JavaScript

Expressions, Comparison operators, Loops, Arrays

Атрибуты defer, async

В современных сайтах скрипты обычно «тяжелее», чем HTML: они весят больше, дольше обрабатываются.

Когда браузер загружает HTML и доходит до тега <script>...</script>, он не может продолжать строить DOM. Он должен сначала выполнить скрипт.

Это ведёт к двум важным проблемам:

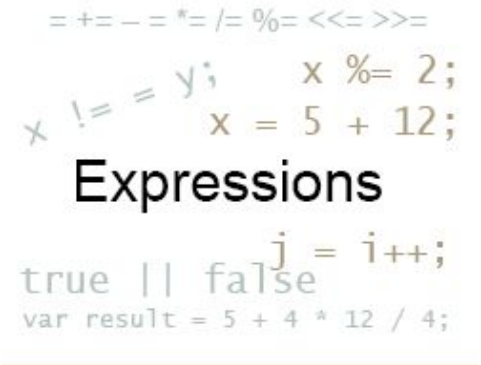
1. Скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя добавить обработчики и т.д.
2. Если вверху страницы объёмный скрипт, он «блокирует» страницу. Пользователи не видят содержимое страницы, пока он не загрузится и не запустится:

Атрибуты defer, async

1. Атрибут **defer** сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда DOM дерево будет полностью построено.
2. Атрибут **async** означает, что скрипт абсолютно независим - страница не ждёт асинхронных скриптов, содержимое обрабатывается и отображается, а остальные скрипты не ждут async, и скрипты с async не ждут другие скрипты

```
<script defer async src="https://javascript.js"></script>
```

Выражения



В JavaScript поддерживаются следующие математические операторы:

Сложение **+**,
Вычитание **–**,
Умножение *****,
Деление **/**,
Взятие остатка от деления **%**,
Возведение в степень ******.

Сложение

+

Объединение чисел в одно целое.

```
let a = 5;
let b = 10;
let result = a + b;
console.log(result); // Выведет 15
```

Вычитание

-

Вычитает правое число от левого.

```
let a = 10;
let b = 2;
let result = a - b;
console.log(result); // Выведет 8
```

Умножение

*

Умножает два числа вместе.

```
let a = 3;
let b = 7;
let result = a * b;
console.log(result); // Выведет 21
```

Деление

/

Делит левое число на правое.

```
let a = 10;
let b = 2;
let result = a / b;
console.log(result); // Выведет 5
```

Взятие остатка %

Оператор взятия остатка %, несмотря на обозначение, никакого отношения к процентам не имеет.

Результат a % b – это остаток от целочисленного деления a на b.

Например:

```
1  alert( 5 % 2 ); // 1, остаток от деления 5 на 2
2  alert( 8 % 3 ); // 2, остаток от деления 8 на 3
3  alert( 8 % 4 ); // 0, остаток от деления 8 на 4
```

Возведение в степень **

Оператор возведения в степень a ** b возводит a в степень b.

В школьной математике мы записываем это как a^b.

Например:

```
1  alert( 2 ** 2 ); // 2² = 4
2  alert( 2 ** 3 ); // 2³ = 8
3  alert( 2 ** 4 ); // 2⁴ = 16
```

Сложение строк при помощи +

Давайте рассмотрим специальные возможности операторов JavaScript, которые выходят за рамки школьной арифметики.

Обычно при помощи плюса '+' складывают числа.

Но если оператор '+' применить к строкам, то он их объединяет в одну:

```
1  let s = "моя" + "строка";
2  alert(s); // моястрока
```

Сложение строк при помощи +

Обратите внимание, если хотя бы один операнд является строкой, то второй будет также преобразован в строку.
Например:

```
1  alert( '1' + 2 ); // "12"
2  alert( 2 + '1' ); // "21"
```

Какой будет результат?

```
1  alert(2 + 2 + '1' );
```

Другие операторы

Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа.

Например, вычитание и деление:

```
1  alert( 6 - '2' ); // 4, '2' приводится к числу
2  alert( '6' / '2' ); // 3, оба операнда приводятся к числам
```

Строковое преобразование

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки, мы можем использовать функцию **String(value),** чтобы преобразовать значение к строке:

```
1  let value = true;
2  alert(typeof value); // boolean
3
4  value = String(value); // теперь value это строка "true"
5  alert(typeof value); // string
```

Численное преобразование

Мы можем использовать функцию **Number(value),** чтобы явно преобразовать value к числу:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // становится числом 123

alert(typeof num); // number
```

Логическое преобразование

Логическое преобразование может быть выполнено явно с помощью функции **Boolean(value)**.

```
1  alert( Boolean(1) ); // true
2  alert( Boolean(0) ); // false
3
4  alert( Boolean("Привет!") ); // true
5  alert( Boolean("") ); // false
```

Инкремент/декремент

Одной из наиболее частых числовых операций является увеличение или уменьшение на единицу.

Для этого существуют даже специальные операторы:

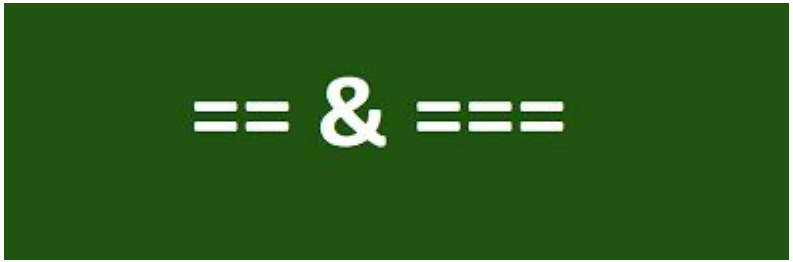
Инкремент ++ увеличивает переменную на 1:

```
let counter = 2;
counter++;          // работает как counter = counter + 1,
alert( counter ); // 3
```

Декремент −− уменьшает переменную на 1:

```
let counter = 2;
counter--;          // работает как counter = counter - 1,
alert( counter ); // 1
```

Операторы сравнения



В JavaScript существуют следующие операторы сравнения:

- Больше/меньше: **a > b, a < b.**
- Больше/меньше или равно: **a >= b, a <= b**.
- Равно: **a == b.** Обратите внимание, для сравнения используется двойной знак равенства ==. Один знак равенства a = b означал бы присваивание.
- Не равно: **a != b.**

Все операторы сравнения возвращают значение логического типа:

true – означает «да», «верно», «истина».

false – означает «нет», «неверно», «ложь».

Больше, меньше >, <

Больше или равно >=

Меньше или равно <=

```
let num1 = 8;
let num2 = 12;

console.log(num1 > num2); // false
console.log(num1 < num2); // true
console.log(num1 >= num2); // false
console.log(num1 <= num2); // true
```

Оператор нестрогого равенства == проверяет равенство с приведением к общему типу.

Использование обычного сравнения == может вызывать проблемы. Например, оно не отличает 0 от false:

```
1  alert( 0 == false ); // true
```

```
1  alert( '' == false ); // true
```

Оператор строгого равенства === проверяет равенство без приведения типов.

Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования.

```
1  alert( 0 === false ); // false, так как сравниваются разные типы
```

Нестрогое неравенство, не выполняет приведение типов перед сравнением - **!=**.

Проверяет неравество значений

```
let x = 5;
let y = "5";


console.log(x != y); // false
```

Строгое неравенство, выполняет приведение типов перед сравнением - **!==**.

Проверяет неравество значений

```
let x = 5;
let y = "5";


console.log(x !== y); // true
```

Сравнение разных типов

При сравнении значений разных типов JavaScript приводит каждое из них к числу.

Например:

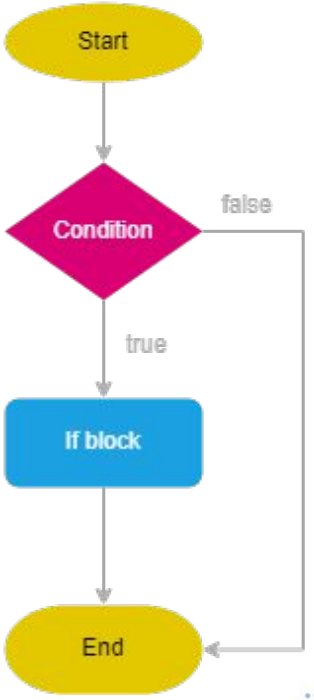
```
1  alert( '2' > 1 ); // true, строка '2' становится числом 2
2  alert( '01' == 1 ); // true, строка '01' становится числом 1
```

Логическое значение true становится 1, а false – 0.

Например:

```
1  alert( true == 1 ); // true
2  alert( false == 0 ); // true
```

Условные операторы



Иногда нам нужно выполнить различные действия в зависимости от условий.
Для этого мы можем использовать инструкцию **if** и условный оператор **?**, который также называют оператором «вопросительный знак».

Инструкция «if»

Инструкция **if(...)** вычисляет условие в скобках и, если результат **true**, то выполняет блок кода.

```
1  if (year == 2015) {
2    alert( "Правильно!" );
3    alert( "Вы такой умный!" );
4  }
```

Преобразование к логическому типу

Инструкция if (…) вычисляет выражение в скобках и преобразует результат к логическому типу.

Преобразование типов:

- Число **0**, пустая строка **""**, **null**, **undefined** и **NaN** становятся false. Из-за этого их называют «ложными» («falsy») значениями.
- Остальные значения становятся true, поэтому их называют «правдивыми» («truthy»).

Блок «else»

Инструкция if может содержать необязательный блок «else» («иначе»). Он выполняется, когда условие ложно.

Например:

```
if (year == 2015) {
  alert( 'Да вы знаток!' );
} else {
  alert( 'А вот и неправильно!' ); // любое значение, кроме 2015
}
```

Несколько условий: «else if»

Иногда нужно проверить несколько вариантов условия. Для этого используется блок else if.

```
if (year < 2015) {
  alert( 'Это слишком рано...' );
} else if (year > 2015) {
  alert( 'Это поздновато' );
} else {
  alert( 'Верно!' );
}
```

Условный оператор „?“

Иногда нам нужно определить переменную в зависимости от условия.

Например:

```
1  let accessAllowed;
2  let age = prompt('Сколько вам лет?', '');
3
4  if (age > 18) {
5    accessAllowed = true;
6  } else {
7    accessAllowed = false;
8  }
9
```

Условный оператор „?“

Так называемый «условный» оператор «вопросительный знак» позволяет нам сделать это более коротким и простым способом.

Оператор представлен знаком вопроса ?. Его также называют «тернарный», так как этот оператор, единственный в своём роде, имеет три аргумента.

```
1  let result = условие ? значение1 : значение2;
```

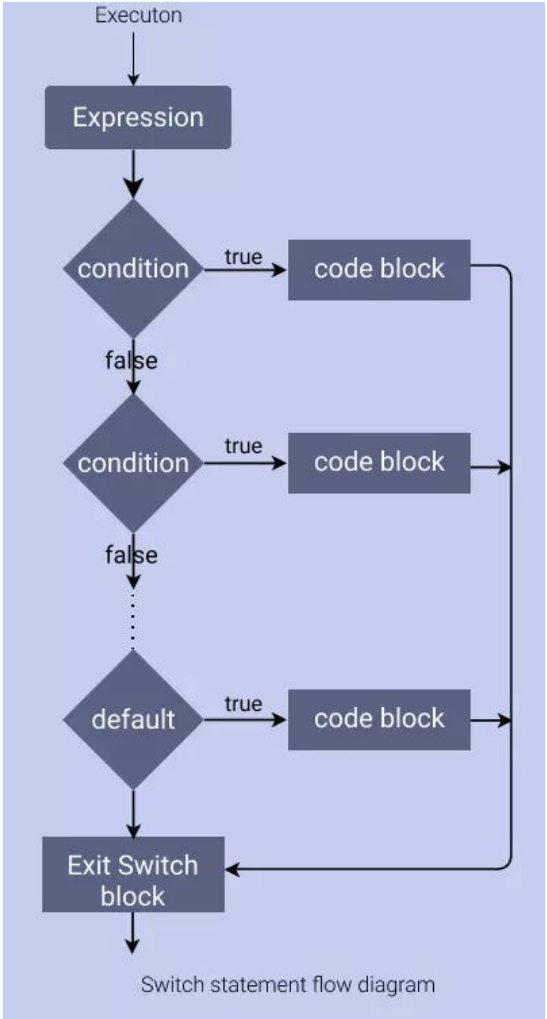


```
1  let accessAllowed = (age > 18) ? true : false;
```

Конструкция "switch"

Конструкция **switch** заменяет собой сразу несколько if.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.



Конструкция **switch** имеет один или более блок **case** и необязательный блок default.

Выглядит она так:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

Переменная **x** проверяется на строгое равенство первому значению **value1**, затем второму **value2** и так далее.

Если соответствие установлено – **switch** начинает выполняться от соответствующей директивы case и далее, до ближайшего **break** (или до конца **switch**).

Если ни один case не совпал – выполняется (если есть) вариант **default**.

Пример использования **switch** (сработавший код выделен):

```
let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Маловато' );
    break;
  case 4:
    alert( 'В точку!' );
    break;
  case 5:
    alert( 'Перебор' );
    break;
  default:
    alert( "Нет таких значений" );
}
```

Логические операторы

|| (ИЛИ)

Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:

```
result = a || b;
```

ские комбинации:

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```

|| (ИЛИ)

Обычно оператор || используется в if для проверки истинности любого из заданных условий.

```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert( 'Офис закрыт.' );
}
```

Можно передать и больше условий:

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'Офис закрыт.' ); // это выходной
}
```

&& (И)

Оператор И пишется как два амперсанда **&&**

В традиционном программировании И возвращает **true**, если оба аргумента истинны, а иначе – **false**:

```
1  alert( true && true );   // true
2  alert( false && true );  // false
3  alert( true && false );  // false
4  alert( false && false ); // false
```

&& (И)

Пример с if:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert( 'Время 12:30' );
}
```

```
if (1 && 0) { // вычисляется как true && false
  alert( "не сработает, так как результат ложный" );
}
```

! (НЕ)

Оператор НЕ представлен восклицательным знаком **!**
Оператор принимает один аргумент и выполняет следующие действия:

1. Сначала приводит аргумент к логическому типу true/false.
2. Затем возвращает противоположное значение.

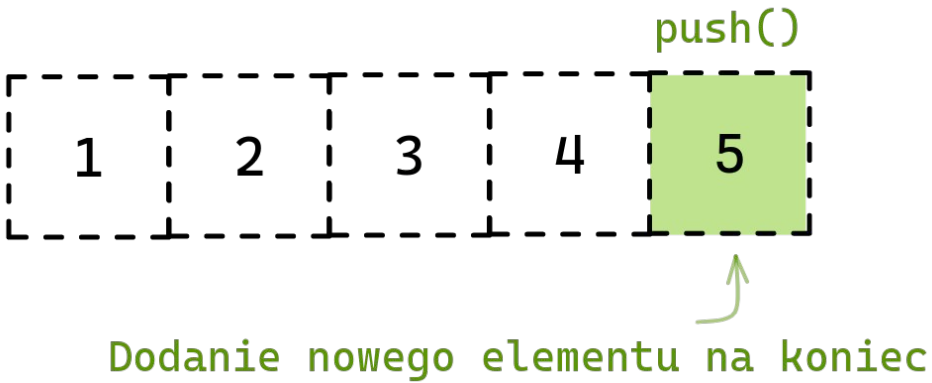
```
1  alert( !true ); // false
2  alert( !0 ); // true
```

! (НЕ)

В частности, двойное НЕ !! используют для преобразования значений к логическому типу:

```
1  alert( !!"непустая строка" ); // true
2  alert( !!null ); // false
```

Массивы



Довольно часто мы понимаем, что нам необходима упорядоченная коллекция данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д.

Например, она понадобится нам для хранения списка чего-либо: **пользователей, товаров, элементов HTML и т.д.**

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, Array.

Объявление

Существует два варианта синтаксиса для создания пустого массива:

```
1  let arr = new Array();
2  let arr = [];
```

Практически всегда используется второй вариант синтаксиса. В скобках мы можем указать начальные значения элементов:

```
1  let fruits = ["Яблоко", "Апельсин", "Слива"];
```

Объявление

Элементы массива нумеруются, начиная с нуля.

Мы можем получить элемент, указав его номер в квадратных скобках:

```
1  let fruits = ["Яблоко", "Апельсин", "Слива"];
2
3  alert( fruits[0] ); // Яблоко
4  alert( fruits[1] ); // Апельсин
5  alert( fruits[2] ); // Слива
```

Мы можем заменить элемент:

```
1  fruits[2] = 'Груша'; // теперь ["Яблоко", "Апельсин", "Груша"]
```

…Или добавить новый к существующему массиву:

```
1  fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]
```

Длина массива

Общее число элементов массива содержится в его свойстве length:

```
1  let fruits = ["Яблоко", "Апельсин", "Слива"];
2
3  alert( fruits.length ); // 3
```

Содержание массива

В массиве могут храниться элементы любого типа.

```
1  // разные типы значений
2  let arr = [ 'Яблоко', { name: 'Джон' }, true, function() { alert('привет'); } ];
3
4  // получить элемент с индексом 1 (объект) и затем показать его свойство
5  alert( arr[1].name ); // Джон
6
7  // получить элемент с индексом 3 (функция) и выполнить её
8  arr[3](); // привет
```

Методы pop/push, shift/unshift

Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

- push добавляет элемент в конец.
- shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

Методы pop/push, shift/unshift

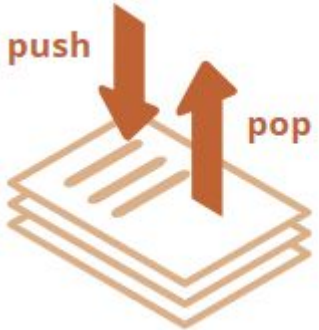
Существует и другой вариант применения для массивов – структура данных, называемая стек.

Она поддерживает два вида операций:

- **push** добавляет элемент в конец.
- **pop** удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца».

Примером стека обычно карты кладутся наверх и берутся тоже сверху:



Методы, работающие с концом массива:

pop

Удаляет последний элемент из массива и возвращает его:

```
1  let fruits = ["Яблоко", "Апельсин", "Груша"];
2
3  alert( fruits.pop() ); // удаляем "Груша" и выводим его
4
5  alert( fruits ); // Яблоко, Апельсин
```

Методы, работающие с концом массива:

push

Добавляет элемент в конец массива:

```
let fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");

alert( fruits ); // Яблоко, Апельсин, Груша
```

Методы, работающие с началом массива:

shift

Удаляет из массива первый элемент и возвращает его:

```
1  let fruits = ["Яблоко", "Апельсин", "Груша"];
2
3  alert( fruits.shift() ); // удаляем Яблоко и выводим его
4
5  alert( fruits ); // Апельсин, Груша
```

Методы, работающие с началом массива:

unshift

Добавляет элемент в начало массива:

```
let fruits = ["Апельсин", "Груша"];

fruits.unshift('Яблоко');

alert( fruits ); // Яблоко, Апельсин, Груша
```

Методы **push** и **unshift** могут добавлять сразу несколько элементов:

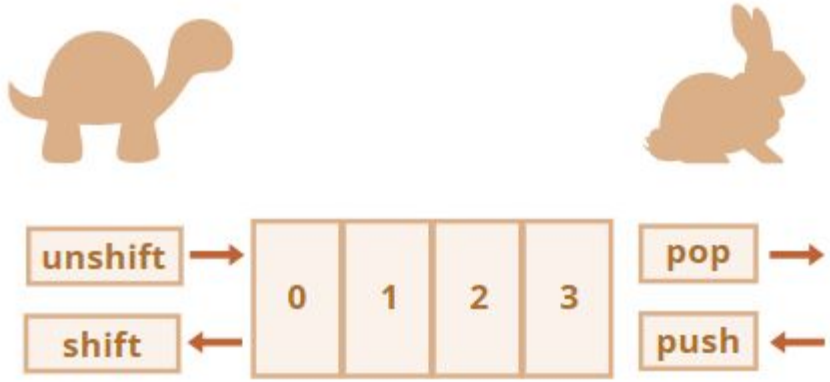
```
let fruits = ["Яблоко"];

fruits.push("Апельсин", "Груша");
fruits.unshift("Ананас", "Лимон");

// ["Ананас", "Лимон", "Яблоко", "Апельсин", "Груша"]
alert( fruits );
```

Эффективность

- Методы `push/pop` выполняются быстро, а методы `shift/unshift` – медленно.



Циклы

Цикл “for”

Более сложный, но при этом самый распространённый цикл — цикл **for**.

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

Цикл “for”

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

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

Цикл “for”

Начало выполняется один раз, а затем каждая итерация заключается в проверке условия, после которой выполняется тело и шаг.

```
1  // for (let i = 0; i < 3; i++) alert(i)
2
3  // Выполнить начало
4  let i = 0;
5  // Если условие == true → Выполнить тело, Выполнить шаг
6  if (i < 3) { alert(i); i++ }
7  // Если условие == true → Выполнить тело, Выполнить шаг
8  if (i < 3) { alert(i); i++ }
9  // Если условие == true → Выполнить тело, Выполнить шаг
10 if (i < 3) { alert(i); i++ }
11 // ...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 );
```

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 только нечётные значен

```
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
}
```

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 только нечётные значения:

```
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
}
```

Перебор элементов массива - for

Одним из самых старых способов перебора элементов массива является цикл **for** по цифровым индексам:

```
1  let arr = ["Яблоко", "Апельсин", "Груша"];
2
3  for (let i = 0; i < arr.length; i++) {
4    alert( arr[i] );
5  }
```

Перебор элементов массива - for…of

Оператор for...of выполняет цикл обхода

```
1 let iterable = [10, 20, 30]
2
3 for (let value of iterable) {
4   value += 1
5   console.log(value)
6 }
7 // 11
8 // 21
9 // 31
```

Цикл while

Цикл **while** имеет следующий синтаксис:

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

Цикл while

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++;
}
```

Одно выполнение тела цикла по-научному называется ***итерация***.
Цикл в примере выше совершает три итерации.

Цикл while

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):**

```
1  let i = 3;
2  while (i) { // когда i будет равно 0, условие станет ложным
3    alert( i );
4    i--;
5  }
```

Цикл “do…while”

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

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

Цикл “do…while”

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

```
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**