

Цели занятия

- разобраться с базовыми циклами в JavaScript;
- научиться создавать и использовать собственные функции;
- поговорить про область видимости;

Полезные материалы

- [Современный учебник JavaScript](#)
- [Очень подробно про циклы в JavaScript](#)
- [Подробнее про области видимости](#)

Циклы

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*.

Цикл while

Это самый простой цикл, который имеет следующий синтаксис:

```
while (condition) {  
    // код, который будет по  
    // 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++; // Сокращенная запись i = i + 1;
}
```

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.

Цикл for

Более сложный, но при этом самый распространённый цикл — цикл `for`. Его синтаксис выглядит следующим образом:

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

А теперь напомним код, который будет выводить числа от 0 до 3 с помощью данного цикла:

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

И подробно разберём алгоритм работы данного цикла:

- начало `let i = 0;` — выполняется только 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.

Пропуск частей for

Любая часть `for` может быть пропущена. Для примера, мы можем пропустить начало если нам ничего не нужно делать перед стартом цикла.

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

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

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`.

Переход к следующей итерации

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 пропустить вывод цифры 2:

```
for (let i = 1; i < 5; i++) {
    // если true, пропустить оставшуюся часть тела цикла
    if (i === 2) continue;

    alert(i); // 1, затем 3, 4, 5
}
```

Функции

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, необходимо красиво вывести сообщение при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции, которые являются основными «строительными блоками» программы. Примеры встроенных функций мы уже видели – это `alert(message)`, `prompt(message)` и т.д., но можно создавать и свои.

Объявление функции

Вначале идёт ключевое слово `function`, после него *имя функции*, затем список *параметров* в круглых скобках через запятую (их может не быть) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

```
function имя(параметры) {  
    ...тело...  
}
```

Например, давайте напишем свою функцию для показа сообщения:

```
function showMessage() {  
    alert( 'Всем привет!' );  
}
```

Наша новая функция может быть вызвана по своему имени `showMessage()`.

```
function showMessage() {  
    alert("Всем привет!");  
}  
  
showMessage();  
showMessage();
```

Вызов `showMessage()` выполняет код функции. Здесь мы увидим сообщение дважды.

Этот пример явно демонстрирует одно из главных предназначений функций — избавление от дублирования кода.

Если понадобится поменять сообщение или способ его вывода — достаточно изменить его в одном месте: в функции, которая его выводит.

Параметры функции

Мы можем передать внутрь функции любую информацию, используя параметры (также называемые *аргументами функции*). Параметров может быть несколько, тогда они должны записываться через запятую.

Например, можем передавать внутрь следующей функции имя человека, чтобы подставлять его внутри приветствия.

```
function sayHello(name) {  
    alert(`Привет, ${name}!`);  
}  
  
sayHello("Аня"); // Привет, Аня!  
sayHello("Юрий"); // Привет, Юрий!
```

Параметры по умолчанию

Если параметр не указан, то его значением становится `undefined`.

Например, вышеупомянутая функция `sayHello(name)` может быть вызвана без аргументов. Это не приведёт к ошибке и скрипт выполнится, но сообщение будет таким: `Привет, undefined!`

Этот случай можно отдельно обработать, если задать параметру `name` значение по умолчанию, которое пишется после символа `=`

```
function sayHello(name = "Незнакомец") {  
    alert(`Привет, ${name}!`);  
}  
  
sayHello("Аня"); // Привет, Аня!  
sayHello(); // Привет, Незнакомец!
```

Возврат значения

Функция может вернуть результат, который будет передан в вызвавший её код.

Простейшим примером может служить функция сложения двух чисел:

```
function sum(a, b) {  
    return a + b;  
}  
  
let result = sum(1, 2);  
alert(result); // 3
```

Директива `return` может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной `result` выше).

Выбор имени функции

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть кратким, точным и описывать действие функции, чтобы программист, который будет читать код, получил верное представление, что делает функция.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов.

Например, функции, начинающиеся с "show" обычно что-то показывают.

Функции, начинающиеся с...

- "get..." – возвращают значение,
- "calc..." – что-то вычисляют,
- "create..." – что-то создают,
- "check..." – что-то проверяют и возвращают логическое значение, и т.д.

Примеры таких имён:

- `showMessage(..)` — показывает сообщение
- `getAge(..)` — возвращает возраст (получая его каким-то образом)
- `calcSum(..)` — вычисляет сумму и возвращает результат
- `createForm(..)` — создаёт форму (и обычно возвращает её)
- `checkPermission(..)` — проверяет доступ, возвращая `true/false`

Благодаря префиксам, при первом взгляде на имя функции становится понятным, что делает её код, и какое значение она может возвращать.

Область видимости

Мы используем переменные, чтобы хранить в них временные значения, а потом в нужный момент получать к ним доступ. Но не все переменные в нашем коде одинаково доступны. Доступна ли переменная и как получить к ней доступ, определяют *области видимости*.

Область видимости — это часть программы, в которой мы можем обратиться к переменной, функции или объекту.

Глобальная область видимости — это когда мы «просто объявляем переменную», вне функций, вне модулей, то эта переменная попадает в глобальную область видимости.

Переменная в примере сейчас находится в глобальной области видимости. Это значит, что она будет доступна откуда угодно внутри скрипта:

```
let userName = "Вася";

function showMessage() {
    let message = "Привет, " + userName;

    alert(message);
}

showMessage(); // Привет, Вася
```

Функциональная область видимости — это область видимости в пределах тела функции. Можно сказать, что она ограничена { и } функции.

```
function showMessage() {
    let message = "Привет, я JavaScript!"; // локальная
переменная
    alert(message);
}
```

```
showMessage(); // Привет, я JavaScript!  
  
alert(message); // <-- будет ошибка
```

Глобальная переменная используется, только если внутри функции нет такой локальной.

Если одноимённая переменная объявляется внутри функции, тогда она перекрывает глобальную. Например, в коде ниже функция использует локальную переменную `userName`. Внешняя будет проигнорирована:

```
let userName = "Вася";  
  
function showMessage() {  
    let userName = "Петя"; // объявляем локальную переменную  
    let message = "Привет, " + userName; // Петя  
  
    alert(message);  
}  
  
// функция создаст и будет использовать собственную переменную  
// userName  
showMessage();  
  
alert(userName); // Вася
```