала при запуске программы, например так: `node program.js 3 7` (используйте массив `process.argv`).
5. Напишите программу, как в задании №2, только для Node.js. Пусть пользователь вводит размеры таблицы в процессе диалога (используйте модуль `readline-sync`).
6. Напишите программу, как в задании №3, только для Node.js. Пусть пользователь имеет возможность вводить размеры таблицы как в процессе диалога, так и перед запуском программы в командной строке терминала.

Сначала программа проверяет наличие параметра, переданного из командной строки, если его нет, то организует диалог с пользователем. Если из командной строки или в процессе диалога в программу было передано нечисловое значение, то программа назначает значение по умолчанию равное 8.



Событийное программирование.

Задания для самостоятельного исполнения.

Уровни сложности задач: #1 - простая, #2 - средней сложности, #3 - немного посложнее.
Для тех, кто хорошо разбирается в событийном программировании можете заменить эти типовые задачи на свой собственный проект - это может быть игра, тест, прикладная программа и т.п.

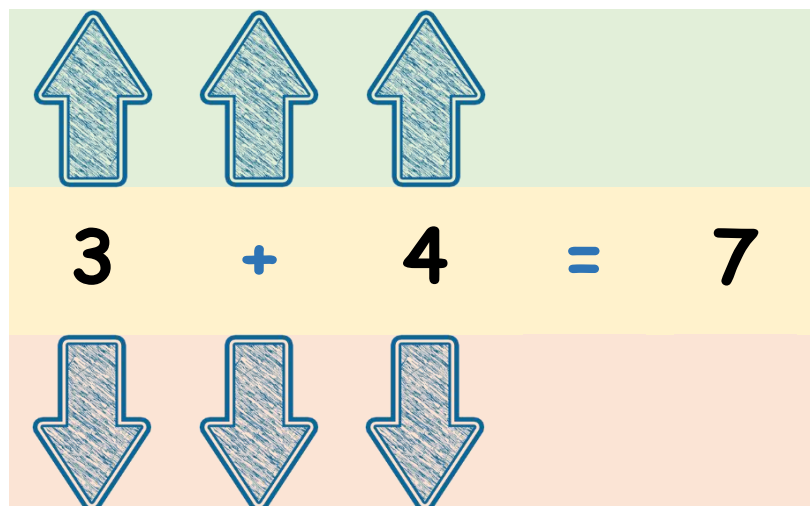
Видео лекции

Шаблон программы смотрите тут: [ОРЁЛ/РЕШКА](#).

1. Напишите программу, которая обрабатывает события мышки. На экране браузера таблица с тремя ячейками: в центре целое число, например, ноль, а по бокам стрелки. При клике мышкой по стрелкам число уменьшается/увеличивается на единицу, при клике по числу - сбрасывается в ноль.



2. На основе предыдущей программы сделайте простой калькулятор. Пользователь мышкой может выбрать значение первого числа, значение второго числа, арифметическое действие
- + * /
и нажать на клавишу равно для получения результата.



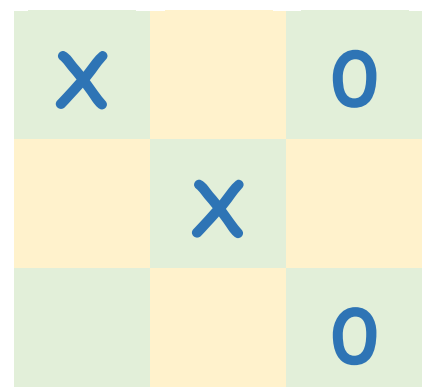
При запуске программы первое и второе число выставлены на ноль.

Пусть операция деления возвращает результат целочисленного деления.

2. Напишите программу для игры в крестики-нолики 3x3. Человек играет крестиками, программа ноликами. Добавьте клавишу сброс. Пусть человек ходит первый.

Если есть желание, то можете добавить возможность выбора:

- кто ходит первый,
- каким фишками играет человек.





Программирование микроконтроллеров.

Задания для самостоятельного исполнения.

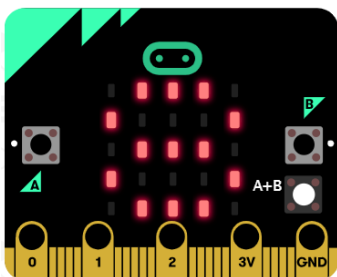
Уровни сложности задач: #1 – простая, #2 – средней сложности, #3 – сложная.

Видео лекции

Будем кодить тут: <https://makecode.microbit.org/>

Так как у вас на руках нет реальных приборов *micro:bit*, то будем использовать ограниченное количество событий и датчиков, а сконцентрируемся на алгоритмической сложности.

1. Разработать программу «Счётчик» с функционалом:



- При старте программы на индикаторном поле отображается цифра ноль.
- При нажатии на клавишу A показания счётчика увеличиваются на 1.
- При нажатии на клавишу B показания счётчика уменьшаются на единицу.
- При нажатии на клавиши AB – счётчик обнуляется.

2. Разработать программу «Звёздное небо». Эта программа использует счётчик из предыдущего задания. В программу добавьте ограничение: счётчик может устанавливать число в диапазоне от 0 до 25. Пользователь имеет возможность выбрать количество звёзд (от 0 до 25). Далее, при событии «Встряхивание», индикаторное поле заполняется включёнными на случайно выбранных позициях индикаторами.

Уточнение (для получения максимальной оценки за данную задачу):

- количество звёзд должно быть таким как выбрал пользователь,
- яркость индикатора и его позиция выбирается случайным образом.

Чтобы звезда была видимой делайте её яркость в диапазоне от 1 до 255 включительно.

Уважающий себя **программный инженер** разобьёт программу на две части:

- в первой части будут продекларированы функции и переменные,
- во второй – обработчики событий, например так:

```
// тут декларации переменных и функций
// . . .
let countDec = function () {
    basic.showNumber(--count);
}
// . . .
// тут декларации обработчиков событий
input.onButtonPressed(Button.B, countDec);
// . . .
```


Как сделать проверку несовпадения новой сгенерированной позиции звёздочки на несовпадение с ранее сгенерированными?

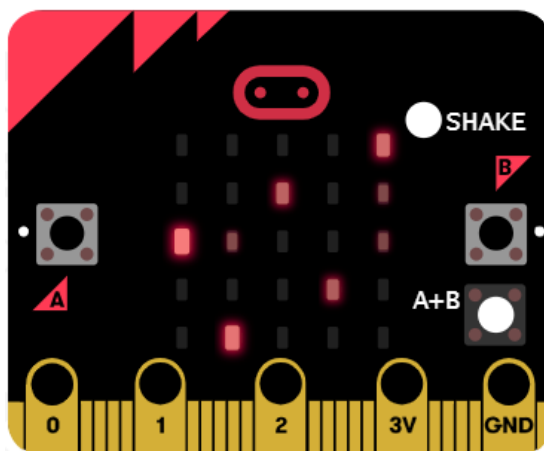
Один из подходов такой:

— пусть позиции будут в диапазоне от 0 до 24-х включительно (так как поле $5 \times 5 = 25$ позиций),

— сгенерированные позиции добавлять в массив позиций,

— предварительно проверяя, нет ли там уже такого элемента (например, так: `arrayPositions.indexOf(newPos)` – этот метод возвращает -1, если элемента нет в массиве),

— после добавление в массив позиций неповторяющихся элементов в количестве указанном пользователем можно вывести их на индикаторное поле.

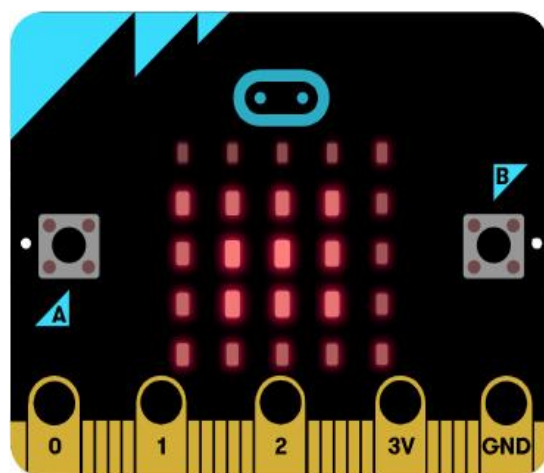


3. Разработать программу «Змейка».

Заполнить двумерную матрицу индикаторов 5×5 по спирали, так чтобы яркость индикатора возрастала с некоторым равномерным шагом от левого верхнего пикселя по часовой стрелке по спирали к центру, где яркость будет максимальной.

События для запуска заполнения и сброса индикаторов определите сами.

Индикаторы должны загораться не все сразу, а последовательно с некоторой паузой, чтобы имитировать движение по спирали.





darkNet.

Задания для самостоятельного исполнения.

Задание одно: сделать часть своего сайта, например, с лабораторными работами от 3-й до последней скрытыми от стороннего пользователя. Вход в эту часть сайта по паролю.

Выполнить задание можно в одном из двух вариантов: через код на js или php. Как именно это делать смотрите в разделе: [тут](#).



ИТ-Хакатон.

Задания для самостоятельного исполнения.

1. По определённой теме (темы обозначены ниже) подготовить эссе (короткая статья на 2-3 страницы) с картинками, схемами, алгоритмами и программным кодом (при необходимости), объяснениями и своим личным мнением по рассматриваемому вопросу. Разместить эссе в формате pdf на своём сайте.
2. Подготовить презентацию к публичному выступлению по одной из обозначенных тем. Объем презентации до 10-ти слайдов.
3. Публично выступить на одном из заключительных занятий. Выступление до 5-ти минут.



*Можно выбрать темы не из списка
по согласованию с преподавателем.*

Темы для выступлений:

1. Рекурсивные функции.
2. IDE для web-программиста.
3. Первый язык - с чего начинать учить программирование.
4. Способы организации ветвления в js.
5. Архитектура программы: html+css+js+frameworks.
6. Мой проект на js (игра, прикладная программа, сайт).
7. Устарел ли Тест Тьюринга?
8. Цветокodирование: CMYK vs RGB.
9. Алгоритм сортировки _____ (нужно выбрать самому).
10. Почему Macbook лучше ноута на Windows.
11. Почему ноут на Windows лучше Macbook.
12. Зачем играм облачные сервисы?
13. Мой список TOP 5 программ для смартфона.
14. Насколько хорош искусственный интеллект в играх?
15. За что получает зарплату геймдизайнер?
16. ИТ-профессии будущего...
17. Как собрать игровой компьютер.
18. Как найти экстремум функции методом Золотого сечения?
19. Один день из жизни студента ИТ-ишника в 2038 году.
20. Мой преподаватель - ИИ (искусственный интеллект).
21. Какой софтверный стартап я бы предложил.

Задачи для размышления

Интересная задача

```
var y = 0;
if (y == (1 || 11 || 111))
    { b = true; }
else
    { b = false}
if (y == 1 || 11 || 111)
    { b = true; }
else
    { b = false}
```

Почему if`ы выдают разный результат?

Эти материалы ещё редактируются...

В разных частях программы можно использовать для разных целей один и тот же идентификатор. Имеется ввиду, что одна и та же переменная (с одним и тем же именем) используется в одном месте для хранения некоторой одной величины, а в другом месте программы эта же переменная используется для хранения другой величины. Это плохая практика, но такое встречается повсеместно. Чтобы как-то

В программах можно объявлять переменные для хранения данных. Это делается при помощи операторов `var` или `let` – они по-разному задают области видимости переменных, то есть те части кода, в которых эти переменные доступны для использования:

В большинстве случаев декларацию переменной и присваивание ей значения совмещают: `var x = 10`.

```
let x = 0;
console.log(x);

for (i=0; i<5; i++) {
  let x = i;
  console.log(x);
}

console.log(x);
```

Переменные, которые названы **БОЛЬШИМИ_БУКВАМИ**, являются константами, то есть, никогда не меняются. Как правило, они используются для удобства, чтобы было меньше ошибок.

Важность директивы `var`

`num = 5`; // переменная `num` будет создана, если ее не было

В режиме "use strict" так делать уже нельзя.

Следующий код выдаст ошибку:

```
"use strict";
num = 5; // error: num is not defined
var something;
"use strict"; // слишком поздно
num = 5; // ошибки не будет, так как строгий режим не активирован
<div id="test"></div>
<script>
test = 5; // здесь будет ошибка!
alert( test ); // не сработает
</script>
```

Константы

Константа — это переменная, которая никогда не меняется. Как правило, их называют большими буквами, через подчёркивание. Например:

```
var COLOR_RED = "#F00";
var COLOR_GREEN = "#0F0";
var COLOR_BLUE = "#00F";
var COLOR_ORANGE = "#FF7F00";
var color = COLOR_ORANGE;
alert( color ); // #FF7F00
```

Технически, константа является обычной переменной, то есть её можно изменить. Но мы договариваемся этого не делать. Зачем нужны константы? Почему бы просто не писать `var color = "#FF7F00"`?

1. Во-первых, константа `COLOR_ORANGE` — это понятное имя. По присвоению `var color="#FF7F00"` непонятно, что цвет — оранжевый. Иными словами, константа `COLOR_ORANGE` является «понятным псевдонимом» для значения `#FF7F00`.

2. Во-вторых, опечатка в строке, особенно такой сложной как `#FF7F00`, может быть не замечена, а в имени константы её допустить куда сложнее.

Константы используют вместо строк и цифр, чтобы сделать программу понятнее и избежать ошибок.