Циклы и массивы

Что такое цикл

Цикл - разновидность управляющей конструкции в высокоуровневых языках программирования, предназначенная для организации многократного исполнения набора инструкций. Говоря простым языком, цикл – это конструкция, которая заставляет определённую группу действий повторять до наступления нужного условия. К примеру:

- Опрашивать клиентов пока не наберётся 100 успешных ответов;
- Читать строчки из файла, пока не доберемся до конца самого файла.

Набор этих инструкций называется телом цикла. Каждое выполнение тела цикла называется итерацией. Условие, которое определяет, делать ли следующую итерацию или завершить цикл, называется условием выхода. Очень удобно знать, сколько итераций уже сделано – т.е. хранить их число в переменной. Такая переменная называется счётчиком итераций цикла, но очевидно, что она не является обязательной для всех циклов.

Итак, цикл состоит из следующих шагов

1. Инициализация переменных цикла в начале цикла.
2. Проверка условия выхода на каждой итерации (до или после неё).
3. Исполнение тела цикла на каждой итерации.
4. Обновление счетчика итераций.

Кроме того, в большинстве языков программирования есть инструменты досрочного прерывания всего цикла или же выполнения текущей итерации.

Циклы в JavaScript

Как и у любого языка программирования, в JavaScript также есть циклы. Давайте рассмотрим несколько видов циклов на интересных примерах. Одно из обычных применений циклов - обход элементов массива.

Цикл while

Цикл while является примером цикла с предусловием. Это цикл, который выполняется, пока истинно некоторое условие, указанное перед его началом. Поскольку условие проверяется до выполнения самой первой итерации, вполне может не выполниться ни одной итерации, если условие изначально ложно.

```
while (condition) {
//Тело цикла
}
```

Тело цикла, содержащее операторы, будет выполняться до тех пор, пока истинно условие, указанное в начале цикла. Алгоритм такого цикла в сравнении с базовым будет выглядеть так:

1. Проверка условия.
2. Выполнение тела, если условие истинно. Выход, если условие ложно.

Для управления циклом обычно требуется одна или несколько переменных. К примеру, некое значение boolean, которое обращается в false при достижении некоего граничного условия. Давайте рассмотрим пример, наша задача, чтобы программа выводила в консоль значения начиная с 1 до значения N которое как раз пользователь введет с клавиатуры.

```
const number = Number(prompt('Введите значение N'));
let i = 1;
while (i <= number) {
   console.log(i++);
}
```

В коде мы создали 2 переменные, первую пользователь вводит с клавиатуры и мы ее сразу преобразовывает в числовой формат, вторая это старт нашего отсчета и конечно цикл while в котором мы будем выполнять условие до тех пор, покаа оно не станет истиной, допустим мы ввели цифру 2, он сравнивает 1 <= 2, нет, значит выполняет тело цикла, тем самым увеличивая i на единицу.

Цикл do..while

Цикл do…while – это уже цикл с постусловием, работающий по алгоритму:

1. Выполнение блока операторов.
2. Проверка условия.
3. Выход, если условие ложно.

Цикл с постусловием – это цикл, в котором условие проверяется после выполнения тела цикла. Отсюда следует, что тело всегда выполняется хотя бы один раз.

Закономерный вопрос зачем нам еще один вид цикла, если он так сильно похож на while, давайте для этого рассмотрим пример:

```
let pass = Number(prompt('Введите пароль в числовом формате'));
while (pass !== 123) {
   pass = Number(prompt('Введите пароль в числовом формате'));
}
```

Что мы можем заметить, конечно же что нам 2 раза пришлось создавать переменную и записывать в нее значение, а если нужно будет поменять текст для пользователя, то нам необходимо будет ее исправлять так же в 2х местах

Пример такой же программы, только с помощью **do while**

```
let pass;
do {
   pass = Number(prompt('Введите пароль в числовом формате'));
} while (pass !== 123);
```

На первой строчке кода мы можем увидеть объявление переменной, так как у нее будет глобальная область видимости и скорее всего мы с ней будем работать еще отдельно

Далее мы видим саму конструкцию, которая полностью исключила дублирование кода. Простыми словами мы сначала делаем что-то, а потом уже проверяем.

Цикл for

Цикл for – это цикл со счётчиком. Сам по себе этот цикл представляет прекрасный пример лаконичной организации кода, имея максимально схожий вид в большинстве языков программирования.

Цикл for значительно упрощает объявление циклов. Как Вы уже увидели, даже без прямого на то требования, циклы имеют счётчик. Он объявляется перед началом цикла и проверяется на каждой итерации, изменяя свое значение внутри тела цикла согласно установленным программистом правилом. Инициализация, проверка и обновление - это три ключевых операции, выполняемых со счетчиком. Цикл for сводит эти три шага воедино:

```
for(инициализация; проверка; инкремент)
```

```
{
   инструкция
}
```

Если провести аналогию с известным нам уже циклом while, получится вот такой псевдокод:

```
инициализация;
while(проверка)
{
   инструкция;
   инкремент;
}
```

Итак, в цикле for перед началом цикла выполняется инициализация, в которой обычно и объявляется переменная-счётчик. Проверка вычисляется в начале каждой итерации цикла. Если она возвращает true, цикл продолжается, иначе – останавливается. По окончанию итерации производится инкрементирование счетчика.

И снова в цикле выведем числа от 1 до N, но уже с применением цикла for

```
const number = Number(prompt('Введите значение N'));
for (let i = 1; i <= number; i++) {
   console.log(i);
}
```

Давайте подведём итоги работы с циклами, какой же выбрать и какой является самым популярным, ответ тут простой, если говорить про популярность, то это определённо цикл for, так что тут вне конкуренции. Если же полагаться на рациональный выбор, то давайте рассмотрим инструкцию:

1. Если нас интересует бесконечный цикл

```
while (true) {
//будет выполняться бесконечное количество раз
}
```

2. Если нам требуется сначала что-то сделать, а потом уже реализовать проверку, то тут к нам на помощь придёт **do while**
3. Если нас интересует счетчик или возможно потребуется считать что либо, то это определённо **for**

Конечно все циклы взаимозаменяемы, но хорошо, когда есть выбор, чем просто одно решение, которое не всегда кажется подходящим.

Массив и его методы

Массив - это упорядоченный список элементов. Массивы дают удобство при обработке однородных данных, позволяя производить операции в циклах для преобразования, фильтрации или сбора нужной информации из исходного массива. Чтобы все эти операции было проще проводить в JavaScript для массивов имеется множество встроенных методов. Большая часть из которых пришла в него со стандартом ES2015. Методы позволяют получить более удобный доступ к элементам массива, а также позволяют удобно с ними взаимодействовать. Давайте посмотрим на самые основные из них и самые используемые.

Давайте посмотрим на примерах когда нам нужны массивы и как мы можем с ними работать.

Предположим, что нас попросили написать программу, облегчающую жизнь деканату, которая бы позволяла создавать группы для студентов, добавлять или исключать их из этой группы, вносить дополнительную информацию о пройденной практике и успеваемости студентов, а также рассчитывать стипендию, если студент хорошо занимается.

Мы могли бы в нашем алгоритме создать переменные строкового типа для хранения каждого студента, но тогда нам понадобилось бы их очень много, и каждый раз, когда в группу добавился бы новый студент, нам приходилось бы дописывать нашу программу, это могло бы выглядеть вот так:

```
const student1 = 'Ivanov';
const student2 = 'Petrov';
const student3 = 'Sidorov';

const student1PracticeTime = 20;
const student2PracticeTime = 135;
const student3PracticeTime = 87;

const fullPracticeTime = student1PracticeTime + student2PracticeTime +
student3PracticeTime;

console.log(student1, ' - practice time: ', student1PracticeTime); // Ivanov  -
practice time:  20
console.log(student2, ' - practice time: ', student2PracticeTime); // Petrov  -
practice time:  135
console.log(student3, ' - practice time: ', student3PracticeTime); // Sidorov  -
practice time:  87
console.log('Full practice time: ', fullPracticeTime); // Full practice time:  242
```

Мы добавили много переменных и нам часто приходится повторять однотипные операции. Если в нашу группу придет еще 5 студентов, нам нужно будет завести еще 10 переменных и дописать много строк кода для вывода информации, что очень неудобно.
В таких случаях, когда информация однотипная, мы можем организовать её в виде массивов, и работать с ней будет намного удобнее:

```
// Массивы всегда называются множественным числом, т.к. массив хранит список чего-либо.
const students = ['Ivanov', 'Petrov', 'Sidorov'];
const studentsPraticeTimes = [20, 135, 87];
// Воспользуемся циклом, чтобы сложить время практики каждого студента. Пока у нас только три студента, цикл выглядит страшнее, чем просто сложения отдельных переменных, но когда у нас будет 30 студентов, обрабатывать их в цикле будет намного проще, чем писать 30 переменных вручную и не запутаться в них. Тут же в цикле мы можем выводить информацию по каждому студенту.
let fullPracticeTime = 0;
for (let index = 0; index < students.length; index++) {
  fullPracticeTime = fullPracticeTime + studentsPraticeTimes[index];
  console.log(students[index], ' - practice time: ', studentsPraticeTimes[index]);
}

// Ivanov  - practice time:  20
// Petrov  - practice time:  135
// Sidorov  - practice time:  87
console.log('Full practice time: ', fullPracticeTime); // Full practice time:  242
```

Как мы можем видеть, количество переменных в коде сократилось, и теперь уже не зависит от количества студентов в нашей программе, мы можем добавлять новых студентов в массив и нам не придется переписывать остальной код программы.

Рассмотрим методы, с помощью которых мы можем проводить операции над массивами. Для начала давайте создадим простой массив студентов:

```
const students = [];
```

Объявляем массив как константу, т.к. массив является объектным типом, то переменная массива - это указатель на область памяти, где хранятся данные массива, и мы не будем менять этот указатель, поэтому объявляем его константой, хотя данные в самом массиве мы можем свободно менять.
Изначально наш массив пустой, нам нужно добавить в него элементы. Студенты могут записываться на курс и мы можем добавить их в массив, для этого нам нужен метод **push**.

push

Метод **push** принимает один или несколько аргументов, которые будут добавлены в конец массива. Давайте добавим в наш массив студентов трёх студентов Иванова, Петрова и Сидорова:

```
const students = [];

students.push('Иванов');
students.push('Петров');
students.push('Сидоров');

console.log(students ); // ['Иванов', 'Петров', 'Сидоров']
```

Еще один пример, когда мы добавляем сразу несколько элементов:

```
const students = [];

students.push('Иванов', 'Петров', 'Сидоров');

console.log(students ); // ['Иванов', 'Петров', 'Сидоров']
```

Смотрите полное описание метода на MDN.

Когда студенты приходят сдавать экзамен, они могут встать в очередь, и преподаватель будет вызывать их по одному. Мы можем работать с нашим массивом точно также, брать последний элемент из него, при этом этот элемент будет удаляться из массива. Для это есть удобный метод **pop**.

pop

Метод **pop** позволяет извлечь из массива последний элемент, при этом он удаляется из массива, а если массив пустой, то вернется **undefined**. Этот метод часто используется чтобы получить последний элемент массива,

```
const students = ['Иванов', 'Петров', 'Сидоров'];

const lastStudent = students.pop();

console.log(lastStudent); // 'Сидоров'
console.log(students); // ['Иванов', 'Петров']
```

Также этот метод позволяет получать последний элемент из строки, которую можно разделить на части, например у нас есть полный путь до файла C:/projects/bestProject/src/images/background-image.png и мы хотим из этого пути получить только файл. Всю строку можно разбить на части, разделенные символом косой черты (слеш “/”), и взять последнюю такую часть, давайте посмотрим пример кода:

```
const filePath = "C:/projects/bestProject/src/images/background-image.png";
const fileName = filePath.split('/').pop(); // Разделим строку на составляющие и превратим её в массив по средствам split('/'), а потом уже вызовем новый метод pop()
```

```
console.log(fileName); // "background-image.png"
```

Смотрите полное описание метода на MDN.

Метод **pop** позволяет получить последний элемент массива, но часто нам бывает нужно получить не последний, а первый элемент, например узнать кто из студентов первым записался на курс - для этого есть метод **shift**, который позволяет извлечь первый элемент массива, при этом все последующие сдвигаются влево на его место.

shift

Этот метод произошел от английского слова shift - сдвигать. Данный метод извлекает нулевой элемент из массива, при этом сдвигает все оставшиеся элементы массива на одну ячейку влево. Получим первого студента, записавшегося на курс.

```
const students = ['Иванов', 'Петров', 'Сидоров'];

const firstStudent = students.shift();

console.log(firstStudent); // 'Иванов'
console.log(students); // ['Петров', 'Сидоров']
```

Также методом shift можно получить имя диска, из полного пути до файла:

```
const filePath = "C:/projects/bestProject/src/images/background-image.png";
const diskName = filePath.split('/').shift(); // Разделим строку на составляющие и превратим её в массив по средствам split('/'), а потом уже вызовим новый метод shift()

console.log(diskName); // "C:"
```

Смотрите полное описание метода на MDN.

Когда нам надо провести контрольную работу среди студентов, нам надо разделить их на группы, чтобы дать разные варианты задания каждой группе. Мы можем разделить наш массив с помощью метода **slice**, который позволяет отделить часть массива в соответствии с переданными аргументами.

slice

На практике часто бывает необходимо сделать копию массива, но т.к. массив - это объект, поэтому переменная массива - это указатель на область памяти, и мы не можем просто присвоить его новой переменной, у нас тогда будет две переменных, ссылающихся на один и

тот же массив, что может привести к изменению исходного массива, когда мы будем работать с новым, давайте посмотрим на примере:

```
const students = ['Иванов', 'Петров', 'Сидоров'];
// Попробуем скопировать массив students в новую переменную.
const students2 = students;
// Добавим в новую переменную нового студента.
students2.push('Белкин');

console.log(students); // ['Иванов', 'Петров', 'Сидоров', 'Белкин']
console.log(students2); // ['Иванов', 'Петров', 'Сидоров', 'Белкин']
```

Как видно из результатов мы меняли второй массив, но вместе с ним изменился и первый, т.к. вторая переменная получила тот же указатель на данные что и первая. Чтобы скопировать массив целиком, мы можем воспользоваться методом **slice**, вызванным без аргументов, тогда он вернет нам полную копию массива:

```
const students = ['Иванов', 'Петров', 'Сидоров'];
// Попробуем скопировать массив students в новую переменную.
const students2 = students.slice();
// Добавим в новую переменную нового студента.
students2.push('Белкин');

console.log(students); // ['Иванов', 'Петров', 'Сидоров']
console.log(students2); // ['Иванов', 'Петров', 'Сидоров', 'Белкин']
```

Теперь наши массивы независимы.

Помимо копирования массивов, метод **slice** (с английского - отрезать) позволяет отрезать от исходного массива часть, при этом исходный массив остается целым, а метод возвращает новый массив. Например когда нам нужно получить 2х первых студентов группы мы можем сделать так:

```
const students = ['Иванов', 'Петров', 'Сидоров', 'Белкин'];
const firstTwoStudents = students.slice(0, 2);

console.log(firstTwoStudents ); // ['Иванов', 'Петров']
```

В зависимости от аргументов мы можем получить разные срезы массива.

Данный метод принимает два аргумента, стартовая позиция - с какого элемента начинать резать данные, и конечная позиция - какой элемент включить в отрезанный кусочек последним. Эти два аргумента имеют хитрое поведение в зависимости от того что в них задано, давайте сведем в таблицу, чтобы было легче понимать что мы получим в итоге:

Старт	Конец	Результат
Не задан	Не задан	Метод **slice** возьмет весь исходный массив и создаст его копию.
Задан в пределах длины массива	Не задан	Метод **slice** отрежет кусочек от исходного массива начиная с элемента, заданного аргументом **старт** и до конца исходного массива.
Задан больше, чем длина массива	Не задан	Вернется пустой массив.
Задан отрицательным	Не задан	Метод **slice** возьмет часть исходного массива, начиная отсчет начального индекса с конца массива.
Задан в пределах длины массива	Задан в пределах длины массива, больше чем аргумент **старт**	Метод **slice** возьмет часть массива, начиная с элемента указанного аргументом **старт**, и заканчивания элементом указанным аргументом **конец**.
Задан в пределах длины массива	Задан отрицательным	Метод **slice** возьмет часть массива, начиная с элемента указанного аргументом **старт**, и заканчивания элементом, позиция которого будет вычислена от конца массива в соответствии с аргументом **конец**.

Давайте посмотрим все это на примерах:

```
const students = ["Ivanov", "Petrov", "Sidorov", "Alexandrov", "Belkin", "Avdeev"];
console.log(students.slice()); // "Ivanov", "Petrov", "Sidorov", "Alexandrov",
"Belkin", "Avdeev" - копия исходного массива.
console.log(students.slice(1)); // "Petrov", "Sidorov", "Alexandrov", "Belkin",
"Avdeev" - часть исходного массива от первого элемента и до конца.
console.log(students.slice(7)); // [] - пустой массив, т.к. Стартовыйы аргумент
больше длины массива.
console.log(students.slice(-2)); // "Belkin", "Avdeev" - часть исходного массива со
второго элемента с конца.
console.log(students.slice(2, 3)); // "Sidorov" - часть исходного массива со второго
элемента по третий.
console.log(students.slice(2, -1)); // "Sidorov", "Alexandrov", "Belkin", - часть
исходного массива со второго элемента по предпоследний.
```

Смотрите полное описание метода на MDN.

Порой, нам надо проверить, есть ли студент с определенной фамилией на нашем курсе. Для этого мы можем искать его в массиве с помощью метода indexOf, который осуществляет поиск в массиве, и если находит, то возвращает индекс найденного элемента.

indexOf

Метод **indexOf** вычисляет индекс определенного элемента, совпадающего со значением, переданным в качестве аргумента. Проще говоря, выполняет поиск элемента. В случае успеха метод возвращает индекс первого найденного элемента, в противном случае возвращается значение -1. При использовании этого метода нужно всегда помнить, что искомый элемент в массиве может оказаться в нулевой позиции, и метод вернет 0, поэтому при проверке, есть ли искомый элемент в массиве, нельзя просто проверить возвращаемое значение, нужно сравнивать с -1. Давайте попробуем проверить, есть ли в нашей группе студентов, студент по фамилии Ivanov:

```
const students = ["Ivanov", "Petrov", "Sidorov", "Alexandrov", "Belkin", "Avdeev"];

// Неправильная проверка.
if (students.indexOf("Ivanov")) {
  console.log("Среди студентов есть Иванов!"); // Ничего не будет выведено, т.к. Иванов является нулевым элементом массива, а ноль приводится к false значению, поэтому такая проверка не сработает.
}

// Правильная проверка.
if (students.indexOf("Ivanov") !== -1) {
  console.log("Среди студентов есть Иванов!"); // "Среди студентов есть Иванов!"
}

const indexOfBelkin = students.indexOf("Belkin"); // 4
```

Этот метод также может принимать второй аргумент, который указывает с какого элемента в массиве начинать поиск, это позволяет отсечь первые элементы в массиве.

Смотрите полное описание метода на MDN

Комбинирование массивов и циклов

Вам необходимо запомнить одно простое правило, работа с массивами очень часто связана с циклами, поэтому мы очень часто будет использовать циклы, для перебора элементов массива

Теперь давайте рассмотрим простую задачу, но которая уже будет максимально приближена к программированию которому мы привыкли видеть. У нас будет массив названий товаров, как в обычном интернет магазине и при клике на кнопку, мы будем удалять последний из товаров.

```
const products = ['Кофта', 'Куртка', 'Футболка', 'Брюки'];
for (let i = 0; i < products.length; i++) {
    console.log(products[i]);
}
```

Первой строчкой мы видим массив продуктов, это намного удобней, чем создавать для каждого товара свою переменную. С помощью цикла **for** мы можем пробежаться по данному массиву и получить каждый элемент массива, в итоге у нас в консоли браузера появятся все элементы массива.

Давайте усложним задачу и сделаем кнопку, при нажатии на которую мы будем удалять последний элемент массива

```
<button onclick="delProduct()">Удалить товар</button>

<script>
   const products = ['Кофта', 'Куртка', 'Футболка', 'Брюки'];
   console.log('Список всех товаров');
   for (let i = 0; i < products.length; i++) {
       console.log(products[i]);
   }

   function delProduct() {
       products.pop(); // удаляем последний элемент массива
       console.log('Список товаров после нажатия на кнопку');
           for (let i = 0; i < products.length; i++) { // выводим
обновлённый список товаров
           console.log(products[i]);
       }
   }
</script>
```

Создаём функцию удаления товара, которая используем метод pop() и выводим обновленный список товаров с помощью цикла for

Дополнительные материалы

1. Документация о массивах на MDN - https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array
2. Массив: перебирающие методы - https://learn.javascript.ru/array-iteration
3. Что такое чистые функции в JavaScript - https://habr.com/ru/post/437512/

Используемые источники

1. https://learn.javascript.ru/rest-parameters-spread-operator
2. https://learn.javascript.ru/destructuring
3. https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array