7 сентября 2022 г.

Циклы while и for

При написании скриптов зачастую встаёт задача сделать однотипное действие много раз.

Например, вывести товары из списка один за другим. Или просто перебрать все числа от `1` до `10` и для каждого выполнить одинаковый код.

Для многократного повторения одного участка кода предусмотрены *циклы*.

Циклы for...of и for...in

Небольшое объявление для продвинутых читателей.

В этой статье рассматриваются только базовые циклы: `while`, `do..while` и `for(..;..;..)`.

Если вы пришли к этой статье в поисках других типов циклов, вот указатели:

- См. for...in для перебора свойств объекта.
- См. for...of и Перебираемые объекты для перебора массивов и перебираемых объектов.

В противном случае, продолжайте читать.

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++;
}
```

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

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 остановитс
  alert( i );
  i--;
}
```

Фигурные скобки не требуются для тела цикла из одной строки

Если тело цикла состоит лишь из одной инструкции, мы можем опустить фигурные скобки `{…}`:

```
let i = 3;
while (i) alert(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);
}
```

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.

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.

Вот в точности то, что происходит 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
```

Встроенное объявление переменной

В примере переменная счётчика `i` была объявлена прямо в цикле. Это так называемое «встроенное» объявление переменной. Такие переменные существуют только внутри цикла.

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // ошибка, нет такой переменной
```

Вместо объявления новой переменной мы можем использовать уже существующую:

```
let i = 0;

for (i = 0; i < 3; i++) { // используем существующую переменную
  alert(i); // 0, 1, 2
}

alert(i); // 3, переменная доступна, т.к. была объявлена снаружи цикла
```

Пропуск частей «for»

Любая часть `for` может быть пропущена.

Для примера, мы можем пропустить `начало` если нам ничего не нужно делать перед стартом цикла.

Вот так:

```
let i = 0; // мы уже имеем объявленную i с присвоенным значением

for (; i < 3; i++) { // нет необходимости в "начале"
  alert( i ); // 0, 1, 2
}
```

Можно убрать и `шаг` :

```
let i = 0;

for (; i < 3;) {
  alert( i++ );
}
```

Это сделает цикл аналогичным `while (i < 3)` .

А можно и вообще убрать всё, получив бесконечный цикл:

```
for (;;) {
  // будет выполняться вечно
}
```

При этом сами точки с запятой `;` обязательно должны присутствовать, иначе будет ошибка синтаксиса.

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 );
```

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`» – отличная штука 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 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
}
```

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.

Директива `continue` позволяет избегать вложенности

Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

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

  if (i % 2) {
    alert( i );
  }

}
```

С технической точки зрения он полностью идентичен. Действительно, вместо `continue` можно просто завернуть действия в блок `if`.

Однако мы получили дополнительный уровень вложенности фигурных скобок. Если код внутри `if` более длинный, то это ухудшает читаемость, в отличие от варианта с `continue`.

Нельзя использовать `break/continue` справа от оператора „?“

Обратите внимание, что эти синтаксические конструкции не являются выражениями и не могут быть использованы с тернарным оператором `?`. В частности, использование таких директив, как `break/continue`, вызовет ошибку.

Например, если мы возьмём этот код:

```
if (i > 5) {
  alert(i);
} else {
  continue;
}
```

...и перепишем его, используя вопросительный знак:

```
(i > 5) ? alert(i) : continue; // continue здесь приведёт к ошибке
```

...то будет синтаксическая ошибка.

Это ещё один повод не использовать оператор вопросительного знака `?` вместо `if`.

Метки для break/continue

Бывает, нужно выйти одновременно из нескольких уровней цикла сразу.

Например, в коде ниже мы проходимся циклами по `i` и `j`, запрашивая с помощью `prompt` координаты `(i, j)` с `(0,0)` до `(2,2)`:

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

  for (let j = 0; j < 3; j++) {

    let input = prompt(`Значение на координатах (${i},${j})`, '');

    // Что если мы захотим перейти к Готово (ниже) прямо отсюда?
  }
}

alert('Готово!');
```

Нам нужен способ остановить выполнение, если пользователь отменит ввод.

Обычный `break` после `input` лишь прервёт внутренний цикл, но этого недостаточно. Достичь желаемого поведения можно с помощью меток.

Метка имеет вид идентификатора с двоеточием перед циклом:

```
labelName: for (...) {
  ...
}
```

Вызов `break <labelName>` в цикле ниже ищет ближайший внешний цикл с такой меткой и переходит в его конец.

```
outer: for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 3; j++) {

    let input = prompt(`Значение на координатах (${i},${j})`, '');

    // если пустая строка или Отмена, то выйти из обоих циклов
    if (!input) break outer; // (*)

    // сделать что-нибудь со значениями...
  }
}

alert('Готово!');
```

В примере выше это означает, что вызовом `break outer` будет разорван внешний цикл до метки с именем `outer`.

Таким образом управление перейдёт со строки, помеченной `(*)`, к `alert('Готово!')`.

Можно размещать метку на отдельной строке:

```
outer:
for (let i = 0; i < 3; i++) { ... }
```

Директива `continue` также может быть использована с меткой. В этом случае управление перейдёт на следующую итерацию цикла с меткой.

Метки не позволяют «прыгнуть» куда угодно

Метки не дают возможности передавать управление в произвольное место кода.

Например, нет возможности сделать следующее:

```
break label; // не прыгает к метке ниже

label: for (...)
```

Директива `break` должна находиться внутри блока кода. Технически, подойдет любой маркированный блок кода, например:

```
label: {
  // ...
  break label; // работает
  // ...
}
```

...Хотя в 99.9% случаев `break` используется внутри циклов, как мы видели в примерах выше.

К слову, `continue` возможно только внутри цикла.

Итого

Мы рассмотрели 3 вида циклов:

- `while` – Проверяет условие перед каждой итерацией.
- `do..while` – Проверяет условие после каждой итерации.
- `for (;;)` – Проверяет условие перед каждой итерацией, есть возможность задать дополнительные настройки.

Чтобы организовать бесконечный цикл, используют конструкцию `while (true)`. При этом он, как и любой другой цикл, может быть прерван директивой `break`.

Если на данной итерации цикла делать больше ничего не надо, но полностью прекращать цикл не следует – используют директиву `continue`.

Обе этих директивы поддерживают *метки*, которые ставятся перед циклом. Метки – единственный способ для `break/continue` выйти за пределы текущего цикла, повлиять на выполнение внешнего.

Заметим, что метки не позволяют прыгнуть в произвольное место кода, в JavaScript нет такой возможности.

Задачи

Последнее значение цикла

важность: 3

Какое последнее значение выведет этот код? Почему?

```
let i = 3;

while (i) {
  alert( i-- );
}
```

решение

Ответ: 1 .

```
let i = 3;

while (i) {
  alert( i-- );
}
```

Каждое выполнение цикла уменьшает `i` . Проверка `while(i)` остановит цикл при `i = 0` .

Соответственно, будет такая последовательность шагов цикла («развернём» цикл):

```
let i = 3;

alert(i--); // выведет 3, затем уменьшит i до 2

alert(i--) // выведет 2, затем уменьшит i до 1

alert(i--) // выведет 1, затем уменьшит i до 0

// все, проверка while(i) не даст выполняться циклу дальше
```

Какие значения выведет цикл while?

важность: 4

Для каждого цикла запишите, какие значения он выведет. Потом сравните с ответом.

Оба цикла выводят `alert` с одинаковыми значениями или нет?

1.

Префиксный вариант `++i` :

```
let i = 0;
while (++i < 5) alert( i );
```

2.

Постфиксный вариант `i++`

```
let i = 0;
while (i++ < 5) alert( i );
```

решение

Задача демонстрирует, как постфиксные/префиксные варианты могут повлиять на результат, когда используются в сравнениях.

1.

От 1 до 4

```
let i = 0;
while (++i < 5) alert( i );
```

Первое значение: `i = 1`, так как операция `++i` сначала увеличит `i`, а потом уже произойдёт сравнение и выполнение `alert`.

Далее `2, 3, 4…` Значения выводятся одно за другим. Для каждого значения сначала происходит увеличение, а потом – сравнение, так как `++` стоит перед переменной.

При `i = 4` произойдёт увеличение `i` до `5`, а потом сравнение `while (5 < 5)` – это неверно. Поэтому на этом цикл остановится, и значение `5` выведено не будет.

2.

От 1 до 5

```
let i = 0;
while (i++ < 5) alert( i );
```

Первое значение: `i = 1`. Остановимся на нём подробнее. Оператор `i++` увеличивает `i`, возвращая старое значение, так что в сравнении `i++ < 5` будет участвовать старое `i = 0`.

Но последующий вызов `alert` уже не относится к этому выражению, так что получит новый `i = 1`.

Далее `2, 3, 4…` Для каждого значения сначала происходит сравнение, а потом – увеличение, и затем срабатывание `alert`.

Окончание цикла: при `i = 4` произойдёт сравнение `while (4 < 5)` – верно, после этого сработает `i++`, увеличив `i` до `5`, так что значение `5` будет выведено. Оно станет последним.

Значение `i = 5` последнее, потому что на следующем шаге `while (5 < 5)` ложно.

Какие значения выведет цикл for?

важность: 4

Для каждого цикла запишите, какие значения он выведет. Потом сравните с ответом.

Оба цикла выведут `alert` с одинаковыми значениями или нет?

1.

Постфиксная форма:

```
for (let i = 0; i < 5; i++) alert( i );
```

2.

Префиксная форма:

```
for (let i = 0; i < 5; ++i) alert( i );
```

решение

Ответ: от `0` до `4` в обоих случаях.

```
for (let i = 0; i < 5; ++i) alert( i );

for (let i = 0; i < 5; i++) alert( i );
```

Такой результат обусловлен алгоритмом работы `for` :

1. Выполнить единожды присваивание `i = 0` перед чем-либо (начало).
2. Проверить условие `i < 5`
3. Если `true` – выполнить тело цикла `alert(i)` , и затем `i++`

Увеличение `i++` выполняется отдельно от проверки условия `(2)` , значение `i` при этом не используется, поэтому нет никакой разницы между `i++` и `++i` .

Выведите чётные числа

важность: 5

При помощи цикла `for` выведите чётные числа от `2` до `10` .

Запустить демо

решение

```
for (let i = 2; i <= 10; i++) {
  if (i % 2 == 0) {
    alert( i );
  }
}
```

Для проверки на чётность мы здесь используем оператор получения остатка от деления `%`.

Замените for на while

важность: 5

Перепишите код, заменив цикл `for` на `while`, без изменения поведения цикла.

```
for (let i = 0; i < 3; i++) {
  alert( `number ${i}!` );
}
```

решение

```
let i = 0;
while (i < 3) {
  alert( `number ${i}!` );
  i++;
}
```

Повторять цикл, пока ввод неверен

важность: 5

Напишите цикл, который предлагает `prompt` ввести число, большее `100`. Если посетитель ввёл другое число – попросить ввести ещё раз, и так далее.

Цикл должен спрашивать число пока либо посетитель не введёт число, большее `100`, либо не нажмёт кнопку Отмена (ESC).

Предполагается, что посетитель вводит только числа. Предусматривать обработку нечисловых строк в этой задаче необязательно.

Запустить демо

решение

```
let num;

do {
  num = prompt("Введите число больше 100?", 0);
} while (num <= 100 && num);
```

Цикл `do..while` повторяется, пока верны две проверки:

1. Проверка `num <= 100` – то есть, введённое число всё ещё меньше `100`.
2. Проверка `&& num` вычисляется в `false`, когда `num` имеет значение `null` или пустая строка `''`. В этом случае цикл `while` тоже нужно прекратить.

Кстати, сравнение `num <= 100` при вводе `null` даст `true`, так что вторая проверка необходима.

Вывести простые числа

важность: 3

Натуральное число, большее `1`, называется простым, если оно ни на что не делится, кроме себя и `1`.

Другими словами, `n > 1` – простое, если при его делении на любое число кроме `1` и `n` есть остаток.

Например, `5` это простое число, оно не может быть разделено без остатка на `2`, `3` и `4`.

Напишите код, который выводит все простые числа из интервала от `2` до `n`.

Для `n = 10` результат должен быть `2,3,5,7`.

P.S. Код также должен легко модифицироваться для любых других интервалов.

решение

Существует множество алгоритмов решения этой задачи.

Давайте воспользуемся вложенными циклами:

```
Для всех i от 1 до 10 {
  проверить, делится ли число i на какое-либо из чисел до него
  если делится, то это i не подходит, берём следующее
  если не делится, то i - простое число
}
```

Решение с использованием метки:

```
let n = 10;

nextPrime:
for (let i = 2; i <= n; i++) { // Для всех i...
```

```
5
6    for (let j = 2; j < i; j++) { // проверить, делится ли число..
7      if (i % j == 0) continue nextPrime; // не подходит, берём следующе
8    }
9
10   alert( i ); // простое число
11 }
```

Конечно же, его можно оптимизировать с точки зрения производительности. Например, проверять все `j` не от `2` до `i`, а от `2` до квадратного корня из `i`. А для очень больших чисел – существуют более эффективные специализированные алгоритмы проверки простоты числа, например квадратичное решето и решето числового поля.

Предыдущий урок Следующий урок

Поделиться

Карта учебника

Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>`, для нескольких строк кода — тег `<pre>`, если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen...)
- Если что-то непонятно в статье — пишите, что именно и с какого места.