

## Тема 1. Основы JavaScript

1. Введение в JavaScript.
2. Отладка скриптов.
3. Синтаксис JavaScript.
4. Переменные.
5. Типы данных.
6. Преобразование типов.
7. Операторы JavaScript.
8. Операторы сравнения.
9. Модальные окна.
10. Условные операторы.
11. Логические операторы.
12. Циклы while, for.
13. Конструкция switch.
14. Функции. Объявление функции.
15. Функциональные выражения и функции-стрелки.

Содержание данной темы включает материалы, доступные по адресу <https://learn.javascript.ru>.

### 1. Введение в JavaScript

#### История

Язык JavaScript (JS) был представлен в 1995 году как способ добавлять программы на веб-страницы в браузере Netscape Navigator. Изначально он назывался LiveScript. Но в то время широко использовался язык Java, и с целью популярности LiveScript переименовали в JavaScript.

Планировалось, что JavaScript будет упрощенной версией языка Java. Однако, JavaScript очень развился и сейчас это совершенно независимый язык, и к языку Java не имеет никакого отношения. JavaScript имеет свою спецификацию, которая называется ECMAScript.

После того, как язык стал широко использоваться в других браузерах, был составлен документ, описывающий работу языка. Он называется стандарт ECMAScript по имени организации ECMA (European Computer Manufacturers Association – Европейская ассоциация производителей компьютеров). ECMAScript является стандартом, а JavaScript – это самая популярная реализация этого стандарта.

#### Версии ECMAScript

ES – принятое сокращение для ECMAScript. Каждое издание ECMAScript получает аббревиатуру ES с последующим его номером.

- ES1 был выпущен в июне 1997 года,
- ES2 – в июне 1998 года,
- ES3 – в декабре 1999 года,
- ES4 – не была принята,
- ES5 был выпущен в декабре 2009 года,

- ES6/ES2015 вышел в июне 2015 года. С выходом этого стандарта комитет принял решение перейти к ежегодным обновлениям. Поэтому издание было переименовано в ES2015, чтобы отражать год релиза. Последующие версии также называются в соответствии с годом их выпуска.
- ES2016 (ES7) вышла в июне 2016 года,
- ES2017 (ES8) – в июне 2017 года,
- ES2018 – в июне 2018 года,
- ES2019 – в июне 2019 года.

Термин ES.Next ссылается на новую версию ECMAScript. Стоит отметить, что каждая новая версия приносит новые функции для языка.

## **Выполнение JS**

Программы на языке JavaScript называются скриптами. В браузере они подключаются напрямую к HTML и выполняются сразу после загрузки страницы.

JavaScript может выполняться не только в браузере, а где угодно, нужна лишь специальная программа – *интерпретатор*. Процесс выполнения скрипта называют *интерпретацией*.

*Транслятор* (англ. translator - переводчик) представляет собой программу, на основе которой компьютер преобразует вводимые в него программы на машинный язык, поскольку он может выполнять программы, записанные только на языке его процессора, и алгоритмы, заданные на другом языке, должны быть перед их выполнением переведены на машинный язык. Трансляторы реализуются в виде компиляторов или интерпретаторов. С точки зрения выполнения работы компилятор и интерпретатор существенно различаются.

*Компилятор* (англ. compiler - составитель, собиратель) читает всю программу целиком, делает ее перевод и создает законченный вариант программы на машинном языке, который затем и выполняется. Результат работы компилятора — бинарный исполняемый файл.

*Интерпретатор* (англ. interpreter - истолкователь, устный переводчик) переводит и выполняет программу строка за строкой. После того, как программа откомпилирована, ни исходный текст программы, ни компилятор более не нужны для исполнения программы. В то же время программа, обрабатываемая интерпретатором, должна заново переводиться на машинный язык при каждом очередном запуске программы. То есть исходный файл является непосредственно исполняемым.

Откомпилированные программы работают быстрее, но интерпретируемые проще исправлять и изменять.

Современные интерпретаторы перед выполнением преобразуют JavaScript в машинный код или близко к нему (байт-код), оптимизируют и затем выполняют. И даже во время выполнения скрипта оптимизируют его. Поэтому JavaScript работает очень быстро.

Во все основные браузеры встроен интерпретатор («движок») JavaScript, именно поэтому они могут выполнять скрипты на странице. Но JavaScript можно использовать не только в браузере. Это полноценный язык, программы на котором можно запускать на сервере, и даже в других устройствах, если в них установлен соответствующий «движок». Разные движки имеют разные «кодовые имена». Например, V8 – для Chrome и Opera, SpiderMonkey – для Firefox, Trident и Chakra

для разных версий IE, ChakraCore для Microsoft Edge, Nitro и SquirrelFish для Safari и т.д.

## **Возможности JavaScript в браузере**

Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что он 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 с пользователем и веб-сервером:

- Добавлять новый HTML 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»).

Возможности JavaScript в браузере ограничены ради безопасности пользователя. Цель заключается в предотвращении доступа вредоносной веб-страницы к личной информации или нанесению 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 операционной системы.

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 могут взаимодействовать между собой. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае, JavaScript с одной страницы не имеет доступа к другой, если они из разных сайтов (с другого домена, протокола или порта).

Это называется «Политика одинакового источника» (Same Origin Policy). Чтобы обойти это ограничение, обе страницы должны быть на это согласны и содержать JavaScript-код, который специальным образом обменивается данными.

– JavaScript может легко взаимодействовать с сервером, с которого пришла текущая страница. Но его способность получать данные с других сайтов/доменов ограничена. Хотя это возможно, требуется явное соглашение (выраженное в заголовках HTTP) с удалённой стороной.

Подобные ограничения не действуют, если JavaScript используется вне браузера, например — на сервере. Современные браузеры предоставляют плагины/расширения с помощью которых можно запрашивать дополнительные разрешения.

## Языки «над» JavaScript

Существует много языков, которые *транспируются* (конвертируются) в JavaScript прежде, чем запустятся в браузере. Современные инструменты делают транспилицию очень быстрой и прозрачной, фактически позволяя разработчикам писать код на другом языке, автоматически преобразуя его в JavaScript.

Примеры таких языков:

- *CoffeeScript* добавляет «синтаксический сахар» для JavaScript. Он вводит более короткий синтаксис, который позволяет писать чистый и лаконичный код.
- *TypeScript* концентрируется на добавлении «строгой типизации» для упрощения разработки и поддержки больших и сложных систем. Разработан Microsoft.
- *Flow* тоже добавляет типизацию, но иначе. Разработан Facebook.
- *Dart* имеет собственный движок работающий вне браузера (например, в мобильных приложениях). Первоначально был предложен Google как замена JavaScript, но на данный момент, необходима его транспилиция для запуска, так же как для языков выше.

## Справочники и спецификации

*Спецификация* ECMA-262 содержит самую глубокую, детальную и формализованную информацию о JavaScript. Она определяет сам язык.

Самые последние возможности, которые включены или будут включены в следующую версию ES, можно найти на <https://tc39.es/ecma262/> и <https://github.com/tc39/proposals>.

MDN (Mozilla) JavaScript Reference – это *справочник* с примерами и другой информацией. Хороший источник для получения подробных сведений о функциях языка, методах встроенных объектов и так далее.

Располагается по адресу <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference>.

MSDN – справочник от Microsoft, содержащий много информации, в том числе по JavaScript. Если вам нужно найти что-то специфическое по разработке для браузера Internet Explorer, то лучше искать тут: <http://msdn.microsoft.com/>.

JavaScript – это развивающийся язык, в который постоянно добавляется что-то новое. Посмотреть, какие возможности поддерживаются в разных браузерах и других движках, можно в следующих источниках:

- <http://caniuse.com> – таблицы совместимости с информацией о поддержке по каждой возможности языка.
- <https://kangax.github.io/compat-table> – таблица с возможностями языка и движками, которые их поддерживают и не поддерживают.

## Редакторы кода

Термином IDE (Integrated Development Environment, «интегрированная среда разработки») называют мощные редакторы с множеством функций, которые работают в рамках целого проекта. Это не просто редактор, а нечто большее.

IDE загружает проект (который может состоять из множества файлов), позволяет переключаться между файлами, предлагает автодополнение по коду всего проекта (а не только открытого файла), также она интегрирована с системой контроля версий (например, такой как git), средой для тестирования и другими инструментами на уровне всего проекта.

Кроссплатформенные IDE:

- Visual Studio Code (бесплатная).
- WebStorm (платная).

Обычные редакторы менее мощные, чем IDE, но они отличаются скоростью, удобным интерфейсом и простотой. Их используют для того, чтобы быстро открыть и отредактировать нужный файл с кодом. Такие редакторы могут иметь множество плагинов, включая автодополнение и анализаторы синтаксиса на уровне директории. Наиболее популярные редакторы кода:

- Atom (кроссплатформенный, бесплатный).
- Sublime Text (кроссплатформенный, частично бесплатный).
- Notepad++ (Windows, бесплатный).
- Vim и Emacs.

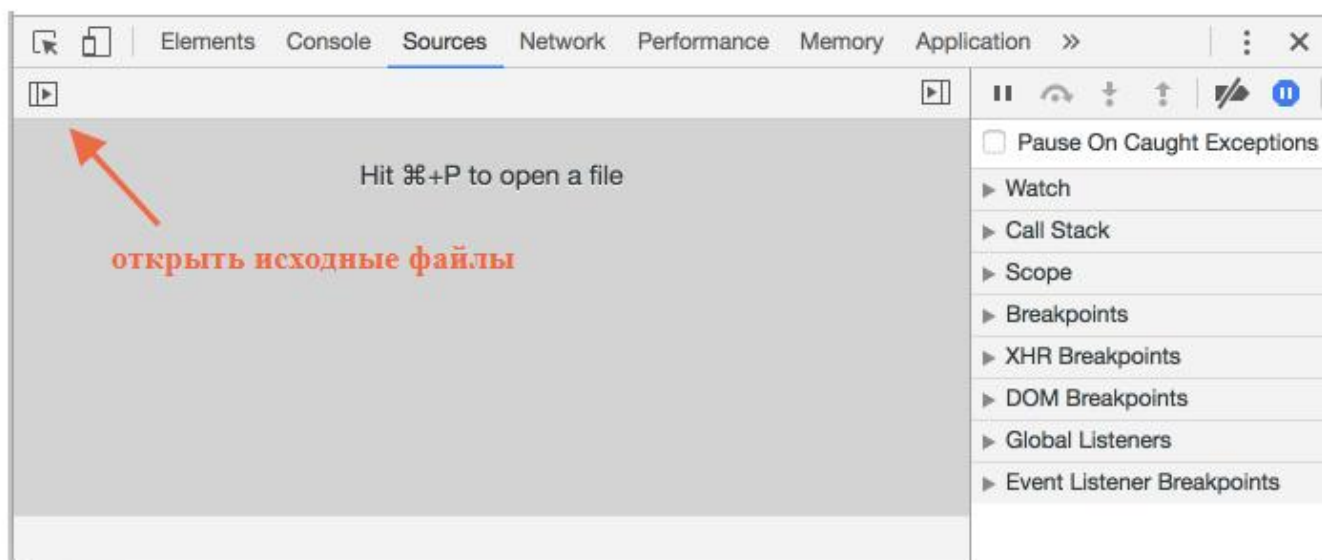
## 2. Отладка скриптов


Все современные браузеры и многие среды разработки поддерживают средства отладки кода — специальный графический интерфейс для быстрого поиска и устранения ошибок. Он также позволяет по шагам отследить, что именно происходит в коде.

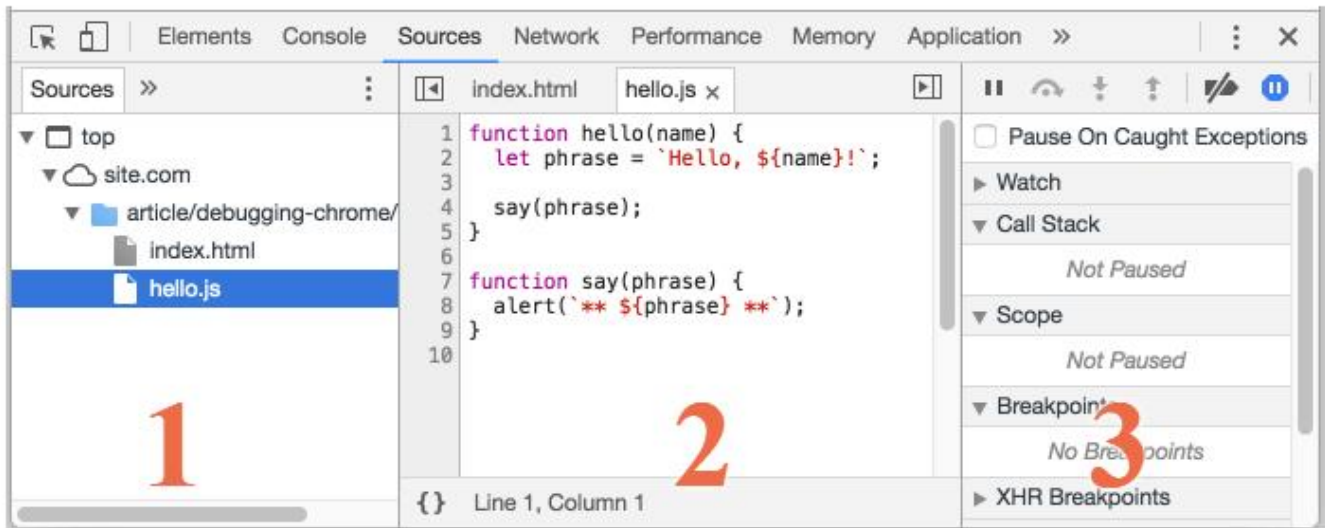
Рассмотрим процесс отладки в браузере Chrome, так как у него достаточно возможностей, в большинстве других браузеров процесс будет схожим.

### Панель «Исходный код» («Sources»)

Чтобы включить инструменты разработчика, необходимо нажать F12 (Mac: Cmd+Opt+I). Щёлкните по панели sources («исходный код»). При первом запуске получаем следующее:




Кнопка-переключатель  откроет вкладку со списком файлов. Кликните на неё и файл с кодом JS. Исходный код появится в соответствующей вкладке:



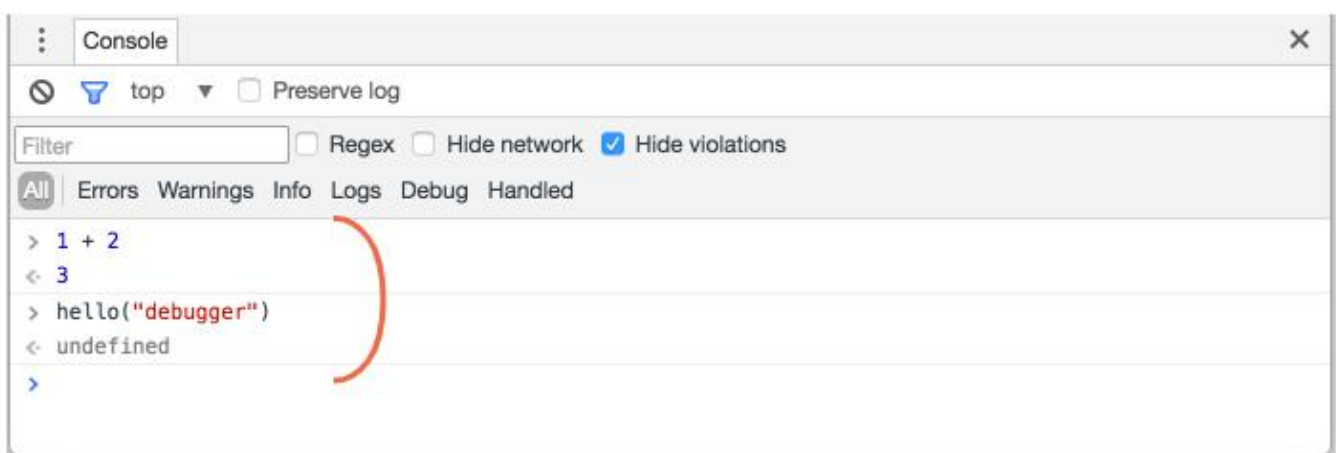
Интерфейс состоит из трёх зон:

1. В зоне Resources (Ресурсы) показаны файлы HTML, JavaScript, CSS, включая изображения, используемые на странице. Здесь также могут быть файлы различных расширений Chrome.
2. Зона Source показывает исходный код.
3. Зона Information and control (Сведения и контроль) отведена для отладки.

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

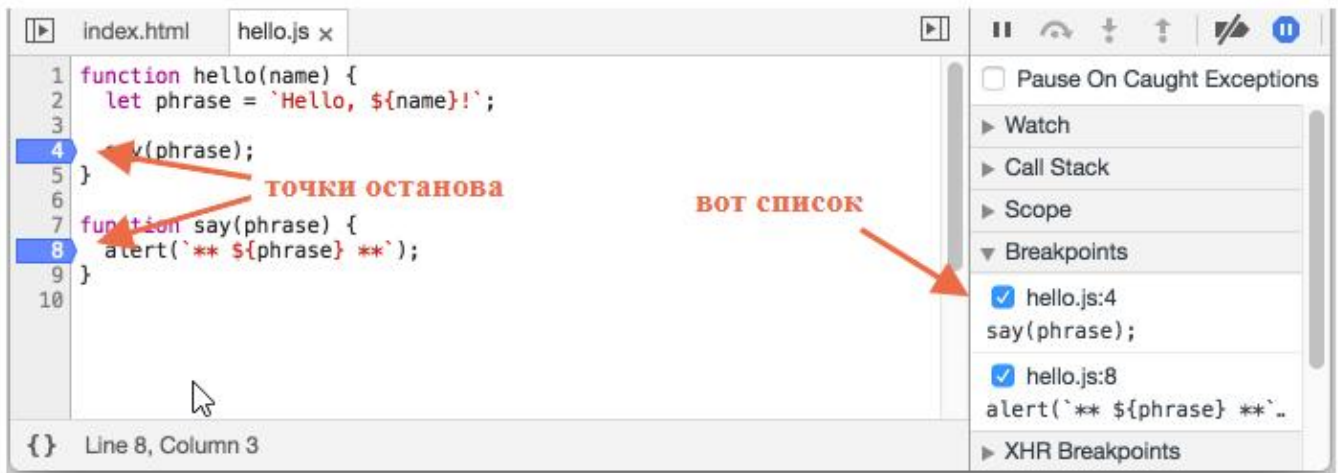
## Консоль

При нажатии на клавишу Esc в нижней части экрана появляется 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 инструкция `hello("debugger")` ничего не возвращает, так что получаем `undefined`:



## Точки останова (breakpoints)

Точки останова позволяют разобраться как работает код. Чтобы поставить точку останова необходимо щёлкнуть по номеру строки. Номер строки будет окрашен в синий цвет. Вот что в итоге должно получиться:



Точка останова – это участок кода, где отладчик автоматически приостановит исполнение JavaScript. Пока исполнение поставлено «на паузу», можно просмотреть текущие значения переменных, выполнить команды в консоли и т.д., т.е. выполнить отладку кода.

Список точек останова располагается в правой части графического интерфейса. Когда точек выставлено много, да ещё и в разных файлах, этот список поможет эффективно ими управлять:

- Быстро переместиться к любой точке останова в коде – нужно щёлкнуть по точке в правой части экрана.
- Временно деактивировать точку – в общем списке нужно снять галочку напротив ненужной в данный момент точки.
- Удалить точку – щёлкнуть по ней правой кнопкой мыши и выбрать Remove (Удалить).
- и так далее.

Можно задать и так называемую условную точку останова. Для этого необходимо щёлкнуть правой кнопкой мыши по номеру строки в коде. Если задать выражение, то именно при его истинности выполнение кода будет приостановлено.

Этот метод используется, когда выполнение кода нужно остановить при присвоении определённого выражения какой-либо переменной или при определённых параметрах функции.

### Команда Debugger

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

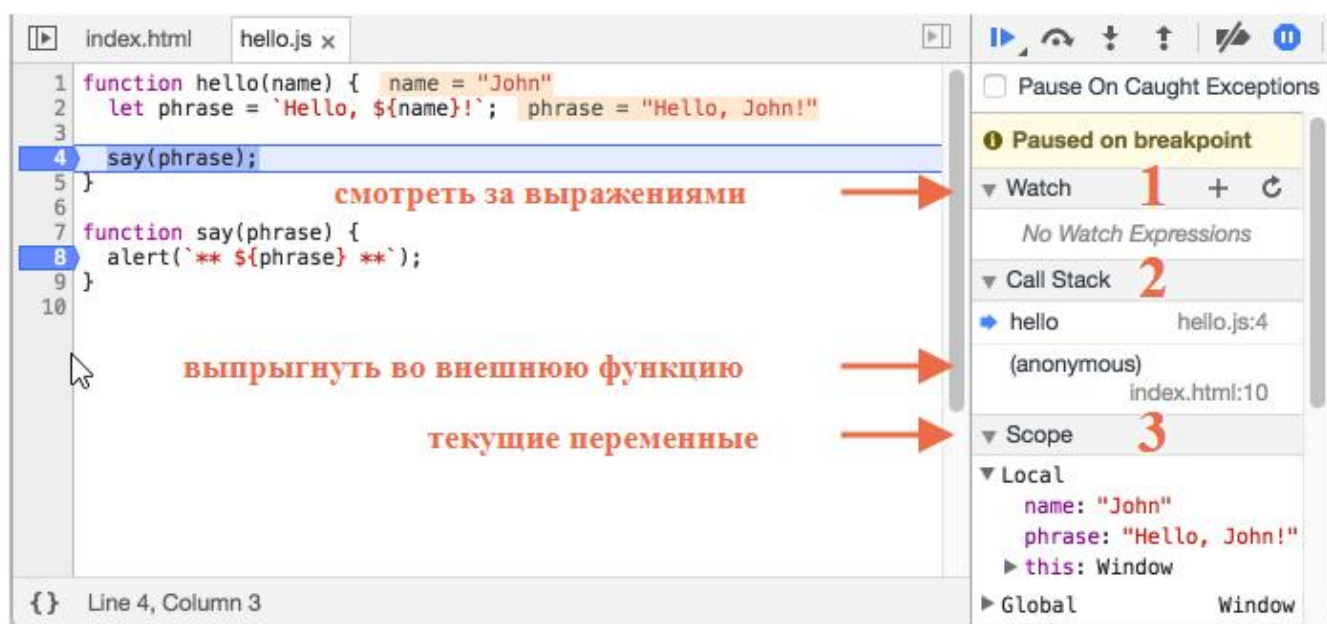
```
function hello(name) {  
  let phrase = `Привет, ${name}!`;   
  
  debugger; // <-- здесь выполнение прерывается  
  
  say(phrase);  
}
```

Способ удобен тем, что можно продолжить работать в редакторе кода без необходимости переключения в браузер для выставления точки останова.

### Пошаговое выполнение скрипта



В примере функция `hello()` вызывается во время загрузки страницы, поэтому для начала отладки (после того, как выставлены точки останова) необходимо её перезагрузить (F5 (Windows, Linux) или Cmd+R (Mac)). Выполнение прервётся на четвёртой строчке:



Содержимое панели справа:

1. *Watch* показывает текущие значения выражений. Чтобы ввести выражение необходимо нажать на +. В процессе выполнения отладчик автоматически пересчитывает и выводит его значение.

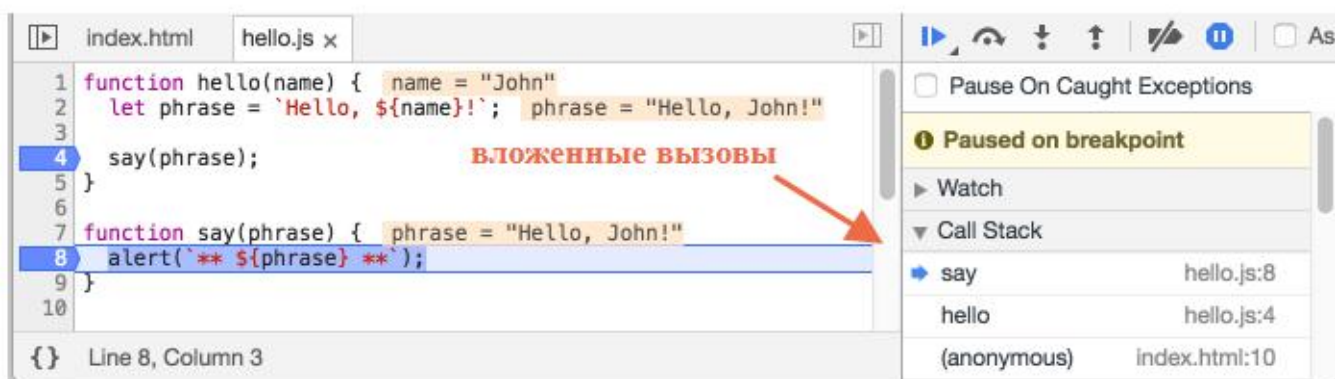
2. *Call Stack* показывает последовательность вызовов функций. Так как в примере отладчик работает с функцией `hello()`, вызванной скриптом из файла `index.html`, и там нет функции, то вызов «анонимный». При нажатии на элемент списка (например, на «anonymous») отладчик переходит к соответствующему коду, и его можно проанализировать.

3. *Scope* показывает текущие переменные. В *Local* отображаются локальные переменные функций, а их значения подсвечены в исходном коде. В *Global* перечисляются глобальные переменные (т.е. объявленные за пределами функций).

Для отладки кода по шагам в правой части панели есть несколько кнопок:


– ► продолжить выполнение. Быстрая клавиша – F8.

Возобновляет выполнение кода. Если больше нет точек останова, отладчик прекращает работу и позволяет приложению работать дальше:






Выполнение кода возобновилось, дошло до другой точки останова внутри `say()`, и отладчик снова приостановил выполнение. В списке Call stack (справа) появился ещё один вызов. Выполнение кода перешло внутрь функции `say()`.

-  сделать шаг (выполнить следующую команду), не заходя в функцию (полностью игнорируя её содержимое). Быстрая клавиша – F10.

-  сделать шаг. Быстрая клавиша – F11.


В отличие от предыдущей команды, здесь выполнение «заходит» во вложенные функции и шаг за шагом проходим по скрипту.

-  продолжить выполнение до завершения текущей функции. Быстрая клавиша – Shift+F11.

Выполнение кода остановится на самой последней строчке текущей функции. Используется, когда разработчик случайно нажал и зашел в функцию и хочет из неё выйти.

-  активировать/деактивировать все точки останова.

Эта кнопка не влияет на выполнение кода, она лишь позволяет массово включить/отключить точки останова.

-  разрешить/запретить остановку выполнения в случае возникновения ошибки.

Если опция включена и инструменты разработчика открыты, любая ошибка в скрипте приостанавливает выполнение кода, что позволяет его проанализировать. Поэтому если скрипт завершается с ошибкой, необходимо открыть отладчик, включить эту опцию, перезагрузить страницу и локализовать проблему.

Если щёлкнуть правой кнопкой мыши по строчке кода, в контекстном меню можно выбрать опцию «*Continue to here*» («продолжить до этого места»). Этот метод используется, когда нужно продвинуться на несколько шагов вперёд до нужной строки, но не хочется выставлять точки останова.

## Логирование

Если нужно что-то вывести в консоль из кода, применяется функция `console.log`. К примеру, выведем в консоль значения от нуля до четырёх:

```
for (let i = 0; i < 5; i++) {  
  console.log("значение", i);  
}
```

Консоль можно открыть через инструменты разработчика – выбрав вкладку «Консоль» или нажать Esc, находясь в другой вкладке – консоль откроется в нижней части интерфейса. Если правильно выстроить логирование в приложении, то можно и без отладчика разобраться, что происходит в коде.

## 3. Синтаксис JavaScript

Код должен быть максимально читаемым и понятным. Для этого нужен хороший стиль написания кода.

### Команды

Каждая команда пишется на отдельной строке – так код лучше читается:

```
alert('Привет');  
alert('Мир');
```

### Точка с запятой

В JavaScript рекомендуется точку с запятой ставить в конце команды. Это не требование стандарта, но в некоторых ситуациях может привести к ошибкам.

Рассмотрим некоторые примеры. Точку с запятой во многих случаях можно не ставить, если есть переход на новую строку:

```
alert('Привет')  
alert('Мир')
```

В этом случае JavaScript интерпретирует переход на новую строку как разделитель команд и автоматически вставляет «виртуальную» точку с запятой между ними. Однако, так происходит не всегда. Например, этот код:

```
alert(3 +  
1  
+ 2);
```

выведет 6. То есть, точка с запятой не ставится, так как это незавершённое выражение, конца которого JavaScript ждёт с первой строки.

Но в некоторых важных ситуациях JavaScript не вставит точку с запятой там, где она нужна. Например, такой код работать не будет:

```
alert("Сейчас будет ошибка")  
[1, 2].forEach(alert)
```

Выведется первый alert, а дальше – ошибка. Потому что перед квадратной скобкой JavaScript точку с запятой не ставит, а она здесь нужна.

### Комментарии

Со временем программа становится большой и сложной. Появляется необходимость добавить комментарии, которые объясняют, что происходит и почему.

Один из показателей хорошего разработчика – качество комментариев, которые позволяют эффективно поддерживать код, возвращаться к нему после любой паузы и легко вносить изменения.

Должен быть минимум комментариев, которые отвечают на вопрос "что происходит в коде?". Хороший код и так понятен. Если вам кажется, что нужно добавить комментарий для улучшения понимания, это значит, что ваш код недостаточно прост и стоит его переписать.

Комментарий должен содержать:

- Архитектурный комментарий – «как оно, вообще, устроено».

Какие компоненты есть, какие технологии использованы, поток взаимодействия. О чём и зачем этот скрипт. Эти комментарии особенно нужны, если вы не один, а проект большой.

- Справочный комментарий перед функцией – о том, что именно она делает, какие параметры принимает и что возвращает.

Для таких комментариев существует синтаксис JSDoc.

```
1  /**
2   * Возвращает x в степени n, только для натуральных n
3   *
4   * @param {number} x Число для возведения в степень.
5   * @param {number} n Показатель степени, натуральное число.
6   * @return {number} x в степени n.
7   */
8  function pow(x, n) {
9      ...
10 }
```

Такие комментарии позволяют сразу понять, что принимает и что делает функция, не вникая в код. Они автоматически обрабатываются многими редакторами, например Aptana и редакторами от JetBrains, которые учитывают их при автодополнении, а также выводят их в автоподсказках при наборе кода. Есть инструменты, например JSDoc 3, которые умеют генерировать по таким комментариям документацию в формате HTML.

Более важными могут быть комментарии, которые объясняют не *что*, а *почему* в коде происходит именно это. Например, есть несколько способов решения задачи. Вы выбрали именно это решение. Вы пробовали решить задачу по-другому, но не получилось – напишите об этом. Особенно это важно в тех случаях, когда используется не первый приходящий в голову способ, а какой-то другой. Вы открываете код, который был написан какое-то время назад, и видите, что он «неоптимален», захотите его переписать. Только этот вариант вы уже обдумали раньше. И отказались, а почему – забыли. Комментарии, которые объясняют выбор решения, очень важны. Они помогают понять происходящее и предпринять правильные шаги при развитии кода.

Комментарии могут находиться в любом месте программы и никак не влияют на её выполнение. Интерпретатор JavaScript попросту игнорирует их.

*Однострочные комментарии* начинаются с двойного слэша //. Текст считается комментарием до конца строки:

```
// Команда ниже говорит "Привет"
alert( 'Привет' );

alert( 'Мир' ); // Второе сообщение выводим отдельно
```

*Многострочные комментарии* начинаются слешем-звездочкой «/\*» и заканчиваются звездочкой-слешем «\*/», вот так:

```
/* Пример с двумя сообщениями.
Это - многострочный комментарий.
*/
alert( 'Привет' );
alert( 'Мир' );
```

Всё содержимое комментария игнорируется. Если поместить код внутри /\* ... \*/ или после // – он не выполнится.

```
/* Закомментировали код
```

```
alert( 'Привет' );
*/
alert( 'Мир' );
```

В большинстве редакторов комментариев можно поставить горячей клавишей, обычно это Ctrl+/ для однострочных и Ctrl+Shift+/ – для многострочных комментариев (нужно выделить блок и нажать сочетание клавиш). Вложенные комментарии не поддерживаются.

В этом коде будет ошибка:

```
/*
/* вложенный комментарий ?!? */
*/
alert('Мир');
```

Не бойтесь комментариев. Чем больше кода в проекте – тем они важнее. Что же касается увеличения размера кода – это не страшно, т.к. существуют инструменты сжатия JavaScript, которые при публикации кода легко их удалят.

### Фигурные скобки

Фигурные скобки располагаются на той же строке, это так называемый «египетский» стиль. Перед скобкой ставится пробел.

```
1  /* Так писать плохо! */
2  if (n < 0) {alert("Число отрицательное.");}
3
4  /* Так допустимо! */
5  if (n < 0) alert("Число отрицательное.");
6
7  /* Лучший вариант! */
8  if (n < 0) {
9      alert("Число отрицательное.");
10 }
```

### Длина строки

Максимальную длину строки согласовывают в команде. Как правило, это либо 80, либо 120 символов, в зависимости от того, какие мониторы у разработчиков. Более длинные строки необходимо разбивать для улучшения читаемости.

### Отступы

Отступы нужны двух типов:

*Горизонтальный отступ, при вложенности – два (или четыре) пробела.*

Как правило, используются именно пробелы, т.к. они позволяют установить более точное выравнивание, чем символ «Tab», например, выровнять аргументы относительно открывающей скобки:

```
1  show("Строки" +
2      " выровнены" +
3      " строго" +
4      " одна под другой");
5
```

*Вертикальный отступ, для лучшей разбивки кода – перевод строки.*

Используется, чтобы разделить логические блоки внутри одной функции. В примере разделены инициализация переменных, главный цикл и возвращение результата:

```
1 function pow(x, n) {  
2     var result = 1;  
3     //  
4     for (var i = 0; i < n; i++) {  
5         result *= x;  
6     }  
7     //  
8     return result;  
9 }
```

Вставляйте дополнительный перевод строки туда, где это сделает код более читаемым. Не должно быть более 9 строк кода подряд без вертикального отступа.

## Именованние

Общее правило:

Имя переменной – существительное.

Имя функции – глагол или начинается с глагола.

Для имён используется английский язык (не транслит) и верблюжья нотация.

Верблюжья нотация (CamelCase) – стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово внутри фразы пишется с прописной буквы.

## Уровни вложенности

Уровней вложенности должно быть немного.

Например, проверки в циклах можно делать через «continue», чтобы не было дополнительного уровня if(..) { ... }.

Вместо:

```
1 for (var i = 0; i < 10; i++) {  
2     if (i подходит) {  
3         ... // <- уровень вложенности 2  
4     }  
5 }
```

Используйте:

```
1 for (var i = 0; i < 10; i++) {  
2     if (i не подходит) continue;  
3     ... // <- уровень вложенности 1  
4 }
```

Аналогичная ситуация – с if/else и return. Следующие две конструкции идентичны. Первая:

```
1 function isEven(n) { // проверка чётности  
2     if (n % 2 == 0) {  
3         return true;  
4     } else {  
5         return false;  
6     }  
7 }
```

Вторая:

```

1  function isEven(n) { // проверка чётности
2      if (n % 2 == 0) {
3          return true;
4      }
5
6      return false;
7  }

```

Если в блоке if есть return, то else за ним не нужен.

Рекомендуется проверять сначала простые условия, вернуть результат, а затем обрабатывать более сложные, без дополнительного уровня вложенности.

## Функции

Функции должны быть небольшими. Если функция большая – желательно разбить её на несколько. Сравните, например, две функции showPrimes(n) для вывода простых чисел до n. В первом варианте используется метка:

```

1  function showPrimes(n) {
2      nextPrime: for (var i = 2; i < n; i++) {
3
4          for (var j = 2; j < i; j++) {
5              if (i % j == 0) continue nextPrime;
6          }
7
8          alert( i ); // простое
9      }
10 }

```

Во втором варианте используется дополнительная функция isPrime(n) для проверки на простоту:

```

1  function showPrimes(n) {
2
3      for (var i = 2; i < n; i++) {
4          if (!isPrime(i)) continue;
5
6          alert(i); // простое
7      }
8  }
9
10 function isPrime(n) {
11     for (var i = 2; i < n; i++) {
12         if ( n % i == 0) return false;
13     }
14     return true;
15 }

```

Второй вариант проще и понятнее. Вместо участка кода видно описание действия, которое там совершается (проверка isPrime).

Рекомендуется располагать функции под кодом, который их использует, так как при чтении кода разработчик в первую очередь хочет знать, что код делает, а уже затем какие функции реализованы.



```

1  // код, использующий функции
2  var elem = createElement();
3  setHandler(elem);
4  walkAround();
5
6  // --- функции ---
7
8  function createElement() {
9      ...
10 }
11
12 function setHandler(elem) {
13     ...
14 }
15
16 function walkAround() {
17     ...
18 }

```

### Руководства по стилю

Когда написанием проекта занимается целая команда, то должен существовать один стандарт кода, описывающий где и когда ставить пробелы, запятые, переносы строк и т.п. Существует много готовых руководств, которые можно использовать. Большинство из них на английском языке:

- Google JavaScript Style Guide;
- jQuery JavaScript Style Guide;
- Airbnb JavaScript Style Guide;
- Idiomatic.JS;
- Dojo Style Guide.

### Автоматизированные средства проверки стиля

Существуют средства, проверяющие стиль кода. Самые известные – это:

- JSLint – проверяет код на соответствие стилю JSLint, в онлайн-интерфейсе можно ввести код и различные настройки проверки, чтобы сделать её более мягкой.
- JSHint – это JSLint с большим количеством настроек.

В частности, JSLint и JSHint интегрированы с большинством редакторов, они гибко настраиваются под нужный стиль и совершенно незаметно улучшают разработку, подсказывая, где и что поправить.

### Полифилы

JavaScript – динамично развивающийся язык программирования. Регулярно появляются новые возможности. Разработчики JavaScript-движков сами принимают решение, какие возможности реализовать в первую очередь и часто реализуется только часть стандарта. Поэтому современные возможности JavaScript, некоторые движки могут не поддерживать.

Новые возможности языка могут включать встроенные функции и синтаксические конструкции. Существуют транспилеры (например, Babel), которые переписывают синтаксические конструкции в старые. Новые встроенные функции, нужно реализовать с помощью полифилов.



*Полифил* – это код, реализующий какую-либо функциональность, которая не поддерживается в некоторых версиях веб-браузеров.

Два популярных полифила:

- `core.js` поддерживает много функций, можно подключать только нужные.
- `polyfill.io` – сервис, который автоматически создаёт скрипт с полифилом в зависимости от необходимых функций и браузера пользователя.

Таким образом, чтобы современные функции поддерживались в старых движках, необходимо установить транспILER и добавить полифил.

### «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.

Так было до 2009 года, когда появился ECMAScript 5 (ES5). Он добавил новые возможности в язык и изменил некоторые из существующих. Чтобы старый код работал, большинство таких модификаций по умолчанию отключены. Необходимо явно включить их с помощью специальной директивы: "use strict". Когда она находится в верхней части скрипта, весь сценарий работает в «современном» режиме.

Например:

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

"use strict" можно расположить в начале большинства видов функций, вместо всего скрипта. Это позволяет включить строгий режим только в этой функции. Но обычно, он используется для всего файла.

Нет какой-либо директивы или другого способа, который возвращает движок к старому поведению. Как только включается строгий режим, отменить это невозможно.

В консоле браузера use strict по умолчанию выключен.

## 4. Переменные.

Для создания переменной в JavaScript, используйте ключевое слово `let`:

```
let message;  
  
message = 'Hello'; // сохранить строку
```

Можно объявить несколько переменных в одной строке, но это не рекомендуется, так как ухудшает читаемость кода:

```
let user = 'John', age = 25, message = 'Hello';
```

Многострочный вариант немного длиннее, но легче для чтения:

```
let user = 'John';  
let age = 25;  
let message = 'Hello';
```

Можно объявлять переменные на нескольких строках и даже с запятой в начале строки:

```
let user = 'John',  
    age = 25,  
    message = 'Hello';  
  
let user = 'John'  
    , age = 25  
    , message = 'Hello';
```

В старых скриптах можно найти ключевое слово: var вместо let:

```
var message = 'Hello';
```

Ключевое слово var — почти то же самое, что и let. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.

## Имена переменных

Название переменной должно иметь ясный и понятный смысл, говорить о том, какие данные в ней хранятся.

Именованние переменных — это один из самых важных и сложных навыков в программировании. Быстрый взгляд на имена переменных может показать, какой код был написан новичком, а какой опытным разработчиком.

В реальном проекте большая часть времени тратится на изменение и расширение существующей кодовой базы, а не на написание чего-то совершенно нового с нуля. Когда мы возвращаемся к коду после какого-то промежутка времени, гораздо легче найти информацию, которая хорошо размечена. Или, другими словами, когда переменные имеют хорошие имена.

Несколько советов:

- Используйте легко читаемые имена, такие как userName или shoppingCart.
- Избегайте использования аббревиатур или коротких имён, таких как a, b, c, за исключением тех случаев, когда вы точно знаете, что так нужно.
- Делайте имена максимально описательными и лаконичными. Примеры плохих имён: data и value. Такие имена ничего не говорят. Их можно использовать только в том случае, если из контекста кода очевидно, какие данные хранит переменная.
- Договоритесь с вашей командой о используемых терминах. Если посетитель сайта называется «user» тогда мы должны назвать связанные с ним переменные currentUser или newUser вместо того, чтобы называть их currentVisitor или newManInTown.

В JavaScript есть два ограничения, касающиеся имён переменных:

1. Имя переменной должно содержать только буквы, цифры или символы \$ и \_.
2. Первый символ не должен быть цифрой.

Если имя содержит несколько слов, обычно используется верблюжья нотация, например, myVeryLongName.

Примеры допустимых имён:

```
let userName;  
let test123;  
  
let $ = 1; // объявили переменную с именем "$"  
let _ = 2; // а теперь переменную с именем "_"
```

Примеры неправильных имён переменных:

```
let 1a; // не может начинаться с цифры  
  
let my-name; // дефис '-' не разрешён в имени
```

Регистр имеет значение. Переменные с именами apple and AppLE – это две разные переменные.

Нелатинские буквы разрешены, но не рекомендуются. Можно использовать любой язык, включая кириллицу или даже иероглифы, например:

```
let имя = '...';  
let 我 = '...';
```

Технически здесь нет ошибки, такие имена разрешены, но есть международная традиция использовать английский язык в именах переменных.

Существует список зарезервированных слов, которые нельзя использовать в качестве имён переменных, потому что они используются самим языком. Например: let, class, return и function зарезервированы. Приведённый ниже код даёт синтаксическую ошибку:

```
let let = 5; // нельзя назвать переменную "let", ошибка!  
let return = 5; // также нельзя назвать переменную "return", ошибка!
```

Обычно нужно определить переменную перед её использованием. Но старые стандарты позволяли создавать переменную простым присвоением значения без использования let. Это все ещё работает, если не включить use strict, для поддержания совместимости со старыми скриптами.

```
// заметка: "use strict" в этом примере не используется  
num = 5; // если переменная "num" не существовала, она создаётся  
alert(num); // 5
```

Это плохая практика и приведёт к ошибке в строгом режиме:

```
"use strict";
```

```
num = 5; // error: num is not defined
```

## Константы

Чтобы объявить константную переменную необходимо использовать `const` вместо `let`:

```
const myBirthday = '18.04.1982';
```

Изменение константы приведет к ошибке:

```
const myBirthday = '18.04.1982';
```

```
myBirthday = '01.01.2001'; // ошибка, константу нельзя перезаписать!
```

Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта. Названия таких констант пишутся с использованием заглавных букв и подчёркивания, например, для различных цветов в шестнадцатичном формате:

```
const COLOR_RED = "#F00";  
const COLOR_GREEN = "#0F0";  
const COLOR_BLUE = "#00F";  
const COLOR_ORANGE = "#FF7F00";
```

```
let color = COLOR_ORANGE;  
alert(color); // #FF7F00
```

Преимущества такого подхода:

- `COLOR_ORANGE` гораздо легче запомнить, чем `"#FF7F00"`.
- Гораздо легче допустить ошибку при вводе `"#FF7F00"`, чем при вводе `COLOR_ORANGE`.
- При чтении кода, `COLOR_ORANGE` намного понятнее, чем `#FF7F00`.

Когда мы должны использовать для констант заглавные буквы, а когда называть их нормально? Давайте разберёмся и с этим.

## 5. Типы данных.

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:

```
// Не будет ошибкой  
let message = "hello";  
message = 123456;
```

Языки программирования, в которых такое возможно, называются динамически типизированными. Это значит, что типы данных есть, но переменные не привязаны ни к одному из них. Есть семь основных типов данных в JavaScript, пять из них называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка или число, или что-то ещё).

## Число

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

Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой. Существует множество операций для чисел, например, умножение \*, деление /, сложение +, вычитание - и так далее.

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

*Infinity* представляет собой математическую бесконечность  $\infty$ . Это особое значение, которое больше любого числа. Мы можем получить его в результате деления на ноль или задать его явно:

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

*NaN* означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:

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

Любая операция с NaN возвращает NaN:

```
alert( "не число" / 2 + 5 ); // NaN
```

Математические операции в JavaScript позволяют делать что угодно: делить на ноль, обращаться со строками как с числами и т.д. Скрипт никогда не остановится с фатальной ошибкой. В худшем случае просто в результате выполнения будет возвращен NaN.

Тип *BigInt* – представляет собой встроенный объект, который предоставляет способ представлять целые числа больше  $\text{pow}(2, 53) - 1$ , наибольшего числа, которое JavaScript может надежно представить с Number примитивом.

Чтобы обозначить, что число относится к типу BigInt нужно добавить n в конце числа. n означает, что это BigInt.

```
let oldMax = Number.MAX_SAFE_INTEGER; // 9007199254740991  
++oldMax; // 9007199254740992  
++oldMax; // 9007199254740992 <- такое же значение  
  
let newMax = 9007199254740992n;  
++newMax; // 9007199254740993n  
++newMax; // 9007199254740994n  
  
const a = 9007199254740991n; // 9007199254740991n  
const b = BigInt(9007199254740991n); // 9007199254740991n  
a === b; // true  
  
typeof 10; // "number";
```

```
typeof 10n; // "bigint";
```

## 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 кавычки же имеют расширенный функционал. 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. Можно записать туда всё, что угодно: переменную `name` или выражение `1 + 2`, или что-то более сложное.

В некоторых языках для одного символа существует специальный «символьный» тип. Например, в C и Java это `char`. В JavaScript подобного типа нет, есть только тип `string`. Строка может содержать один символ или множество.

## 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, не правильно». Например:

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

Булевы 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;
```

В JavaScript `null` не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно». В приведённом выше коде указано, что переменная `age` неизвестна или не имеет значения по какой-то причине.

### Значение «undefined»

Специальное значение `undefined` также стоит особняком. Оно 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 её значением будет `undefined`:

```
let x;  
  
alert(x); // выведет "undefined"
```

Технически можно присвоить значение `undefined` любой переменной:

```
let x = 123;  
  
x = undefined;  
  
alert(x); // "undefined"
```

Но так делать не рекомендуется. Обычно `null` используется для присвоения переменной «пустого» или «неизвестного» значения, а `undefined` для проверки, была ли переменная назначена.

### Объекты и символы

Тип `object` (объект) используется для хранения коллекций данных или более сложных объектов.

Тип `symbol` (символ) используется для создания уникальных идентификаторов объектов.

Рассмотрим объекты и символы позднее после того, как изучим подробнее примитивы.

### Оператор `typeof`

Оператор `typeof` возвращает тип аргумента. Это полезно, когда необходимо обрабатывать значения различных типов по-разному или просто сделать проверку.

У него есть два синтаксиса, результат одинаковый:

- Синтаксис оператора: `typeof x`.
- Синтаксис функции: `typeof(x)`.

Вызов `typeof x` возвращает строку с именем типа:

```
typeof undefined // "undefined"  
typeof 0 // "number"  
typeof true // "boolean"  
typeof "foo" // "string"  
typeof null // "object" (1)  
typeof alert // "function" (2)
```



Результатом вызова `typeof null` в строке (1) является `"object"`. Это неверно. Это официально признанная ошибка в `typeof`, сохранённая для совместимости. `null` не является объектом, это специальное значение с отдельным типом.

Вызов `typeof alert` в строке (2) возвращает `"function"`, потому что `alert` является функцией. В JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но `typeof` обрабатывает их особым образом, возвращая `"function"`. Формально это неверно, но очень удобно на практике.

## 6. Преобразование типов

Чаще всего, операторы и функции автоматически приводят переданные им значения к нужному типу. Например, `alert` автоматически преобразует любое значение к строке, а математические операторы преобразуют значения к числам. Есть также случаи, когда нужно явно преобразовать значение в ожидаемый тип.

### 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 способом. `False` становится `"false"`, `null` становится `"null"` и т.п.

Почти все математические операторы выполняют численное преобразование. Исключение составляет `+`. Если одно из слагаемых является строкой, тогда и все остальные приводятся к строкам и они конкатенируются (присоединяются) друг к другу:

```
alert( 1 + '2' ); // '12' (строка справа)
alert( '1' + 2 ); // '12' (строка слева)
```

Так происходит, только если хотя бы один из аргументов является строкой. Во всех остальных случаях, значения складываются как числа.

### 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 необходимо получить число из строковых источников, вроде форм текстового ввода. 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	Результат
undefined	NaN
null	0
true / false	1 / 0
string	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки «считывается» число. При ошибке результат NaN.

Примеры:

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

null и undefined ведут себя по-разному: null становится нулём, undefined приводится к NaN.

### Логическое преобразование

Логическое преобразование происходит в логических операторах но так же может быть выполнено явно с помощью функции Boolean(value).

Правила 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
```

Строка с нулём "0" преобразуется в true. Некоторые языки 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
```

## 7. Операторы.

*Операнд* – то, к чему применяется оператор. Например, в умножении  $5 * 2$  есть два операнда: левый операнд равен 5, а правый операнд равен 2. Иногда их называют аргументами.

*Унарным* называется оператор, который применяется к одному операнду. Например, оператор унарный минус "-" меняет знак числа на противоположный:

```
let x = 1;

x = -x;
alert( x ); // -1, применили унарный минус
```

*Бинарным* называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:

```
let x = 1, y = 3;
alert( y - x ); // 2, бинарный минус
```

Формально мы говорим о двух разных операторах: унарное отрицание (один операнд: меняет знак) и бинарное вычитание (два операнда: вычитает).

### Сложение строк, бинарный +

Обычно при помощи оператора плюс '+' складывают числа. Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну. Если хотя бы один операнд является строкой, то второй будет также преобразован к строке.

```
let s = "моя" + "строка";
alert(s); // моястрока
```

Например:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Важно то, что операции выполняются слева направо. Если перед строкой идут два числа, то числа будут сложены перед преобразованием в строку:

```
alert(2 + 2 + '1' ); // будет "41", а не "221"
```

Сложение и преобразование строк – это особенность бинарного плюса +. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа. Например, вычитание и деление:

```
alert( 2 - '1' ); // 1
alert( '6' / '2' ); // 3
```

### Преобразование к числу, унарный плюс +

Плюс + существует в двух формах: бинарной, которая рассматривалась выше, и унарной. Унарный плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число. Это то же самое, что и Number(...), только короче. Например:

```
// Не влияет на числа
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// Преобразует нечисла в число
alert( +true ); // 1
alert( +"" ); // 0
```

Необходимость преобразовывать строки в числа возникает очень часто. Например, обычно значения полей HTML-формы – это строки. Если их надо сложить, то сначала необходимо преобразовать к числам. Бинарный плюс сложит их как строки:

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // "23"
```

Поэтому используются унарный плюс, чтобы преобразовать к числу:

```
let apples = "2";
let oranges = "3";

// оба операнда предварительно преобразованы в числа
alert( +apples + +oranges ); // 5

// более длинный вариант
// alert( Number(apples) + Number(oranges) ); // 5
```

Сначала выполняются унарные плюсы, приведут строки к числам, а затем – бинарный '+' их сложит. Этот порядок определяет приоритет оператора.

### Приоритет операторов

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется приоритетом, или, другими словами, существует определённый порядок выполнения операторов.

Известно, что умножение в выражении  $2 * 2 + 1$  выполнится раньше сложения. Это потому, что умножение имеет более высокий приоритет, чем сложение.

Скобки важнее, чем приоритет, так что, если необходимо изменить порядок по умолчанию, можно использовать их, чтобы изменить приоритет. Например, написать  $(1 + 2) * 2$ .

В JavaScript много операторов. Каждый оператор имеет соответствующий номер приоритета. Тот, у кого это число больше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.

Ниже представлена часть таблицы приоритетов. Обратите внимание, что у унарных операторов приоритет выше, чем у соответствующих бинарных:

Приоритет	Название	Обозначение
...	...	...
16	унарный плюс	+
16	унарный минус	-
14	умножение	*
14	деление	/
13	сложение	+
13	вычитание	-
...	...	...
3	присваивание	=
...	...	...

Так как «унарный плюс» имеет приоритет 16, который выше, чем 13 у «сложения» (бинарный плюс), то в выражении `" +apples + +oranges "` сначала выполнятся унарные плюсы, а затем сложение.

### Присваивание

В таблице приоритетов также есть оператор присваивания `=`. У него один из самых низких приоритетов – 3. Именно поэтому, когда переменной что-либо присваивают, например,  $x = 2 * 2 + 1$ , то сначала выполнится арифметика, а уже затем произойдёт присваивание `=`.

```
let x = 2 * 2 + 1;
```

```
alert( x ); // 5
```

Возможно присваивание по цепочке:

```
let a, b, c;
```

```
a = b = c = 2 + 2;
```

```
alert( a ); // 4
```

```
alert( b ); // 4
alert( c ); // 4
```

Такое присваивание работает справа-налево. Сначала вычисляется самое правое выражение  $2 + 2$ , и затем оно присвоится переменным слева:  $c$ ,  $b$  и  $a$ . В конце у всех переменных будет одно значение.

### Оператор "=" возвращает значение

Все операторы возвращают значение. Для некоторых это очевидно, например, сложение  $+$  или умножение  $*$ . Но и оператор присваивания не является исключением. Вызов  $x = \text{value}$  записывает  $\text{value}$  в  $x$  и возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения:

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```

В примере выше результатом  $(a = b + 1)$  будет значение, которое присваивается в  $a$ , то есть 3. Потом оно используется для дальнейших вычислений. Писать в таком стиле не рекомендуется, так как это делает ваш код менее понятным и читабельным.

### Остаток от деления %

Оператор взятия остатка в выражении  $a \% b$  возвращает остаток от деления  $a$  на  $b$ . Например:

```
alert( 5 % 2 ); // 1, остаток от деления 5 на 2
alert( 8 % 3 ); // 2, остаток от деления 8 на 3
alert( 6 % 3 ); // 0, остаток от деления 6 на 3
```

### Возведение в степень \*\*

Оператор возведения в степень  $**$  недавно добавили в язык. Для натурального числа  $b$  результат  $a ** b$  равен  $a$ , умноженному на само себя  $b$  раз. Например:

```
alert( 2 ** 2 ); // 4 (2 * 2)
alert( 2 ** 3 ); // 8 (2 * 2 * 2)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2)
```

Оператор работает и для нецелых чисел. Например:

```
alert( 4 ** (1/2) ); // 2
alert( 8 ** (1/3) ); // 2
```

### Инкремент/декремент

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу. Для этого существуют даже специальные операторы:

*Инкремент* ++ увеличивает на 1:

```
let counter = 2;
counter++;      // работает как counter = counter + 1, просто запись короче
alert( counter ); // 3
```

*Декремент* -- уменьшает на 1:

```
let counter = 2;
counter--;      // работает как counter = counter - 1, просто запись короче
alert( counter ); // 1
```

Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа 5++, вернёт ошибку.

Операторы ++ и -- могут быть расположены не только после, но и до переменной. Когда оператор идёт после переменной – это *постфиксная форма*: counter++. *Префиксная форма* – это когда оператор идёт перед переменной: ++counter. Обе эти формы записи делают одно и то же: увеличивают counter на 1. Разница в том, что префиксная форма возвращает новое значение, а постфиксная форма возвращает старое значение (до увеличения/уменьшения числа). Чтобы увидеть разницу, вот небольшой пример:

```
let counter = 1;
let a = ++counter; // (*)

alert(a); // 2
```

В строке (\*) префиксная форма увеличения counter, она возвращает новое значение 2.

Пример использования постфиксной формы:

```
let counter = 1;
let a = counter++; // (*)

alert(a); // 1
```

В строке (\*) постфиксная форма counter++ также увеличивает counter, но возвращает старое значение, которое было до увеличения. Так что alert покажет 1.

Операторы ++/-- могут также использоваться внутри выражений. Их приоритет выше, чем у арифметических операций. Например:

```
let counter = 1;
alert( 2 * ++counter ); // 4
```

Сравните с:

```
let counter = 1;
alert( 2 * counter++ ); // 2
```

Такая запись делает код менее читабельным. При беглом чтении кода можно с лёгкостью пропустить counter++, и будет неочевидно, что переменная увеличивается. Лучше использовать стиль «одна строка – одно действие»:



```
let counter = 1;
alert( 2 * counter );
counter++;
```

## Побитовые операторы

Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), на уровне их внутреннего двоичного представления. Эти операторы не являются чем-то специфичным для JavaScript, они поддерживаются в большинстве языков программирования. Поддерживаются следующие побитовые операторы:

- AND (и) ( & )
- OR (или) ( | )
- XOR (побитовое исключающее или) ( ^ )
- NOT (не) ( ~ )
- LEFT SHIFT (левый сдвиг) ( << )
- RIGHT SHIFT (правый сдвиг) ( >> )
- ZERO-FILL RIGHT SHIFT (правый сдвиг с заполнением нулями) ( >>> )

Они используются редко. Чтобы понять их, нужно углубиться в низкоуровневое представление чисел.

## Сокращённая арифметика с присваиванием

Часто нужно применить оператор к переменной и сохранить результат в ней же. Например:

```
let n = 2;
n = n + 5;
n = n * 2;
```

Эту запись можно укоротить при помощи совмещённых операторов += и \*:=

```
let n = 2;
n += 5; // теперь n=7 (работает как n = n + 5)
n *= 2; // теперь n=14 (работает как n = n * 2)

alert( n ); // 14
```

Подобные краткие формы записи существуют для всех арифметических и побитовых операторов: /=, -= и так далее. Вызов с присваиванием имеет в точности такой же приоритет, как обычное присваивание, то есть выполнится после большинства других операций:

```
let n = 2;

n *= 3 + 5;

alert( n ); // 16
```

## Оператор запятая

Оператор «запятая», редко используется и является одним из самых необычных. Иногда он используется для написания более короткого кода, поэтому нужно знать его, чтобы понимать, что при этом происходит.

Оператор запятая предоставляет возможность вычислять несколько выражений, разделяя их запятой. Каждое выражение выполняется, но возвращается результат только последнего. Например:

```
let a = (1 + 2, 3 + 4);  
  
alert( a ); // 7
```

Первое выражение  $1 + 2$  выполняется, а результат отбрасывается. Затем идёт  $3 + 4$ , выражение выполняется и возвращается результат.

Оператор запятая имеет очень низкий приоритет, приоритет которого ниже  $=$ , поэтому скобки важны в приведённом примере выше. Без них в  $a = 1 + 2, 3 + 4$  сначала выполнится  $+$ , суммируя числа в  $a = 3, 7$ , затем оператор присваивания  $=$  присвоит  $a = 3$ , а то что идёт дальше, будет игнорировано. Всё так же, как в  $(a = 1 + 2), 3 + 4$ .

Этот оператор иногда используют в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:

```
// три операции в одной строке  
for (a = 1, b = 3, c = a * b; a < 10; a++) {  
    ...  
}
```

Такое написание кода используется во многих JavaScript-фреймворках и о нем стоит знать. Но обычно это не улучшает читабельность кода, поэтому этот оператор не рекомендуется использовать.

## 8. Операторы сравнения.

Операторы сравнения известные из математики:

- Больше/меньше:  $a > b, a < b$ .
- Больше/меньше или равно:  $a \geq b, a \leq b$ .
- 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  $\neq$ . В JavaScript записывается как знак равенства с предшествующим ему восклицательным знаком:  $a != b$ .

Операторы сравнения, как и другие операторы, возвращают значение. Это значение имеет логический тип:

- `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
```

## 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( 'Кот' > 'Код' ); // true
alert( 'Сонный' > 'Сон' ); // true
```

Алгоритм сравнения двух строк довольно прост:

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
4. Сравнение продолжается, пока не закончится одна из строк.
5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

В примерах выше сравнение 'Я' > 'А' завершится на первом шаге, тогда как строки "Кот" и "Код" будут сравниваться посимвольно:

1. К равна К.
2. о равна о.
3. т больше чем д. На этом сравнение заканчивается. Первая строка больше.

Строчная "а" больше заглавной буквы "А". Потому что строчные буквы имеют больший код во внутренней таблице кодирования (Unicode), которую использует JavaScript.

## 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.

### 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:

```
alert( 0 === false ); // false, так как сравниваются разные типы
```

Оператор строгого равенства делает код более очевидным и оставляет меньше мест для ошибок.

Ещё есть оператор строгого неравенства `!==`, аналогичный `!=`.

### Сравнение с null и undefined

Сравнение null и undefined между собой и с другими значениями возвращает неожиданные результаты:

- 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.

Сравнение null с нулём:

```
alert( null > 0 ); // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

Результат последнего сравнения говорит о том, что "null больше или равно нулю", тогда результат одного из сравнений выше должен быть true, но они оба ложны. Причина в том, что нестрогое равенство и сравнения >, <, >=, <= работают по-разному. Сравнения преобразуют null в число, рассматривая его как 0. Поэтому выражение (3) null >= 0 истинно, а null > 0 ложно.

С другой стороны, для нестрогого равенства == значений undefined и null действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому (2) null == 0 ложно.

Значение undefined несравнимо с другими значениями:

```
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

Сравнение undefined с нулём всегда ложно, по следующим причинам:

- Сравнения (1) и (2) возвращают false, потому что undefined преобразуется в NaN, а NaN – это специальное числовое значение, которое возвращает false при любых сравнениях.
- Нестрогое равенство (3) возвращает false, потому что undefined равно только null и ни чему больше.

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

## 9. Модальные окна

Рассмотрим базовые UI операции: alert, prompt и confirm, которые позволяют работать с данными, полученными от пользователя.

### alert

Синтаксис:

```
alert(сообщение)
```

alert выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «ОК».

```
alert( "Привет" );
```

Окно сообщения, которое выводится, является *модальным окном*. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберётся с окном. В данном случае – пока не нажмёт на «ОК».

## prompt

Функция prompt принимает два аргумента:

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

Она выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL. Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием Esc на клавиатуре. Вызов prompt возвращает то, что ввёл посетитель – строку или специальное значение null, если ввод отменён. Как и в случае с alert, окно prompt модальное.

```
var years = prompt('Сколько вам лет?', 100);  
  
alert('Вам ' + years + ' лет!')
```

Всегда указывайте default. Второй параметр может отсутствовать. Однако при этом браузер IE вставит в диалог значение по умолчанию "undefined". Поэтому рекомендуется *всегда* указывать второй аргумент, хотябы пустую строку.

## confirm

Синтаксис:

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

confirm выводит окно с вопросом question с двумя кнопками: OK и CANCEL. Результатом будет true при нажатии OK и false – при CANCEL (Esc). Например:

```
var isAdmin = confirm("Вы - администратор?");  
  
alert( isAdmin );
```

## 10. Условные операторы.

Чтобы выполнить различные действия в зависимости от условий, нам нужно использовать оператор if и условный оператор ?, который также называют «оператор вопросительный знак».

### Оператор «if»

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

```
let year = prompt('В каком году появилась 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 надо 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 оператор `if`, даже если выполняется только одна команда. Это улучшает читаемость кода.

### Преобразование к логическому типу

Оператор `if (...)` вычисляет выражение в скобках и преобразует результат к логическому типу. Вспомним правила преобразования типов:

- Число 0, пустая строка `""`, `null`, `undefined` и `NaN` становятся `false`. Из-за этого их называют «ложными» значениями.
  - Остальные значения становятся `true`, поэтому их называют «правдивыми».
- Таким образом, код при таком условии никогда не выполнится:

```
if (0) {  
    ...  
}
```

А при таком – выполнится всегда:

```
if (1) {  
    ...  
}
```

Также можно передать заранее вычисленное в переменной логическое значение в `if`, например, так:

```
let cond = (year == 2015); // преобразуется к true или false  
  
if (cond) {  
    ...  
}
```

### Блок «else»

Оператор `if` может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно. Например:

```
let year = prompt('В каком году появилась 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`. Например:

```
let year = prompt('В каком году появилась 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.

### Условный оператор „?“

Иногда нужно назначить 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 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 – значение 2. 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 читабельным, поэтому рекомендуется их использовать.

В примере выше, можно избежать использования оператора вопросительного знака ?, т.к. сравнение само по себе уже возвращает true/false:

```
// то же самое
let accessAllowed = age > 18;
```

### Несколько операторов „?“

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

```
let age = prompt('Возраст?', 18);

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

alert( message );
```

Поначалу может быть сложно понять, что происходит. Но при ближайшем рассмотрении видно, что это обычная последовательная проверка:

1. Первый знак вопроса проверяет `age < 3`.
2. Если верно – возвращает 'Здравствуй, малыш!'. В противном случае, проверяет выражение после двоеточия „:““, вычисляет `age < 18`.
3. Если это верно – возвращает 'Привет!'. В противном случае, проверяет выражение после следующего двоеточия „:““, вычисляет `age < 100`.
4. Если это верно – возвращает 'Здравствуйте!'. В противном случае, возвращает выражение после последнего двоеточия – 'Какой необычный возраст!'.

Те же проверки при использовании `if..else`:

```
if (age < 3) {
  message = 'Здравствуй, малыш!';
} else if (age < 18) {
  message = 'Привет!';
} else if (age < 100) {
  message = 'Здравствуйте!';
} else {
  message = 'Какой необычный возраст!';
}
```

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. Не рекомендуется использовать оператор вопросительного знака таким образом, так как она менее читабельна.

Тот же код, использующий `if`:

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

if (company == 'Netscape') {
  alert('Верно!');
} else {
  alert('Неправильно.');
```

## 11. Логические операторы.

В JavaScript есть три логических оператора: `||` (ИЛИ), `&&` (И) и `!` (НЕ). 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 противной ситуации возвращается `false`. В JavaScript этот оператор работает несколько иным образом.

Существует всего четыре возможные логические комбинации:

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

Как видно, 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) {
  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.

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 (т.к. все ложно, возвращается последнее значение)
```

Такой принцип действия позволяет применять оператор нетрадиционным способом для решения некоторых задач. Например, чтобы получить первое истинное значение из списка переменных или выражений. Допустим имеется ряд переменных, которые могут содержать данные или быть `null/undefined`. С помощью || можно найти первую переменную с данными:

```
let currentUser = null;
let defaultUser = "John";

let name = currentUser || defaultUser || "unnamed";

alert( name ); // выбирается "John" – первое истинное значение
```

Еще один пример использования ИЛИ для сокращённого вычисления. 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 не достаточно 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
```

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.

## **&& (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( "won't work, because the result is falsy" );
}
```

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. Примеры:

```
// Если первый операнд истинный,
// И возвращает второй:
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5

// Если первый операнд ложный,
// И возвращает его. Второй операнд игнорируется
alert( null && 5 ); // null
alert( 0 && "no matter what" ); // 0
```

Можно передать несколько значений подряд. В таком случае возвратится первое «ложное» значение, на котором остановились вычисления.

```
alert( 1 && 2 && null && 3 ); // null
```

Когда все значения верны, возвращается последнее

```
alert( 1 && 2 && 3 ); // 3
```

Приоритет оператора И && больше, чем ИЛИ ||, поэтому 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. Например:

```
let x = 1;

(x > 0) && alert( 'Greater than zero!' );
```

Действие в правой части && выполнится только в том случае, если до него дойдут вычисления. То есть, alert сработает, если в левой части `(x > 0)` будет true:

```
let x = 1;

if (x > 0) {
```

```
    alert( 'Greater than zero!' );  
}
```

Как правило, вариант с `if` лучше читается и воспринимается. Он более очевиден, поэтому лучше использовать его.

## ! (НЕ)

Оператор НЕ представлен восклицательным знаком `!`. Синтаксис:

```
result = !value;
```

Оператор принимает один аргумент и выполняет следующие действия:

- Сначала приводит аргумент к логическому типу `true/false`.
- Затем возвращает противоположное значение.

Например:

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

Двойное НЕ используют для преобразования значений к логическому типу:

```
alert( !! "non-empty string" ); // true  
alert( !! null ); // false
```

Первое НЕ преобразует значение в логическому типу и возвращает обратное, а второе НЕ снова инвертирует его. В результате получится простое преобразование значения в логическое. С помощью встроенной функции `Boolean` можно сделать то же самое:

```
alert( Boolean("non-empty string") ); // true  
alert( Boolean(null) ); // false
```

Приоритет НЕ `!` является наивысшим из всех логических операторов, поэтому он всегда выполняется первым, перед `&&` или `||`.

## ?? (нулевое слияние)

Оператор нулевого слияния `??` это логический оператор, который возвращает значение правого операнда когда значение левого операнда равно `null` или `undefined`, в противном случае будет возвращено значение левого операнда.

В отличие от логического ИЛИ (`||`), левая часть оператора вычисляется и возвращается даже если его результат после приведения к логическому типу оказывается ложным, но не является `null` или `undefined`. Другими словами, если вы используете `||` чтобы установить значение по умолчанию, вы можете столкнуться с неожиданным поведением если считаете некоторые ложные значения пригодными для использования (например, `""` или `0`). Примеры:

```
const foo = null ?? 'default string';  
console.log(foo); // "default string"  
  
const baz = 0 ?? 42;
```



```
console.log(baz); // 0
```

## 12. Циклы while, 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*.

### Цикл «while»

Цикл while имеет следующий синтаксис:

```
while (condition) {  
    // код – тело 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 *итерация*. Цикл в примере выше совершает три итерации. Если бы строка i++ отсутствовала, то цикл бы повторялся (в теории) вечно. Но в действительности браузер 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--;  
}
```

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

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

### Цикл «do...while»

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

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

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

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

Цикл `do..while` стоит использовать, если необходимо, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным. На практике чаще используется форма с предусловием: `while(...) {...}`.

### Цикл «for»

Более сложный, но при этом самый распространённый цикл – цикл `for`. Синтаксис:

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

Рассмотрим пример. 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);  
}
```

То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке *условия*, после которой выполняется *тело* и *шаг*.

В примере переменная счётчика `i` была объявлена прямо в цикле. Это так называемое встроенное объявление переменной. Такие переменные видны 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 можно использовать уже существующую:

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

```
alert(i); // 3, переменная доступна
```

Любая часть `for` может быть пропущена. Например, можно пропустить начало если ничего не нужно делать перед стартом цикла:

```
let i = 0;

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

Можно убрать шаг, это сделает цикл аналогичным `while (i < 3)`:

```
let i = 0;

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

Можно убрать всё, получив бесконечный цикл:

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

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

### Прерывание цикла: «`break`»

Обычно цикл завершается при вычислении *условия* в `false`. Но можно 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`. Использование директивы `break` в бесконечном цикле удобно в том случае, если условие, по которому нужно прерваться, находится не в начале или конце цикла, а посередине.

## Переход к следующей итерации: `continue`

При выполнении директивы `continue` цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно `true`). Её используют, если известно, что в текущей итерации цикла нет необходимости. Например, цикл ниже использует `continue`, чтобы выводить только нечётные значения:

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

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

Директива `continue` позволяет избегать вложенности. Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

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

Но здесь появился дополнительный уровень вложенности. Если код внутри `if` более длинный, то это ухудшает читаемость, в отличие от варианта с `continue`.

Эти синтаксические конструкции `break/continue` не являются выражениями и не могут быть использованы с тернарным оператором `?`. Это приведёт к синтаксической ошибке. Например:

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

## 13. 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*. Синтаксис:

```
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:

```
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. Пример без break:

```
let a = 2 + 2;

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

В примере выше последовательно выполняются три alert:

```
alert( 'В точку!' );
alert( 'Перебор' );
alert( "Нет таких значений" );
```

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 результатом выражения +a будет 1, что совпадает с выражением b + 1 в case, и, следовательно, код в этом блоке будет выполнен.

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. 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.

Стоит отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство. Для примера, 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' );
}

```

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

## 14. Функции.

Зачастую нам надо повторять одно и то же действие во многих частях программы. Чтобы не повторять один и тот же код во многих местах, придуманы функции. Примеры встроенных функций: alert(message), prompt(message, default) и confirm(question). Можно создавать и свои.

### Объявление функции (Function Declaration)

Для создания функций можно использовать объявление функции. Такой синтаксис называется *Function Declaration*. 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 примере выше он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

```

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

```

Функция может быть вызвана по её имени: showMessage(). Например:

```

function showMessage() {

```



```
    alert( 'Всем привет!' );
}

showMessage();
showMessage();
```

Вызов `showMessage()` выполняет код функции. В результате появится 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.

Переменные объявленные внутри функции, видны только внутри этой функции и являются *локальными*. 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
```

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

showMessage();

alert( userName ); // Петя
```

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

```

Переменные, объявленные снаружи всех функций, такие как внешняя переменная `userName` в коде выше — 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 их нет или они используются редко.

## Параметры

Можно передать внутрь функции любую информацию, используя параметры (также называемые аргументы функции). В примере ниже 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.

В примере ниже, есть переменная `from`, и она передаётся функции. Функция изменяет значение `from`, но это изменение не видно снаружи, так она получает только копию значения:

```

function showMessage(from, text) {
  from = '*' + from + '*';
  alert( from + ': ' + text );
}

let from = "Аня";

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

alert( from ); // Аня

```

## 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 необходимо задать параметру `text` значение по умолчанию, то надо 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  
}
```

### Вычисление параметров по умолчанию

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра. В примере выше `anotherFunction()` будет вызываться каждый раз, когда `showMessage()` вызывается без параметра `text`.

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах. Например, явная проверка на `undefined` или с помощью оператора `||`:

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

```
function showMessage(from, text) {  
    // Если значение text ложно, тогда присвоить параметру text значение по умолчанию  
    text = text || 'текст не добавлен';  
    ...  
}
```

## 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 то же самое, что она возвращает undefined:

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

Пустой return аналогичен return undefined:

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

Для длинного выражения в return не стоит добавлять перевод строки между return и его значением, например так:

```
return  
(some + long + expression + or + whatever * f(a) + f(b))
```

Код не выполнится, потому что интерпретатор JavaScript подставит точку с запятой после return. Для него это будет выглядеть так:

```
return;  
(some + long + expression + or + whatever * f(a) + f(b))
```

Таким образом, это фактически стало пустым return. Если необходимо, чтобы возвращаемое выражение занимало несколько строк, то нужно начать его на той же строке, что и return. Или, хотя бы, поставить там открывающую скобку:

```
return (  
    some + long + expression  
    + or +  
    whatever * f(a) + f(b)  
)
```

## 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. Примеры префиксов:

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

Примеры имён функций с префиксами:

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

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. Например, функция `getAge` должна только возвращать возраст, а не выводить `alert` с возрастом; `createForm` — должна только создавать форму и возвращать её, а не изменять документ, добавляя форму в него; `checkPermission` — должна только выполнять проверку и возвращать её результат, а не отображать сообщение с текстом доступ разрешён/запрещён и т.д.

Имена функций, которые используются очень часто, иногда делают сверхкороткими. Например, во фреймворке `jQuery` есть функция с именем `$`. В библиотеке `Lodash` основная функция представлена именем `_`. Это исключения. В основном имена функций должны быть в меру краткими и описательными.

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Небольшие функции не только облегчают тестирование и отладку, но и являются хорошим комментарием. Например, сравним ниже две функции `showPrimes(n)`. Каждая из них выводит простое число до  $n$ .

Первый вариант использует метку `nextPrime`:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert( i ); // простое
  }
}
```

Второй вариант использует дополнительную функцию `isPrime(n)` для проверки на простое:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i); // простое
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

Второй вариант легче для понимания. Не надо разбираться с кодом, сразу видно название действия (`isPrime`). Разработчики называют такой код самодокументируемым.

Таким образом, рекомендуется создавать функции даже если не планируется повторно использовать их. Такие функции структурируют код и делают его более понятным.

## 15. Функциональные выражения и функции-стрелки

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

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

В коде выше функция создаётся и явно присваивается переменной, как любое другое значение. Независимо от того, как определена функция (Function Expression или Function Declaration), это просто значение, хранимое в переменной sayHi. Можно даже вывести это значение с помощью alert:

```
function sayHi() {  
    alert( "Привет" );  
}  
  
alert( sayHi ); // выведет код функции
```

Обратите внимание, что последняя строка не вызывает функцию sayHi, так как после её имени нет круглых скобок. В JavaScript функции – это значения, поэтому и обращаться с ними, надо как со значениями. Код выше выведет строковое представление функции, которое является её исходным кодом.

С функцией можно делать то же самое, что и с любым другим значением. Например, скопировать функцию в другую переменную:

```
function sayHi() {    // (1)  
    alert( "Привет" );  
}  
  
let func = sayHi;      // (2)  
  
func(); // Привет     // (3)  
sayHi(); // Привет
```

Рассмотрим пример подробнее:

1. Объявление Function Declaration (1) создало функцию и присвоило её значение переменной с именем sayHi.
2. В строке (2) её значение скопировано в переменную func. Обратите 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().

Можно использовать и Function Expression для того, чтобы создать sayHi в первой строке. Результат будем таким же:



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

Заметьте, что в Function Expression ставится точка с запятой ; в конце, а в Function Declaration нет:

```
function sayHi() {  
    // ...  
}  
  
let sayHi = function() {  
    // ...  
};
```

Это потому, что Function Expression использует внутри себя инструкции присваивания `let sayHi = ...`; как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой не относится непосредственно к Function Expression, она лишь завершает инструкцию.

### Функции-«колбэки»

Рассмотрим 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». В браузерах такие функции обычно отображают красивые диалоговые окна. Функция задает 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." );  
}  
  
ask("Вы согласны?", showOk, showCancel);
```

Аргументы функции `ask` ещё называют функциями-колбэками или просто колбэками (от англ. «call back» – обратный вызов). Т.е. функция передаётся в качестве аргумента и вызывается обратно тогда, когда это необходимо. В нашем случае, `showOk` становится колбэком для ответа «yes», а `showCancel` – для ответа «no».

Можно переписать пример 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 имён, поэтому они называются анонимными. Такие функции недоступны снаружи ask (потому что они не присвоены переменным).

Одним из отличий Function Expression от Function Declaration, кроме синтаксиса, является то, что Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться. После того, как поток выполнения достигнет правой части выражения присваивания `let sum = function...` — с этого момента, функция считается созданной и может быть использована (присвоена переменной, вызвана и т.д.).

Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке). Другими словами, когда движок JavaScript готовится выполнять скрипт или блок кода, прежде всего он ищет в нём Function Declaration и создаёт все такие функции. Обычно этот процесс называют «стадией инициализации». И только после того, как все объявления Function Declaration будут обработаны, продолжится выполнение. В результате, функции, созданные, как Function Declaration могут быть вызваны раньше своих определений. Например, так будет работать:

```
sayHi("Вася"); // Привет, Вася

function sayHi(name) {
  alert( `Привет, ${name}` );
}
```

Функция sayHi была создана, когда движок JavaScript подготавливал скрипт к выполнению, и такая функция видна повсюду в этом скрипте. Если бы это было Function Expression, то такой код привел бы к ошибке:

```
sayHi("Вася");

let sayHi = function(name) {
  alert( `Привет, ${name}` );
};
```

Ещё одна важная особенность Function Declaration заключается в их блочной области видимости. В строгом режиме, когда Function Declaration находится в блоке {...}, функция доступна везде внутри блока, но не снаружи него. Для примера представим, что нужно создать функцию welcome() в зависимости от

значения переменной `age`, которое будет получено во время выполнения кода. Такой код, использующий Function Declaration, работать не будет:

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

// в зависимости от условия объявляем функцию
if (age < 18) {

  function welcome() {
    alert("Привет!");
  }

} else {

  function welcome() {
    alert("Здравствуйте!");
  }

}

welcome(); // Error: welcome is not defined
```

Это произошло, так как объявление Function Declaration видимо только внутри блока кода, в котором располагается.

Верным подходом будет воспользоваться функцией, объявленной при помощи Function Expression, и присвоить значение `welcome` переменной, объявленной снаружи `if`, что обеспечит нужную видимость. Такой код работает как требуется:

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

let welcome;

if (age < 18) {

  welcome = function() {
    alert("Привет!");
  };

} else {

  welcome = function() {
    alert("Здравствуйте!");
  };

}

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

Можно упростить этот код ещё больше, используя условный оператор `?:`:

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

let welcome = (age < 18) ?
```

```
function() { alert("Привет!"); } :  
function() { alert("Здравствуйте!"); };  
  
welcome();
```

## Функции-стрелки

Существует ещё 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 создаёт функцию func с аргументами arg1..argN и вычисляет expression с правой стороны с их использованием, возвращая результат. Это то же самое, что и:

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

Рассмотрим пример:

```
let sum = (a, b) => a + b;  
  
alert( sum(1, 2) ); // 3
```

Если у передается только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:

```
let double = n => n * 2;  
  
alert( double(3) ); // 6
```

Если нет аргументов, используются пустые круглые скобки (их указывать обязательно):

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

В примерах выше аргументы использовались слева от =>, а справа вычислялось выражение с их значениями. Но если требуется вычислить несколько выражений или инструкций, то необходимо заключить такие выражения в фигурные скобки с использованием директивы return внутри них, как в обычной функции. Например:

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

Важной особенностью стрелочных функций является то, что у них нет переменной `arguments`.