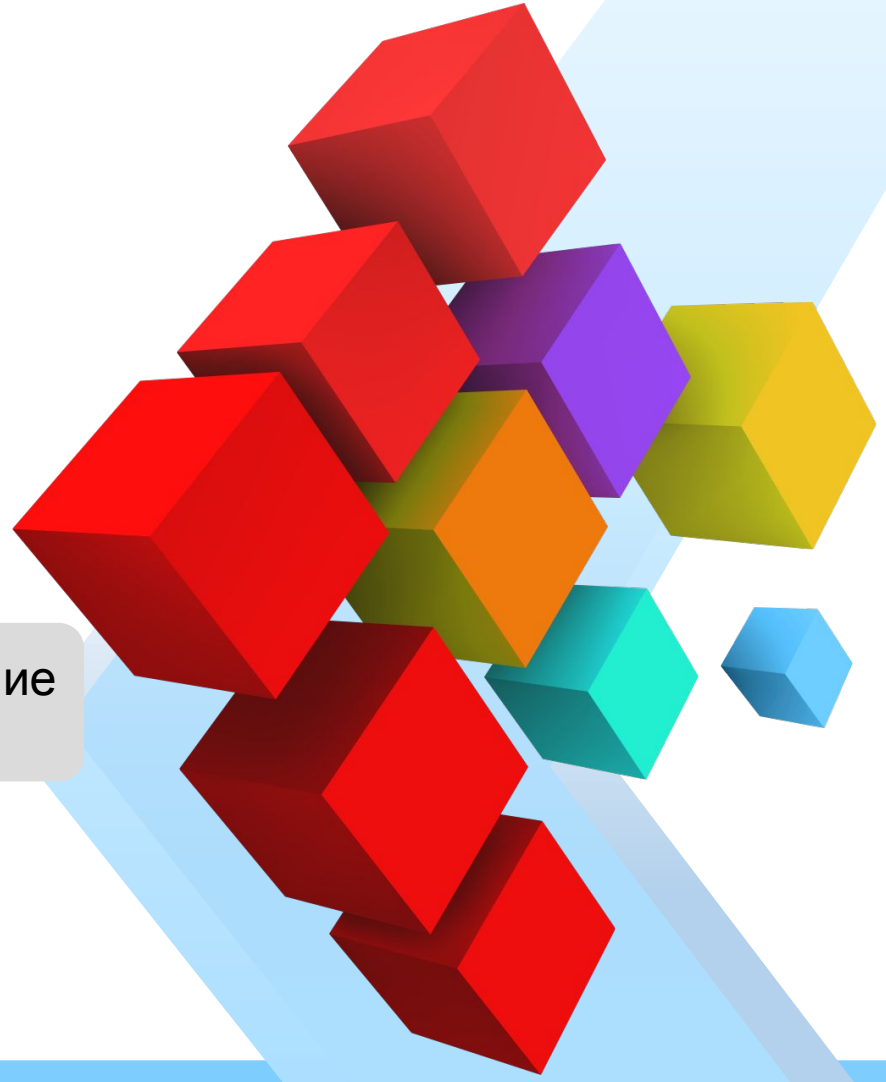
СИНЕРГИЯ

Основы программирования на языке **JavaScript** **1** часть

"Верстка CSS и программирование на языке JavaScript"



Колобова Луиза Владимировна

JavaScript варианты подключения

```
<script>
...
</script>
```

HTML5

```
<script type="text/javascript">
...
</script>
```

HTML < 5

```
<script type="text/javascript" src="./js_files/my_script.js"></script>
```

HTML < 5, внешний файл сценария.

*Тег **<script>** может присутствовать в любом месте документа. Но чаще всего его размещают в блоке **<head>**.*

Однако JavaScript код можно писать и атрибутах тегов

```
1  <html>
2  <body>
3      <p onclick = " this.style.color = 'red'; this.style.fontSize = '24pt';
       this.style.background='yellow'; "
4      ondblclick = " this.style.color = 'black'; this.style.fontSize = 'medium';
       this.style.background='white';" >
5      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin lacus elit,
       commodo vitae dui in, consectetur vulputate nisl. Mauris lacinia enim non
       rhoncus tristique.
6      </p>
7  </body>
8  </html>
```

Но это привод к «распылению» кода по странице.

«Допустимый» синтаксис

```
4  <script>
5
6      var message = "Text, text, text.";
7
8      console.log(message);
9
10 </script>
```

В процессе обучения мы можем ограничиваться только тегами <script></script> для написания кода, и опускать полную разметку документа.

СИНЕРГИЯ

1

Переменные

Переменные

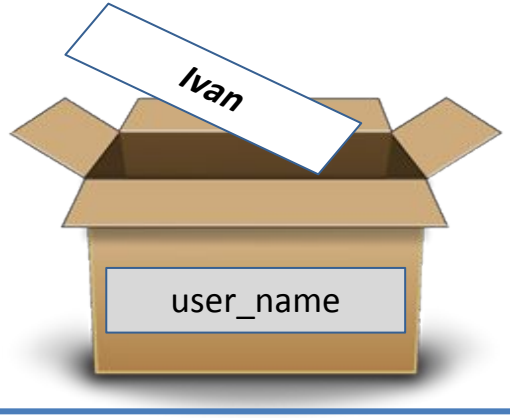
Переменная – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

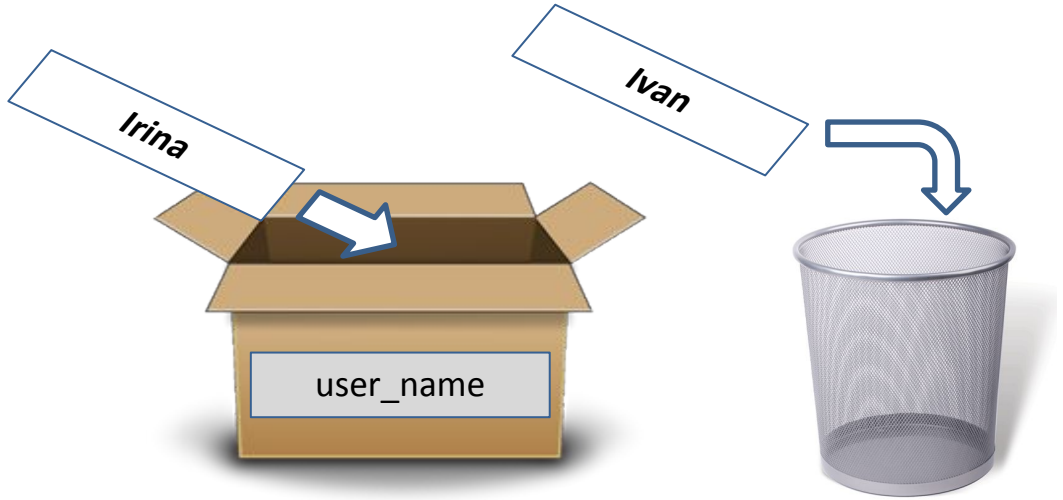
Для создания переменной в JavaScript используйте ключевое слово let, var и const.

Вне зависимости от того, для чего делается скрипт, понадобится работать с информацией

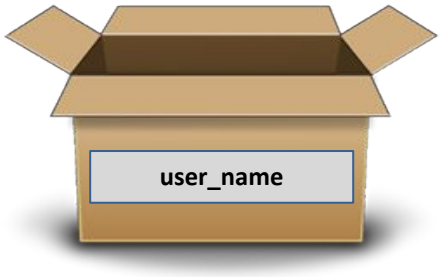
Для хранения информации, используются ***переменные***.

```
1  <script>
2
3      var user_name = "Ivan";
4
5      alert(user_name);
6
7      user_name = "Irina";
8
9      alert(user_name);
10
11 </script>
```





Переменные



Для хранения информации, используются **переменные**.

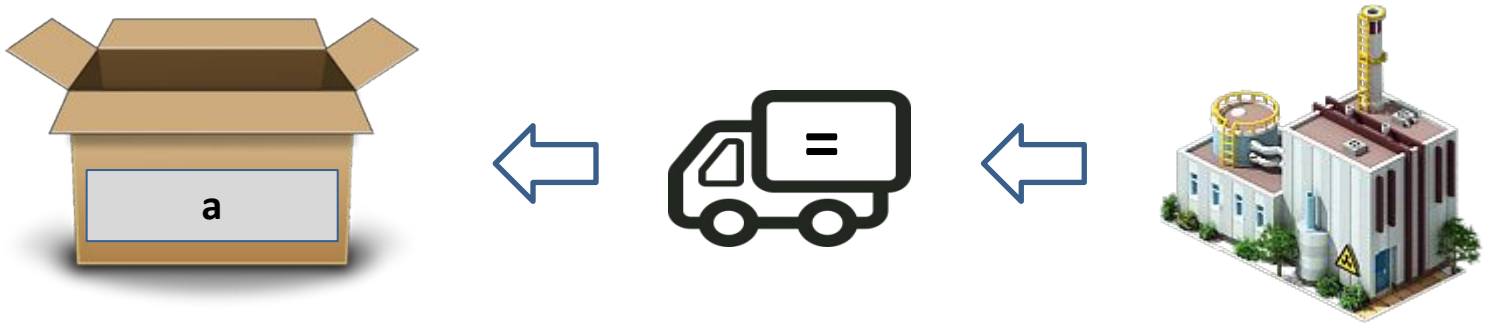
```
var user_name = "Ivan";
```

Перед использованием переменной мы должны попросить выделить под неё место с памяти. Для этого используется ключевое слово **var**. С его помощью происходит т.н. определение переменной. Определение переменной нужно делать только одни раз. В дальнейшем можно использовать переменную по имени, без слова **var**.

Оператор присвоения

Чтобы сказать компьютеру, что именно нужно записать в перемененную используется оператор присвоения **=**

```
a = 2 + 3 * 4;
```



Оператор присвоения берёт то что справа от него и записывает в переменную имя которой расположено слева от него.

Имена переменных

При именовании переменных нам нужно придерживаться определенных правил:

1. Имя переменной должно содержать только буквы, цифры или символы $ и _.
2. Первый символ в имени не должен быть цифрой.
3. Имена переменных должны быть написаны на латинице. Вы конечно можете называть переменные на кириллице, а также на любом другом языке кроме английского, но это считается плохой практикой.
4. Имя переменной должно отражать смысл того что она хранит.

Константы

При именовании переменных нам нужно придерживаться определенных правил:

1. Имя переменной должно содержать только буквы, цифры или символы $ и _.
2. Первый символ в имени не должен быть цифрой.
3. Имена переменных должны быть написаны на латинице. Вы конечно можете называть переменные на кириллице, а также на любом другом языке кроме английского, но это считается плохой практикой.
4. Имя переменной должно отражать смысл того что она хранит.

Типы данных (переменных)

Типы данных — одна из базовых конструкций в любом языке программирования, на основе которых строятся другие структуры данных. В этой статье преподаватель кампуса Эльбрус Буткемп Адам Балкоев в рамках подготовки к пре курсу JavaScript рассказывает о типах данных в этом языке программирования.
JavaScript (JS) — язык программирования с динамической типизацией. Это означает, что во время объявления переменной не нужно указывать ее тип данных — он определится автоматически при присвоении значения.

Типов данных в JS восемь: number, string, boolean, undefined, object, bigInt, symbol и null. Начинающим программистам предстоит пользоваться шестью из них — на них и остановимся.

number

Тип данных number — числовой. Если вы записываете значение переменной в виде числа, как в коде ниже, то JS автоматически определяет его как

```
let age = 30;
```

С числами в JS можно выполнять различные математические операции: складывать (+), вы жет, например, так:

```
let quantity = 10;

let price = 2.99;

let total = quantity * price;

console.log(total); // 29.9
```

string

Текст, заключенный в одинарные (‘) или двойные (") кавычки язык программирования автоматически относит к типу данных string, строка.

```
let name = "Adam";
```

Важно уточнить, что строки в JS неизменяемые: нельзя заменить отдельный символ после ее создания. С уже готовыми строками можно выполнять различные операции, в том числе объединять их, как в примере ниже:

```
let firstName = "Adam";

let lastName = "Balkoev";

let fullName = firstName + " " + lastName;
```

boolean

Следующий тип данных в JS — boolean. Это логический тип, который может принимать только два значения: true или false, правда или ложь.
Например, если мы создадим переменную haveCar со значением true, JS автоматически определит тип как boolean.
Этот тип данных часто используется в операторах if и циклах for и помогает понять, какую часть кода выполнять. Например:

```
let isTrue = true;
let isFalse = false;

if (isTrue) {
  console.log("The value is true");
} else {
  console.log("The value is false");
}
```

null

null в JS — это еще один тип данных, который имеет смысл «ничего» или «значение неизвестно».
Создадим переменную корзина (trash) с пустым значением, которое свидетельствует, что мы не знаем, что находится в корзине:

```
let trash = null;
```

undefined

Если в JS создать переменную и не присвоить ей никакого значения, язык программирования автоматически определит тип данных в ней как undefined.

Рассмотрим пример: создадим переменную box с пустым значением, которое свидетельствует, что в коробке пока ничего нет:

```
let box;

condole.log(box) // undefined
```

object

Рассмотрим этот тип на примере. Напишем объект shop, внутрь которого поместим несколько свойств: название, описание и количество отделов.

```
let shop = {
  title: "Goshan",
  description: "Some text",
  countShop: 10,
}
```

Значение этой переменной JS автоматически отнесет к типу object. Язык программирования при определении типа данных не проверяет, что находится внутри объекта: он видит структуру и по ней определяет тип.

Ключевое слово typeof

Оператор typeof позволяет определить, какой тип данных скрывается за той или иной переменной.

Поскольку JS присваивает типы данных автоматически, иногда полезно узнать, как язык видит значение переменной. Чтобы увидеть тип данных в терминале, нужно написать консоль с ключевым словом и указанием переменной. Например, так:

```
let name = "Adam";
console.log(typeof name); // string

let age = 30;
console.log(typeof age); // number
```

Директива “use strict”

Директива “use strict” говорит браузеру, что следует относиться к JavaScript коду строго по стандарту ECMAScript 5. Все «попустительства» поддерживаемые для совместимости со старыми стандартами перестают действовать.

let и область видимости

Оператор let объявляет переменную, но такие переменные существуют только в той области видимости (тех операторных скобках) в которой они объявлены, и не видны снаружи, в отличии от переменных объявленных через var.

Операторы, операнды и операции

Для выполнения действий (операций) над переменными (или значениями) используются операторы, операторов существует много. С некоторыми из них все знакомы, например с арифметические операторами.

Унарный оператор – тот который взаимодействует только с одной переменной (операндом).

```
6       ++a;
7       b--;
```

Бинарный оператор – тот который взаимодействует с двумя переменными (операндами).

```
6       var c = (a + b) * 4 - (++b);
```

У операторов есть приоритеты, какой приоритет выше, какой ниже запомнить непросто. Поэтому в случае сомнений какая операция будет первой а какая второй – смело используйте скобки. Принцип их применения такой же как и в математике – скобки повышают приоритет операции в них записанной.

Операторы

Унарный оператор – тот который взаимодействует только с одной переменной (операндом).

```
<script>

    var a = 5;
    var b = -a;

    ++a;
    b--;

</script>
```

```
<script>
    var x = 5;

    alert(++x);
    alert(x);
</script>
```

[6, 6]

```
<script>
    var x = 5;

    alert(x++);
    alert(x);
</script>
```

[5, 6]

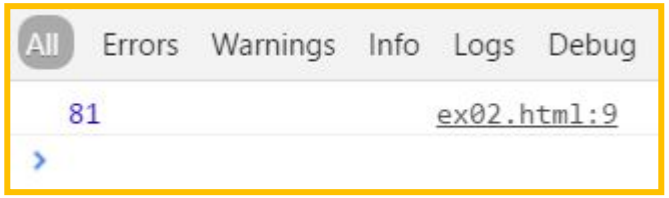
Бинарный оператор – тот который взаимодействует с двумя переменными (операндами).

```
<script>

    var a = 5;
    var b = 6;

    var c = (a + b) * 4 - (++b);

</script>
```

Выражения

По правую сторону от оператора присвоения может быть как конкретное значение (5 или 9 или “Ivan”), а также может быть выражение – формула рассчитав которую компьютер получит результат который будет записан в переменную имя которой стоит слева от знака присвоения. В выражении могут участвовать как и конкретные значения так и другие переменные.

```
1  <script>
2
3      var a = 5;
4      var b = 7;
5      var c = 3;
6
7      var c = (a + b) * c + 45;
8
9      console.log(c);
10
11 </script>
```



Операторы и операции (их приоритеты)

operator	Описание
. [ ] ( )	Доступ к полям, индексация массивов, вызовы функций и группировка выражений
++ -- - ~ ! delete new typeof void	Унарные операторы, тип возвращаемых данных, создание объектов, неопределенные значения
* / %	Умножение, деление, деление по модулю
+ - +	Сложение, вычитание, объединение строк
<< >> >>>	Сдвиг битов
< <= > >= instanceof	Меньше, меньше или равно, больше, больше или равно, instanceof
== != === !==	Равенство, неравенство, строгое равенство, строгое неравенство
&	Побитовое И
^	Побитовое исключающее ИЛИ
|	Побитовое ИЛИ
&&	Логическое И
||	Логическое ИЛИ
?:	Условный оператор
= *OP*=	Присваивание, присваивание с операцией (например += и &=)
,	Вычисление нескольких выражений

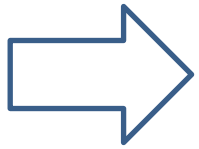
Добавление скобок

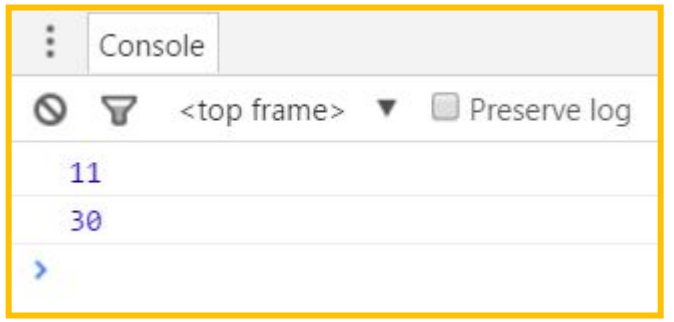
У операторов есть приоритеты, уровней приоритета полтора десятка и помнить затруднительно. Поэтому в случае сомнений какая операция будет первой а какая второй – смело используйте скобки. Принцип их применения такой же как и в математике – скобки повышают приоритет операции в них записанной.

```
var c = (a + b) * 4 - (++b);
```

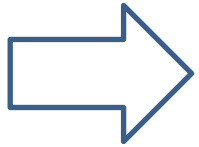
Пример

```
1  <script>
2
3      var a = 5;
4      var b = 6;
5
6      console.log(a + b);
7      console.log(a * b);
8
9  </script>
```






```
1  <script>
2
3      var a = "5";
4      var b = "6";
5
6      console.log(a + b);
7      console.log(a * b);
8
9  </script>
```



?!?

Преобразование данных в JS

Нередко возникает необходимость преобразовать одни данные в другие. Некоторые преобразования JavaScript выполняет автоматически.

При сложении преобразования в JavaScript производятся по принципу:

- Если оба операнда представляют числа, то происходит обычное арифметическое сложение.
- Если предыдущее условие не выполняется, то оба операнда преобразуются в строки и производится объединение строк.

Функция parseInt()

Для преобразования строки в целое число применяется функция parseInt()

```
1  let number1 = "56";
2  let number2 = 4;
3  let result = parseInt(number1) + number2;
4  console.log(result); // 60
```

У parseInt есть особенность. Если в качестве аргумента функции parseInt передана строка в которой присутствуют и символы (буквы) и цифры, то parseInt вернёт число только в случае, если переданная в качестве аргумента строка начинается с числа.

Значение NaN и функция isNaN

Если функции parseInt() не удастся выполнить преобразование, то она возвращает значение NaN (Not a Number), которое говорит о том, что строка не представляет число и не может быть преобразована.

```
console.log(parseInt("abc")); // NaN
```

С помощью специальной функции isNaN() можно проверить, представляет ли строка число. Функция возвращает true, если строка не является числом, в ином случае функция isNaN возвращает false.

```
1  let num1 = "javascript";
2  console.log(isNaN(num1)); // true
```

Функция parseFloat

Для преобразования строк в дробные числа применяется функция parseFloat()

```
1  let number1 = "46.07";
2  let number2 = "4.98";
3  let result = parseFloat(number1) + parseFloat(number2);
4  console.log(result); //51.05
```

СИНЕРГИЯ

2

Условные ветвления

JavaScript и условные операторы

Условная инструкция (или условный оператор) – это оператор (конструкция), которая в JS обеспечивает выполнение команд и алгоритмов в зависимости от истинности заданного изначально условия (логического выражения). Соответствующая запись отвечает за формирование ветвления в программном коде. Этот прием часто применяется тогда, когда встречается цикл – «петля» будет повторяться до тех пор, пока не достигнет условие остановки.

Тернарный оператор – единственный в JS, у которого осуществляется прием трех операндов. Он обычно применяется разработчиками в качестве укороченной формы записи условного оператора if.

Классификация

Рассматривая записи типа if … else в JS, необходимо понимать – условные операторы бывают нескольких видов. Каждый вариант поддерживает свой собственный синтаксис и области применения.
Записи типа if … else в JavaScript бывают:

- с одной веткой;
- двойными;
- с несколькими ветвями;

поддерживающие инструкцию выбора дальнейшего действия.
Отдельно выделяется команда, которая носит название «тернарный оператор». Далее предстоит более подробно изучить и научиться использовать все перечисленные варианты ветвления.

Инструкции с If

Изучая оператор if в JavaScript, необходимо обратить внимание на простейшее ветвление. Это – структура с одной «веткой». Функция будет выполняться, если условие, написанное в if = true. Записать соответствующий тип ветвления проще простого. Для этого используется следующая syntax форма:

```
// condition – это условие
if (condition) {
  // блок кода, который выполняется один раз, если condition истинно
}
```

В виде условия может быть указано совершенно любое выражение. Если соответствующая запись приводится к истине, блок будет реализован. В противном случае программа будет обрабатывать operator, идущий после if condition.

примеры

Оператор if … else с простейшим ветвлением (одной «веткой») используется для решения простейших задач. Пример – когда есть цикл. Для него рассмотренная структура создан условие остановки.
Вот – простейшие наглядные примеры, в которых используется конструкция ветвления с «одной веткой». Первый вариант:

```
if (true) {
  console.log('Привет, мир!');
}
```

Условие здесь после if = true, поэтому блок кода будет выполнен.
Предложенный фрагмент выведет в консоль сообщение «Привет, мир!».
Данный код не имеет никакого реального и сложного применения – он взят только для наглядного примера.

примеры

Если алгоритм, необходимый для реализации, состоит из одной функции, записывать его можно в одну строку, опуская фигурные скобки. Выглядеть это будет так:

```
if (true) console.log('Привет, мир!');
```

Несмотря на это, рекомендуется не пренебрегать фигурными скобками. Они помогут сделать код более читабельным и понятным.

примеры

Еще один пример – увеличение значения переменной с именем num на 5, если изначально она больше 4:

```
let num = 8;
if (num > 4) {
  num += 5;
}
console.log(num); // 13;
```

Здесь условие будет приведено к истине. Результатом система выводит цифру 13.

```
const article = {
  date: '18.05.2022'
}
if (!article.title) {
  console.log('Не указан заголовок!');
}
```

ования простейшего ветвления в JS – с «НЕ

Двойное ветвление

Двойное ветвление поддерживает два условия, которые будут реализовываться в зависимости от истинности заданного выражение. Структура такого if … else является элементарной. Она не слишком сильно отличается от предыдущего варианта.

Здесь предстоит написать фрагмент, включающий в себя два «смысловых блока»:

- то, что будет обрабатываться системой, если условие – истина;
- фрагмент для else – какие действия выполнять, если условие – ложь.

Двойное ветвление

Записать if … else с несколькими условиями можно так:

```
// condition – это условие
if (condition) {
  // блок кода, который выполняется, если условие истинно
} else {
  // блок кода, который выполняется, если условие ложно
}
```

Если написать код по предложенному шаблону, система будет всегда выполнять хотя бы одну из частей. Пример – с else или только с if. Одновременно оба алгоритма при двойном ветвлении не могут быть реализованы. Это противоречит принципам оператора.

Двойное ветвление

Чтобы лучше понимать принцип работы if … else, можно сделать такой файл с кодом:

```
const num = 5;
if (num % 2) {
  console.log('Число нечётное!');
} else {
  console.log('Число чётное!');
}
```

Здесь предлагается сделать следующее: Определить, каким будет заданное число.
Вывести характерное сообщение о четности/нечетности имеющегося элемента.
В предложенной форме if … else система вводит в консоль результат «Число нечетное!». Если значение num поменять на 2, 4 или 6, результатом станет сообщение «Число четное!».

Множественное ветвление

Два условия в ветвлении JavaScript используются для решения более сложных задач. Иногда требуется ввести код с выбором алгоритма из многочисленных доступных вариантов. Такая концепция называется множественным ветвлением. Для нее необходимо писать конструкции типа

```
if (condition1) {
  // блок кода, который выполняется, если условие condition1 истинно
} else if (condition2) {
  // блок кода, который выполняется, если условие condition2 истинно
} else if (condition3) {
  // блок кода, который выполняется, если условие condition3 истинно
} else {
  // блок кода, который выполняется, если предыдущие условия ложны
}
```

Множественное ветвление

Работает такой фрагмент if else так:
Если condition1 – это истина, система выполнит соответствующий код в фигурных скобках. Остальные варианты даже не просматриваются в приложении.
Когда condition is false, система переходит в проверке condition2. Если оно – истина, выполняется соответствующий код
.
Проверки условий conditionN с elseif продолжаются до тех пор, пока не будет обнаружено первое выражение со значением true. Если они отсутствуют – система выполнит требования, написанные в разделе else.

Все это значит, что минимум один из фрагментов кода, которые набрал программист, будет выполнен. При помощи множественных условий else удается создавать весьма сложные выражения и реализовывать сложные задачи.

Множественное ветвление

Вот – пример с else, в котором задаем значение val. В зависимости от него система будет выводить в консоль на печать различные тексты:

```
const val = 7;
if (val < 5) {
  console.log('Less than 5');
} else if (val < 10) {
  console.log('Less than 10');
} else {
  console.log('Greater than or equal to 10');
}
```

Множественное ветвление

А вот – вариант, в котором предыдущую инструкцию можно ввести, используя только if, исключая else:

```
const val = 7;
if (val < 5) {
  console.log('Less than 5');
}
if (val > 5 && val < 10) {
  console.log('Less than 10');
}
if (val >= 10) {
  console.log('Greater than or equal to 10');
}
```

Множественное ветвление

Необходимо запомнить, что else – это необязательная часть. Фрагмент кода без него выглядит так:

```
const lang = 'ru';
if (lang === 'ru') {
  console.log('Это русский текст');
} else if (lang === 'en') {
  console.log('Это английский текст');
} else if (lang !== 'ru' || lang !== 'en') {
  console.log('Это не русский и не английский текст');
}
```

Есть и другие команды, напоминающие if … else, которые использует цикл для формирования ветвления.

Тернарный оператор

Тернарный оператор – некая запись в JavaScript. Она вернет результат первого или второго выражения в зависимости от истинности имеющегося выражения.

Синтаксическая форма записи:

```
// condition – условие
// expression1 – первое выражение
// expression2 – второе выражение
condition ? expression1 : expression2
```

Тернарный оператор

Тернарный оператор – выражение. Он будет в обязательном порядке возвращать значение. Операнда тут три:

- condition – условие, с которым предстоит работать (подобно if … false или true);
- expression1 – выражение, реализуемое при истинности;
- expression2 – операция, выполняемая if … = false.

Тернарный оператор

В качестве разделителей используются символы «вопросительный знак» и «двоеточие». Вот – пример с присваиванием переменной того или иного значения тернарного оператора:

```
const value = 10;
const result = value > 10 ? 'Число больше 10!' : 'Число равно или меньше 10!';
console.log(result); // "Число равно или меньше 10!"
```

JavaScript поддерживает множественные тернарные операторы, но они используются в основном опытными разработчиками.

Условие switch

Изучая циклы, а также ветвления с if … false, необходимо обратить внимание на инструкцию switch. Она позволяет выбирать алгоритм из нескольких представленных в зависимости от того, какое значение получено в вычисляемом выражении. Решение зависит от строгого соответствия результата case.

```
// expression – выражение
switch (expression) {
  case valueA:
    // действия, которые будут выполнены, если expression === valueA
    break; // прерываем дальнейшее выполнение switch
  case valueB:
    // действия, которые будут выполнены, если expression === valueB
    break; // прерываем дальнейшее выполнение switch
  // ...
  case valueN:
  // действия, которые будут выполнены, если expression === valueN
    break; // прерываем дальнейшее выполнение switch
  default:
    // действия по умолчанию, если expression не равно valueA, valueB, ..., valueN
}
```

Условие switch

Default, подобно else, является необязательной частью. Данный фрагмент применяется тогда, когда необходимо обозначить, что делать, если система не обнаружит ни одного соответствия с case. Это своеобразная замена else в случае с ранее рассмотренными ветвлениями.

Break – тоже необязательная часть. Она применяется для прерывания реализации switch. После этого система передаст управление приложением фрагменту, написанному после switch-case.

Условие switch

Вот наглядный пример работы со switch без if … else:

```
const countCandyBoys = 1;
const countCandyGirls = 2;
let message;
switch (countCandyBoys + countCandyGirls) {
  case 1:
    message = 'Одна конфета';
    break;
  case 2:
  case 3:
    message = 'Две или три конфеты';
    break;
  case 4:
    message = 'Четыре конфеты';
    break;
  default:
    message = 'Не одна, не две, не три и не четыре конфеты';
}
// выведем сообщение в консоль
console.log(message);
```

Условие switch

Вот наглядный пример работы со switch без if … else:

```
const countCandyBoys = 1;
const countCandyGirls = 2;
let message;
switch (countCandyBoys + countCandyGirls) {
  case 1:
    message = 'Одна конфета';
    break;
  case 2:
  case 3:
    message = 'Две или три конфеты';
    break;
  case 4:
    message = 'Четыре конфеты';
    break;
  default:
    message = 'Не одна, не две, не три и не четыре конфеты';
}
// выведем сообщение в консоль
console.log(message);
```

Здесь в консоль браузера выводится сообщение о количестве конфет у пользователя. После реализации блока кода срабатывает break. Это приведет к передаче управления console.log и к печати сообщения «Две или три конфеты».

СИНЕРГИЯ

3

Циклы

Циклы

Цикл — это повторяющаяся последовательность действий.

Цикл состоит из условия и тела цикла.
Перед запуском цикла проверяется условие. Если условие истинное, то выполняется блок кода, который называется телом цикла. Затем этот шаг повторяется. Так будет продолжаться, пока условие не станет ложным. Каждое выполнение тела цикла называется итерацией.

JavaScript предоставляет несколько способов создания цикла. Самые распространённые из них — while и for (инициализация; условие; завершающая операция) {}

Циклы

В программировании есть много задач, когда нужно выполнять заранее неизвестное количество однотипных шагов:

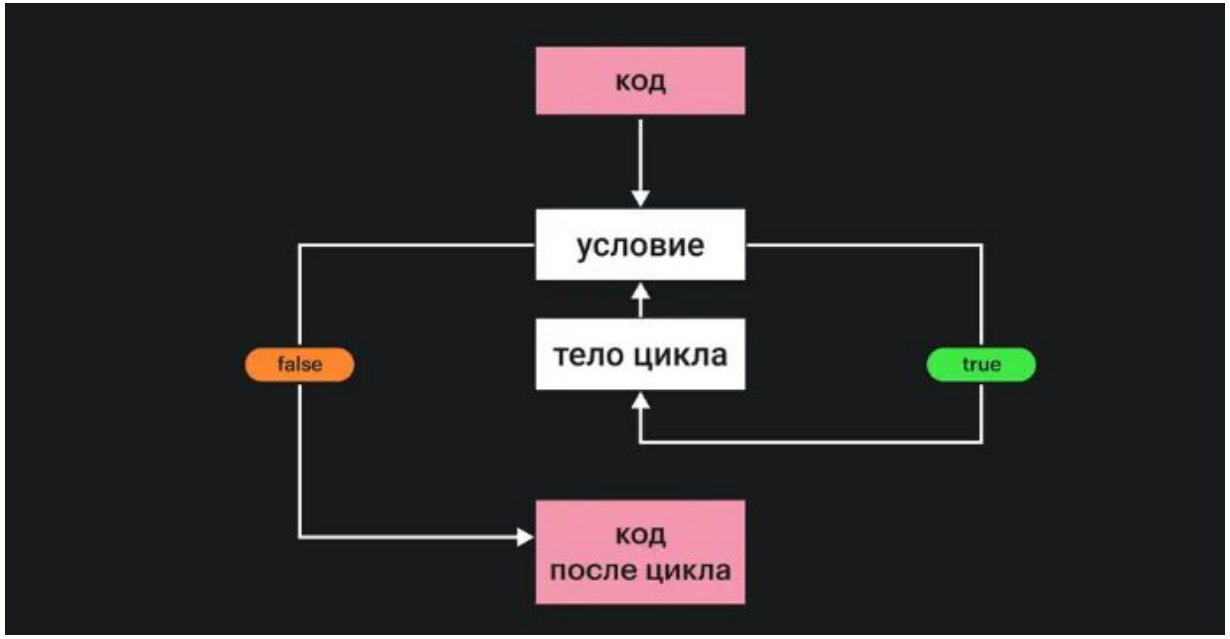
- напечатать все сообщения из списка;
- обозначить на карте присланные пользователем координаты;
- уточнять значения при вычислении функций (например, считать квадратный корень).

До выполнения программы невозможно знать, сколько будет сообщений или сколько раз нужно уточнить значение для точного расчёта квадратного корня. Такие задачи решают циклы.

Циклы

Опишем работу цикла словами:

- проверь, выполняется ли условие.
- если условие выполняется, выполни тело цикла. Затем вернись в пункт 1.
- если условие не выполняется, цикл завершён.



Условие цикла

Условие Скопировать ссылку "Условие"
Нужно хорошо понимать, как работает условие, чтобы уверенно работать с циклами. Условие — это выражение, которое JavaScript вычислит в значение. В простом случае, условие вычисляется в логический тип: true, либо false. Такие выражения получаются при использовании операторов сравнения ==, ===, >, <, >=, <=, !==, !=.
Например:

```
let count = 10
while (count > 0) {
  console.log(count)
  count--
}
```

Код напечатает числа от 10 до 1 на экран.

Условие цикла

Выражения в условии можно комбинировать с помощью логических операторов.

В более сложном случае условие будет вычисляться в какой-либо другой тип: число, строку, массив, объект и т.д. В этом случае JavaScript будет приводить получившееся значение к логическому типу.

```
let count = 10

while (count) {
  console.log(count)
  count--
}
```

Условие цикла

Чтобы понять, какой когда цикл остановится, нужно запомнить правила приведения различных типов к логическому. Главное правило:
Все, что не приводится к false, будет true
Осталось запомнить 8 значений, которые приводятся к false:

- false
- 0
- -0
- ' '
- null
- undefined
- NaN

```
let count = 10

while (count) {
  console.log(count)
  count--
}
```

Зная это правило, мы поймём, что цикл перестанет работать после 10 итераций и напечатает числа от 10 до 1.

Цикл «while»

Цикл while имеет следующий синтаксис:

```
while (condition) {
  // код
  // также называемый "телом цикла"
}
```

Код из тела цикла выполняется, пока условие condition истинно.
Например, цикл ниже выводит i, пока i < 3:

```
let i = 0;
while (i < 3) { // выводит 0, затем 1, затем 2
  alert( i );
  i++;
}
```

Цикл «while»

Если бы строка i++ отсутствовала в примере выше, то цикл бы повторялся (в теории) вечно. На практике, конечно, браузер не позволит такому случиться, он предоставит пользователю возможность остановить «подвисший» скрипт, а JavaScript на стороне сервера придётся «убить» процесс.

Любое выражение или переменная может быть условием цикла, а не только сравнение: условие while вычисляется и преобразуется в логическое значение.

Например, while (i) – более краткий вариант while (i != 0):

```
let i = 3;
while (i) { // когда i будет равно 0, условие станет ложным, и цикл остановится
  alert( i );
  i--;
}
```

Цикл «do…while»

Проверку условия можно разместить под телом цикла, используя специальный синтаксис do..while:

```
do {
  // тело цикла
} while (condition);
```

Цикл сначала выполнит тело, а затем проверит условие condition, и пока его значение равно true, он будет выполняться снова и снова.

Например:

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

Такая форма синтаксиса оправдана, если вы хотите, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным. На практике чаще используется форма с предусловием: while(…) {…}.

Цикл «for»

Более сложный, но при этом самый распространённый цикл — цикл for. Выглядит он так:

```
for (начало; условие; шаг) {
  // ... тело цикла ...
}
```

Давайте разберёмся, что означает каждая часть, на примере. Цикл ниже выполняет alert(i) для i от 0 до (но не включая) 3:

```
for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2
  alert(i);
}
```

Цикл «for»

Рассмотрим конструкцию for подробней:

часть		
начало	`let i = 0`	Выполняется один раз при входе в цикл
условие	`i < 3`	Проверяется *перед* каждой итерацией цикла. Если оно вычислится в `false`, цикл остановится.
тело	`alert(i)`	Выполняется снова и снова, пока условие вычисляется в `true`.
шаг	`i++`	Выполняется *после* тела цикла на каждой итерации *перед* проверкой условия.

Цикл «for»

В целом, алгоритм работы цикла выглядит следующим образом:

```
Выполнить начало
→ (Если условие == true → Выполнить тело, Выполнить шаг)
→ (Если условие == true → Выполнить тело, Выполнить шаг)
→ (Если условие == true → Выполнить тело, Выполнить шаг)
→ ...
```

То есть, начало выполняется один раз, а затем каждая итерация заключается в проверке условия, после которой выполняется тело и шаг. Если тема циклов для вас нова, может быть полезным вернуться к примеру выше и воспроизвести его работу на листе бумаги, шаг за шагом.

Цикл «for»

В нашем случае:

```
// for (let i = 0; i < 3; i++) alert(i)

// Выполнить начало
let i = 0;
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// ...конец, потому что теперь i == 3
```

Прерывание цикла: «break»

Обычно цикл завершается при вычислении условия в false.
Но мы можем выйти из цикла в любой момент с помощью специальной директивы break.
Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

```
let sum = 0;

while (true) {

  let value = +prompt("Введите число", '');

  if (!value) break; // (*)

  sum += value;

}
alert( 'Сумма: ' + sum );
```

Прерывание цикла: «break»

Обычно цикл завершается при вычислении условия в false.
Но мы можем выйти из цикла в любой момент с помощью специальной директивы break.
Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

```
let sum = 0;

while (true) {

  let value = +prompt("Введите число", '');

  if (!value) break; // (*)

  sum += value;

}
alert( 'Сумма: ' + sum );
```

Прерывание цикла: «break»

```
let sum = 0;

while (true) {

  let value = +prompt("Введите число", '');

  if (!value) break; // (*)

  sum += value;

}
alert( 'Сумма: ' + sum );
```

Директива break в строке (*) полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на alert.
Вообще, сочетание «бесконечный цикл + break» – отличное решение для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале или конце цикла, а посередине или даже в нескольких местах его тела.

Переход к следующей итерации: continue

Директива continue – «облегчённая версия» break. При её выполнении цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true). Её используют, если понятно, что на текущем повторе цикла делать больше нечего. Например, цикл ниже использует continue, чтобы выводить

```
for (let i = 0; i < 10; i++) {

  // если true, пропустить оставшуюся часть тела цикла
  if (i % 2 == 0) continue;

  alert(i); // 1, затем 3, 5, 7, 9
}
```

Для чётных значений i, директива continue прекращает выполнение тела цикла и передаёт управление на следующую итерацию for (со следующим числом). Таким образом alert вызывается только для нечётных значений.

Переход к следующей итерации: continue

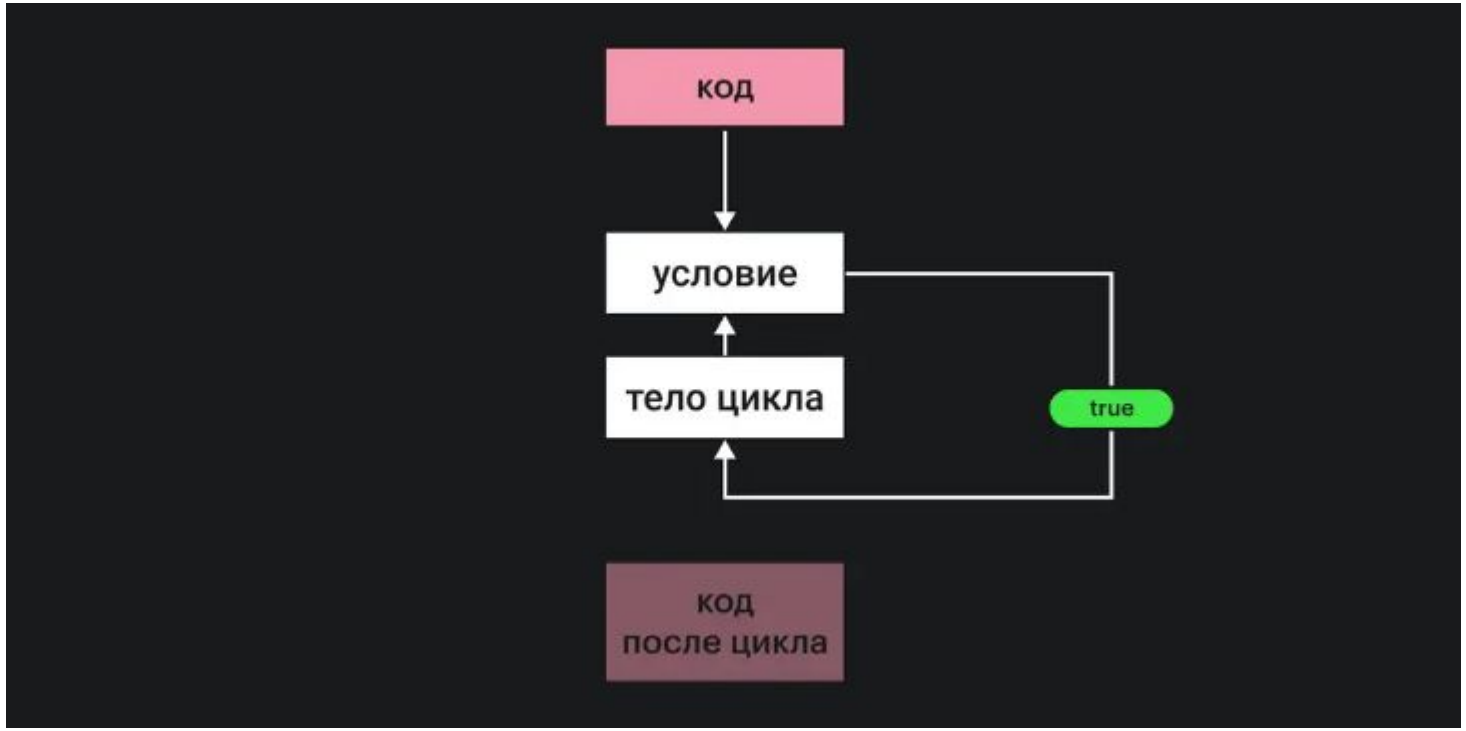
Директива continue – «облегчённая версия» break. При её выполнении цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true). Её используют, если понятно, что на текущем повторе цикла делать больше нечего. Например, цикл ниже использует continue, чтобы выводить

```
for (let i = 0; i < 10; i++) {

  // если true, пропустить оставшуюся часть тела цикла
  if (i % 2 == 0) continue;

  alert(i); // 1, затем 3, 5, 7, 9
}
```

Для чётных значений i, директива continue прекращает выполнение тела цикла и передаёт управление на следующую итерацию for (со следующим числом). Таким образом alert вызывается только для нечётных значений.

Бесконечные циклы

Если условие цикла написано так, что оно никогда не станет ложным, цикл будет выполняться бесконечно. Такой цикл занимает все выделенные ресурсы компьютера. В итоге вкладка браузера или целая программа зависает.



СИНЕРГИЯ

4

Массивы

Использование массивов в JS

Для хранения набора данных в языке JavaScript предназначены массивы. Массивы в JavaScript представлены объектом Array. Объект Array предоставляет ряд свойств и методов, с помощью которых мы можем управлять массивом и его элементами.

Создание массива

Можно создать пустой массив, используя квадратные скобки или конструктор Array:

```
const users = new Array();
const people = [];

console.log(users); // Array[0]
console.log(people); // Array[0]
```

Можно сразу же инициализировать массив некоторым количеством элементов:

```
const users = new Array("Tom", "Bill", "Alice");
const people = ["Sam", "John", "Kate"];

console.log(users); // ["Tom", "Bill", "Alice"]
console.log(people); // ["Sam", "John", "Kate"]
```

Использование массивов

Еще один способ инициализации массивов представляет метод Array.of() - он принимает элементы и инициизирует ими массив:

```
const people = Array.of("Tom", "Bob", "Sam");
console.log(people); // ["Tom", "Bob", "Sam"]
```

Можно определить массив и по ходу определять в него новые элементы:

```
const users = [];
users[1] = "Tom";
users[2] = "Kate";
console.log(users[1]); // "Tom"
console.log(users[0]); // undefined
```

При этом не важно, что по умолчанию массив создается с нулевой длиной. С помощью индексов мы можем подставить на конкретный индекс в массиве тот или иной элемент.

```
Array.from();
```

Использование массивов

И еще один способ представляет функция Array.from(). Она имеет много вариантов, рассмотрим самые распространенные:

```
Array.from(arrayLike)
Array.from(arrayLike, function mapFn(element) { ... })
Array.from(arrayLike, function mapFn(element, index) {...})
```

В качестве первого параметра - arrayLike функция принимает некий объект, который, условно говоря, "похож на массив", то есть может быть представлен в виде набора элементов. Это может быть и другой массив, это может быть и строка, которая по сути предоставляет набор символов. Вообщем какой-то набор элементов, который можно преобразовать в массив. Кроме того, это может и некий объект, в котором определено свойство length. Например:

```
const array = Array.from("Hello");
console.log(array); // ["H", "e", "l", "l", "o"]
```

В данном случае в функцию передается строка и возвращается массив, каждый элемент которого предоставляет один из символов этой строки.

Использование массивов

И еще одна версия функции Array.from() в качестве второго параметра принимает функцию преобразования, в которую кроме элемента из перебираемого набора передается и индекс этого элемента: Array.from(arrayLike, function mapFn(element, index) { ... }). Используем эту версию и передадим в функцию объект со свойством length:

```
const array = Array.from({length:3}, (element, index) => {
    console.log(element);   // undefined
    return index;
});
console.log(array); // [0, 1, 2]
```

Здесь в функцию передается объект, у которого свойство length равно 3. Для функции Array.from это будет сигналом, в возвращаемом массиве должно быть три элемента. При этом неважно, что функция преобразования из второго параметра принимает элемент набора (параметр element) - в данном случае он будет всегда undefined, тем не менее значение length:3 будет указателем, что возвращаемый массив будет иметь три элемента с соответственно индексами от 0 до 2. И через второй параметр функции преобразования - параметр index мы можем и получить текущий индекс элемента.

Использование массивов

Тем не менее мы можем передать объект, где в качестве названий свойств применяются индексы. В этом случае объект превратится в массив- подобный объект, который можно перебрать:

```
const array = Array.from({length:3, "0": "Tom", "1": "Sam", "2": "Bob"}, (element) => {
    console.log(element);
    return element;
});
console.log(array); // ["Tom", "Sam", "Bob"]
```

length

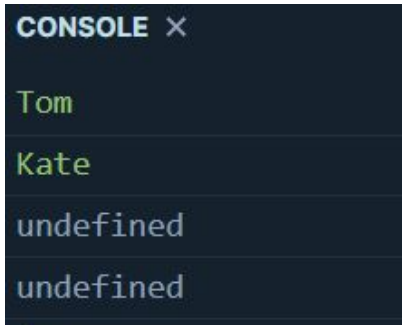
Чтобы узнать длину массива, используется свойство length:

```
const fruit = [];
fruit[0] = "яблоки";
fruit[1] = "груши";
fruit[2] = "сливы";

console.log("В массиве fruit ", fruit.length, " элемента");
for(let i=0; i < fruit.length; i++)
    console.log(fruit[i]);
```

По факту длиной массива будет индекс последнего элемента с добавлением единицы.
Например:

```
const users = []; // в массиве 0 элементов
users[0] = "Tom";
users[1] = "Kate";
users[4] = "Sam";
for(let i=0; i<users.length;i++)
    console.log(users[i]);
```

Вывод браузера:



Несмотря на то, что для индексов 2 и 3 мы не добавляли элементов, но длиной массива в данном случае будет число 5. Просто элементы с индексами 2 и 3 будут иметь значение undefined.

Копирование массива. slice()

Язык JavaScript предоставляет богатые возможности для работы с массивами, которые реализуются с помощью методов объекта Array. Рассмотрим применение этих методов

Копирование массива. slice()

Копирование массива может быть поверхностным или неглубоким (shallow copy) и глубоким (deep copy).

При неглубоком копировании достаточно присвоить переменной значение другой переменной, которая хранит массив:

```
const users = ["Tom", "Sam", "Bill"];
console.log(users);     //  ["Tom", "Sam", "Bill"]
const people = users;       //  неглубокое копирование

people[1] = "Mike";     //  изменяем второй элемент
console.log(users);     //  ["Tom", "Mike", "Bill"]
```

Копирование массива. slice()

В данном случае переменная people после копирования будет указывать на тот же массив, что и переменная users. Поэтому при изменении элементов в people, изменятся элементы и в users, так как фактически это один и тот же массив. Такое поведение не всегда является желательным. Например, мы хотим, чтобы после копирования переменные указывали на отдельные массивы. И в этом случае можно использовать глубокое копирование с помощью метода slice():

```
const users = ["Tom", "Sam", "Bill"];
console.log(users);              //  ["Tom", "Sam", "Bill"]
const people = users.slice();    //  глубокое копирование

people[1] = "Mike";              //  изменяем второй элемент
console.log(users);              //  ["Tom", "Sam", "Bill"]
console.log(people);             //  ["Tom", "Mike", "Bill"]
```

В данном случае после копирования переменные будут указывать на разные массивы, и мы сможем изменять их отдельно друг от друга.

Копирование массива. slice()

Но тут стоит отметить, что то же самое копирование по сути можно выполнить и с помощью spread-оператора(`"..."`):

```
const users = ["Tom", "Sam", "Bill"];
console.log(users);     //  ["Tom", "Sam", "Bill"]
const people = [...users];

people[1] = "Mike";     //  изменяем второй элемент
console.log(users);     //  ["Tom", "Sam", "Bill"]
console.log(people);     //  ["Tom", "Mike", "Bill"]
```

Также метод slice() позволяет скопировать часть массива. Для этого он принимает два параметра:

```
slice(начальный_индекс, конечный_индекс)
```

Первый параметр указывает на начальный индекс элемента, с которого которые используются для выборки значений из массива. А второй параметр - конечный индекс, по который надо выполнить копирование.

Копирование массива. slice()

Например, выберем в новый массив элементы, начиная с 1 индекса по индекс 4:

```
const users = ["Tom", "Sam", "Bill", "Alice", "Kate"];
const people = users.slice(1, 4);
console.log(people);        // ["Sam", "Bill", "Alice"]
```

И поскольку индексация массивов начинается с нуля, то в новом массиве окажутся второй, третий и четвертый элемент.
Если указан только начальный индекс, то копирование выполняется до конца массива:

```
const users = ["Tom", "Sam", "Bill", "Alice", "Kate"];
const people = users.slice(2);  // со второго индекса до конца
console.log(people);        // ["Bill", "Alice", "Kate"]
```

push()

Метод push() добавляет элемент в конец массива:

```
const people = [];
people.push("Tom");
people.push("Sam");
people.push("Bob","Mike");

console.log("В массиве people элементов: ", people.length);
console.log(people); // ["Tom", "Sam", "Bob", "Mike"]
```

pop()

Метод pop() удаляет последний элемент из массива:

```
const people = ["Tom", "Sam", "Bob", "Mike"];

const lastPerson = people.pop(); // извлекаем из массива последний элемент
console.log(lastPerson );   // Mike
console.log(people);    // ["Tom", "Sam", "Bob"]
```

shift()

Метод shift() извлекает и удаляет первый элемент из массива:

```
const people = ["Tom", "Sam", "Bob", "Mike"];

const firstPerson = people.shift(); // извлекаем из массива первый элемент
console.log(firstPerson);   // Tom
console.log(people);    // ["Sam", "Bob", "Mike"]
unshift()
```

unshift()

Метод unshift() добавляет новый элемент в начало массива:

```
const people = ["Tom", "Sam", "Bob"];

people.unshift("Alice");
console.log(people);    // ["Alice", "Tom", "Sam", "Bob"]
```

splice()

Метод splice() удаляет элементы с определенного индекса. Например, удаление элементов с третьего индекса:

```
const people = ["Tom", "Sam", "Bill", "Alice", "Kate"];
const deleted = people.splice(3);
console.log(deleted);       // [ "Alice", "Kate" ]
console.log(people);         // [ "Tom", "Sam", "Bill" ]
```

Метод splice возвращает удаленные элементы в виде нового массива.
В данном случае удаление идет с начала массива. Если передать отрицательный индекс, то удаление будет производиться с конца массива. Например, удалим последний элемент:

```
const people = ["Tom", "Sam", "Bill", "Alice", "Kate"];
const deleted = people.splice(-1);
console.log(deleted);       // [ "Kate" ]
console.log(people);         // ["Tom", "Sam", "Bill", "Alice"]
```

splice()

Дополнительная версия метода позволяет задать количество элементов для удаления. Например, удалим с первого индекса три элемента:

```
const people = ["Tom", "Sam", "Bill", "Alice", "Kate"];
const deleted = people.splice(1, 3);
console.log(deleted);       // ["Sam", "Bill", "Alice"]
console.log(people);         // ["Tom", "Kate"]
```

Еще одна версия метода splice позволяет вставить вместо удаляемых элементов новые элементы:

```
const people = ["Tom", "Sam", "Bill", "Alice", "Kate"];
const deleted = people.splice(1, 3, "Ann", "Bob");
console.log(deleted);       // ["Sam", "Bill", "Alice"]
console.log(people);         // ["Tom", "Ann", "Bob", "Kate"]
```

В данном случае удаляем три элемента с 1-го индекса и вместо них вставляем два элемента.

concat()

Метод concat() служит для объединения массивов. В качестве результата он возвращает объединенный массив:

```
const men = ["Tom", "Sam", "Bob"];
const women = ["Alice", "Kate"];
const people = men.concat(women);
console.log(people);          // ["Tom", "Sam", "Bob", "Alice", "Kate"]
```

join()

Метод join() объединяет все элементы массива в одну строку, используя определенный разделитель, который передается через параметр:

```
const people = ["Tom", "Sam", "Bob"];
const peopleToString = people.join("; ");
console.log(peopleToString);          // Tom; Sam; Bob
```

В метод join() передается разделитель между элементами массива. В данном случае в качестве разделителя будет использоваться точка с запятой и пробел ("; ").

sort()

Метод sort() сортирует массив по возрастанию:

```
const people = ["Tom", "Sam", "Bob"];
people.sort();
console.log(people);        // ["Bob", "Sam", "Tom"]
```

Стоит отметить, что по умолчанию метод sort() рассматривает элементы массива как строки и сортирует их в алфавитном порядке. Что может привести к неожиданным результатам, например:

```
const numbers = [200, 15, 5, 35];
numbers.sort();
console.log(numbers);         // [15, 200, 35, 5]
```

sort()

Здесь мы хотим отсортировать массив чисел, но результат может нас обескуражить: [15, 200, 35, 5]. В этом случае мы можем настроить метод, передав в него функцию сортировки. Логику функции сортировки мыы определяем сами:

```
const numbers = [200, 15, 5, 35];
numbers.sort( (a, b) =>  a - b);
console.log(numbers);          // [5, 15, 35, 200]
```

Функция сортировки получает два рядом расположенных элемента массива и возвращает положительное число, если первый элемент должен находится перед вторым элементом. Если первый элемент должен располагаться после второго, то возвращается отрицательное число.
Если элементы равны, возвращается 0.

reverse()

Метод reverse() переворачивает массив задом наперед:

```
const people = ["Tom", "Sam", "Bob"];
people.reverse();
console.log(people);          // ["Bob", "Sam", "Tom"]
```

Поиск индекса элемента

Методы indexOf() и lastIndexOf() возвращают индекс первого и последнего включения элемента в массиве. Например:

```
const people = ["Tom", "Sam", "Bob", "Tom", "Alice", "Sam"];
const firstIndex = people.indexOf("Tom");
const lastIndex = people.lastIndexOf("Tom");
const otherIndex = people.indexOf("Mike");
console.log(firstIndex); // 0
console.log(lastIndex);  // 3
console.log(otherIndex); // -1
```

firstIndex имеет значение 0, так как первое включение строки "Tom" в массиве приходится на индекс 0, а последнее на индекс 3.
Если же элемент отсутствует в массиве, то в этом случае методы indexOf() и lastIndexOf() возвращают значение -1.

Проверка наличия элемента

Метод includes() проверяет, есть ли в массиве значение, переданное в метод через параметр. Если такое значение есть, то метод возвращает true, если значения в массиве нет, то возвращается false. Например:

```
const people = ["Tom", "Sam", "Bob", "Tom", "Alice", "Sam"];
console.log(people.includes("Tom"));    // true - Tom есть в массиве
console.log(people.includes("Kate"))    // false - Kate нет в массиве
```

В качестве второго параметра метод includes() принимает индекс, с которого надо начинать поиск:

```
const people = ["Tom", "Sam", "Bob", "Tom", "Alice", "Sam"];
console.log(people.includes("Bob", 2)); // true
console.log(people.includes("Bob", 4))  // false
```

В данном случае мы видим, что при поиске со 2-го индекса в массиве есть строка "Bob", тогда как начиная с 4-го индекса данная строка отсутствует.
Если если этот параметр не передается, то по умолчанию поиск идет с 0-го индекса.

every()

Метод every() проверяет, все ли элементы соответствуют определенному условию:

```
const numbers = [ 1, -12, 8, -4, 25, 42 ];
const passed = numbers.every(n => n > 0);
console.log(passed); // false
```

В метод every() в качестве параметра передается функция, которая представляет условие. Эта функция в качестве параметра принимает элемент и возвращает true (если элемент соответствует условию) или false (если не соответствует).
Если хотя бы один элемент не соответствует условию, то метод every() возвращает значение false.
В данном случае условие задается с помощью лямбда-выражения n => n > 0, которое проверяет, больше ли элемент нуля.

some()

Метод some() похож на метод every(), только он проверяет, соответствует ли хотя бы один элемент условию. И в этом случае метод some() возвращает true. Если элементов, соответствующих условию, в массиве нет, то возвращается значение false:

```
const numbers = [ 1, -12, 8, -4, 25, 42 ];
const passed = numbers.some(n => n > 0);
console.log(passed); // true
```

filter()

Метод filter(), как some() и every(), принимает функцию условия. Но при этом возвращает массив тех элементов, которые соответствуют этому условию:

```
const numbers = [ 1, -12, 8, -4, 25, 42 ];
const filteredNumbers = numbers.filter(n => n > 0);
console.log(filteredNumbers); // [1, 8, 25, 42]
```

forEach() и map()

Методы forEach() и map() осуществляют перебор элементов и выполняют с ними определенные операции. Например, используем метод forEach() для вычисления квадратов чисел в массиве:

```
const numbers = [ 1, 2, 3, 4, 5, 6];
numbers.forEach(n =>
    console.log("Квадрат числа", n, "равен", n * n )
)
```

Метод forEach() в качестве параметра принимает функцию, которая имеет один параметр - текущий перебираемый элемент массива. А в теле функции над этим элементом можно выполнить различные операции.

Консольный вывод программы:

```
CONSOLE ×

Квадрат числа 1 равен 1
Квадрат числа 2 равен 4
Квадрат числа 3 равен 9
Квадрат числа 4 равен 16
Квадрат числа 5 равен 25
Квадрат числа 6 равен 36
```

forEach() и map()

Метод map() похож на метод forEach, он также в качестве параметра принимает функцию, с помощью которой выполняются операции над перебираемыми элементами массива, но при этом метод map() возвращает новый массив с результатами операций над элементами массива. Например, применим метод map к вычислению квадратов чисел массива:

```
const numbers = [ 1, 2, 3, 4, 5, 6];
const squares = numbers.map(n => n * n);
console.log(squares);   //  [1, 4, 9, 16, 25, 36]
```

Функция, которая передается в метод map() получает текущий перебираемый элемент, выполняет над ним операции и возвращает некоторое значение. Это значение затем попадает в результирующий массив squares

Поиск в массиве

Метод **find()** возвращает первый элемент массива, который соответствует некоторому условию. В качестве параметра метод find принимает функцию условия:

```
const numbers = [1, 2, 3, 5, 8, 13, 21, 34];
// получаем первый элемент, который больше 10
let found = numbers.find(n => n > 10 );
console.log(found); // 13
```

В данном случае получаем первый элемент, который больше 10. Если элемент, соответствующий условию, не найден, то возвращается undefined.

Поиск в массиве

Метод **findIndex()** также принимает функцию условия, только возвращает индекс первого элемента массива, который соответствует этому условию:

```
const numbers = [1, 2, 3, 5, 8, 13, 21, 34];
// получаем индекс первого элемента, который больше 10
let foundIndex = numbers.findIndex(n => n > 10 );
console.log(foundIndex);    // 5
```

Если элемент не найден, то возвращается число -1.

Метод flat()

Метод flat() упрощает массив с учетом указанной вложенности элементов:

```
const people = ["Tom", "Bob", ["Alice", "Kate", ["Sam", "Ann"]]];
const flattenPeople = people.flat();
console.log(flattenPeople); // ["Tom", "Bob", "Alice", "Kate", ["Sam", "Ann"]]
```

То есть метод flat() фактически из вложенных массивов переводит элементы во внешний массив самого верхнего уровня. Однако мы видим, что элементы массива второго уровня вложенности перешли в массив первого уровня вложенности, но тем не менее по-прежнему находятся во вложенном массиве. Дело в том, что метод flat() по умолчанию применяется только к вложенным массивам первого уровня вложенности. Но мы можем передать в метод уровень вложенности:

```
const people = ["Tom", "Bob", ["Alice", "Kate", ["Sam", "Ann"]]];
const flattenPeople = people.flat(2);
console.log(flattenPeople); // ["Tom", "Bob", "Alice", "Kate", "Sam", "Ann"]
```

Если массив содержит вложенные массивы гораздо более глубоких уровней вложенности, или мы даже не знаем, какие уровни вложенности есть в массиве, но мы хотим, чтобы все элементы были преобразованы в один массив, то можно использовать значение Infinity:

```
const people = ["Tom", "Bob", ["Alice", "Kate", ["Sam", "Ann"]]];
const flattenPeople = people.flat(Infinity);
console.log(flattenPeople); // ["Tom", "Bob", "Alice", "Kate", "Sam", "Ann"]
```

Примеры

Массивы в js нужны для нахождения или отсчета каких-либо игровых событий или величин. К примеру, разберем задачу: у нас есть 3 объекта, у которых есть поле “деньги” и каждый из этих объектов соответственно будет тратить их. Как же считать кто сколько потратил, писать скрипт для каждого объекта? А если объект не один, и не два, и не десять?
В таком случае реализовать подсчет через написания скрипта каждому объекту будет слишком долго. Есть вариант получше. Нужно создать массив из объектов, над которыми нужно проводить пересчет денег:кому нужно отнимать, а кому нужно добавлять.
Как примерно это будет выглядеть:

```
const mass = [];
for(i=0;i<10;i++){
    mass.Append(new heroes());     //заполняем массив объектами шахтерами
}
for(i==0;i<mass.lenght;i++){
if(mass[i].gold_ore > 0){ //проверяем нашли ли шахтеры золото
mass[i].total_price -= 1500 * mass[i].gold_ore}   // если нашли тогда даем им 1500 монет за каждую руду
 else{
Mass[i].total_price -= 10;   // иначе ничего не выдаем
}
}
```