УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационной безопасности

Кафедра инфокоммуникационных технологий

ВЕБ-ТЕХНОЛОГИИ В ИНФОКОММУНИКАЦИЯХ

Язык программирования JavaScript

Минск 2022

Макейчик Е.Г.

Содержание

Лабораторная работа №5
Язык программирования JavaScript

Основы JavaScript

JavaScript создавался как скриптовый язык для браузеров Netscape. Компания Microsoft также признала его потенциал и включила под именем JScript в Internet Explorer 3, обеспечив частичную поддержку стандартов языка, что привело в итоге к неразберихе со стандартами и версиями JavaScript. Поэтому Netscape, Microsoft и другие заинтересованные компании обратились в организацию ECMA (Европейская ассоциация производителей компьютеров), где была одобрена первая спецификация языка ECMA-262. В связи с тем, что название «JavaScript» являлось зарегистрированным товарным знаком, для нового стандарта было решено использовать ECMAScript (или сокращенно ES). ECMAScript изначально был разработан для использования в качестве языка сценариев, но позже стал широко использоваться в качестве языка программирования общего назначения.

В основу создания JavaScript была положена идея динамического управления объектами HTML-документов без перезагрузки текущей страницы (так называемые бессерверные сценарии). Со временем возможности языка расширились:

- С помощью JavaScript можно запускать одностраничные приложения на стороне клиента;
- JavaScript используется на стороне сервера с такими технологиями, как Node.js;
- JavaScript помогает создавать настольные приложения с Electron и может использоваться для работы с одноплатными компьютерами типа Raspberry Pi;
- Также, JavaScript используется для обучения моделей машинного обучения в браузере с помощью технологии TensorFlow.js.

Что такое JavaScript?

JavaScript – язык сценариев, или скриптов. Скрипт представляет собой программный код – набор инструкций, который не требует предварительной обработки (например, компиляции) перед запуском. Код JavaScript интерпретируется движком браузера во время загрузки веб-страницы. Интерпретатор браузера выполняет построчный анализ, обработку и выполнение исходной программы или запроса.

JavaScript – объектно-ориентированный язык с прототипным наследованием. Он поддерживает несколько встроенных объектов, а также позволяет создавать или удалять свои собственные (пользовательские) объекты. Объекты могут наследовать свойства непосредственно друг от друга, образуя цепочку объект-прототип.

Подключение сценариев к html-документу

Сценарии JavaScript бывают встроенные, т.е. их содержимое является частью документа, и внешние, хранящиеся в отдельном файле с расширением .js. Сценарии можно внедрить в html-документ следующими способами.

В виде гиперссылки

Для этого нужно разместить код в отдельном файле и включить ссылку на файл в заголовок

```
<head>
<script src="script.js"></script>
</head>
```

или тело страницы.

```
<body>
<script src="script.js"></script>
</body>
```

Этот способ обычно применяется для сценариев большого размера или сценариев, многократно используемых на разных веб-страницах.

В виде обработчика события

Каждый html-элемент имеет JavaScript-события, которые срабатывают в определенный момент. Нужно добавить необходимое событие в html-элемент как атрибут, а в качестве значения этого атрибута указать требуемую функцию. Функция, вызываемая в ответ на срабатывание события, является **обработчиком события**. В результате срабатывания события исполнится связанный с ним код. Этот способ применяется в основном для коротких сценариев, например, можно установить смену цвета фона при нажатии на кнопку:

```
<script>
let colorArray = ["#5A9C6E", "#A8BF5A", "#FAC46E", "#FAD5BB", "#F2FEFF"]; // создаем массив с цветами фона
let i = 0;

function changeColor(){
    document.body.style.background = colorArray[i];
    i++;
    if( i > colorArray.length - 1){
        i = 0;
    }
}
</script>
<button onclick="changeColor();">Сменить цвет фона</button>
```

Смотреть пример обработчика события

Внутрь элемента <script>

Элемент <script> может вставляться в любое место документа. Внутри тега располагается код, который выполняется сразу после прочтения браузером, или содержит описание функции, которая выполняется в момент ее вызова. Описание функции можно располагать в любом месте, главное, чтобы к моменту ее вызова код функции уже был загружен.

Обычно код JavaScript размещается в заголовке документа (элемент <head>) или после открывающего тега <body>. Если скрипт используется после загрузки страницы, например, код счетчика, то его лучше разместить в конце документа:

```
<footer>
<script>
document.write("Введите свое имя");
</script>
</footer>
</body>
```

Типы данных и переменные в JavaScript

Компьютеры обрабатывают информацию – данные. Данные могут быть представлены в различных формах или типах. Большая часть функциональности JavaScript реализуется за счет простого набора объектов и типов данных. Функциональные возможности, связанные со строками, числами и логикой, базируются на строковых, числовых и логических типах данных. Другая функциональная возможность, включающая регулярные выражения, даты и математические операции, осуществляется с помощью объектов RegExp, Date и Math.

Литералы в JavaScript представляют собой особый класс типа данных, фиксированные значения одного из трех типов данных – строкового, числового или логического:

```
"это строка"
3.14
true
alert("Hellow"); // "Hellow" - это литерал
let myVariable = 15; // 15 - это литерал
```

Примитивный тип данных является экземпляром определенного типа данных, таких как строковый, числовой, логический, null и undefined.

Комментарии

Синтаксис комментариев является таким же, как и в C и во многих других языках:

```
// Комментарий, занимающий одну строку

/* Комментарий,
   занимающий несколько строк
*/

/* Нельзя вкладывать /* комментарий в комментарий */ SyntaxError */
```

Объявления

В JavaScript существует три вида объявлений:

- var – объявляет переменную, инициализация переменной значением является необязательной (устаревшее объявление).
- let – объявляет локальную переменную в области видимости блока, инициализация переменной значением является необязательной.
- const – объявляет именованную константу, доступную только для чтения.

Переменные в JavaScript

Данные, обрабатываемые сценарием JavaScript, являются **переменными**. Переменные представляют собой именованные контейнеры, хранящие данные (значения) в памяти компьютера, которые могут изменяться в процессе выполнения программы. Переменные имеют **имя**, **тип** и **значение**.

Имя переменной, или **идентификатор**, может включать только буквы a-z, A-Z, цифры 0-9 (цифра не может быть первой в имени переменной), символ $ (может быть только первым символом в имени переменной или функции) и символ подчеркивания _, наличие пробелов не допускается. Длина имени переменной не ограничена. Можно, но не рекомендуется записывать имена переменных буквами русского алфавита, для этого они должны быть записаны в Unicode.

В качестве имени переменной нельзя использовать ключевые слова JavaScript. Имена переменных в JavaScript чувствительные к регистру, что означает, что переменная let message; и let Message; – разные переменные.

Переменная создается (объявляется) с помощью ключевого слова let, за которым следует имя переменной, например, let message;. Объявлять переменную необходимо перед ее использованием.

Переменная **инициализируется** значением с помощью операции присваивания =, например, let message="Hellow";, т.е. создается переменная message и в ней сохраняется ее первоначальное значение "Hellow". Переменную можно объявлять без значения, в этом случае ей присваивается значение по умолчанию undefined. Значение переменной может изменяться во время исполнения скрипта. Разные переменные можно объявлять в одной строке, разделив их запятой:

```
let message="Hellow", number_msg = 6, time_msg = 50;
```

Типы данных переменных

JavaScript является нетипизированным языком, тип данных для конкретной переменной при ее объявлении указывать не нужно. Тип данных переменной зависит от значений, которые она принимает. Тип переменной может изменяться в процессе совершения операций с данными (**динамическое приведение типов**). Преобразование типов выполняется автоматически в зависимости от того, в каком контексте они используются. Например, в выражениях, включающих числовые и строковые значения с оператором `+`, JavaScript преобразует числовые значения в строковые:

```
let message = 10 + " дней до отпуска"; // вернет "10 дней до отпуска"
```

Получить тип данных, который имеет переменная, можно с помощью оператора `typeof`. Этот оператор возвращает строку, которая идентифицирует соответствующий тип.

```
typeof 35; // вернет "number"
typeof "text"; // вернет "string"
typeof true; // вернет "boolean"
typeof [1, 2, 4]; // вернет "object"
typeof undefined; // вернет "undefined"
typeof null; // вернет "object"
```

Все типы данных в JavaScript делятся на две группы – **простые** типы данных (primitive data types) и **составные** типы данных (composite data types).

Число

Числовой тип данных (`number`) представляет как целочисленные значения, так и числа с плавающей точкой.

```
let n = 123;
n = 12.345;
```

Существует множество операций для чисел, например, умножение `*`, деление `/`, сложение `+`, вычитание `–` и так далее.

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: `Infinity`, `–Infinity` и `NaN`.

`Infinity` представляет собой математическую бесконечность ∞. Это особое значение, которое больше любого числа.

Результат деления на ноль:

```
alert( 1 / 0 ); // Infinity
```

Задано явно:

```
alert( Infinity ); // Infinity
```

`NaN` означает вычислительную ошибку. Это результат неправильной или неопределенной математической операции, например:

```
alert( "не число" / 2 ); // NaN, такое деление является ошибкой
```

BigInt

В JavaScript тип «number» не может содержать числа больше, чем $(2^{53}-1)$ (т.е. `9007199254740991`), или меньше, чем $-(2^{53}-1)$ для отрицательных чисел. Это техническое ограничение вызвано их внутренним представлением.

Для большинства случаев этого достаточно. Но иногда нужны действительно гигантские числа, например, в криптографии или при использовании метки времени («timestamp») с микросекундами.

Тип BigInt был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины.

Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала:

```
// символ "n" в конце означает, что это BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Строка

Строка (string) в JavaScript должна быть заключена в кавычки.

```
let str = "Привет";
let str2 = 'Одинарные кавычки тоже подойдут';
let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;
```

В JavaScript существует три типа кавычек.

- Двойные кавычки: "Привет".
- Одинарные кавычки: 'Привет'.
- Обратные кавычки: `Привет`.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные же кавычки имеют расширенную функциональность. Они позволяют встраивать выражения в строку, заключая их в ${…}. Например:

```
let name = "Иван";

// Вставим переменную
alert( `Привет, ${name}!` ); // Привет, Иван!

// Вставим выражение
alert( `результат: ${1 + 2}` ); // результат: 3
```

Выражение внутри ${…} вычисляется, и его результат становится частью строки.

Булевый (логический) тип

Булевый тип (boolean) может принимать только два значения: true (истина) и false (ложь).

Такой тип, как правило, используется для хранения значений да/нет: true значит «да, правильно», а false значит «нет, неправильно».

Например:

```
let nameFieldChecked = true; // да, поле отмечено
let ageFieldChecked = false; // нет, поле не отмечено
```

Булевые значения также могут быть результатом сравнений:

```
let isGreater = 4 > 1;

alert( isGreater ); // true (результатом сравнения будет "да")
```

Значение «null»

Специальное значение null не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение null:

```
let age = null;
```

Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

В приведенном выше коде указано, что значение переменной age неизвестно.

Значение «undefined»

Специальное значение undefined формирует тип из самого себя так же, как и null.

Оно означает, что «значение не было присвоено».

Если переменная объявлена, но ей не присвоено никакого значения, то ее значением будет undefined:

```
let age;

alert(age); // выведет "undefined"
```

Обычно null используется для присвоения переменной «пустого» или «неизвестного» значения, а undefined – для проверок, была ли переменная назначена.

Объекты и символы

Тип object (объект) – особенный. В объектах хранят коллекции данных или более сложные структуры.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка, или число, или что-то еще).

Тип symbol (символ) используется для создания уникальных идентификаторов в объектах.

Глобальные и локальные переменные

Переменные по области видимости делятся на **глобальные** и **локальные**. Область видимости представляет собой часть сценария, в пределах которой имя переменной связано с этой переменной и возвращает ее значение. Переменные, объявленные внутри тела функции, называются локальными, их можно использовать только в этой функции. Локальные переменные создаются и уничтожаются вместе с соответствующей функцией.

Переменные, объявленные внутри элемента <script>, или внутри функции, но без использования ключевого слова let, называются глобальными. Доступ к ним может осуществляться на протяжении всего времени, пока страница загружена в браузере. Такие переменные могут использоваться всеми функциями, позволяя им обмениваться данными.

Глобальные переменные попадают в глобальное пространство имен, которое является местом взаимодействия отдельных компонентов программы. Не рекомендуется объявлять переменные таким способом, так как аналогичные имена переменных уже могут использоваться любым другим кодом, вызывая сбой в работе скрипта.

Глобальное пространство в JavaScript представляется глобальным объектом window. Добавление или изменение глобальных переменных автоматически обновляет глобальный объект. В свою очередь, обновление глобального объекта автоматически приводит к обновлению глобального пространства имен.

Если глобальная и локальная переменная имеют одинаковые имена, то локальная переменная будет иметь преимущество перед глобальной.

Локальные переменные, объявленные внутри функции в разных блоках кода, имеют одинаковые области видимости. Тем не менее, рекомендуется помещать объявления всех переменных в начале функции.

Взаимодействие: alert(), prompt(), confirm()

alert()

Функция показывает сообщение и ждет, пока пользователь нажмет кнопку «OK».

```
alert("Hello");
```

Это небольшое окно с сообщением называется модальным окном. Понятие модальное означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «OK».

prompt()

Функция `prompt()` принимает два аргумента:

```
result = prompt(title, [default]);
```

Этот код отобразит модальное окно с текстом, полем для ввода текста и кнопками «OK»/«Отмена».

`title` – текст для отображения в окне.

`default` – необязательный второй параметр, который устанавливает начальное значение в поле для текста в окне. Квадратные скобки вокруг `default` в описанном выше синтаксисе означают, что параметр факультативный, необязательный.

Пользователь может напечатать что-либо в поле ввода и нажать «OK». Введенный текст будет присвоен переменной `result`. Пользователь также может отменить ввод нажатием на кнопку «Отмена» или нажав на клавишу Esc. В этом случае значением `result` станет `null`.

Вызов `prompt` возвращает текст, указанный в поле для ввода, или `null`, если ввод отменен пользователем.

```
let age = prompt('Сколько тебе лет?', 100);

alert(`Тебе ${age} лет!`); // Тебе 100 лет!
```

confirm()

Функция `confirm()` отображает модальное окно с текстом вопроса `question` и двумя кнопками: OK и Отмена.

```
result = confirm(question);
```

Результат – `true`, если нажата кнопка OK. В других случаях – `false`.

```
let isBoss = confirm("Ты здесь главный?");

alert( isBoss ); // true, если нажата OK
```

Все эти функции являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

На все указанные методы распространяются два ограничения:

1. Расположение окон определяется браузером. Обычно окна находятся в центре.
2. Визуальное отображение окон зависит от браузера, нельзя изменить их вид.

Преобразование типов

Чаще всего операторы и функции автоматически приводят переданные им значения к нужному типу.

Например, alert() автоматически преобразует любое значение к строке. Математические операторы преобразуют значения к числам.

Строковое преобразование

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки.

Например, alert(value) преобразует значение к строке.

Также можно использовать функцию String(value), чтобы преобразовать значение к строке:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // теперь value это строка "true"
alert(typeof value); // string
```

Преобразование происходит очевидным образом. false становится "false", null становится "null" и т.п.

Численное преобразование

Численное преобразование происходит в математических функциях и выражениях.

Например, когда операция деления / применяется не к числу:

```
alert( "6" / "2" ); // 3, строки преобразуются в числа
```

Можно 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
```

Явное преобразование часто применяется, когда нужно получить число из строкового контекста, например из текстовых полей форм.

Если строка не может быть явно приведена к числу, то результатом преобразования будет NaN. Например:

```
let age = Number("Любая строка вместо числа");

alert(age); // NaN, преобразование не удалось
```

Правила численного преобразования:

Значение	Преобразуется в…
undefined	NaN
null	0
true / false	1 / 0
string	Пробельные символы по краям обрезаются. Далее, если остается пустая строка, то получаем 0, иначе из непустой строки «считывается» число. При ошибке результат NaN.

null и undefined ведут себя по-разному. Так, null становится нулем, тогда как undefined приводится к NaN.

```
alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (ошибка чтения числа на месте символа "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

Большинство математических операторов также производит данное преобразование.

Логическое преобразование

Логическое преобразование происходит в логических операциях, но также может быть выполнено явно с помощью функции Boolean(value).

Правило преобразования:

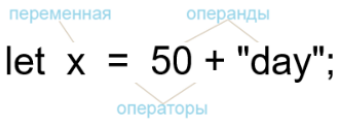
- Значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined и NaN, становятся false.
- Все остальные значения становятся true.

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

alert( Boolean("Привет!") ); // true
alert( Boolean("") ); // false
```

Выражения в JavaScript

Выражения в JavaScript представляют собой комбинации **операндов** и **операторов**.



Операции в выражениях выполняются последовательно в соответствии со значением приоритета (чем больше значение приоритета, тем он выше). Возвращаемый результат не всегда имеет значение того же типа, что и тип обрабатываемых данных. Например, в операциях сравнения участвуют операнды различных типов, но возвращаемый результат всегда будет логического типа.

Операнды – это данные, обрабатываемые сценарием JavaScript. В качестве операндов могут быть как простые типы данных, так и сложные, а также другие выражения.

Операторы – это символы языка, выполняющие различные операции с данными. Операторы могут записываться с помощью символов пунктуации или ключевых слов.

В зависимости от количества операндов различают следующие типы операторов:

- унарный – в операции участвует один операнд;
- бинарный – в операции участвуют два операнда;
- тернарный – комбинирует три операнда.

Простейшая форма выражения – литерал – нечто, вычисляемое само в себя, например, число 100, строка "Hellow world". Переменная тоже может быть выражением, так как она вычисляется в присвоенное ей значение.

Арифметические операторы

Арифметические операторы предназначены для выполнения математических операций, они работают с числовыми операндами (или переменными, хранящими числовые значения), возвращая в качестве результата числовое значение.

Если один из операндов является строкой, интерпретатор JavaScript попытается преобразовать его в числовой тип, а после выполнить соответствующую операцию. Если преобразование типов окажется невозможным, будет получен результат NaN (не число).

Оператор/Операция	Описание	Приоритет
+ Сложение	Складывает числовые операнды. Если один из операндов – строка, то результатом выражения будет строка.	12
– Вычитание	Выполняет вычитание второго операнда из первого.	12
– Унарный минус	Преобразует положительное число в отрицательное, и наоборот.	14
* Умножение	Умножает два операнда.	13
/ Деление	Делит первый операнд на второй. Результатом деления может являться как целое, так и число с плавающей точкой.	13
% Деление по модулю (остаток от деления)	Вычисляет остаток, получаемый при целочисленном делении первого операнда на второй. Применяется как к целым числам, так и числам с плавающей точкой.	13

```
let x = 5, y = 8, z;
z = x + y; // вернет 13
z = x - y; // вернет -3
z = - y; // вернет -8
z = x * y; // вернет 40
z = x / y; // вернет 0.625
z = y % x; // вернет 3
```

Операторы присваивания

Операторы присваивания используются для присваивания значений переменным. Комбинированные операторы позволяют сохранить первоначальное и последующее значение в одной переменной.

Оператор/Операция	Описание	Приоритет
= Присваивание	Используется для присваивания значения переменной.	2
+=, -=, *=, /=, %= Комбинированный оператор	Выполняет присваивание с операцией. Между первым и вторым операндом выполняется соответствующая операция, затем результат присваивается первому операнду.	2

```
let a = 5; // присваиваем переменной a числовое значение 5
let b = "hellow"; // сохраняем в переменной b строку hellow
let m = n = z = 10; // присваиваем переменным m, n, z числовое значение 10
x += 10; // равнозначно x = x + 10;
x -= 10; // равнозначно x = x - 10;
x *= 10; // равнозначно x = x * 10;
x /= 10; // равнозначно x = x / 10;
x %= 10; // равнозначно x = x % 10;
```

Операторы инкремента и декремента

Операции **инкремента** и **декремента** являются унарными и производят увеличение и уменьшение значения операнда на единицу. В качестве операнда может быть переменная, элемент массива, свойство объекта. Чаще всего такие операции используются для увеличения счетчика в цикле.

Оператор/Операция	Описание	Приоритет
++x Префиксный инкремент	Увеличивает операнд на единицу.	14
x++ Постфиксный Инкремент	Прибавляет к операнду единицу, но результатом выражения будет являться первоначальное значение операнда.	14
--x Префиксный декремент	Уменьшает на единицу операнд, возвращая уменьшенное значение.	14
x-- Постфиксный декремент	Уменьшает на единицу операнд, возвращая первоначальное значение.	14

```
let x = y = m = n = 5, z, s, k, l;
z = ++x * 2; /* в результате вычислений вернет значение z = 12, x = 6, т.е. значение x сначала увеличивается на 1, а после выполняется операция умножения */
s = y++ * 2; /* в результате вычислений вернет значение s = 10, y = 6, т.е. сначала выполняется операция умножения, а после в переменной y сохраняется увеличенное на 1 значение */
k = --m * 2; // вернет значение k = 8, m = 4
l = n-- * 2; // вернет значение l = 10, n = 4
```

Операторы сравнения

Операторы сравнения используются для сопоставления операндов, результатом выражения может быть одно из двух значений – true или false. Операндами могут быть не только числа, но и строки, логические значения и объекты. Однако сравнение может выполняться только для чисел и строк, поэтому операнды, не являющиеся числами или строками, преобразуются.

Если оба операнда не могут быть успешно преобразованы в числа или строки, операторы всегда возвращают false.

Если оба операнда являются строками/числами или могут быть преобразованы в строки/числа, они будут сравниваться как строки/числа.

Если один операнд является строкой/преобразуется в строку, а другой является числом/преобразуется в число, то оператор попытается преобразовать строку в число и выполнить сравнение чисел. Если строка не является числом, она преобразуется в значение NaN и результатом сравнения будет false.

Чаще всего операции сравнения используются при организации ветвлений в программах.

Оператор/Операция	Описание	Приоритет
== Равенство	Проверяет две величины на совпадение, допуская преобразование типов. Возвращает true, если операнды совпадают, и false, если они различны.	9
!= Неравенство	Возвращает true, если операнды не равны	9
=== Идентичность	Проверяет два операнда на «идентичность», руководствуясь строгим определением совпадения. Возвращает true, если операнды равны без преобразования типов.	9
!== Неидентичность	Выполняет проверку идентичности. Возвращает true, если операнды не равны без преобразования типов.	9

> Больше	Возвращает true, если первый операнд больше второго, в противном случае возвращает false.	10
>= Больше или равно	Возвращает true, если первый операнд не меньше второго, в противном случае возвращает false.	10
< Меньше	Возвращает true, если первый операнд меньше второго, в противном случае возвращает false.	10
<= Меньше или равно	Возвращает true, если первый операнд не больше второго, в противном случае возвращает false.	10

```
5 == "5"; // вернет true
5 != -5.0; // вернет true
5 === "5"; // вернет false
false === false; // вернет true
1 !== true; // вернет true
1 != true; // вернет false, так как true преобразуется в 1
3 > -3; // вернет true
3 >= "4"; // вернет false
```

Логические операторы

Логические операторы позволяют комбинировать условия, возвращающие логические величины. Чаще всего используются в условном выражении if.

Оператор/Операция	Описание	Приоритет
&& Логическое И	Возвращает true, только если оба операнда истинны. При выполнении операции сначала проверяется значение первого операнда. Если оно имеет значение false, то значение второго оператора не учитывается и результату выражения присваивается false.	5
|| Логическое ИЛИ	Возвращает true, если хотя бы один операнд истинен, т.е. проверяет истинность как минимум одного условия.	4
! Логическое НЕ	Изменяет значение оператора на обратное – с true на false и наоборот.	14

```
(2 < 3) && (3===3); // вернет true, так как выражения в обеих скобках дают true
(x < 10 && x > 0); // вернет true, если значение x принадлежит промежутку от 0 до 10
!false; // вернет true
```

Побитовые операторы

Побитовые операторы работают с операндами как с 32-битной последовательностью нулей и единиц и возвращают числовое значение, означающее результат операции, записанное в десятичной системе счисления. В качестве операндов рассматриваются целые числа, дробная часть операнда отбрасывается. Побитовые операции могут использоваться, например, при шифровании данных, для работы с флагами, разграничения прав доступа.

Оператор/Операция	Описание	Приоритет
& Побитовый И	Если оба бита равны 1, то результирующий бит будет равен 1. В противном случае результат равен 0.	8

`|` Побитовый ИЛИ	Если один из операндов содержит в позиции `1`, результат тоже будет содержать `1` в этой позиции, в противном случае результат в этой позиции будет равен `0`.	6
`^` Исключающее ИЛИ	Если одно, и только одно значение содержит `1` в какой-либо позиции, то и результат будет содержать `1` в этой позиции, в противном случае результат в этой позиции будет равен `0`.	7
`~` Отрицание	Выполняется операция побитового отрицания над двоичным представлением значения выражения. Любая позиция, содержащая `1` в исходном выражении, заменяется на `0`. Любая позиция, содержащая `0` в исходном выражении, становится равной `0`. Положительные числа начинаются с `0`, отрицательные – с `–1`, поэтому `~ n == –(n+1)`.	14
`<<` Побитовый сдвиг влево	Оператор сдвигает биты первого операнда влево на число битовых позиций, установленных вторым операндом. Для заполнения позиций справа используются нули. Возвращают результат того же типа, что левый операнд.	11
`>>` Побитовый сдвиг вправо	Оператор сдвигает биты первого операнда вправо на число битовых позиций, установленных вторым операндом. Цифры, сдвинутые за пределы диапазона, удаляются. Самый старший бит (32-й) не меняется, чтобы сохранить знак результата. Если первый операнд положителен, старшие биты результата заполняются нулями; если первый операнд отрицателен, старшие биты результата заполняются единицами. Сдвиг значения вправо на одну позицию эквивалентен делению на 2 (с отбрасыванием остатка), а сдвиг вправо на две позиции эквивалентен делению на 4 и т.д.	11
`>>>` Побитовый сдвиг вправо без учета знака	Оператор сдвигает биты первого операнда вправо на число битовых позиций, установленных вторым операндом. Слева добавляются нули независимо от знака первого операнда. Цифры, сдвинутые за пределы диапазона, удаляются.	11

```
let x = 9, y = 5, z = 2, s = -5, result; // 9 эквивалентно 1001, 5 эквивалентно 0101
result = x & y; // вернет 1 (эквивалентно 0001)
result = x | y; // вернет 13 (эквивалентно 1101)
result = x ^ y; // вернет 12 (эквивалентно 1100)
result = ~ y; // вернет -6 (эквивалентно 1100)
result = x << y; // вернет 288 (эквивалентно 100100000)
result = x >> z; // вернет 2 (эквивалентно 10)
result = s >>> z; // вернет 1073741822 (эквивалентно 11111111111111111111111111111110)
```

Строковые операторы

Существует несколько операторов, которые работают со строками особым образом.

Оператор/Операция	Описание	Приоритет
+ Конкатенация	Оператор работает слева направо, выполняя объединение строк. Если первый операнд является строкой, последующие операнды будут преобразованы в строки и далее выполнится их объединение.	12
+= Конкатенация с присваиванием	Выполняется объединение двух строк и результат присваивается переменной.	12
>, <, >=, <=, == Сравнение	Строки сравниваются по алфавиту, буквы в верхнем регистре всегда меньше букв в нижнем регистре. Сравнение строк основывается на номерах символов, указанных в стандарте Unicode, где прописные буквы идут раньше, чем строчные.	10

```
"1" + "10"; // вернет "110"
"1" + 10; // вернет "110"
2 + 5 + " цветных карандашей"; // вернет "7 цветных карандашей"
"Цветных карандашей " + 2 + 5; // вернет "Цветных карандашей 25"
"1" > "10"; // вернет false
"10" <= 10; // вернет true
"СССР" == "ссср"; // вернет false
x = "micro"; x+= "soft"; // вернет "microsoft"
```

Специальные операторы

Оператор/Операция	Описание	Приоритет
. Обращение к свойству	Осуществляет доступ к свойству объекта.	15
, Множественное вычисление	Вычисляет несколько независимых выражений, записанных в одну строку.	1
[] Индексация массива	Осуществляет доступ к элементам массива или свойствам объекта.	15
() Вызов функции, группировка	Группирует операции или вызывает функцию.	15
typeof Определение типа данных	Унарный оператор, возвращает тип данных операнда.	14
instanceof Проверка типа объекта	Оператор проверяет, является ли объект экземпляром определенного класса. Левый операнд должен быть объектом, правый – должен содержать имя класса объектов. Результат будет true, если объект, указанный слева, представляет собой экземпляр класса, указанного справа, в противном случае – false.	10
in Проверка наличия свойства	В качестве левого операнда должна быть строка, а правым – массив или объект. Если левое значение является свойством объекта, вернется результат true.	10

new Создание объекта	Оператор создает новый объект с неопределенными свойствами, затем вызывает функцию-конструктор для его инициализации (передачи параметров). Также может применяться для создания массива.	1
delete Удаление	Оператор позволяет удалять свойство из объекта или элемент из массива. Возвращает true, если удаление прошло успешно, в противном случае false. При удалении элемента массива его длина не меняется.	14
void Определение выражения без возвращаемого значения	Унарный оператор, отбрасывает значение операнда и возвращает underfined.	14
?: Операция условного выражения	Тернарный оператор, позволяет организовать простое ветвление. В выражении участвуют три операнда, первый должен быть логическим значением или преобразовываться в него, а второй и третий – любыми значениями. Если первый операнд равен true, то условное выражение примет значение второго операнда; если false – то третьего.	3

```
document.write("hello world"); // выводит на экран строку hello world

i = 0, j = 1; // сохраняет значения в переменных

function1(10, 5); // вызов функции function1 с параметрами 10 и 5

let year = [2014, 2015]; // создает массив с элементами

typeof {a:1}; // вернет "object"

let d = new Date(); // создаем новый объект с помощью конструктора Date()
d instanceof Date; // вернет true

let mycar = {make: "Honda", model: "Accord", year: 2005};
"make" in mycar; // вернет true

let obj = new Object(); // создает пустой объект

let food = ["milk", "bread", "meat", "olive oil", "cheese"];
delete food[3]; // удаляет четвертый элемент из массива food

Нажмите здесь, ничего не произойдет

x > 10 ? x * 2 : x / 2; // возвращает значение x * 2, если x > 10, в противном случае x / 2
```

Циклы JavaScript

Циклы JavaScript обеспечивают многократное выполнение повторяющихся вычислений. Они оптимизируют процесс написания кода, выполняя одну и ту же инструкцию или блок инструкций, образующих тело цикла, заданное число раз (используя переменную-счетчик) или пока заданное условие истинно. **Циклы** выполняют обход последовательности значений. Однократное выполнение цикла называется **итерацией**.

На производительность цикла влияют количество итераций и количество операций, выполняемых в теле цикла каждой итерации.

В JavaScript существуют следующие операторы цикла:

- for используется, когда вы заранее знаете, сколько раз вам нужно что-то сделать;
- for...in используется для обхода свойств объектов;

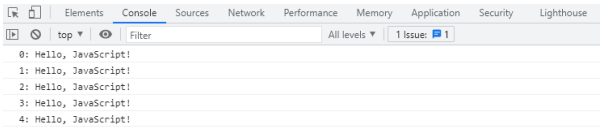
- while используется, когда вы не знаете, сколько раз нужно что-то сделать;
- do...while работает аналогично с оператором while. Отличается тем, что do...while всегда выполняет выражение в фигурных скобках, по крайней мере один раз, даже если проверка условия возвращает false.

Цикл for

Цикл for используется для выполнения итераций по элементам массивов или объектов, напоминающих массивы, таких как arguments и HTMLCollection. Условие проверяется перед каждой итерацией цикла. В случае успешной проверки выполняется код внутри цикла, в противном случае код внутри цикла не выполняется, и программа продолжает работу с первой строки, следующей непосредственно после цикла.

Следующий цикл выведет на консоль строчку Hello, JavaScript! пять раз.

```
for (let i = 0; i < 5; i++) {
  console.log(i + ": Hello, JavaScript!");
}
```



Как работает цикл for

Цикл for состоит из трех разных операций:

Шаг 1. инициализация let i = 0; – объявление переменной-счетчика, которая будет проверяться во время выполнения цикла. Эта переменная инициализируется со значением 0. Чаще всего в качестве счетчиков цикла выступают переменные с именами i, j и k.

Шаг 2. проверка условия i < 5; – условное выражение, если оно возвращает true, тело цикла (инструкция в фигурных скобках) будет выполнено. В данном примере проверка условия идет до тех пор, пока значение счетчика меньше 5.

Шаг 3. завершающая операция i++ – операция приращения счетчика, увеличивает значение переменной let i на единицу. Вместо операции инкремента также может использоваться операция декремента.

По завершении цикла в переменной let i сохраняется значение 1. Следующий виток цикла выполняется для for (let i = 1; i < 5; i++) { }. Условное выражение вычисляется снова, чтобы проверить, является ли значение счетчика i все еще меньше 5. Если это так, операторы в теле цикла выполняются еще раз. Завершающая операция снова увеличивает значение переменной на единицу. Шаги 2 и 3 повторяются до тех пор, пока условие i < 5; возвращает true.

Вывод значений массива

Чтобы вывести значения массива с помощью цикла for, нужно задействовать свойство массива length. Это поможет определить количество элементов в массиве и выполнить цикл такое же количество раз.

Приведенный ниже скрипт выведет на экран пять сообщений с названиями цветов:

```
let flowers = ["Rose", "Lily", "Tulip", "Jasmine", "Orchid"];
for (let i = 0; i < flowers.length; i++){
  alert(flowers[i] + " - это цветок.");
}
```

Если значение свойства length не изменяется в ходе выполнения цикла, можно сохранить его в локальной переменной, а затем использовать эту переменную в условном выражении. Таким образом можно повысить скорость выполнения цикла, так как значение свойства length будет извлекаться всего один раз за все время работы цикла.

```
let flowers = ["Rose", "Lily", "Tulip", "Jasmine", "Orchid"], len = flowers.length;
for (let i = 0; i < len; i++){
  alert(flowers[i] + " - это цветок.");
```

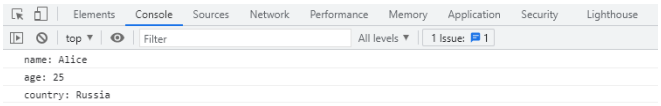
Цикл for...in

Циклы for...in используются для обхода свойств объектов, не являющихся массивами. Такой обход также называется перечислением.

Для примера создадим объект с помощью литерала объекта.

```
let user = {
  name: 'Alice',
  age: 25,
  country: 'Russia'
};

for (let prop in user) {
  console.log(prop + ": " + user[prop]);
}
```



Цикл while

Цикл while – цикл с предварительной проверкой условного выражения. Инструкция внутри цикла (блок кода в фигурных скобках) будет выполняться в случае, если условное выражение вычисляется в true. Если первая проверка даст результат false, блок инструкций не выполнится ни разу.

После завершения итерации цикла условное выражение опять проверяется на истинность и процесс будет повторяться до тех пор, пока выражение не будет вычислено как false. В этом случае программа продолжит работу с первой строки, следующей непосредственно после цикла (если таковая имеется).

Данный цикл выведет на экран таблицу умножения для числа 3:

1 x 3 = 3
2 x 3 = 6
3 x 3 = 9
4 x 3 = 12
5 x 3 = 15
6 x 3 = 18
7 x 3 = 21
8 x 3 = 24
9 x 3 = 27

```
let i = 1;
let msg = '';
while (i < 10) {
  msg+= i + ' x 3 = ' + (i * 3) + '<br>';
  i++;
}
document.write(msg);
```

Цикл do...while

Цикл do...while; проверяет условие продолжения после выполнения цикла. В отличие от цикла while, в do...while; тело цикла выполняется как минимум один раз, так как условие проверяется в конце цикла, а не в начале. Данный цикл используется реже, чем while, так как на практике ситуация, когда требуется хотя бы однократное исполнение цикла, встречается редко.

```
1 2 3 4 5
```

```
let result = '';
let i = 0;
do {
   i += 1;
   result += i + ' ';
} while (i < 5);
document.write(result);
```

В следующем примере операторы внутри цикла выполняются один раз, даже если условие не выполняется.

```
let i = 10;
do {
    document.write(i + ' ');
    i++;
} while (i < 10);
```

Бесконечные циклы

При создании любого цикла можно создать бесконечный цикл, который никогда не завершится. Такой цикл может потенциально продолжать работать до тех пор, пока работает компьютер пользователя. Большинство современных браузеров могут обнаружить это и предложат пользователю остановить выполнение скрипта. Чтобы избежать создания бесконечного цикла, вы должны быть уверены, что заданное условие в какой-то момент вернет false. Например, следующий цикл задает условие, которое никогда не возвращает ложь, так как переменная i никогда не будет меньше 10:

```
for (let i = 25; i > 10; i++) {
document.write("Это предложение будет выводиться бесконечно...<br>");
}
```

Вложенные циклы

Цикл внутри другого цикла называется вложенным. При каждой итерации цикла вложенный цикл выполняется полностью. Вложенные циклы можно создавать с помощью цикла for и цикла while.

```
1. Строка цикла
Строка вложенного цикла
Строка вложенного цикла
2. Строка цикла
Строка вложенного цикла
Строка вложенного цикла
```

```
for (let count = 1; count < 3; count++) {
  document.write(count + ". Строка цикла<br>");
  for (let nestcount = 1; nestcount< 3; nestcount++) {
    document.write("Строка вложенного цикла<br>");
  }
}
```

Управление циклом

Циклом можно управлять с помощью операторов break; и continue;.

Оператор break

Оператор break завершает выполнение текущего цикла. Он используется в исключительных случаях, когда цикл не может выполняться по какой-то причине, например, если приложение обнаруживает ошибку. Чаще всего оператор break является частью конструкции if.

Когда оператор break используется без метки, он позволяет выйти из цикла или из инструкции switch. В следующем примере создается счетчик, значения которого должны изменяться от 1 до 99, однако оператор break прерывает цикл после 14 итераций.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
for (let i = 1; i < 100; i++) {
  if (i == 15) {
    break;
    }
  document.write(i);
  document.write(' <br>');
}
```

Для вложенных циклов оператор break используется с меткой, с помощью которой завершается работа именованной инструкции. Метка позволяет выйти из любого блока кода. Именованной инструкцией может быть любая инструкция, внешняя по отношению к оператору break. В качестве метки может быть имя инструкции if или имя блока инструкций, заключенных в фигурные скобки только для присвоения метки этому блоку. Между ключевым словом break и именем метки не допускается перевод строки.

```
i = 0 j = 0
i = 0 j = 1
i = 0 j = 2
i = 0 j = 3
i = 1 j = 0
i = 1 j = 1
i = 1 j = 2
i = 1 j = 3
i = 3 j = 0
i = 3 j = 1
i = 3 j = 2
i = 3 j = 3
FINAL i = 4 j = 0
```

```
outerloop:
  for(let i = 0; i < 10; i++) {
    innerloop:
      for(let j = 0; j < 10; j++) {
        if (j > 3) break; // Выход из самого внутреннего цикла
        if (i == 2) break innerloop; // То же самое
        if (i == 4) break outerloop; // Выход из внешнего цикла
        document.write("i = " + i + " j = " + j + "<br>");
      }
  }
document.write("FINAL i = " + i + " j = " + j + "<br>");
```

Оператор continue

Оператор continue останавливает текущую итерацию цикла и запускает новую итерацию. При этом, цикл while возвращается непосредственно к своему условию, а цикл for сначала вычисляет выражение инкремента, а затем возвращается к условию.

В этом примере на экран будут выведены все четные числа:

чётное число = 2
чётное число = 4
чётное число = 6
чётное число = 8
чётное число = 10

```
let i;
  for(i = 1; i <= 10; i++) {
    if (i % 2 !== 0) {
      continue;
    }
    document.write("<br><b>чётное число</b> = " + i);
  }
```

Оператор continue также может применяться во вложенных циклах с меткой.

```
внешний цикл: 0
вложенный цикл: 0
вложенный цикл: 1
вложенный цикл: 2
внешний цикл: 1
внешний цикл: 2
вложенный цикл: 0
вложенный цикл: 1
вложенный цикл: 2
Все циклы выполнены
```

```
outerloop:
for (let i = 0; i < 3; i++)
{
  document.write("внешний цикл: "+i+"");
  for (let j = 0; j < 5; j++)
  {
    if (i == 1)
    break;
    if (j == 3)
    continue outerloop;
    document.write("вложенный цикл: "+j+"");
  }
}
document.write("Все циклы выполнены"+"");
```

Условное ветвление: if, '?'

Иногда нужно выполнить различные действия в зависимости от условий.

Для этого можно 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.

```
let year = prompt('В каком году была опубликована спецификация ECMAScript-2015?', '');

if (year == 2015) alert( 'Вы правы!' );
```

В примере выше, условие – это простая проверка на равенство (year == 2015), но оно может быть и гораздо более сложным.

Если нужно выполнить более одной инструкции, то нужно заключить блок кода в фигурные скобки:

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

Рекомендуется использовать фигурные скобки {} всегда, когда используется инструкция if, даже если выполняется только одна команда. Это улучшает читабельность кода.

Блок else

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

```
let year = prompt('В каком году была опубликована спецификация ECMAScript-2015?', '');

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.

```
let year = prompt('В каком году была опубликована спецификация ECMAScript-2015?', '');

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

В приведенном выше коде JavaScript сначала проверит year < 2015. Если это неверно, он переходит к следующему условию year > 2015. Если оно тоже ложно, тогда сработает последний alert().

Блоков else if может быть и больше. Присутствие блока else не является обязательным.

Условный оператор ?

Иногда нужно определить переменную в зависимости от условия.

```
let accessAllowed;
let age = prompt('Сколько вам лет?', '');

if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}

alert(accessAllowed);
```

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

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

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

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение2.

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

Технически, можно опустить круглые скобки вокруг age > 18. Оператор вопросительного знака имеет низкий приоритет, поэтому он выполняется после сравнения >.

```
// оператор сравнения "age > 18" выполняется первым в любом случае
// (нет необходимости заключать его в скобки)
let accessAllowed = age > 18 ? true : false;
```

Несколько операторов ?

Последовательность операторов вопросительного знака ? позволяет вернуть значение, которое зависит от более чем одного условия.

```
let age = prompt('Возраст?', 18);

let message = (age < 3) ? 'Здравствуй, малыш!' :
  (age < 18) ? 'Привет!' :
  (age < 100) ? 'Здравствуйте!' :
  'Какой необычный возраст!';

alert( message );
```

Первый знак вопроса проверяет age < 3.

Если верно – возвращает 'Здравствуй, малыш!'. В противном случае, проверяет выражение после двоеточия :, вычисляет age < 18.

Если это верно – возвращает 'Привет!'. В противном случае, проверяет выражение после следующего двоеточия :, вычисляет age < 100.

Если это верно – возвращает 'Здравствуйте!'. В противном случае, возвращает выражение после последнего двоеточия – 'Какой необычный возраст!'.

Нетрадиционное использование ?

Иногда оператор «вопросительный знак» ? используется в качестве замены if:

```
let company = prompt('Какая компания создала JavaScript?', '');

(company == 'Netscape') ?
   alert('Верно!') : alert('Неправильно.');
```

В зависимости от условия company == 'Netscape', будет выполнена либо первая, либо вторая часть после ?.

Здесь не присваивается результат переменной. Вместо этого выполняется различный код в зависимости от условия.

Оператор объединения с null ??

Оператор объединения с null представляет собой два вопросительных знака ??.

Результат выражения a ?? b будет следующим:

- a, если значение a определено,
- b, если значение a не определено.

То есть оператор ?? возвращает первый аргумент, если он не null/undefined, иначе второй.

Как правило, оператор ?? нужен для того, чтобы задать значение по умолчанию для потенциально неопределенной переменной.

Например, в следующем примере, если переменная user не определена, покажем модальное окно с надписью "Аноним":

```
let user;

alert(user ?? "Аноним"); // Аноним
```

Кроме этого, можно записать последовательность из операторов ??, чтобы получить первое значение из списка, которое не является null/undefined.

Допустим, у нас есть данные пользователя в переменных firstName, lastName или nickName. Все они могут быть неопределенными, если отсутствует соответствующая информация.

Выведем имя пользователя, используя одну из этих переменных, а в случае, если все они не определены, то покажем "Аноним".

Для этого воспользуемся оператором ??:

```
let firstName = null;
let lastName = null;
let nickName = "Суперкодер";

// показывает первое определённое значение:
alert(firstName ?? lastName ?? nickName ?? "Аноним"); // Суперкодер
```

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.

```
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.

```
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" );
}
```

Здесь оператор switch последовательно сравнит a со всеми вариантами из case.

Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert('В точку!') и далее, до ближайшего break, который прервет выполнение.

Если break нет, то выполнение пойдет ниже по следующим case, при этом остальные проверки игнорируются.

```
let a = 2 + 2;

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

Здесь последовательно выполнятся три alert().

Группировка «case»

Несколько вариантов case, использующих один код, можно группировать.

Для примера, выполним один и тот же код для case 3 и case 5, сгруппировав их:

```
let a = 2 + 2;

switch (a) {
  case 4:
    alert('Правильно!');
    break;

  case 3: // (*) группируем оба case
  case 5:
    alert('Неправильно!');
    alert("Может вам посетить урок математики?");
    break;

  default:
    alert('Результат выглядит странновато. Честно.');
}
```

Теперь оба варианта 3 и 5 выводят одно сообщение.

Функции

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

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

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций – это alert(message), prompt(message, default) и confirm(question). Но можно создавать и свои.

Объявление функции

Для создания функций можно использовать объявление функции (Function Declaration).

Пример объявления функции:

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

Вначале идет ключевое слово function, после него имя функции, затем список параметров в круглых скобках через запятую (в вышеприведенном примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

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

Новая функция может быть вызвана по ее имени: showMessage().

Например:

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

showMessage();
showMessage();
```

Вызов showMessage() выполняет код функции. Выведется 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.

Локальные переменные

Переменные, объявленные внутри функции, видны только внутри этой функции.

Например:

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

  alert( message );
}

showMessage(); // Привет, я JavaScript!

alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции
```

Внешние переменные

У функции есть доступ к внешним переменным, например:

```
let userName = 'Вася';

function showMessage() {
  let message = 'Привет, ' + userName;
  alert(message);
}

showMessage(); // Привет, Вася
```

Функция обладает полным доступом к внешним переменным и может изменять их значение.

Например:

```
let userName = 'Вася';

function showMessage() {
  userName = "Петя"; // (1) изменяем значение внешней переменной

  let message = 'Привет, ' + userName;
  alert(message);
}

alert( userName ); // Вася перед вызовом функции

showMessage();

alert( userName ); // Петя, значение внешней переменной было изменено функцией
```

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

Если одноименная переменная объявляется внутри функции, тогда она перекрывает внешнюю. 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 свою собственную локальную переменную userName
showMessage();

alert( userName ); // Вася, не изменилась, функция не трогала внешнюю переменную
```

Параметры

Можно передать внутрь функции любую информацию, используя параметры (также называемые аргументами функции).

В нижеприведенном примере функции передаются два параметра: `from` и `text`.

```
function showMessage(from, text) { // аргументы: from, text
  alert(from + ': ' + text);
}

showMessage('Аня', 'Привет!'); // Аня: Привет! (*)
showMessage('Аня', "Как дела?"); // Аня: Как дела? (**)
```

Когда функция вызывается в строках (*) и (**), переданные значения копируются в локальные переменные `from` и `text`. Затем они используются в теле функции.

Параметры по умолчанию

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

Например, вышеупомянутая функция `showMessage(from, text)` может быть вызвана с одним аргументом:

```
showMessage("Аня");
```

Это не приведет к ошибке. Такой вызов выведет `"Аня: undefined"`. В вызове не указан параметр `text`, поэтому предполагается, что `text === undefined`.

Если необходимо задать параметру `text` значение по умолчанию, нужно указать его после `=`:

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}

showMessage("Аня"); // Аня: текст не добавлен
```

Теперь, если параметр `text` не указан, его значением будет `"текст не добавлен"`.

В данном случае `"текст не добавлен"` это строка, но на ее месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра. Например:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() выполнится только если не передан text
  // результатом будет значение text
}
```

Возврат значения

Функция может вернуть результат, который будет передан в вызвавший ее 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 ее код (присваивается переменной result выше).

Вызовов return может быть несколько, например:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('А родители разрешили?');
  }
}

let age = prompt('Сколько вам лет?', 18);

if ( checkAge(age) ) {
  alert( 'Доступ получен' );
} else {
  alert( 'Доступ закрыт' );
}
```

Возможно использовать return и без значения. Это приведет к немедленному выходу из функции.

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Вам показывается кино" ); // (*)
  // ...
}
```

В коде выше, если checkAge(age) вернет false, showMovie() не выполнит alert().

Выбор имени функции

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть простым, точным и описывать действие функции, чтобы программист, который будет читать код, получил верное представление о том, 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.

```
showMessage(..)     // показывает сообщение
getAge(..)          // возвращает возраст (в каком либо значении)
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 ее код, и какое значение она может возвращать.

Функциональное выражение

Синтаксис, который использовался до этого, называется Function Declaration (Объявление Функции):

```
function sayHi() {
  alert( "Привет" );
}
```

Существует еще один синтаксис создания функций, который называется Function Expression (Функциональное Выражение):

```
let sayHi = function() {
  alert( "Привет" );
};
```

В коде выше функция создается и явно присваивается переменной, как любое другое значение. По сути, без разницы, как определили функцию, это просто значение, хранимое в переменной sayHi.

Смысл обоих примеров кода одинаков: "создать функцию и поместить ее значение в переменную sayHi".

```
function sayHi() {
  alert( "Привет" );
}

alert( sayHi ); // выведет код функции
```

Последняя строка не вызывает функцию sayHi(), после ее имени нет круглых скобок. Существуют языки программирования, в которых любое упоминание имени функции совершает ее вызов. JavaScript – не один из них.

В JavaScript функции – это значения, поэтому и обращаться с ними нужно, как со значениями. Код выше выведет строковое представление функции, которое является ее исходным кодом.

Можно скопировать функцию в другую переменную:

```
1  function sayHi() {   // (1) создаём
2    alert( "Привет" );
3  }
4
5  let func = sayHi;    // (2) копируем
6
7  func(); // Привет    // (3) вызываем копию (работает)!
8  sayHi(); // Привет   //     прежняя тоже работает (почему бы нет)
```

1. Объявление Function Declaration (1) создало функцию и присвоило ее значение переменной с именем sayHi.
2. В строке (2) происходит копирование ее значение в переменную func. Внимание: нет круглых скобок после sayHi. Если бы они были, то выражение func = sayHi() записало бы результат вызова sayHi() в переменную func, а не саму функцию sayHi.
3. Теперь функция может быть вызвана с помощью обеих переменных sayHi() и func().

Функции-«колбэки»

Рассмотрим примеры функциональных выражений и передачи функции как значения.

Напишем функцию ask(question, yes, no) с тремя параметрами:

- question – Текст вопроса.
- yes – Функция, которая будет вызываться, если ответ будет «Yes».
- no –Функция, которая будет вызываться, если ответ будет «No».

Функция должна задать вопрос question и, в зависимости от того, как ответит пользователь, вызвать yes() или no():

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "Вы согласны." );
}

function showCancel() {
  alert( "Вы отменили выполнение." );
}

// использование: функции showOk, showCancel передаются в качестве аргументов ask
ask("Вы согласны?", showOk, showCancel);
```

На практике подобные функции очень полезны. Основное отличие «реальной» функции ask() от примера выше будет в том, что она использует более сложные способы взаимодействия с пользователем, чем простой вызов confirm(). В браузерах такие функции обычно отображают красивые диалоговые окна.

Аргументы функции ask еще называют **функциями-колбэками** или просто **колбэками**.

Ключевая идея в том, что передается функция и ожидается, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо. В нашем случае, showOk становится колбэком для ответа «yes», а showCancel – для ответа «no».

Можно переписать этот пример значительно короче, используя Function Expression:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Вы согласны?",
  function() { alert("Вы согласились."); },
  function() { alert("Вы отменили выполнение."); }
);
```

Здесь функции объявляются прямо внутри вызова ask(...). У них нет имен, поэтому они называются анонимными. Такие функции недоступны снаружи ask (потому что они не присвоены переменным).

Функции-стрелки

Существует еще более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression.

Он называется «функции-стрелки» или «стрелочные функции» (arrow functions), т.к. выглядит следующим образом:

```
let func = (arg1, arg2, ...argN) => expression
```

Такой код создает функцию func с аргументами arg1…argN и вычисляет expression с правой стороны с их использованием, возвращая результат.

Другими словами, это более короткий вариант такой записи:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Пример:

```
let sum = (a, b) => a + b;

/* Более короткая форма для:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

То есть, `(a, b) => a + b` задает функцию с двумя аргументами `a` и `b`, которая при запуске вычисляет выражение справа `a + b` и возвращает его результат.

Если используется только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись еще короче:

```
// тоже что и
// let double = function(n) { return n * 2 }
let double = n => n * 2;

alert( double(3) ); // 6
```

Если нет аргументов, указываются пустые круглые скобки:

```
let sayHi = () => alert("Hello!");

sayHi();
```

Функции-стрелки могут быть использованы так же, как и Function Expression.

Например, для динамического создания функции:

```
let age = prompt("Сколько Вам лет?", 18);

let welcome = (age < 18) ?
  () => alert('Привет') :
  () => alert("Здравствуйте!");

welcome(); // теперь всё в порядке
```

Многострочные стрелочные функции

В примерах выше аргументы использовались слева от `=>`, а справа вычислялось выражение с их значениями.

Порой необходимо, например, выполнить несколько инструкций. Это также возможно, нужно лишь заключить инструкции в фигурные скобки. И использовать `return` внутри них, как в обычной функции.

Например:

```
let sum = (a, b) => {  // фигурная скобка, открывающая тело многострочной функции
  let result = a + b;
  return result; // при фигурных скобках для возврата значения нужно явно вызвать return
};

alert( sum(1, 2) ); // 3
```

Объекты

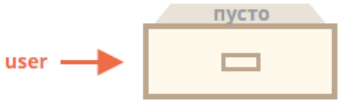
В JavaScript существует 8 типов данных. Семь из них называются «примитивными», так как содержат только одно значение (будь то строка, число или что-то другое).

Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка.

Объект может быть создан с помощью фигурных скобок `{…}` с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

Можно представить объект в виде ящика с подписанными папками. Каждый элемент данных хранится в своей папке, на которой написан ключ. По ключу папку легко найти, удалить или добавить в нее что-либо.

Пустой объект можно создать, используя один из двух вариантов синтаксиса:




```
let user = new Object(); // синтаксис "конструктор объекта"
let user = {};  // синтаксис "литерал объекта"
```

Обычно используют вариант с фигурными скобками `{...}`. Такое объявление называют литералом объекта или литеральной нотацией.

Литералы и свойства

При использовании литерального синтаксиса `{...}` можно поместить в объект несколько свойств в виде пар «ключ: значение»:

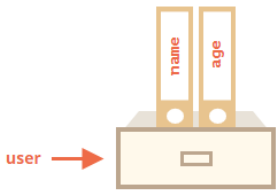
```
let user = {     // объект
  name: "John",  // под ключом "name" хранится значение "John"
  age: 30        // под ключом "age" хранится значение 30
};
```

У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие `:`, и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте `user` сейчас находятся два свойства:

- Первое свойство с именем `name` и значением `"John"`.
- Второе свойство с именем `age` и значением `30`.

Можно сказать, что объект `user` – это ящик с двумя папками, подписанными «name» и «age».

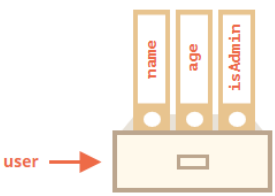


Можно в любой момент добавить в него новые папки, удалить папки или прочитать содержимое любой папки.

Для обращения к свойствам используется запись «через точку»:

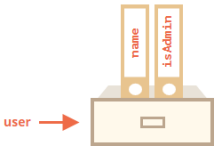
```
// получаем свойства объекта:
alert( user.name ); // John
alert( user.age ); // 30
```

Значение может быть любого типа. Добавим свойство с логическим значением:



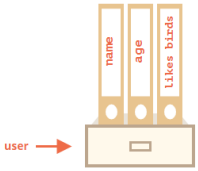

```
user.isAdmin = true;
```

Для удаления свойства используется оператор delete:




```
delete user.age;
```

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:




```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // имя свойства из нескольких слов должно быть в кавычках
};
```

Последнее свойство объекта может заканчиваться запятой:

```
let user = {
  name: "John",
  age: 30,
}
```

Это называется «висячая запятая». Такой подход упрощает добавление, удаление и перемещение свойств, так как все строки объекта становятся одинаковыми.

Квадратные скобки

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

```
// это вызовет синтаксическую ошибку
user.likes birds = true
```

JavaScript видит, что идет обращение к свойству user.likes, а затем идет непонятное слово birds. В итоге синтаксическая ошибка.

Точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме $ и _.

Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

```
let user = {};

// присваивание значения свойству
user["likes birds"] = true;

// получение значения свойства
alert(user["likes birds"]); // true

// удаление свойства
delete user["likes birds"];
```

Строка в квадратных скобках заключена в кавычки (подойдет любой тип кавычек).

Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной:

```
let key = "likes birds";

// то же самое, что и user["likes birds"] = true;
user[key] = true;
```

Здесь переменная key может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого используем ее для доступа к свойству. Это дает большую гибкость.

Пример:

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("Что вы хотите узнать о пользователе?", "name");

// доступ к свойству через переменную
alert( user[key] ); // John (если ввели "name")
```

Запись «через точку» такого не позволяет:

```
let user = {
  name: "John",
  age: 30
};

let key = "name";
alert( user.key ); // undefined
```

Вычисляемые свойства

Можно использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства.

Пример:

```
let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {
  [fruit]: 5, // имя свойства будет взято из переменной fruit
};

alert( bag.apple ); // 5, если fruit="apple"
```

Запись [fruit] означает, что имя свойства необходимо взять из переменной fruit.

И если посетитель введет слово "apple", то в объекте bag теперь будет лежать свойство {apple: 5}.

Можно использовать и более сложные выражения в квадратных скобках:

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Квадратные скобки дают намного больше возможностей, чем запись через точку. Они позволяют использовать любые имена свойств и переменные, хотя и требуют более громоздких конструкций кода.

Свойство из переменной

В реальном коде часто необходимо использовать существующие переменные как значения для свойств с тем же именем.

Например:

```
function makeUser(name, age) {
  return {
    name: name,
    age: age
    // ...другие свойства
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

Название свойств name и age совпадают с названиями переменных, которые подставлены в качестве значений этих свойств. Такой подход настолько распространен, что существуют специальные короткие свойства для упрощения этой записи.

Вместо name: name можно написать просто name:

```
function makeUser(name, age) {
  return {
    name, // то же самое, что и name: name
    age   // то же самое, что и age: age
    // ...
  };
}
```

Можно использовать как обычные свойства, так и короткие в одном и том же объекте:

```
let user = {
  name,  // тоже самое, что и name:name
  age: 30
};
```

Ограничения на имена свойств

Имя переменной не может совпадать с зарезервированными словами, такими как for, let, return и т.д. Для свойств объекта такого ограничения нет!

```
// эти имена свойств допустимы
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return );  // 6
```

Методы объекта, this

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

```
// Объект пользователя
let user = {
  name: "Джон",
  age: 30
};
```

И так же, как и в реальном мире, пользователь может совершать действия: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п.

Такие действия в JavaScript представлены свойствами-функциями объекта.

```
let user = {
  name: "Джон",
  age: 30
};

user.sayHi = function() {
  alert("Привет!");
};

user.sayHi(); // Привет!
```

Здесь использовалось Function Expression (функциональное выражение), чтобы создать функцию для приветствия, и присвоили ее свойству user.sayHi объекта.

Функцию, которая является свойством объекта, называют методом этого объекта.

Ключевое слово «this» в методах

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода).

Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user.

Для доступа к информации внутри объекта метод может использовать ключевое слово this.

Значение this – это объект «перед точкой», который использовался для вызова метода.

Например:

```
let user = {
  name: "Джон",
  age: 30,

  sayHi() {
    // this - это "текущий объект"
    alert(this.name);
  }

};

user.sayHi(); // Джон
```

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user).

Конструкторы, создание объектов через new

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую необходимо создать множество однотипных объектов, таких как пользователи, элементы меню и т.д.

Это можно сделать при помощи функции-конструктора и оператора new.

Когда функция вызывается как new User(...), происходит следующее:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Вася");

alert(user.name); // Вася
alert(user.isAdmin); // false
```

1. Создается новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

Другими словами, вызов new User(...) выполняет:

```
function User(name) {
  // this = {};  (неявно)

  // добавляет свойства к this
  this.name = name;
  this.isAdmin = false;

  // return this;  (неявно)
}
```

То есть, результат вызова new User("Вася") – это тот же объект, что и:

```
let user = {
  name: "Вася",
  isAdmin: false
};
```

Теперь, когда необходимо будет создать других пользователей, можно использовать new User("Маша"), new User("Даша") и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Создание методов в конструкторе

Использование конструкторов для создания объектов дает большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать.

В this можно добавлять не только свойства, но и методы.

В примере ниже, new User(name) создает объект с данным именем name и методом sayHi:

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "Меня зовут: " + this.name );
  };
}

let vasya = new User("Вася");

vasya.sayHi(); // Меня зовут: Вася

/*
vasya = {
    name: "Вася",
    sayHi: function() { ... }
}
*/
```

Массивы

Объекты позволяют хранить данные со строковыми ключами. Это замечательно.

Но довольно часто необходимо упорядоченная коллекция данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д. Например, она нужна для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Нельзя вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

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

Объявление

Существует два варианта синтаксиса для создания пустого массива:

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

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

Элементы массива нумеруются, начиная с нуля.

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

Можно получить элемент, указав его номер в квадратных скобках:

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

alert( fruits[0] ); // Яблоко
alert( fruits[1] ); // Апельсин
alert( fruits[2] ); // Слива
```

Можно заменить элемент:

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

Добавить новый к существующему массиву:

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

Общее число элементов массива содержится в его свойстве length:

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

alert( fruits.length ); // 3
```

Вывести массив целиком можно при помощи alert().

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

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

В массиве могут храниться элементы любого типа.

```
// разные типы значений
let arr = [ 'Яблоко', { name: 'Джон' }, true, function() { alert('привет'); } ];

// получить элемент с индексом 1 (объект) и затем показать его свойство
alert( arr[1].name ); // Джон

// получить элемент с индексом 3 (функция) и выполнить её
arr[3](); // привет
```

Строки

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренний формат для строк – всегда UTF-16, вне зависимости от кодировки страницы.

Многострочные строки

С помощью обратных кавычек можно занимать более одной строки:

```
let guestList = `Guests:
 * John
 * Pete
 * Mary
`;

alert(guestList); // список гостей, состоящий из нескольких строк
```

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя так называемый «символ перевода строки», который записывается как \n:

```
let guestList = "Guests:\n * John\n * Pete\n * Mary";

alert(guestList); // список гостей, состоящий из нескольких строк
```

В частности, эти две строки эквивалентны, просто записаны по-разному:

```
// перевод строки добавлен с помощью символа перевода строки
let str1 = "Hello\nWorld";

// многострочная строка, созданная с использованием обратных кавычек
let str2 = `Hello
World`;

alert(str1 == str2); // true
```

Длина строки

Свойство length содержит длину строки:

```
alert(`My`.length); // 2
```

Доступ к символам

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод charAt: str.charAt(pos). Первый символ занимает нулевую позицию:

```
let str = "Hello";

// получаем первый символ
alert(str[0]); // H
alert(str.charAt(0)); // H

// получаем последний символ
alert(str[str.length - 1]); // o
```

Квадратные скобки – современный способ получить символ, в то время как charAt существует в основном по историческим причинам.

Разница только в том, что если символ с такой позицией отсутствует, тогда [] вернет undefined, а charAt – пустую строку:

```
let str = "Hello";

alert(str[1000]); // undefined
alert(str.charAt(1000)); // '' (пустая строка)
```

Строки неизменяемы

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана – она такая навсегда.

```
let str = "Hello";

str[0] = 'h'; // ошибка
alert(str[0]); // не работает
```

Изменение регистра

Методы toLowerCase() и toUpperCase() меняют регистр символов:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

Можно перевести в нижний регистр какой-то конкретный символ:

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

Поиск подстроки

Существует несколько способов поиска подстроки.

str.indexOf – ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо –1 при отсутствии совпадений.

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, потому что подстрока 'Widget' найдена в начале
alert( str.indexOf('widget') ); // -1, совпадений нет, поиск чувствителен к регистру

alert( str.indexOf("id") ); // 1, подстрока "id" найдена на позиции 1 (..idget with id)
```

Необязательный второй аргумент позволяет начать поиск с определенной позиции.

Например, первое вхождение "id" – на позиции 1. Для того, чтобы найти следующее, начнем поиск с позиции 2:

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Каждый раз, получив очередную позицию, начинаем новый поиск со следующей:

```
let str = 'Ослик Иа-Иа посмотрел на виадук';

let target = 'Иа'; // цель поиска

let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Найдено тут: ${foundPos}` );
  pos = foundPos + 1; // продолжаем со следующей позиции
}
```

Также есть похожий метод str.lastIndexOf(substr, position), который ищет с конца строки к ее началу. Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определенной позиции.

Более современный метод str.includes(substr, pos) возвращает true, если в строке str есть подстрока substr, либо false, если нет.

Его нужно использовать при проверки на совпадение, но если позиция не нужна:

```
alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false
```

Необязательный второй аргумент str.includes позволяет начать поиск с определенной позиции:

```
alert( "Midget".includes("id") ); // true
alert( "Midget".includes("id", 3) ); // false, поиск начат с позиции 3
```

Методы str.startsWith и str.endsWith проверяют, соответственно, начинается ли и заканчивается ли строка определенной строкой:

```
alert( "Widget".startsWith("Wid") ); // true, "Wid" – начало "Widget"
alert( "Widget".endsWith("get") ); // true, "get" – окончание "Widget"
```

Получение подстроки

В JavaScript есть 3 метода для получения подстроки: substring, substr и slice.

str.slice(start [, end]) – возвращает часть строки от start до (не включая) end.

```
let str = "stringify";
// 'strin', символы от 0 до 5 (не включая 5)
alert( str.slice(0, 5) );
// 's', от 0 до 1, не включая 1, т. е. только один символ на позиции 0
alert( str.slice(0, 1) );
```

Если аргумент end отсутствует, slice возвращает символы до конца строки:

```
let str = "stringify";
alert( str.slice(2) ); // ringify, с позиции 2 и до конца
```

Также для start/end можно задавать отрицательные значения. Это означает, что позиция определена как заданное количество символов с конца строки:

```
let str = "stringify";

// начинаем с позиции 4 справа, а заканчиваем на позиции 1 справа
alert( str.slice(-4, -1) ); // gif
```

str.substring(start [, end]) – возвращает часть строки между start и end.

Это – почти то же, что и slice, но можно задавать start больше end.

```
let str = "stringify";

// для substring эти два примера – одинаковы
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// …но не для slice:
alert( str.slice(2, 6) ); // "ring" (то же самое)
alert( str.slice(6, 2) ); // "" (пустая строка)
```

Отрицательные значения substring, в отличие от slice, не поддерживает, они интерпретируются как 0.

str.substr(start [, length]) – возвращает часть строки от start длины length.

В противоположность предыдущим методам, этот позволяет указать длину вместо конечной позиции:

```
let str = "stringify";
// ring, получаем 4 символа, начиная с позиции 2
alert( str.substr(2, 4) );
```

Значение первого аргумента может быть отрицательным, тогда позиция определяется с конца:

```
let str = "stringify";
// gi, получаем 2 символа, начиная с позиции 4 с конца строки
alert( str.substr(-4, 2) );
```

Перспективы

То, что описано выше является лишь малой частью для изучения языка JavaScript. Рекомендуется использовать ресурс javascript.ru для дальнейшего изучения, начиная с нуля и заканчивая продвинутыми концепциями вроде ООП.

Задание к лабораторной работе №5

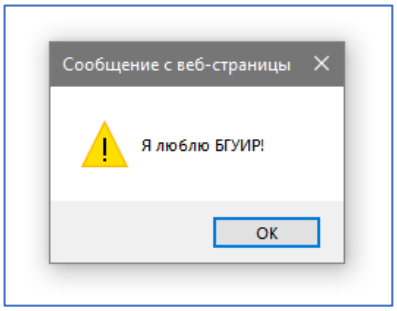
Для выполнения лабораторной работы необходимо установить и настроить редактор кода. При выполнении необходимо использовать в браузере инструменты разработчика.

Задание к лабораторной работе №5 состоит из задач разного уровня сложности. Выполнение задач 1-7 оценивается максимально в 5 баллов, задач 1-10 в 10 баллов.

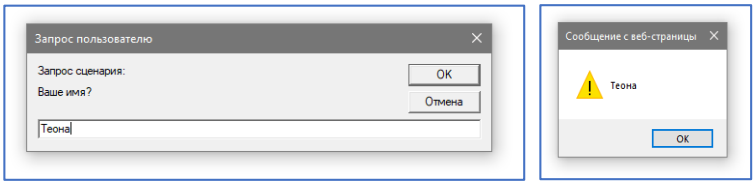
Задача 1

Условие: Необходимо создать два скрипта и внедрить их в html документы.

1. Создайте страницу, которая отобразит сообщение «Я люблю БГУИР!». Скрипт необходимо внедрить в код html.



2. Создайте страницу, которая запрашивает имя у пользователя и выводит его. Содержимое скрипта должно находится внешний файл alert.js.



Задача 2

Условие: Необходимо создать два скрипта и внедрить их в html документы:

1. Создайте переменную name и присвойте ей значение своего имени. Обращаясь к отдельным символам этой строки выведите на экран первый символ, второй символ, третий символ и т.д.

2. Создайте переменную num и присвойте ей значение '12345'. Найдите произведение (умножение) цифр этого числа.

Задача 3

Условие: Необходимо внести правки в скрипты.

1. Необходимо переделать этот код так, чтобы в нем использовались операции +=, -=, *=, /=. Количество строк кода при этом не должно измениться.

```
let num = 47;
num = num + 7;
num = num - 18;
num = num * 10;
num = num / 15;
alert(num);
```

2. Необходимо переделать этот код так, чтобы в нем использовались операции ++ и --. Количество строк кода при этом не должно измениться.

```
let num = 10;
num = num + 1;
num = num + 1;
num = num - 1;
alert(num);
```

Задача 4

Условие: Необходимо создать два скрипта:

1. В переменной time лежит число от 0 до 59. Определите в какую четверть часа попадает это число (в первую, вторую, третью или четвертую).
2. Переменная lang может принимать два значения: 'ru' 'by'. Если она имеет значение 'ru', то в переменную week запишется массив дней недели на русском языке, а если имеет значение 'by' – то на белорусском. Решение необходимо выполнить через switch-case.

Задача 5

Условие: Необходимо создать два скрипта:

1. Дан массив с элементами [5, 6, 7, 8]. С помощью цикла for найдите произведение элементов этого массива.
2. Дан объект obj с ключами 'Минск', 'Москва', 'Киев' с элементами 'Беларусь', 'Россия', 'Украина'. С помощью for…in выведите на экран строки такого формата: 'Минск – это Беларусь.'.

Задача 6

Условие: Необходимо создать скрипт:

1. Напишите функцию, которая возвращает квадрат числа. Число передается параметром.
2. Напишите функцию, которая возвращает сумму двух чисел.
3. Напишите функцию, которая вычитает от первого числа второе и делит на третье.
4. Напишите функцию, которая принимает параметром число от 1 до 7, а возвращает день недели на русском языке.

Задача 7

Условие: Необходимо создать скрипт:

1. Напишите функцию, которая параметрами принимает два числа. Если эти числа равны – пусть функция вернет true, а если не равны – false.
2. Напишите функцию, которая параметрами принимает два числа. Если их сумма больше 10 – пусть функция вернет true, а если нет – false.
3. Напишите функцию, которая параметром принимает число и проверяет – отрицательное оно или нет. Если отрицательное – пусть функция вернет true, а если нет – false.

Задача 8

Условие: Необходимо создать скрипт:

1. Напишите функцию, которая принимает два числа и возвращает 1, если первое число больше, чем второе; –1, если первое число меньше, чем второе, и 0, если числа равны. Реализовать эту функцию необходимо с помощью тернарного оператора, и в виде стрелочной функции.

Задача 9

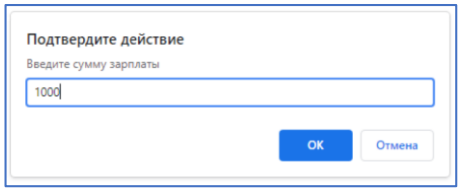
Условие: Необходимо создать скрипт:

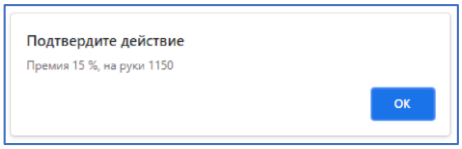
1. Скрипт должен осуществить проверку на наличие в начале строки http://.
2. Скрипт должен осуществить проверку на наличие в конце строки .html.

Задача 10

Условие: Необходимо создать скрипт:

1. Попросить у пользователя ввести сумму его зарплаты.
2. Затем рассчитываете: Добавку в виде премии в сумме 15% от зарплаты. Налоги в сумме 10% от суммы всех начислений (зарплата + премия). Трату в магазине на сумму 90 рублей. Разделите оставшуюся сумму между пользователем и его "женой"/"мужем". При решении используйте арифметические операторы присваивания, например, `result-=100` или `result/=2`, где `result` – это переменная, в которую вы записываете все действия по расчету сумм. Выводите, сколько осталось в результате всех операций денег у пользователя.





...