Практическая работа №9 «Основы JavaScript»

Введение

JavaScript - это язык программирования, выполняющийся на стороне пользователя с помощью браузера. Он позволяет управлять элементами веб-страницы - заставлять их менять свои свойства и расположение, двигаться, реагировать на события, такие как перемещение мыши или нажатия клавиатуры, а также создавать множество других интересных эффектов.

Изначально JavaScript был создан, чтобы «сделать веб-страницы живыми». Программы на этом языке называются скриптами. Они могут встраиваться в HTML и выполняться автоматически при загрузке веб-страницы. Скрипты распространяются и выполняются, как простой текст. Им не нужна специальная подготовка или компиляция для запуска. Это отличает JavaScript от другого языка – Java.

Что может JavaScript в браузере?

Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не требующих этого. Возможности JavaScript сильно зависят от окружения, в котором он работает. Например, Node.js поддерживает функции чтения/записи произвольных файлов, выполнения сетевых запросов и т.д. В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием vc пользователем и веб-сервером.

Например, в браузере JavaScript может:

- Добавлять новый HTML-код на страницу, изменять существующее содержимое, модифицировать стили.
- Реагировать на действия пользователя, щелчки мыши, перемещения указателя, нажатия клавиш.
- Отправлять сетевые запросы на удалённые сервера, скачивать и загружать файлы (технологии AJAX и COMET).
- Получать и устанавливать куки, задавать вопросы посетителю, показывать сообщения.
- Запоминать данные на стороне клиента («local storage»).

Чего НЕ может JavaScript в браузере?

Возможности JavaScript в браузере ограничены ради безопасности пользователя. Цель заключается в предотвращении доступа недобросовестной веб-страницы к личной информации или нанесения ущерба данным пользователя.

Примеры таких ограничений включают в себя:

JavaScript на веб-странице не может читать/записывать произвольные файлы на жёстком диске, копировать их или запускать программы. Он не имеет прямого доступа к системным функциям ОС. Современные браузеры позволяют ему работать с файлами, но с ограниченным доступом, и предоставляют его, только если пользователь выполняет определённые действия, такие как «перетаскивание» файла в окно браузера или его выбор с помощью тега `<input>`. Существуют способы взаимодействия с камерой/микрофоном и другими устройствами, но они требуют явного разрешения пользователя. Таким образом, страница с поддержкой JavaScript не может незаметно включить веб-камеру, наблюдать за происходящим и отправлять информацию куда-либо.

Различные окна/вкладки не знают друг о друге. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае JavaScript с одной страницы не имеет доступа к другой, если они пришли с разных сайтов (с другого домена, протокола или порта). Это называется «Политика одинакового источника» (Same Origin Policy). Чтобы обойти это ограничение, обе страницы должны согласиться с этим и содержать JavaScript-код, который специальным образом обменивается данными. Это ограничение необходимо, опять же, для безопасности пользователя. Страница https://anysite.com, которую открыл пользователь, не должна иметь доступ к другой вкладке браузера с URL https://gmail.com и воровать информацию оттуда.

JavaScript может легко взаимодействовать с сервером, с которого пришла текущая страница. Но его способность получать данные с других сайтов/доменов ограничена. Хотя это возможно в принципе, для чего требуется явное согласие (выраженное в заголовках HTTP) с удалённой стороной. Опять же, это ограничение безопасности. Подобные ограничения не действуют, если JavaScript используется вне браузера, например — на сервере. Современные браузеры предоставляют плагины/расширения, с помощью которых можно запрашивать дополнительные разрешения.

JavaScript имеет синтаксис схожий с языком Си, однако имеет ряд существенных отличий:

- Возможность работы с объектами, в том числе определение типа и структуры объекта во время выполнения программы.

– Возможность передавать и возвращать функции как параметры, а также присваивать их переменной.
– Наличие механизма автоматического приведения типов.
– Автоматическая сборка мусора.
– Использование анонимных функций.

С помощью CSS описывается, например, как выглядит кнопка, но она не работает. Включить её может JavaScript. Этот язык оживляет элементы веб-страниц, управляет их поведением. Код на JavaScript называется «скрипт», в переводе «сценарий». Когда скрипт запускается, браузер совершает операции по определённому сценарию.

Чтобы не запутаться в сценарии, код на JavaScript пишут в отдельном файле и оставляют комментарии. Обычно этот файл называют `script.js`.

В `script.js` описан сценарий, который будет управлять поведением кнопки. После нажатия на кнопку скрипт изменит её цвет.

Например, можно создать 3 новых файлика в отдельной папке на компьютере и запустить `index.html`.

`script.js`

```
// при клике по кнопке скрипт запустит код
button.onclick = function () {
    // если фон кнопки чёрный
    if (button.style.backgroundColor == 'black') {
      // изменим его на белый, а текст сделаем чёрным
      button.style.backgroundColor = 'white';
      button.style.color = 'black';
    } else {
      // иначе сделаем фон чёрным, а текст белым

    }
  }
```

index.html

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <!-- подключение файла стилей -->
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <button id="button">Нравится</button>
  <!-- подключение скрипта -->
  <script src="script.js"></script>
</body>
</html>
```

style.css

```css
button {
    background-color: black;
    color: white;
}
```

Правда, если нажать на кнопку еще раз, цвет уже не изменится. В `script.js` расписан только первый шаг. Попробуйте написать второе условие самостоятельно прямо в этом коде. Нужно, чтобы цвет фона изменился на чёрный, а текст стал белым.

JavaScript не только управляет кнопками, всплывающими окнами и кликами в меню. На самом деле — он всесилен. Во многом это возможно благодаря миллионам людей, которые на досуге совершенствуют этот язык и создают разные библиотеки.

1 Внедрение кода JavaScript

Внедрение кода JavaScript в документ HTML можно выполнить двумя способами:

- Первый - размещение кода непосредственно в HTML-файле,
- Второй - размещение кода в отдельном файле.

Рассмотрим оба способа:

1. **При размещении кода внутри HTML-файла** код JavaScript обрамляется тэгом `<script>`. Выглядит это примерно так:

```html
<script>
    JavaScript-код
</script>
```

Код такой страницы выглядит следующим образом:

```
<html>
<head>
    <title>Страница с примером кода JavaScript</title>

    <script>
    alert("Hello World!");
    </script>

</head>
<body>
    Это текст основной страницы
</body>
</html>
```

При открытии данной страницы появится окошко с надписью "Hello World!" и кнопкой "OK". После нажатия на кнопку "OK" продолжится выполнение страницы и на ней появится надпись: "Это текст основной страницы".

2. Теперь рассмотрим пример **размещения JavaScript-кода в отдельном файле**.

Создадим файл с названием `myscript.js` и поместим в него наш код:

```
alert("Hello World!");
```

А вызов кода из тела основного HTML-файла теперь будет выглядеть вот так:

```
<html>
  <head>
    <title>Страница с примером кода JavaScript</title>
    <script src="myscript.js"></script>
  </head>
  <body>
    Это текст основной страницы
  </body>
</html>
```

Плюсы такого использования очевидны: при любом объеме JavaScript-кода вызов обеспечивается одной небольшой строчкой.

2 Структура программы

Теперь рассмотрим, из чего состоит код JavaScript. Основной единицей кода является инструкция.

Инструкция JavaScript - это фактически команда браузеру, указание, что необходимо выполнить. Инструкцию желательно заканчивать точкой с запятой ";". Это не обязательное требование, согласно стандарту предполагается, что браузер интерпретирует конец строки как конец инструкции. Но использование точки с запятой в конце инструкции считается хорошей программистской практикой и, кроме того, позволяет написание нескольких инструкций в одной строке, что, правда, считается плохим стилем оформления кода. Также необходимо отметить, что отсутствие точки с запятой в конце строки в определенных случаях влияет на поведение кода, поэтому без лишней необходимости эту возможность желательно не использовать.

Код JavaScript - это последовательность инструкций JavaScript. Инструкции выполняются браузером в том порядке, в котором они написаны.

Блоки JavaScript - это способ сгруппировать инструкции вместе. Блок начинается открывающейся фигурной скобкой "{" и заканчивается закрывающейся "}".

Обычно блоки используются для выполнения нескольких команд в функции или внутри условной конструкции.

3 Комментарии

Говорить о том насколько полезны комментарии не будем, скажем только что это очень важная часть кода в работе большой команды. Комментарии в JS - это строчки, отмеченные особым образом и не рассматриваемые браузером как выполняемый код. Таким образом можно размещать как текстовые комментарии, так и помечать отдельные куски кода для предотвращения их выполнения в процессе отладки.

Комментарии бывают однострочные и многострочные.

Однострочные комментарии начинаются с двух слэшей "//" и предотвращают выполнение кода на этой строчке.

Вот пример использования однострочного комментария:

```
<script>
    // Это текстовый комментарий
    alert("Hello World");   // Этот код выполнится.
    // alert("Hello World"); А этот нет
    // Он закомментирован
</script>
```

Многострочные комментарии начинаются с одного слэша и звездочки "/*" и заканчиваются обратной комбинацией - звездочкой и слэшем "*/". Все, что находится

между этими знаками, браузер будет считать комментарием. Выглядит это следующим образом:

```
<script>
    /* Тут начинается комментарий
     alert("Hello World"); Тут код не выполнится.
     alert("Hello World"); И тут тоже.
     Здесь комментарий заканчивается */

    alert("Hi World"); // А этот код выполнится.
</script>
```

Многострочные комментарии нельзя вкладывать друг в друга: первый же встреченный конец комментария "*/" заканчивает блок многострочного комментария.

4 Переменные в JavaScript

Переменная в JavaScript. Например, есть стандартное выражение из алгебры «x=1, y=2...». В частности, буква (например, "x") могла использоваться для хранения некоего переменного значения (например, "1"). Эти буквы называются переменными, и переменные могут использоваться для хранения значений.

Помещение значения в переменную называется "присвоением" значения, для этого используется операция "=". (Ее не следует путать с операцией сравнения, для этого в JavaScript используется операция "==".) В процессе выполнения этой операции значение выражения, расположенного справа от знака "=", присваивается переменной, расположенной слева от знака "=".

Например, выражение "x = 1" присваивает переменной "x" значение "1", выражение "x=y" присваивает переменной "x" значение из переменной "y", а выражение "x = x+1" присваивает переменной "x" ее же значение, увеличенное на единицу.

Переменные в JavaScript, так же, как и в алгебре, используются для хранения значений или выражений. Переменная может иметь имя, например, "x" или более информативное имя, например, "myPetName".

Для именования переменных JavaScript существует набор правил:

- Имена переменных чувствительны к регистру (y и Y это две разных переменных),
- Имена переменных должны начинаться с буквы, символа "$" или символа "_",
- Имя переменной может состоять из любых цифр и букв латинского алфавита, а также символов "$" и "_",
- В качестве имени переменной нельзя использовать зарезервированные и ключевые слова.

Ключевые слова JavaScript:

`break, delete, function, return, typeof, case, do, if, switch, var, catch, else, in, this, void, continue, false, instanceof, throw, while, debugger, finally, new, true, with, default, for, null, try`

Также в JavaScript есть зарезервированные слова, не являющиеся частью языка, но которые могут войти в него в будущем (мы рассматриваем стандарт ECMA-262): `class, const, enum, export, extends, import, super`

Также не рекомендуется, а в некоторых случаях и не разрешается, использовать в качестве идентификаторов следующие слова: `implements, let, private, public, yield, interface, package, protected, static`

Значение переменной может изменяться во время выполнения скрипта, также вы можете обращаться к ней по имени для выполнения различных действий с ее содержимым.

Переменную в JavaScript перед использованием необходимо создать, или, как это еще называют, объявить.

Объявление переменной делается с помощью ключевых слов **`var`** или **`let`**. Объявление следующим образом:

```
var x;
var studentName;

let x;
let studentName;
```

В конце строки должна стоять точка с запятой, и это касается не только объявления переменной, а и любой операции в JavaScript. Это не является строгим требованием языка, но отсутствие закрывающей точки с запятой иногда может приводить к непредсказуемым последствиям.

Присвоить значение переменной или инициализировать ее можно на этапе объявления:

```
var x = 1;
let studentName = "Vasiliy";
```

Инициализация уже объявленной переменной делается без директивы `var`:

```
x = 1;
studentName = "Vasiliy";
```

Объявление с помощью ключевого слова `let` имеет несколько важных отличий от объявления с помощью ключевого слова `var`.

1. Область видимости

В отличие от переменной, объявленной через ключевое слово `var` и видимой внутри всей функции, в которой произошло объявление, переменная, объявленная словом `let`, видна только внутри блока { ... }, в котором она объявлена.

2. Видимость по времени

Переменная, объявленная ключевым словом `let`, видна только после объявления, а переменная, объявленная ключевым словом `var`, может быть доступна в коде и до того места, в котором она объявлена, конечно, в рамках общих правил видимости переменных.

3. При использовании переменной **в качестве счетчика цикла**, объявленная через слово `var` переменная живет в течение всего выполнения цикла и доступна даже после его завершения. Если же использовать объявление через `let`, каждой итерации цикла будет соответствовать своя независимая переменная.

Переменные в JavaScript могут быть Локальными и Глобальными.

1. Локальные переменные - это переменные, объявленные внутри функции JavaScript. Они доступны только в пределах той функции, внутри которой они объявлены. При выходе из этой функции переменные уничтожаются.

Можно объявлять внутри разных функций переменные с одинаковым именем - они никак не будут пересекаться, поскольку используются только внутри функции, в которой они созданы.

2. Глобальные переменные объявляются вне функций и к ним могут обращаться все функции и скрипты на странице. Уничтожаются такие переменные при закрытии страницы.

Если переменную объявить без использования ключевого слова `var`, то она автоматически объявляется глобальной, даже если объявление произведено внутри функции.

Например, выражения `x = 5;` или `surName = "Ivanov";` объявят переменные `x` и `surName` как глобальные, если их еще не существует.

5 Типы данных в JavaScript

Типы данных в JavaScript могут быть следующих видов:

- Числа (`number`)
- Строки (`string`)
- Логические (`boolean`)
- Неопределенные (`undefined`)
- Объект (`object`)
- Пустой (`null`)

Для того, чтобы определить тип данных, записанных в переменной, можно воспользоваться оператором `typeof`.

Например, вот такой скрипт:

```
var myName = "Ivan";
alert (typeof myName);
```

выдаст сообщение "string";

Тип данных `number` образуется, если переменной, в качестве значения, присваивается любое число.

Например, команда `x = 5;` создаст переменную типа `number`.

А если в этом примере цифру 5 заключить в кавычки, `var x = "5";` то тип переменной будет `string`.

Тип переменной `String` получается если значение, присвоенное переменной, заключить в одинарные или двойные кавычки.

Логический тип данных – `boolean`, это всего лишь два варианта значения переменной - `true` (правда или логическая 1) и `false` (ложь или логический 0). Этот тип данных используется при применении операторов сравнения, логических операций и операторов ветвления.

Тип данных `undefined` переменная имеет в тот момент, когда она объявлена, но еще не инициализирована, то есть ее создали, а значение еще не присвоили.

5.1 Строгий режим — "use strict"

На протяжении долгого времени JavaScript развивался без проблем с обратной совместимостью. Новые функции добавлялись в язык, в то время как старая функциональность не менялась. Преимуществом данного подхода было то, что существующий код продолжал работать. А недостатком – что любая ошибка или несовершенное решение, принятое создателями JavaScript, застревали в языке навсегда. В 2009 году появился ECMAScript 5 (ES5). Он добавил новые возможности в язык и изменил некоторые из существующих. Чтобы устаревший код работал, как и раньше, по умолчанию подобные изменения не применяются. Поэтому нам нужно явно их активировать с помощью специальной директивы: "use strict".

Директива выглядит как строка: "use strict" или 'use strict'. Когда она находится в начале скрипта, весь сценарий работает в «современном» режиме.

Например:

```
"use strict";
// этот код работает в современном режиме
...
```

Вместо всего скрипта "use strict" можно поставить в начале большинства видов функций. Это позволяет включить строгий режим только в конкретной функции. Но обычно люди используют его для всего файла.

Убедитесь, что "use strict" находится в начале.

Проверьте, что "use strict" находится в первой исполняемой строке скрипта, иначе строгий режим может не включиться.

Здесь строгий режим не включён:

```
alert("some code");
// "use strict" ниже игнорируется - он должен быть в первой строке

"use strict";

// строгий режим не активирован
```

Над "use strict" могут быть записаны только комментарии.

Нет никакого способа отменить use strict. Нет директивы типа "no use strict", которая возвращала бы движок к старому поведению. Как только мы входим в строгий режим, отменить это невозможно.

Всегда ли нужно использовать `"use strict"`?

Современный JavaScript поддерживает «классы» и «модули» — продвинутые структуры языка, которые автоматически включают строгий режим. Поэтому в них нет нужды добавлять директиву `"use strict"`. Однако, на первое время очень желательно добавлять `"use strict"` в начале ваших скриптов. Позже, когда весь ваш код будет состоять из классов и модулей, директиву можно будет не включать.

5.2 Взаимодействие: `alert`, `prompt`, `confirm`

`alert`

Функция показывает сообщение и ждёт, пока пользователь нажмёт кнопку «OK». Например:

```
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 в синтаксисе [...] - Квадратные скобки вокруг `default` в описанном выше синтаксисе означают, что параметр факультативный, необязательный.

Пользователь может напечатать что-либо в поле ввода и нажать OK. Введённый 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 отменён пользователем. Например:

```
let age = prompt('Сколько тебе лет?', 100);
alert(`Тебе ${age} лет!`); // Тебе 100 лет!
```

`confirm`

Синтаксис:

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

Функция `confirm` отображает модальное окно с текстом вопроса `question` и двумя кнопками: OK и Отмена. Результат – `true`, если нажата кнопка OK. В других случаях – `false`. Например:

```
let isBoss = confirm("Ты здесь главный?");
alert( isBoss ); // true, если нажата OK
```

Все эти методы 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:

- Расположение окон определяется браузером. Обычно окна находятся в центре.
- Визуальное отображение окон зависит от браузера, и мы не можем изменить их вид.

5.3 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. Есть также случаи, когда нам нужно явно преобразовать значение в ожидаемый тип.

5.3.1 Строковое преобразование

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. Например, `String(value)` преобразует значение к строке. Также мы можем 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.

5.3.2 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
```

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

```
let str = "123";
alert(typeof str); // string
let num = Number(str); // становится числом 123
alert(typeof num); // number
```

Явное преобразование часто применяется, когда мы ожидаем 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 (Таблица 1):

Таблица 1 - Правила численного преобразования

`undefined`	NaN
`null`	0
`true / false`	1/0
`string`	Пробельные символы (пробелы, знаки табуляции \t, знаки новой строки \n и т. п.) по краям обрезаются. Далее, если остаётся пустая строка, то получаем 0, иначе из непустой строки «считывается» число. При ошибке результат NaN.

Примеры:

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

Учтите, что `null` и `undefined` ведут себя по-разному. Так, `null` становится нулём, тогда как `undefined` приводится к NaN.

5.3.3 Логическое преобразование

Логическое преобразование самое простое.

Правило преобразования:

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

Например:

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

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

Заметим, что строчка с нулём "0" — это true.

Некоторые языки (к примеру, PHP) воспринимают строку "0" как `false`. Но в JavaScript, если строка не пустая, то она всегда `true`.

```
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // пробел это тоже true
```

6 Операторы сравнения

Многие операторы сравнения известны нам из математики. В JavaScript они записываются так:

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

– Не равно. В математике обозначается символом ≠, но в JavaScript записывается как `a != b`.

Результат сравнения имеет логический тип. Все операторы сравнения возвращают значение логического типа:

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

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

Например:

```
alert( 2 > 1 );  // true (верно)
alert( 2 == 1 ); // false (неверно)
alert( 2 != 1 ); // true (верно)
```

Результат сравнения можно присвоить переменной, как и любое значение:

```
let result = 5 > 4; // результат сравнения присваивается переменной result
alert( result ); // true
```

6.1 Сравнение строк

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок. Другими словами, строки сравниваются посимвольно. Например:

```
alert( 'Я' > 'А' ); // true
alert( 'Коты' > 'Кода' ); // true
alert( 'Сонный' > 'Сон' ); // true
```

Алгоритм сравнения двух строк довольно прост:

– Сначала сравниваются первые символы строк.

– Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй. Сравнение завершено.

– Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.

– Сравнение продолжается, пока не закончится одна из строк.

– Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

Примечание: используется кодировка Unicode, а не настоящий алфавит.

В JavaScript имеет значение регистр символов. Заглавная буква "A" не равна строчной "a". Какая же из них больше? Строчная "a". Почему? Потому что строчные буквы имеют больший код во внутренней таблице кодирования, которую использует JavaScript (Unicode).

6.2 Сравнение разных типов

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

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

Логическое значение `true` становится 1, а `false` – 0.

Например:

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

Возможна следующая ситуация:

Два значения равны. Одно из них `true` как логическое значение, другое – `false`.

Например:

```
let a = 0;
alert( Boolean(a) ); // false
let b = "0";
alert( Boolean(b) ); // true

alert(a == b); // true!
```

С точки зрения JavaScript, результат ожидаем. Равенство преобразует значения, используя числовое преобразование, поэтому "0" становится 0. В то время как явное преобразование с помощью `Boolean` использует другой набор правил.

6.3 Строгое сравнение

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

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

Та же проблема с пустой строкой:

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

Это происходит из-за того, что операнды разных типов преобразуются оператором == к числу. В итоге, и пустая строка, и false становятся нулём. Оператор строгого равенства === проверяет равенство без приведения типов. Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования. Оператор строгого равенства дольше писать, но он делает код более очевидным и оставляет меньше места для ошибок.

Сравнение с null и undefined. Поведение null и undefined при сравнении с другими значениями — особое:

При строгом равенстве === эти значения различны, так как различны их типы.

```
alert( null === undefined ); // false
```

При нестрогом равенстве == эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

```
alert( null == undefined ); // true
```

При использовании математических операторов и других операторов сравнения < > <= >= Значения null/undefined преобразуются к числам: null становится 0, а undefined – NaN.

Относитесь очень осторожно к любому сравнению с undefined/null, кроме случаев строгого равенства ===. Не используйте сравнения >= > < <= с переменными, которые могут принимать значения null/undefined, разве что вы полностью уверены в том, что делаете. Если переменная может принимать эти значения, то добавьте для них отдельные проверки.

7 Условное ветвление: if, '?'

Иногда нам нужно выполнить различные действия в зависимости от условий. Для этого мы можем использовать инструкцию if и условный оператор '?'.

7.1 Инструкция `if`

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

```
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 мы хотим 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 вы используете инструкцию `if`, даже если выполняется только одна команда. Это улучшает читаемость кода.

7.2 Блок `else`

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

```
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
}
```

7.3 Несколько условий: `else if`

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок `else if`. Например:

```
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 приведённом 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.

7.4 Условный оператор '?'

Иногда нам нужно определить переменную в зависимости от условия. Например:

```
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 нам сделать это более коротким и простым способом.

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

Синтаксис:

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

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

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

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

Этот пример будет делать то же самое, что и предыдущий:

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

Но скобки делают код более простым для восприятия.

При чтении глаза сканируют код по вертикали. Блоки кода, занимающие несколько строк, воспринимаются гораздо легче, чем длинный горизонтальный набор инструкций. **Смысл оператора «вопросительный знак» (?) – вернуть то или иное значение, в зависимости от условия.** Используйте его именно для этого. Когда вам нужно выполнить разные ветви кода – используйте `if`.

8 Логические операторы

В JavaScript есть четыре логических оператора: `||` (ИЛИ), `&&` (И) и `!` (НЕ), `??` (Оператор нулевого слияния). Несмотря на своё название, данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.

`||` (ИЛИ)

Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:

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

Традиционно в программировании ИЛИ предназначено только для манипулирования булевыми значениями: в случае, если какой-либо из аргументов `true`, он вернёт `true`, в противоположной ситуации возвращается `false`. В JavaScript, как мы увидим далее, этот оператор работает несколько иным образом. Но давайте сперва посмотрим, что происходит с булевыми значениями. Существует всего четыре возможные логические комбинации:

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

Как мы можем наблюдать, результат операций всегда равен `true`, за исключением случая, когда оба аргумента `false`. Если значение не логического типа, то оно к нему приводится в целях вычислений.

Например, число 1 будет воспринято как `true`, а 0 – как `false`:

```
if (1 || 0) { // работает как if( true || false )
  alert( 'truthy!' );
}
```

Обычно оператор `||` используется в `if` для проверки истинности любого из заданных условий. К примеру:

```
let hour = 9;

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

Можно передать и больше условий:

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

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

ИЛИ "||" находит первое истинное значение

Описанная выше логика соответствует традиционной. Теперь давайте поработаем с «дополнительными» возможностями JavaScript. Расширенный алгоритм работает следующим образом. При выполнении ИЛИ || с несколькими значениями:

```
result = value1 || value2 || value3;
```

Оператор || выполняет следующие действия:

- Вычисляет операнды слева направо.
- Каждый операнд конвертирует в логическое значение. Если результат `true`, останавливается и возвращает исходное значение этого операнда.
- Если все операнды являются ложными (`false`), возвращает последний из них.

Значение возвращается в исходном виде, без преобразования.

Другими словами, цепочка ИЛИ || возвращает первое истинное значение или последнее, если такое значение не найдено. Например:

```
alert( 1 || 0 ); // 1
alert( true || 'no matter what' ); // true

alert( null || 1 ); // 1 (первое истинное значение)
alert( null || 0 || 1 ); // 1 (первое истинное значение)
alert( undefined || null || 0 ); // 0 (поскольку все ложно, возвращается последнее значение)
```

Это делает возможным более интересное применение оператора по сравнению с «чистым, традиционным, только булевым ИЛИ». Получение первого истинного значения из списка переменных или выражений.

8.1 Сокращённое вычисление

Операндами могут быть как отдельные значения, так и произвольные выражения. ИЛИ `||` вычисляет их слева направо. Вычисление останавливается при достижении первого истинного значения. Этот процесс называется «сокращённым вычислением», поскольку второй операнд вычисляется только в том случае, если первого недостаточно для вычисления всего выражения. Это хорошо заметно, когда выражение, указанное в качестве второго аргумента, имеет побочный эффект, например, изменение переменной. В приведённом ниже примере x не изменяется:

```
let x;

true || (x = 1);

alert(x); // undefined, потому что (x = 1) не вычисляется
```

Если бы первый аргумент имел значение `false`, то `||` приступил бы к вычислению второго и выполнил операцию присваивания:

```
let x;

false || (x = 1);

alert(x); // 1
```

Присваивание – лишь один пример. Конечно, могут быть и другие побочные эффекты, которые не проявятся, если вычисление до них не дойдёт. Как мы видим, этот вариант использования `||` является "аналогом `if`". Первый операнд преобразуется в логический. Если он оказывается ложным, начинается вычисление второго. В большинстве случаев лучше использовать «обычный» `if`, чтобы облегчить понимание кода, но иногда это может быть удобно.

`&&` (И)

Оператор `И` пишется как два амперсанда `&&`:

```
result = a && b;
```

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

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

Пример с `if`:

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

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

Как и в случае с ИЛИ, любое значение допускается в качестве операнда И:

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

И «&&» находит первое ложное значение. При нескольких подряд операторах И:

```
result = value1 && value2 && value3;
```

Оператор `&&` выполняет следующие действия:

- Вычисляет операнды слева направо.
- Каждый операнд преобразует в логическое значение. Если результат `false`, останавливается и возвращает исходное значение этого операнда.
- Если все операнды были истинными, возвращается последний.

Другими словами, И возвращает первое ложное значение. Или последнее, если ничего не найдено. Вышеуказанные правила схожи с поведением ИЛИ. Разница в том, что И возвращает первое ложное значение, а ИЛИ– первое истинное.

Приоритет оператора И `&&` больше, чем ИЛИ `||`, так что он выполняется раньше. Таким образом, код `a && b || c && d` по существу такой же, как если бы выражения `&&` были в круглых скобках: `(a && b) || (c && d)`. Как и оператор ИЛИ `||`, И `&&` иногда может заменять `if`.

8.2 Оператор нулевого слияния (??)

Оператор нулевого слияния представляет собой два вопросительных знака '??'. Так как он обрабатывает `null` и `undefined` одинаковым образом, то мы введём специальный

термин. Для краткости будем говорить, что значение «определено», если оно не равняется ни `null`, ни `undefined`.

Результат выражения `a ?? b` будет следующим:

- если a определено, то `a`,
- если a не определено, то `b`.

Иначе говоря, оператор `??` возвращает первый аргумент, если он не `null`/`undefined`, иначе второй. Оператор нулевого слияния не является чем-то принципиально новым. Это всего лишь удобный синтаксис, как из двух значений получить одно, которое «определено». Вот как можно переписать выражение `result = a ?? b`, используя уже знакомые нам операторы:

```
result = (a !== null && a !== undefined) ? a : b;
```

Как правило, оператор `??` нужен для того, чтобы задать значение по умолчанию для потенциально неопределённой переменной. Например, здесь мы отобразим `user`, если её значение не `null`/`undefined`, в противном случае Аноним:

```
let user;

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

А вот пример, когда `user` присвоено значение:

```
let user = "Иван";

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

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 не существовать, если пользователь решил не вводить соответствующие значение. Мы хотели бы отобразить имя пользователя, используя одну из этих переменных, или показать «Аноним», если все они `null`/`undefined`. Для этого воспользуемся оператором `??`:

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

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

8.3 Сравнение ?? с ||

Оператор ИЛИ || можно использовать для того же, что и ??. Например, если в приведённом выше коде заменить ?? на ||, то будет тот же самый результат:

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

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

Исторически сложилось так, что оператор ИЛИ || появился первым. Он существует с самого начала в JavaScript, поэтому разработчики долгое время использовали его для таких целей. С другой стороны, сравнительно недавно в язык был добавлен оператор нулевого слияния ?? – как раз потому, что многие были недовольны оператором ||.

Важное различие между ними заключается в том, что:

- || возвращает первое истинное значение.
- ?? возвращает первое определённое значение.

Проще говоря, оператор || не различает false, 0, пустую строку "" и null/undefined. Для него они все одинаковы, т.е. являются ложными значениями. Если первым аргументом для оператора || будет любое из перечисленных значений, то в качестве результата мы получим второй аргумент. Однако на практике часто требуется использовать значение по умолчанию только тогда, когда переменная является null/undefined. Ведь именно тогда значение действительно неизвестно/не определено.

Рассмотрим следующий пример:

```
let height = 0;
alert(height || 100); // 100
alert(height ?? 100); // 0
```

- height || 100 проверяет height на ложное значение, оно равно 0, да, ложное, поэтому результатом || является второй аргумент, т.е. 100.
- height ?? 100 проверяет, что переменная height содержит null/undefined, а поскольку это не так, то результатом является сама переменная height, т.е. 0.

На практике нулевая высота часто является вполне нормальным значением, которое не следует заменять значением по умолчанию. Таким образом, ?? здесь как раз работает так, как нужно.

Приоритет оператора `??` такой же, как и у `||`. Это означает, что, как и `||`, оператор нулевого слияния `??` вычисляется до `=` и `?`, но после большинства других операций, таких как `+`, `*`. Так что, в выражениях такого вида понадобятся скобки:

```
let height = null;
let width = null;

// важно: используйте круглые скобки
let area = (height ?? 100) * (width ?? 50);

alert(area); // 5000
```

Иначе, если опустить скобки, оператор `*` выполнится первым, так как у него приоритет выше, чем у `??`, и это приведёт к неправильным результатам.

```
// без скобок
let area = height ?? 100 * width ?? 50;
```

```
// ...сработает вот так (совсем не как нам нужно):
let area = height ?? (100 * width) ?? 50;
```

8.4 Использование `??` вместе с `&&` или `||`

По соображениям безопасности JavaScript запрещает использование оператора `??` вместе с `&&` и `||`, если приоритет явно не указан при помощи круглых скобок. Это, безусловно спорное, ограничение было добавлено в спецификацию языка с целью избежать программные ошибки, когда люди начнут переходить с `||` на `??`. Используйте скобки, чтобы обойти это ограничение:

```
let x = (1 && 2) ?? 3; // Работает без ошибок

alert(x); // 2
```

В итоге, оператор нулевого слияния `??` — это быстрый способ выбрать первое «определённое» значение из списка.

9 Циклы `while` и `for`

При написании скриптов зачастую встаёт задача сделать однотипное действие много раз. Например, вывести товары из списка один за другим. Или просто перебрать все числа от 1 до 10 и для каждого выполнить одинаковый код.

Для многократного повторения одного участка кода предусмотрены циклы `for…of` и `for…in`.

9.1 Цикл `while`

Цикл `while` имеет следующий синтаксис:

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

Код из тела цикла выполняется, пока условие `condition` истинно. Например, цикл ниже выводит `i`, пока `i < 3`:

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

Одно выполнение тела цикла называется итерацией. Цикл в примере выше совершает три итерации. Если бы строка `i++` отсутствовала в примере выше, то цикл бы повторялся (в теории) вечно. На практике, конечно, браузер не позволит такому случиться, он предоставит пользователю возможность остановить «подвисший» скрипт, а JavaScript на стороне сервера придётся «убить» процесс.

Любое выражение или переменная может быть условием цикла, а не только сравнение: условие `while` вычисляется и преобразуется в логическое значение. Например, `while (i)` – более краткий вариант `while (i != 0)`:

```
let i = 3;
while (i) { // когда i будет равно 0, условие станет ложным, и цикл остановится
  alert( i );
  i--;
}
```

Фигурные скобки не требуются для тела цикла из одной строки. Если тело цикла состоит лишь из одной инструкции, мы можем опустить фигурные скобки `{…}`:

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

9.2 Цикл do…while

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

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

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

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

Такая форма синтаксиса оправдана, если вы хотите, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным. На практике чаще используется форма с предусловием: `while(…) {…}`.

9.3 Цикл for

Более сложный, но при этом самый распространённый цикл — цикл `for`. Выглядит он так:

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

Давайте разберёмся, что означает каждая часть, на примере:

Цикл ниже выполняет `alert(i)` для `i` от `0` до (но не включая) `3`:

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

Рассмотрим конструкцию `for` подробнее (Таблица 2):

Таблица 2 – Описание цикла `for`

начало	`let i = 0`	Выполняется один раз при входе в цикл.
условие	`i < 3`	Проверяется перед каждой итерацией цикла. Если оно вычислится в `false`, цикл остановится.

тело	`alert(i)`	Выполняется снова и снова, пока условие вычисляется в `true`.
шаг	`i++`	Выполняется после тела цикла на каждой итерации перед проверкой условия.

В целом, алгоритм работы цикла выглядит следующим образом:

1. Выполнить начало
2. → (Если условие == true → Выполнить тело, Выполнить шаг)
3. → (Если условие == true → Выполнить тело, Выполнить шаг)
4. → (Если условие == true → Выполнить тело, Выполнить шаг)
5. → ...

То есть, начало выполняется один раз, а затем каждая итерация заключается в проверке условия, после которой выполняется тело и шаг. Вот в точности то, что происходит в нашем случае:

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

// Выполнить начало
let i = 0;
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// ...конец, потому что теперь i == 3
```

9.4 Встроенное объявление переменной

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

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

Вместо объявления новой переменной мы можем использовать уже существующую:

```
let i = 0;

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

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

9.5 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
}
```

Можно убрать и шаг:

```
let i = 0;

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

Это сделает цикл аналогичным `while (i < 3)`. А можно и вообще убрать всё, получив бесконечный цикл:

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

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

9.6 Прерывание цикла: `break`

Обычно цикл завершается при вычислении условия в `false`. Но мы можем выйти из цикла в любой момент с помощью специальной директивы `break`. Например,

следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

```
let sum = 0;

while (true) {
  let value = +prompt("Введите число", '');
  if (!value) break; // (*)
  sum += value;
}
alert( 'Сумма: ' + sum );
```

Директива `break` в строке `(*)` полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на `alert`. Вообще, сочетание «бесконечный цикл + `break`» – отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале или конце цикла, а посередине или даже в нескольких местах его тела.

9.7 Переход к следующей итерации: `continue`

Директива `continue` – «облегчённая версия» `break`. При её выполнении цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно `true`). Её используют, если понятно, что на текущем повторе цикла делать больше нечего. Например, цикл ниже использует `continue`, чтобы выводить только нечётные значения:

```
for (let i = 0; i < 10; i++) {
  // если true, пропустить оставшуюся часть тела цикла
  if (i % 2 == 0) continue;
  alert(i); // 1, затем 3, 5, 7, 9
}
```

Для чётных значений `i`, директива `continue` прекращает выполнение тела цикла и передаёт управление на следующую итерацию `for` (со следующим числом). Таким образом `alert` вызывается только для нечётных значений. Директива `continue` позволяет избегать вложенности.

9.8 Нельзя использовать `break`/`continue` справа от оператора '?'

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

Обе этих директивы поддерживают метки, которые ставятся перед циклом. Метки – единственный способ для `break`/`continue` выйти за пределы текущего цикла, повлиять на выполнение внешнего. Заметим, что метки не позволяют прыгнуть в произвольное место кода, в JavaScript нет такой возможности.

10 Конструкция `switch`

Конструкция `switch` заменяет собой сразу несколько `if`. Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Синтаксис:

Конструкция `switch` имеет один или более блок `case` и необязательный блок `default`. Выглядит она так:

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

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

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

Переменная x проверяется на строгое равенство первому значению `value1`, затем второму `value2` и так далее. Если соответствие установлено – `switch` начинает выполняться от соответствующей директивы `case` и далее, до ближайшего `break` (или до конца `switch`). Если ни один `case` не совпал – выполняется (если есть) вариант `default`.

Пример использования `switch` (сработавший код выделен):

```
let a = 2 + 2;

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

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

В примере выше последовательно выполнятся три `alert`. Любое выражение может быть аргументом для `switch`/`case`. И `switch` и `case` допускают любое выражение в качестве аргумента. Например:

```
let a = "1";
let b = 0;

switch (+a) {
  case b + 1:
    alert("Выполнится, т.к. значением +a будет 1, что в точности равно b+1");
    break;

  default:
    alert("Это не выполнится");
}
```

В этом примере выражение `+a` вычисляется в `1`, что совпадает с выражением `b + 1` в `case`, и, следовательно, код в этом блоке будет выполнен.

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 = 3;

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.

Возможность группировать `case` – это побочный эффект того, как `switch`/`case` работает без `break`. Здесь выполнение `case 3` начинается со строки `(*)` и продолжается в `case 5`, потому что отсутствует `break`.

Нужно отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство. Для примера, давайте рассмотрим следующий код:

```
let arg = prompt("Введите число?");
switch (arg) {
  case '0':
  case '1':
    alert( 'Один или ноль' );
    break;

  case '2':
    alert( 'Два' );
    break;

  case 3:
    alert( 'Никогда не выполнится!' );
    break;
  default:
    alert( 'Неизвестное значение' );
}
```

- Для `'0'` и `'1'` выполнится первый `alert`.
- Для `'2'` – второй `alert`.
- Но для `3`, результат выполнения `prompt` будет строка "3", которая не соответствует строгому равенству `===` с числом `3`. Таким образом, мы имеем «мёртвый код» в `case 3`! Выполнится вариант `default`.

11 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, ещё где-нибудь. Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы. Примеры встроенных функций вы уже видели – это `alert(message)`, `prompt(message, default)` и `confirm(question)`. Но можно создавать и свои.

11. 1 Объявление функции

Для создания функций мы можем использовать объявление функции. Пример объявления функции:

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

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

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

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

```
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.

11.2 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
```

11.3 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 одноимённая 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
```

11.4 Глобальные переменные

Переменные, объявленные снаружи всех функций, такие как внешняя переменная userName в вышеприведённом коде – называются глобальными. Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные). Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных.

11.5 Параметры

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

В нижеприведённом 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.

Вот ещё один пример: есть переменная `from`, и мы передаём её функции. Обратите внимание: функция изменяет значение `from`, но это изменение не видно снаружи. Функция всегда получает только копию значения:

```
function showMessage(from, text) {

  from = '*' + from + '*'; // немного украсим "from"

  alert( from + ': ' + text );
}

let from = "Аня";

showMessage(from, "Привет"); // *Аня*: Привет

// значение "from" осталось прежним, функция изменила значение локальной переменной
alert( from ); // Аня
```

Значение, передаваемое в качестве параметра функции, также называется аргументом. Иными словами, **параметр** – это переменная, указанная в круглых скобках в

объявлении функции. **Аргумент** – это значение, которое передаётся функции при её вызове. Мы объявляем функции со списком параметров, затем вызываем их, передавая аргументы. В приведённом выше примере можно было бы сказать: «функция `showMessage` объявляется с двумя параметрами, затем вызывается с двумя аргументами: `from` и "Привет"».

11.5.1 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 приведёт к ошибке. Такой вызов выведет «`*Аня*: undefined`». В вызове не указан параметр `text`, поэтому предполагается, что `text === undefined`. Если мы хотим задать параметру `text` значение по умолчанию, мы должны 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 её 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
}
```

11.5.2 Вычисление параметров по умолчанию

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра. В приведённом выше примере, функция `anotherFunction()` не будет вызвана вообще, если указан параметр `text`. С другой стороны, функция будет независимо вызываться каждый раз, когда `text` отсутствует.

11.5.3 Альтернативные параметры по умолчанию

Иногда имеет смысл присваивать значения по умолчанию для параметров не в объявлении функции, а на более позднем этапе. Во время выполнения функции мы можем проверить, передан ли параметр, сравнив его с `undefined`:

```
function showMessage(text) {
  // ...
  if (text === undefined) { // если параметр отсутствует
    text = 'пустое сообщение';
  }
  alert(text);
}
showMessage(); // пустое сообщение
```

Или можно использовать оператор `||`:

```
function showMessage(text) {
  // если значение text ложно или равняется undefined, тогда присвоить text значение 'пусто'
  text = text || 'пусто';
  ...
}
```

Современные движки JavaScript поддерживают оператор нулевого слияния `??`. Его использование будет лучшей практикой, в случае, если большинство ложных значений, таких как `0`, следует расценивать как «нормальные».

```
function showCount(count) {
  // если count равен undefined или null, показать "неизвестно"
  alert(count ?? "неизвестно");
}
showCount(0); // 0
showCount(null); // неизвестно
showCount(); // неизвестно
```

11.6 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). 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 приведёт к немедленному выходу из функции. Например:

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

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

В коде выше, если `checkAge(age)` вернёт `false`, `showMovie` не выполнит `alert`. Результат функции с пустым `return` или без него – `undefined`. Если функция не возвращает значения, это всё равно, как если бы она возвращала `undefined`:

```
function doNothing() { /* пусто */ }

alert( doNothing() === undefined ); // true
```

Пустой return аналогичен `return undefined`:

```
function doNothing() {
  return;
}
alert( doNothing() === undefined ); // true
```

Никогда не добавляйте перевод строки между `return` и его значением. Для длинного выражения в `return` может быть заманчиво разместить его на нескольких отдельных строках. Если мы хотим, чтобы возвращаемое выражение занимало несколько строк, нужно начать его на той же строке, что и `return`. Или, хотя бы, поставить там открывающую скобку, тогда всё сработает, как задумано.

11.7 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 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.

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.

11.7.1 Одна функция – одно действие

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием. Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе (в этом случае мы можем создать третью функцию, которая будет их вызывать).

Несколько примеров, которые нарушают это правило:

- `getAge` – будет плохим выбором, если функция будет выводить `alert` с возрастом (должна только возвращать его).
- `createForm` – будет плохим выбором, если функция будет изменять документ, добавляя форму в него (должна только создавать форму и возвращать её).
- `checkPermission` – будет плохим выбором, если функция будет отображать сообщение с текстом доступ разрешён/запрещён (должна только выполнять проверку и возвращать её результат).

В этих примерах использовались общепринятые смыслы префиксов. Вы должны чётко понимать, что значит префикс, что функция с ним может делать, а чего не может.

11.7.2 Функции == Комментарии

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Иногда следовать этому правилу непросто, но это определённо хорошее правило.

Небольшие функции не только облегчают тестирование и отладку – само существование таких функций выполняет роль хороших комментариев! Таким образом, допустимо создавать функции, даже если мы не планируем повторно использовать их. Такие функции структурируют код и делают его более понятным.

12 Консоль разработчика

Код уязвим для ошибок. Но по умолчанию в браузере ошибки не видны. То есть, если что-то пойдёт не так, мы не увидим, что именно сломалось, и не сможем это починить. Для решения задач такого рода в браузер встроены так называемые «Инструменты разработки» (Developer tools или сокращённо — devtools).

Для вызова консоли нажмите F12 или, если вы используете Mac, Cmd+Opt+J. По умолчанию в инструментах разработчика откроется вкладка Console (консоль). В консоли мы можем увидеть сообщение об ошибке, отрисованное красным цветом.

12.1 Многострочный ввод

Обычно при нажатии Enter введённая строка кода сразу выполняется. Чтобы перенести строку, нажмите Shift+Enter. Так можно вводить более длинный JS-код. Теперь мы явно видим ошибки, для начала этого вполне достаточно.

12. 2 Отладка в браузере

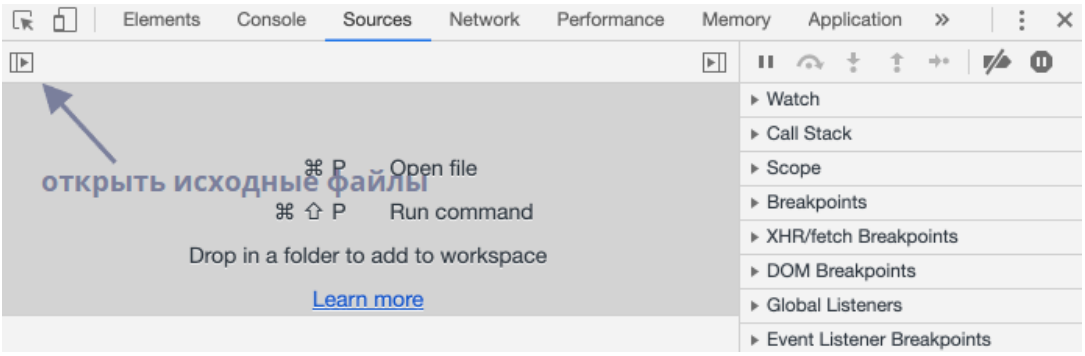
Отладка – это процесс поиска и исправления ошибок в скрипте. Все современные браузеры и большинство других сред разработки поддерживают инструменты для отладки – специальный графический интерфейс, который сильно упрощает отладку. Он также позволяет по шагам отследить, что именно происходит в нашем коде.

12.3 Панель «Исходный код» («Sources»)

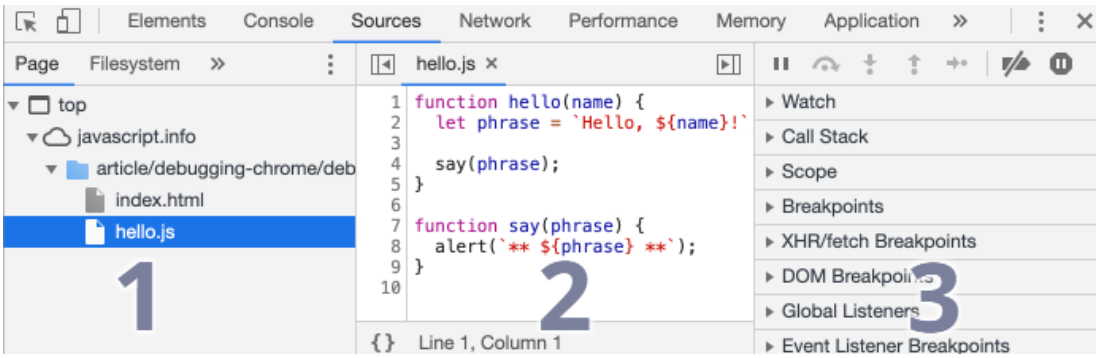
Версия Chrome, установленная у вас, может выглядеть немного иначе, однако принципиальных отличий не будет.

1. Работая в Chrome, откройте свою страницу.
2. Включите инструменты разработчика, нажав F12 (Mac: Cmd+Opt+I).
3. Щёлкните по панели Sources («исходный код»).

При первом запуске получаем следующее:



Кнопка-переключатель откроет вкладку со списком файлов. Выбираем файл скрипта в дереве файлов:



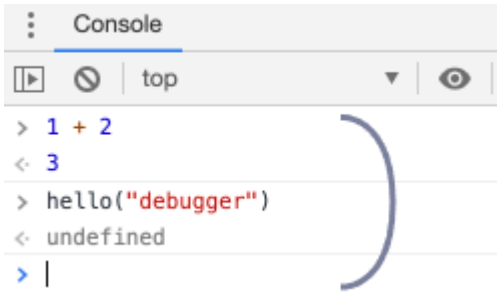
Интерфейс состоит из трёх зон:

1. В зоне File Navigator (панель для навигации файлов) показаны файлы HTML, JavaScript, CSS, включая изображения, используемые на странице. Здесь также могут быть файлы различных расширений Chrome.
2. Зона Code Editor (редактор кода) показывает исходный код.
3. Наконец, зона JavaScript Debugging (панель отладки JavaScript) отведена для отладки, скоро мы к ней вернёмся.

Чтобы скрыть список ресурсов и освободить экранное место для исходного кода, щёлкните по тому же переключателю.

12.4 Консоль

При нажатии на клавишу Esc в нижней части экрана вызывается консоль, где можно вводить команды и выполнять их клавишей Enter. Результат выполнения инструкций сразу же отображается в консоли. Например, результатом 1+2 будет 3, а вызов функции `hello("debugger")` ничего не возвращает, так что результатом будет `undefined`:



12. 5 Точки остановки (breakpoints)

Давайте разберёмся, как работает код нашей тестовой страницы. В файле вашего скрипта щёлкните на номер строки. Таким образом мы поставили точку остановки. Точка остановки – это участок кода, где отладчик автоматически приостановит исполнение JavaScript. Пока исполнение поставлено «на паузу», мы можем просмотреть текущие значения переменных, выполнить команды в консоли, другими словами, выполнить отладку кода. В правой части графического интерфейса мы видим список точек остановки. А когда таких точек выставлено много, да ещё и в разных файлах, этот список поможет эффективно ими управлять.

13 Команда `debugger`

Выполнение кода можно также приостановить с помощью команды `debugger` прямо изнутри самого кода:

```
function hello(name) {
  let phrase = `Привет, ${name}!`;

  debugger;  // <-- тут отладчик остановится

  say(phrase);
}
```

Такая команда сработает только если открыты инструменты разработки, иначе браузер ее проигнорирует. В нашем примере функция `hello()` вызывается во время загрузки страницы, поэтому для начала отладки (после того, как мы поставили точки останова) проще всего её перезагрузить. Нажмите F5 (Windows, Linux) или Cmd+R (Mac). Выполнение прервётся на четвёртой строчке (где находится точка остановки).

Чтобы понять, что происходит в коде, щёлкните по стрелочкам справа:

- Watch – показывает текущие значения для любых выражений. Вы можете нажать на + и ввести выражение. Отладчик покажет его значение, автоматически пересчитывая его в процессе выполнения.
- Call Stack – показывает цепочку вложенных вызовов. В текущий момент отладчик находится внутри вызова hello(), вызываемого скриптом в index.html (там нет функции, поэтому она называется “анонимной”). Если вы нажмёте на элемент стека (например, «anonymous»), отладчик перейдёт к соответствующему коду, и нам представляется возможность его проанализировать.
- Scope показывает текущие переменные.
- Local показывает локальные переменные функций, а их значения подсвечены прямо в исходном коде.
- В Global перечисляются глобальные переменные (то есть вне каких-либо функций). Там также есть ключевое слово `this`, которое мы ещё не изучали, но скоро изучим.

Если правильно выстроить логирование в приложении, то можно и без отладчика разобраться, что происходит в коде.

Практическое задание

1. Создайте страницу входа в панель управления сайтом (или используйте уже созданную форму для входа на сайт).

Используя конструкцию `if…else`, напишите код, который будет спрашивать: «Желаете пройти регистрацию на сайте?». Если пользователь вводит «Да», то показать: «Круто!», в любом другом случае – отобразить: «Попробуй ещё раз».

2. Напишите код, который будет спрашивать логин с помощью `prompt`. Если посетитель вводит «Админ», то `prompt` запрашивает пароль, если ничего не введено или нажата клавиша Esc – показать «Отменено», в противном случае отобразить «Я вас не знаю» (можно использовать свои формулировки).

Пароль нужно проверить следующим образом:

- Если введён пароль «Я главный», то выводить «Здравствуйте!»,
- Иначе – «Неверный пароль»,
- При отмене – «Отменено».

Используйте вложенные блоки `if`. Обращайте внимание на стиль и читаемость кода. Передача пустого ввода в приглашение `prompt` возвращает пустую строку. Нажатие клавиши Esc во время запроса возвращает `null`.

3. Создайте кнопку «Нравится», которая будет менять цвет при нажатии на неё, а при повторном нажатии возвращаться к своему цвету.

4. Дополнительно к этой кнопке (или к новой) добавьте возможность создавать элементы следуя за курсором, при повторном нажатии «рисование» прекращается.