

Здесь вы можете изучить JavaScript, начиная с нуля и заканчивая продвинутыми концепциями вроде ООП.

Мы сосредоточимся на самом языке, изредка добавляя заметки о средах его исполнения.

Введение

Про язык JavaScript и окружение для разработки на нём.

Введение в JavaScript

Давайте посмотрим, что такого особенного в JavaScript, чего можно достичь с его помощью и какие другие технологии хорошо с ним работают.

Что такое JavaScript?

Изначально JavaScript был создан, чтобы «сделать веб-страницы живыми».

Программы на этом языке называются *скриптами*. Они могут встраиваться в HTML и выполняться автоматически при загрузке веб-страницы.

Скрипты распространяются и выполняются, как простой текст. Им не нужна специальная подготовка или компиляция для запуска.

Это отличает JavaScript от другого языка – [Java](#) .

Почему JavaScript?

Когда JavaScript создавался, у него было другое имя – «LiveScript». Однако, язык Java был очень популярен в то время, и было решено, что позиционирование JavaScript как «младшего брата» Java будет полезно.

Со временем JavaScript стал полностью независимым языком со своей собственной спецификацией, называющейся [ECMAScript](#) , и сейчас не имеет никакого отношения к Java.

Сегодня JavaScript может выполняться не только в браузере, но и на сервере или на любом другом устройстве, которое имеет специальную программу, называющуюся «движком» [JavaScript](#) .

У браузера есть собственный движок, который иногда называют «виртуальная машина JavaScript».

Разные движки имеют разные «кодовые имена». Например:

- [V8](#) – в Chrome, Opera и Edge.
- [SpiderMonkey](#) – в Firefox.
- ...Ещё есть «Chakra» для IE, «JavaScriptCore», «Nitro» и «SquirrelFish» для Safari и т.д.

Эти названия полезно знать, так как они часто используются в статьях для разработчиков. Мы тоже будем их использовать. Например, если «функциональность X поддерживается V8», тогда «X», скорее всего, работает в Chrome, Opera и Edge.

Как работают движки?

Движки сложны. Но основы понять легко.

1. Движок (встроенный, если это браузер) читает («парсит») текст скрипта.
2. Затем он преобразует («компилирует») скрипт в машинный язык.
3. После этого машинный код запускается и работает достаточно быстро.

Движок применяет оптимизации на каждом этапе. Он даже просматривает скомпилированный скрипт во время его работы, анализируя проходящие через него данные, и применяет оптимизации к машинному коду, полагаясь на полученные знания. В результате скрипты работают очень быстро.

Что может JavaScript в браузере?

Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не требующих этого.

Возможности JavaScript сильно зависят от окружения, в котором он работает. Например, [Node.js](#) поддерживает функции чтения/записи произвольных файлов, выполнения сетевых запросов и т.д.

В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером.

Например, в браузере JavaScript может:

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

Чего НЕ может JavaScript в браузере?

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

Современные браузеры позволяют ему работать с файлами, но с ограниченным доступом, и предоставляют его, только если пользователь выполняет определённые

действия, такие как «перетаскивание» файла в окно браузера или его выбор с помощью тега `<input>`.

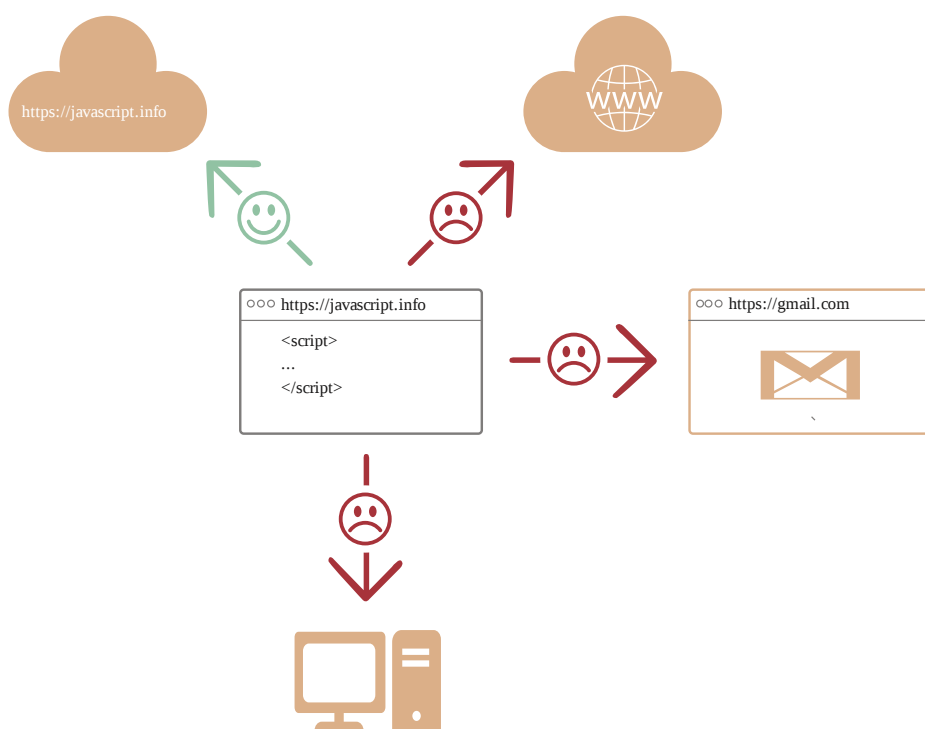
Существуют способы взаимодействия с камерой/микрофоном и другими устройствами, но они требуют явного разрешения пользователя. Таким образом, страница с поддержкой JavaScript не может незаметно включить веб-камеру, наблюдать за происходящим и отправлять информацию в [ФСБ](#).

- Различные окна/вкладки не знают друг о друге. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае JavaScript с одной страницы не имеет доступа к другой, если они пришли с разных сайтов (с другого домена, протокола или порта).

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

Это ограничение необходимо, опять же, для безопасности пользователя. Страница `https://anysite.com`, которую открыл пользователь, не должна иметь доступ к другой вкладке браузера с URL `https://gmail.com` и воровать информацию оттуда.

- JavaScript может легко взаимодействовать с сервером, с которого пришла текущая страница. Но его способность получать данные с других сайтов/доменов ограничена. Хотя это возможно в принципе, для чего требуется явное согласие (выраженное в заголовках HTTP) с удалённой стороной. Опять же, это ограничение безопасности.



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

Что делает JavaScript особенным?

Как минимум, *три* сильные стороны JavaScript:

- Полная интеграция с HTML/CSS.
- Простые вещи делаются просто.
- Поддерживается всеми основными браузерами и включён по умолчанию.

JavaScript – это единственная браузерная технология, сочетающая в себе все эти три вещи.

Вот что делает JavaScript особенным. Вот почему это самый распространённый инструмент для создания интерфейсов в браузере.

Хотя, конечно, JavaScript позволяет делать приложения не только в браузерах, но и на сервере, на мобильных устройствах и т.п.

Языки «над» JavaScript

Синтаксис JavaScript подходит не под все нужды. Разные люди хотят иметь разные возможности.

Это естественно, потому что проекты разные и требования к ним тоже разные.

Так, в последнее время появилось много новых 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 «под капотом».

Примеры таких языков:

- [CoffeeScript](#) ➔ добавляет «синтаксический сахар» для JavaScript. Он вводит более короткий синтаксис, который позволяет писать чистый и лаконичный код. Обычно такое нравится Ruby-программистам.
- [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](#) ➔ стоит особняком, потому что 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 вышеперечисленных языков.
- [Brython](#) ➔ транспирует Python в JavaScript, что позволяет писать приложения на чистом Python без JavaScript.

Есть и другие. Но даже если мы используем один из этих языков, мы должны знать JavaScript, чтобы действительно понимать, что мы делаем.

Итого


- JavaScript изначально создавался только для браузера, но сейчас используется на многих других платформах.

- Сегодня JavaScript занимает уникальную позицию в качестве самого распространённого языка для браузера, обладающего полной интеграцией с HTML/CSS.
- Многие языки могут быть «транспирированы» в JavaScript для предоставления дополнительных функций. Рекомендуется хотя бы кратко рассмотреть их после освоения JavaScript.


Справочники и спецификации


Эта книга является *учебником* и нацелена на то, чтобы помочь вам постепенно освоить язык. Но когда вы хорошо изучите основы, вам понадобятся дополнительные источники информации.

Спецификация

[Спецификация ECMA-262](#)  содержит самую глубокую, детальную и формализованную информацию о JavaScript. Она определяет сам язык.

Вначале спецификация может показаться тяжеловатой для понимания из-за слишком формального стиля изложения. Если вы ищете источник самой достоверной информации, то это правильное место, но она не для ежедневного использования.

Новая версия спецификации появляется каждый год. А пока она не вышла официально, все желающие могут ознакомиться с текущим черновиком на <https://tc39.es/ecma262/> .

Чтобы почитать о самых последних возможностях, включая те, которые «почти в стандарте» (так называемые «stage 3 proposals»), посетите <https://github.com/tc39/proposals> .


Если вы разрабатываете под браузеры, то существуют и другие спецификации, о которых рассказывается во [второй части](#) этого учебника.


Справочники

- **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> .

Хотя зачастую вместо их сайта удобнее использовать какой-нибудь интернет-поисковик, вводя там запрос «MDN [что вы хотите найти]», например <https://google.com/search?q=MDN+parseInt>  для поиска информации о функции `parseInt`.




- **MSDN** – справочник от Microsoft, содержащий много информации, в том числе по JavaScript (который там часто обозначается как JScript). Если вам нужно найти что-то специфическое по браузеру Internet Explorer, лучше искать там: <https://msdn.microsoft.com/> .

Так же, как и в предыдущем случае, можно использовать интернет-поиск, набирая фразы типа «RegExp MSDN» или «RegExp MSDN jscript».

Таблицы совместимости

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 с информацией о поддержке по каждой возможности языка. Например, чтобы узнать, какие движки поддерживают современные криптографические функции, посетите: <http://caniuse.com/#feat=cryptography> .
- <https://kangax.github.io/compat-table>  – таблица с возможностями языка и движками, которые их поддерживают и не поддерживают.

Все эти ресурсы полезны в ежедневной работе программиста, так как они содержат ценную информацию о возможностях использования языка, их поддержке и так далее.


Пожалуйста, запомните эти ссылки (или ссылку на эту страницу) на случай, когда вам понадобится подробная информация о какой-нибудь конкретной возможности JavaScript.


Редакторы кода

Большую часть своего рабочего времени программисты проводят в редакторах кода.



Есть два основных типа редакторов: IDE и «лёгкие» редакторы. Многие используют по одному инструменту каждого типа.


IDE

Термином [IDE](#)  (Integrated Development Environment, «интегрированная среда разработки») называют мощные редакторы с множеством функций, которые работают в рамках целого проекта. Как видно из названия, 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.

Если вы ещё не выбрали себе IDE, присмотритесь к этим:

- [Visual Studio Code](#)  (кросс-платформенная, бесплатная).
- [WebStorm](#)  (кросс-платформенная, платная).

Для Windows есть ещё Visual Studio (не путать с Visual Studio Code). Visual Studio – это платная мощная среда разработки, которая работает только на Windows. Она хорошо подходит для .NET платформы. У неё есть бесплатная версия, которая называется [Visual Studio Community](#) .

Многие IDE платные, но у них есть пробный период. Их цена обычно незначительна по сравнению с зарплатой квалифицированного разработчика, так что попробуйте и выбирайте ту, что вам подходит лучше других.

«Лёгкие» редакторы






«Лёгкие» редакторы менее мощные, чем IDE, но они отличаются скоростью, удобным интерфейсом и простотой.

В основном их используют для того, чтобы быстро открыть и отредактировать нужный файл.

Главное отличие между «лёгким» редактором и IDE состоит в том, что IDE работает на уровне целого проекта, поэтому она загружает больше данных при запуске, анализирует структуру проекта, если это необходимо, и так далее. Если вы работаете только с одним файлом, то гораздо быстрее открыть его в «лёгком» редакторе.

На практике «лёгкие» редакторы могут иметь множество плагинов, включая автодополнение и анализаторы синтаксиса на уровне директории, поэтому границы между IDE и «лёгкими» редакторами размыты.

Следующие варианты заслуживают вашего внимания:

- [Atom](#)  (кроссплатформенный, бесплатный).
- [Sublime Text](#)  (кроссплатформенный, условно-бесплатный).
- [Notepad++](#)  (Windows, бесплатный).
- [Vim](#)  и [Emacs](#)  тоже хороши, если знать, как ими пользоваться.


Не будем ссориться

Редакторы, перечисленные выше, известны автору давно и заслужили много хороших отзывов от коллег.

Конечно же, есть много других отличных редакторов. Выбирайте тот, который вам больше нравится.

Выбор редактора, как и любого другого инструмента, индивидуален и зависит от ваших проектов, привычек и личных предпочтений.

Консоль разработчика

Код уязвим для ошибок. И вы, скорее всего, будете делать ошибки в коде... Впрочем, давайте будем откровенны: вы *точно* будете совершать ошибки в коде. В конце концов, вы человек, а не [робот](#) .

Но по умолчанию в браузере ошибки не видны. То есть, если что-то пойдёт не так, мы не увидим, что именно сломалось, и не сможем это починить.

Для решения задач такого рода в браузер встроены так называемые «Инструменты разработки» (Developer tools или сокращённо — devtools).

Chrome и Firefox снискали любовь подавляющего большинства программистов во многом благодаря своим отменным инструментам разработчика. Остальные браузеры, хотя и оснащены подобными инструментами, но всё же зачастую находятся в роли догоняющих и по качеству, и по количеству свойств и особенностей. В общем, почти у всех программистов есть свой «любимый» браузер. Другие используются только для отлова и исправления специфичных «браузерозависимых» ошибок.

Для начала знакомства с этими мощными инструментами давайте выясним, как их открывать, смотреть ошибки и запускать команды JavaScript.

Google Chrome

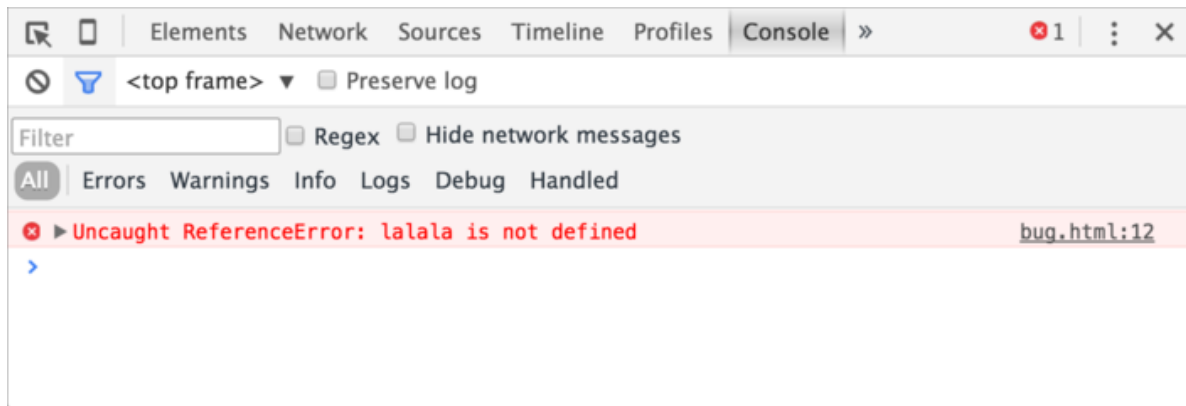
Откройте страницу [bug.html](#).

В её JavaScript-коде закралась ошибка. Она не видна обычному посетителю, поэтому давайте найдём её при помощи инструментов разработки.

Нажмите `F12` или, если вы используете Mac, `Cmd+Opt+J`.

По умолчанию в инструментах разработчика откроется вкладка Console (консоль).

Она выглядит приблизительно следующим образом:



Точный внешний вид инструментов разработки зависит от используемой версии Chrome. Время от времени некоторые детали изменяются, но в целом внешний вид остаётся примерно похожим на предыдущие версии.

- В консоли мы можем увидеть сообщение об ошибке, отрисованное красным цветом. В нашем случае скрипт содержит неизвестную команду «lalala».
- Справа присутствует ссылка на исходный код `bug.html:12` с номером строки кода, в которой эта ошибка и произошла.

Под сообщением об ошибке находится синий символ `>`. Он обозначает командную строку, в ней мы можем редактировать и запускать JavaScript-команды. Для их запуска нажмите `Enter`.

i Многострочный ввод

Обычно при нажатии `Enter` введённая строка кода сразу выполняется.

Чтобы перенести строку, нажмите `Shift+Enter`. Так можно вводить более длинный JS-код.

Теперь мы явно видим ошибки, для начала этого вполне достаточно. Мы ещё вернёмся к инструментам разработчика позже и более подробно рассмотрим отладку кода в главе [Отладка в браузере](#).

Firefox, Edge и другие

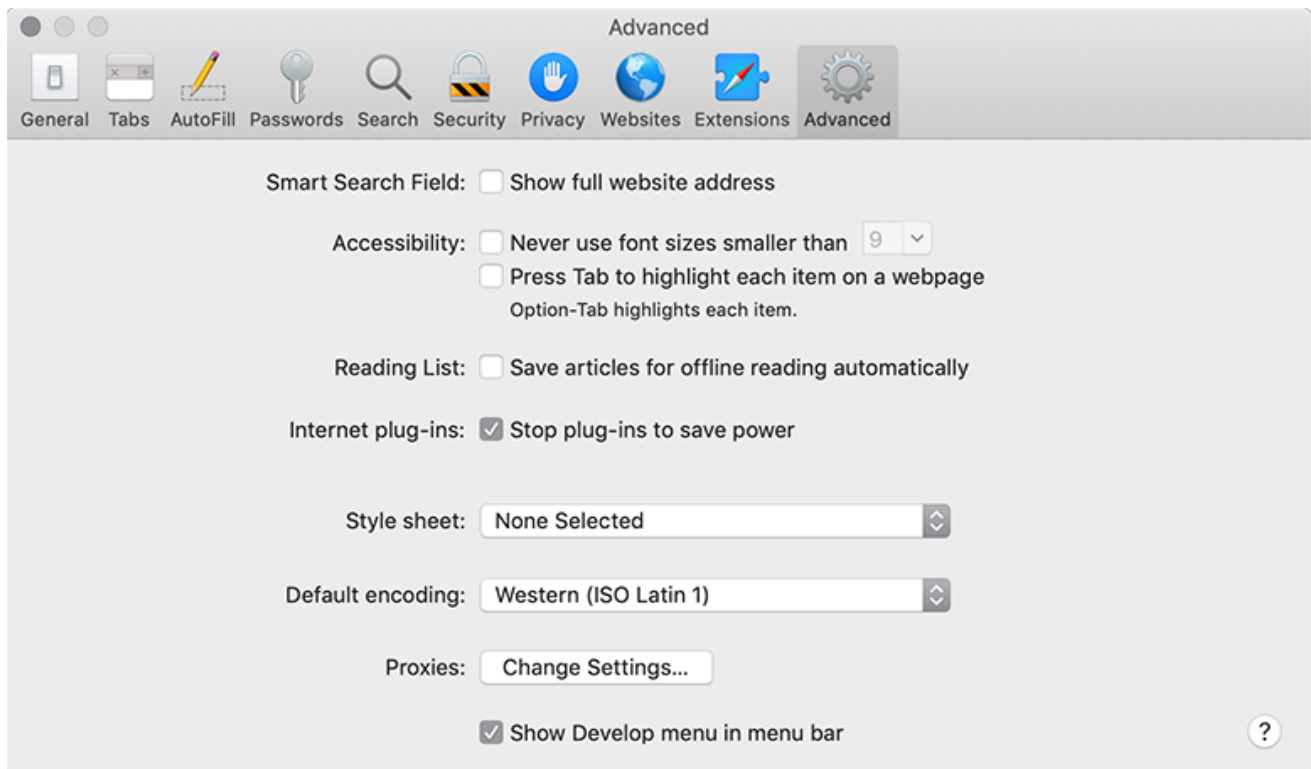
Инструменты разработчика в большинстве браузеров открываются при нажатии на `F12`.

Их внешний вид и принципы работы мало чем отличаются. Разобравшись с инструментами в одном браузере, вы без труда сможете работать с ними и в другом.

Safari

Safari (браузер для Mac, не поддерживается в системах Windows/Linux) всё же имеет небольшое отличие. Для начала работы нам нужно включить «Меню разработки» («Developer menu»).

Откройте Настройки (Preferences) и перейдите к панели «Продвинутые» (Advanced). В самом низу вы найдёте чекбокс:



Теперь консоль можно активировать нажатием клавиш `Cmd+Opt+C`. Также обратите внимание на новый элемент меню «Разработка» («Develop»). В нем содержится большое количество команд и настроек.

Итого

- Инструменты разработчика позволяют нам смотреть ошибки, выполнять команды, проверять значение переменных и ещё много всего полезного.
- В большинстве браузеров, работающих под Windows, инструменты разработчика можно открыть, нажав `F12`. В Chrome для Mac используйте комбинацию `Cmd+Opt+J`, Safari: `Cmd+Opt+C` (необходимо предварительное включение «Меню разработчика»).

Теперь наше окружение полностью настроено. В следующем разделе мы перейдём непосредственно к JavaScript.

Основы JavaScript

Давайте изучим основы создания скриптов.

Привет, мир!

В этой части учебника мы изучаем собственно JavaScript, сам язык.

Но нам нужна рабочая среда для запуска наших скриптов, и, поскольку это онлайн-книга, то браузер будет хорошим выбором. В этой главе мы сократим количество специфичных для браузера команд (например, `alert`) до минимума, чтобы вы не тратили на них время, если планируете сосредоточиться на другой среде (например, Node.js). А на использовании JavaScript в браузере мы сосредоточимся в [следующей части](#) учебника.

Итак, сначала давайте посмотрим, как выполнить скрипт на странице. Для серверных сред (например, Node.js), вы можете выполнить скрипт с помощью команды типа `"node my.js"`. Для браузера всё немного иначе.

Тег «script»

Программы на JavaScript могут быть вставлены в любое место HTML-документа с помощью тега `<script>`.

Для примера:

```
<!DOCTYPE HTML>
<html>

<body>

  <p>Перед скриптом...</p>

  <script>
    alert( 'Привет, мир!' );
  </script>

  <p>...После скрипта.</p>

</body>

</html>
```

Тег `<script>` содержит JavaScript-код, который автоматически выполнится, когда браузер его обработает.

Современная разметка

Тег `<script>` имеет несколько атрибутов, которые редко используются, но всё ещё могут встретиться в старом коде:

Атрибут `type`: `<script type=...>`

Старый стандарт HTML, HTML4, требовал наличия этого атрибута в теге `<script>`. Обычно он имел значение `type="text/javascript"`. На текущий момент этого больше не требуется. Более того, в современном стандарте HTML смысл этого атрибута полностью изменился. Теперь он может использоваться для JavaScript-модулей. Но это тема не для начального уровня, и о ней мы поговорим в другой части учебника.

Атрибут `language`: `<script language=...>`

Этот атрибут должен был задавать язык, на котором написан скрипт. Но так как JavaScript является языком по умолчанию, в этом атрибуте уже нет необходимости.

Обёртывание скрипта в HTML-комментарии.

В очень древних книгах и руководствах вы сможете найти комментарии внутри тега `<script>`, например, такие:

```
<script type="text/javascript">!--  
...  
//--></script>
```

Этот комментарий скрывал код JavaScript в старых браузерах, которые не знали, как обрабатывать тег `<script>`. Поскольку все браузеры, выпущенные за последние 15 лет, не содержат данной проблемы, такие комментарии уже не нужны. Если они есть, то это признак, что перед нами очень древний код.

Внешние скрипты

Если у вас много JavaScript-кода, вы можете поместить его в отдельный файл.

Файл скрипта можно подключить к HTML с помощью атрибута `src`:

```
<script src="/path/to/script.js"></script>
```

Здесь `/path/to/script.js` – это абсолютный путь до скрипта от корня сайта. Также можно указать относительный путь от текущей страницы. Например, `src="script.js"` или `src="./script.js"` будет означать, что файл `"script.js"` находится в текущей папке.

Можно указать и полный URL-адрес. Например:


```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"></script>
```

Для подключения нескольких скриптов используйте несколько тегов:

```
<script src="/js/script1.js"></script>  
<script src="/js/script2.js"></script>  
...
```

На заметку:

Как правило, только простейшие скрипты помещаются в HTML. Более сложные выделяются в отдельные файлы.

Польза отдельных файлов в том, что браузер загрузит скрипт отдельно и сможет хранить его в [кеше](#) .

Другие страницы, которые подключают тот же скрипт, смогут брать его из кеша вместо повторной загрузки из сети. И таким образом файл будет загружаться с сервера только один раз.

Это сокращает расход трафика и ускоряет загрузку страниц.

Если атрибут `src` установлен, содержимое тега `script` будет игнорироваться.

В одном теге `<script>` нельзя использовать одновременно атрибут `src` и код внутри.

Нижеприведённый пример не работает:

```
<script src="file.js">
  alert(1); // содержимое игнорируется, так как есть атрибут src
</script>
```

Нужно выбрать: либо внешний скрипт `<script src="...">`, либо обычный код внутри тега `<script>`.

Вышеприведённый пример можно разделить на два скрипта:

```
<script src="file.js"></script>
<script>
  alert(1);
</script>
```

Итого

- Для добавления кода JavaScript на страницу используется тег `<script>`
- Атрибуты `type` и `language` необязательны.
- Скрипт во внешнем файле можно вставить с помощью `<script src="path/to/script.js"></script>`.

Нам ещё многое предстоит изучить про браузерные скрипты и их взаимодействие со страницей. Но, как уже было сказано, эта часть учебника посвящена именно языку JavaScript, поэтому здесь мы постараемся не отвлекаться на детали реализации в браузере. Мы воспользуемся браузером для запуска JavaScript, это удобно для онлайн-демонстраций, но это только одна из платформ, на которых работает этот язык.

Задачи

Вызвать alert

важность: 5

Создайте страницу, которая отобразит сообщение «Я JavaScript!».

Выполните это задание в песочнице, либо на вашем жёстком диске, где – неважно, главное – проверьте, что она работает.

[Демо в новом окне](#) ↗

[К решению](#)

Покажите сообщение с помощью внешнего скрипта

важность: 5

Возьмите решение предыдущей задачи [Вызвать alert](#), и измените его. Извлеките содержимое скрипта во внешний файл `alert.js`, лежащий в той же папке.

Откройте страницу, убедитесь, что оповещение работает.

[К решению](#)

Структура кода

Начнём изучение языка с рассмотрения основных «строительных блоков» кода.

Инструкции

Инструкции – это синтаксические конструкции и команды, которые выполняют действия.

Мы уже видели инструкцию `alert('Привет, мир!')`, которая отображает сообщение «Привет, мир!».

В нашем коде может быть столько инструкций, сколько мы захотим. Инструкции могут отделяться точкой с запятой.

Например, здесь мы разделили сообщение «Привет Мир» на два вызова `alert`:

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

Обычно каждую инструкцию пишут на новой строке, чтобы код было легче читать:

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

Точка с запятой

В большинстве случаев точку с запятой можно не ставить, если есть переход на новую строку.

Так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

В этом случае JavaScript интерпретирует перенос строки как «неявную» точку с запятой. Это называется [автоматическая вставка точки с запятой](#) [↗](#).

В большинстве случаев новая строка подразумевает точку с запятой. Но «в большинстве случаев» не значит «всегда»!

В некоторых ситуациях новая строка всё же не означает точку с запятой. Например:

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

Код выведет `6`, потому что JavaScript не вставляет здесь точку с запятой. Интуитивно очевидно, что, если строка заканчивается знаком `"+"`, значит, это «незавершённое выражение», поэтому точка с запятой не требуется. И в этом случае всё работает, как задумано.

Но есть ситуации, где JavaScript «забывает» вставить точку с запятой там, где она нужна.

Ошибки, которые при этом появляются, достаточно сложно обнаруживать и исправлять.

Пример ошибки

Если вы хотите увидеть конкретный пример такой ошибки, обратите внимание на этот код:

```
alert('Hello');  
  
[1, 2].forEach(alert);
```

Пока нет необходимости знать значение скобок `[]` и `forEach`. Мы изучим их позже. Пока что просто запомните результат выполнения этого кода: выводится `Hello`, затем `1`, затем `2`.

А теперь давайте уберем точку с запятой после `alert`:

```
alert('Hello')  
  
[1, 2].forEach(alert);
```

Этот код отличается от кода, приведенного выше, только в одном: пропала точка с запятой в конце первой строки.

Если мы запустим этот код, выведется только первый `alert`, а затем мы получим ошибку (вам может потребоваться открыть консоль, чтобы увидеть её)!

Это потому что JavaScript не вставляет точку с запятой перед квадратными скобками `[...]`. И поэтому код в последнем примере выполняется, как одна инструкция.

Вот как движок видит его:

```
alert('Hello')[1, 2].forEach(alert);
```

Выглядит странно, правда? Такое слияние в данном случае неправильное. Мы должны поставить точку с запятой после `alert`, чтобы код работал правильно.

Это может произойти и в некоторых других ситуациях.

Мы рекомендуем ставить точку с запятой между инструкциями, даже если они отделены переносами строк. Это правило широко используется в сообществе разработчиков. Стоит отметить ещё раз – в большинстве случаев *можно* не ставить точку с запятой. Но безопаснее, особенно для новичка, ставить её.

Комментарии

Со временем программы становятся всё сложнее и сложнее. Возникает необходимость добавлять *комментарии*, которые бы описывали, что делает код и почему.

Комментарии могут находиться в любом месте скрипта. Они не влияют на его выполнение, поскольку движок просто игнорирует их.

Однострочные комментарии начинаются с двойной косой черты `//`.

Часть строки после `//` считается комментарием. Такой комментарий может как занимать строку целиком, так и находиться после инструкции.

Как здесь:

```
// Этот комментарий занимает всю строку
alert('Привет');

alert('Мир'); // Этот комментарий следует за инструкцией
```

Многострочные комментарии начинаются косой чертой со звёздочкой `/*` и заканчиваются звёздочкой с косой чертой `*/`.

Как вот здесь:

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

Содержимое комментария игнорируется, поэтому, если мы поместим код внутри `/* ... */`, он не будет исполняться.

Это бывает удобно для временного отключения участка кода:

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

Используйте горячие клавиши!

В большинстве редакторов строку кода можно закомментировать, нажав комбинацию клавиш `Ctrl+/` для однострочного комментария и что-то вроде `Ctrl+Shift+/` – для многострочных комментариев (выделите кусок кода и нажмите комбинацию клавиш). В системе Mac попробуйте `Cmd` вместо `Ctrl` и `Option` вместо `Shift`.

Вложенные комментарии не поддерживаются!

Не может быть `/* ... */` внутри `/* ... */`.

Такой код «умрёт» с ошибкой:

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

Не стесняйтесь использовать комментарии в своём коде.

Комментарии увеличивают размер кода, но это не проблема. Есть множество инструментов, которые минифицируют код перед публикацией на рабочий сервер. Они убирают комментарии, так что они не содержатся в рабочих скриптах. Таким образом, комментарии никоим образом не вредят рабочему коду.

Позже в учебнике будет глава [Качество кода](#), которая объяснит, как лучше писать комментарии.

Строгий режим — "use strict"

На протяжении долгого времени JavaScript развивался без проблем с обратной совместимостью. Новые функции добавлялись в язык, в то время как старая функциональность не менялась.

Преимуществом данного подхода было то, что существующий код продолжал работать. А недостатком — что любая ошибка или несовершенное решение, принятое создателями JavaScript, застревали в языке навсегда.

Так было до 2009 года, когда появился ECMAScript 5 (ES5). Он добавил новые возможности в язык и изменил некоторые из существующих. Чтобы устаревший код работал, как и раньше, по умолчанию подобные изменения не применяются. Поэтому нам нужно явно их активировать с помощью специальной директивы: `"use strict"`.

«use strict»

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

Например:

```
"use strict";

// этот код работает в современном режиме
...
```

Позже мы изучим функции (способ группировки команд). Забегая вперёд, заметим, что вместо всего скрипта `"use strict"` можно поставить в начале большинства видов функций. Это позволяет включить строгий режим только в конкретной функции. Но обычно люди используют его для всего файла.

⚠ Убедитесь, что «use strict» находится в начале

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

Здесь строгий режим не включён:

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

Над `"use strict"` могут быть записаны только комментарии.

⚠ Нет никакого способа отменить `use strict`

Нет директивы типа `"no use strict"`, которая возвращала бы движок к старому поведению.

Как только мы входим в строгий режим, отменить это невозможно.

Консоль браузера

В дальнейшем, когда вы будете использовать [консоль браузера](#) для тестирования функций, обратите внимание, что `use strict` по умолчанию в ней выключен.

Иногда, когда `use strict` имеет значение, вы можете получить неправильные результаты.

Итак, как можно включить `use strict` в консоли?

Можно использовать `Shift+Enter` для ввода нескольких строк и написать в верхней строке `use strict`:

```
'use strict'; <Shift+Enter для перехода на новую строку>  
// ...ваш код...  
<Enter для запуска>
```

В большинстве браузеров, включая Chrome и Firefox, это работает.

Если этого не происходит, например, в старом браузере, есть некрасивый, но надежный способ обеспечить `use strict`. Поместите его в следующую обёртку:

```
(function() {  
  'use strict';  
  
  // ...ваш код...  
})();
```

Всегда ли нужно использовать «use strict»?

Вопрос кажется риторическим, но это не так.

Кто-то посоветует начинать каждый скрипт с `"use strict"` ... Но есть способ покруче.

Современный JavaScript поддерживает «классы» и «модули» — продвинутые структуры языка (и мы, конечно, до них доберёмся), которые автоматически включают строгий режим. Поэтому в них нет нужды добавлять директиву `"use strict"`.

Подытожим: пока очень желательно добавлять `"use strict"`; в начале ваших скриптов. Позже, когда весь ваш код будет состоять из классов и модулей, директиву можно будет опускать.

Пока мы узнали о `use strict` только в общих чертах.

В следующих главах, по мере расширения знаний о возможностях языка, мы яснее увидим отличия между строгим и стандартным режимом. К счастью, их не так много, и все они делают жизнь разработчика лучше.

Все примеры в этом учебнике подразумевают исполнение в строгом режиме, за исключением случаев (очень редких), когда оговорено иное.

Переменные

JavaScript-приложению обычно нужно работать с информацией. Например:

1. Интернет-магазин — информация может включать продаваемые товары и корзину покупок.
2. Чат — информация может включать пользователей, сообщения и многое другое.

Переменные используются для хранения этой информации.

Переменная

[Переменная](#) — это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

Для создания переменной в JavaScript используйте ключевое слово `let`.

Приведённая ниже инструкция создаёт (другими словами: *объявляет* или *определяет*) переменную с именем «message»:

```
let message;
```

Теперь можно поместить в неё данные, используя оператор присваивания `=`:

```
let message;
```

```
message = 'Hello'; // сохранить строку 'Hello' в переменной с именем message
```

Строка сохраняется в области памяти, связанной с переменной. Мы можем получить к ней доступ, используя имя переменной:

```
let message;  
message = 'Hello!';  
  
alert(message); // показывает содержимое переменной
```

Для краткости можно совместить объявление переменной и запись данных в одну строку:

```
let message = 'Hello!'; // определяем переменную и присваиваем ей значение  
  
alert(message); // Hello!
```

Мы также можем объявить несколько переменных в одной строке:

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

Такой способ может показаться короче, но мы не рекомендуем его. Для лучшей читаемости объявляйте каждую переменную на новой строке.

Многострочный вариант немного длиннее, но легче для чтения:

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

Некоторые люди также определяют несколько переменных в таком вот многострочном стиле:

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

...Или даже с запятой в начале строки:

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

В принципе, все эти варианты работают одинаково. Так что это вопрос личного вкуса и эстетики.

i var вместо let

В старых скриптах вы также можете найти другое ключевое слово: `var` вместо `let` :

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

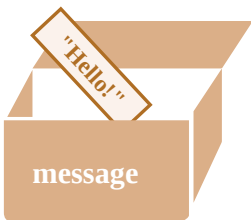
Ключевое слово `var` — *почти* то же самое, что и `let` . Оно объявляет переменную, но немного по-другому, «устаревшим» способом.

Есть тонкие различия между `let` и `var` , но они пока не имеют для нас значения. Мы подробно рассмотрим их в главе [Устаревшее ключевое слово "var"](#).

Аналогия из жизни

Мы легко поймём концепцию «переменной», если представим её в виде «коробки» для данных с уникальным названием на ней.

Например, переменную `message` можно представить как коробку с названием `"message"` и значением `"Hello!"` внутри:

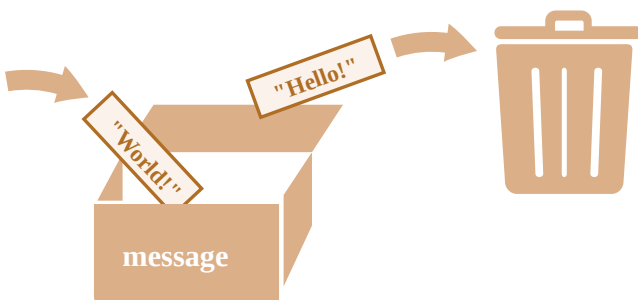


Мы можем положить любое значение в коробку.

Мы также можем изменить его столько раз, сколько захотим:

```
let message;  
  
message = 'Hello!';  
  
message = 'World!'; // значение изменено  
  
alert(message);
```

При изменении значения старые данные удаляются из переменной:



Мы также можем объявить две переменные и скопировать данные из одной в другую.

```
let hello = 'Hello world!';

let message;

// копируем значение 'Hello world' из переменной hello в переменную message
message = hello;

// теперь две переменные содержат одинаковые данные
alert(hello); // Hello world!
alert(message); // Hello world!
```

Повторное объявление вызывает ошибку

Переменная может быть объявлена только один раз.

Повторное объявление той же переменной является ошибкой:

```
let message = "Это";

// повторение ключевого слова 'let' приводит к ошибке
let message = "Другое"; // SyntaxError: 'message' has already been declared
```

Поэтому следует объявлять переменную только один раз и затем использовать её уже без `let`.

Функциональные языки программирования

Примечательно, что существуют [функциональные](#) языки программирования, такие как [Scala](#) или [Erlang](#), которые запрещают изменять значение переменной.

В таких языках однажды сохранённое «в коробку» значение остаётся там навсегда. Если нам нужно сохранить что-то другое, язык заставляет нас создать новую коробку (объявить новую переменную). Мы не можем использовать старую переменную.

Хотя на первый взгляд это может показаться немного странным, эти языки вполне подходят для серьёзной разработки. Более того, есть такая область, как параллельные вычисления, где это ограничение даёт определённые преимущества. Изучение такого языка (даже если вы не планируете использовать его в ближайшее время) рекомендуется для расширения кругозора.

Имена переменных

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.

Примеры допустимых имён:


```
let userName;  
let test123;
```

Если имя содержит несколько слов, обычно используется [верблюжья нотация](#) [↗](#), то есть, слова следуют одно за другим, где каждое следующее слово начинается с заглавной буквы: `myVeryLongName`.

Самое интересное – знак доллара `'$'` и подчёркивание `'_'` также можно использовать в названиях. Это обычные символы, как и буквы, без какого-либо особого значения.

Эти имена являются допустимыми:

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



```
alert($ + _); // 3
```

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` и `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. Даже если мы пишем небольшой скрипт, у него может быть долгая жизнь впереди. Людям из других стран, возможно, придётся прочесть его не один раз.

⚠ Зарезервированные имена

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!
```

⚠ Создание переменной без использования `use strict`

Обычно нам нужно определить переменную перед её использованием. Но в старые времена было технически возможно создать переменную простым присвоением значения без использования `let`. Это все ещё работает, если мы не включаем `use strict` в наших файлах, чтобы обеспечить совместимость со старыми скриптами.

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

Это плохая практика, которая приводит к ошибке в строгом режиме:

```
"use strict";  
  
num = 5; // ошибка: num is not defined
```

Константы

Чтобы объявить константную, то есть, неизменяемую переменную, используйте `const` вместо `let`:

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

Переменные, объявленные с помощью `const`, называются «константами». Их нельзя изменить. Попытка сделать это приведёт к ошибке:

```
const myBirthday = '18.04.1982';  
  
myBirthday = '01.01.2001'; // ошибка, константу нельзя перезаписать!
```

Если программист уверен, что переменная никогда не будет меняться, он может гарантировать это и наглядно донести до каждого, объявив её через `const`.

Константы в верхнем регистре

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, сделаем константы для различных цветов в «шестнадцатеричном формате»:

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

// ...когда нам нужно выбрать цвет
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Преимущества:

- `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.

Название «константа» просто означает, что значение переменной никогда не меняется. Но есть константы, которые известны до выполнения (например, шестнадцатеричное значение для красного цвета), а есть константы, которые *вычисляются* во время выполнения сценария, но не изменяются после их первоначального назначения.

Например:

```
const pageLoadTime = /* время, потраченное на загрузку веб-страницы */;
```

Значение `pageLoadTime` неизвестно до загрузки страницы, поэтому её имя записано обычными, а не прописными буквами. Но это всё ещё константа, потому что она не изменяется после назначения.

Другими словами, константы с именами, записанными заглавными буквами, используются только как псевдонимы для «жёстко закодированных» значений.

Придумывайте правильные имена

В разговоре о переменных необходимо упомянуть, что есть ещё одна чрезвычайно важная вещь.

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.

Пожалуйста, потратьте время на обдумывание правильного имени переменной перед её объявлением. Делайте так, и будете вознаграждены.

Несколько хороших правил:

- 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 об используемых терминах. Если посетитель сайта называется «user», тогда мы должны называть связанные с ним переменные `currentUser` или `newUser`, а не, к примеру, `currentVisitor` или `newManInTown`.

Звучит просто? Действительно, это так, но на практике для создания описательных и кратких имён переменных зачастую требуется подумать. Действуйте.

i Повторно использовать или создавать новую переменную?

И последняя заметка. Есть ленивые программисты, которые вместо объявления новых переменных повторно используют существующие.

В результате их переменные похожи на коробки, в которые люди бросают разные предметы, не меняя на них этикетки. Что сейчас находится внутри коробки? Кто знает? Нам необходимо подойти поближе и проверить.

Такие программисты немного экономят на объявлении переменных, но теряют в десять раз больше при отладке.

Дополнительная переменная – это добро, а не зло.

Современные JavaScript-минификаторы и браузеры оптимизируют код достаточно хорошо, поэтому он не создаёт проблем с производительностью. Использование разных переменных для разных значений может даже помочь движку оптимизировать ваш код.

Итого

Мы можем объявить переменные для хранения данных с помощью ключевых слов `var`, `let` или `const`.

- `let` – это современный способ объявления.

- `var` – это устаревший способ объявления. Обычно мы вообще не используем его, но
- мы рассмотрим тонкие отличия от `let` в главе [Устаревшее ключевое слово "var"](#) на случай, если это всё-таки вам понадобится.
 - `const` – похоже на `let`, но значение переменной не может изменяться.

Переменные должны быть названы таким образом, чтобы мы могли легко понять, что у них внутри.

✓ Задачи

Работа с переменными

важность: 2

1. Объявите две переменные: `admin` и `name`.
2. Запишите строку `"Джон"` в переменную `name`.
3. Скопируйте значение из переменной `name` в `admin`.
4. Выведите на экран значение `admin`, используя функцию `alert` (должна показать «Джон»).

[К решению](#)

Придумайте правильные имена

важность: 3

1. Создайте переменную для названия нашей планеты. Как бы вы её назвали?
2. Создайте переменную для хранения имени текущего посетителя сайта. Как бы вы назвали такую переменную?

[К решению](#)

Какие буквы (заглавные или строчные) использовать для имён констант?

важность: 4

Рассмотрим следующий код:

```
const birthday = '18.04.1982';  
const age = someCode(birthday);
```

У нас есть константа `birthday`, а также `age`, которая вычисляется при помощи некоторого кода, используя значение из `birthday` (в данном случае детали не имеют значения, поэтому код не рассматривается).

Можно ли использовать заглавные буквы для имени `birthday`? А для `age`? Или одновременно для обеих переменных?

```
const BIRTHDAY = '18.04.1982'; // использовать заглавные буквы?  
  
const AGE = someCode(BIRTHDAY); // а здесь?
```

[К решению](#)

Типы данных

Значение в JavaScript всегда относится к данным определённого типа. Например, это может быть строка или число.

Есть восемь основных типов данных в JavaScript. В этой главе мы рассмотрим их в общем, а в следующих главах поговорим подробнее о каждом.

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.

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.

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

- `Infinity` представляет собой математическую [бесконечность](#) ∞ . Это особое значение, которое больше любого числа.

Мы можем получить его в результате деления на ноль:

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

Или задать его явно:

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

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

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

Значение `NaN` «прилипчиво». Любая математическая операция с `NaN` возвращает `NaN`:

```
alert( NaN + 1 ); // NaN
alert( 3 * NaN ); // NaN
alert( "не число" / 2 - 1 ); // NaN
```

Если где-то в математическом выражении есть `NaN`, то оно распространяется на весь результат (есть только одно исключение: `NaN ** 0` равно `1`).

Математические операции – безопасны

Математические операции в JavaScript «безопасны». Мы можем делать что угодно: делить на ноль, обращаться с нечисловыми строками как с числами и т.д.

Скрипт никогда не остановится с фатальной ошибкой (не «умрёт»). В худшем случае мы получим `NaN` как результат выполнения.

Специальные числовые значения относятся к типу «число». Конечно, это не числа в привычном значении этого слова.

Подробнее о работе с числами мы поговорим в главе [Числа](#).

BigInt

В JavaScript тип `number` не может безопасно работать с числами, большими, чем $(2^{53} - 1)$ (т. е. `9007199254740991`) или меньшими, чем $-(2^{53} - 1)$ для отрицательных чисел. Технически, тип `number` может хранить и гораздо большие значения (вплоть до $1.7976931348623157 \cdot 10^{308}$), однако за пределами безопасного диапазона $\pm(2^{53} - 1)$ многие из чисел не могут быть представлены с помощью этого типа данных из-за ограничений, вызванных внутренним представлением чисел в двоичной форме. Например, нечётные числа, большие, чем $(2^{53} - 1)$, невозможно хранить при помощи типа `number`, они с разной точностью будут автоматически округляться до чётных значений. В то же время некоторые чётные числа, большие, чем $(2^{53} - 1)$, при помощи типа `number` хранить технически возможно (однако не стоит этого делать во избежание дальнейших ошибок).

Для большинства случаев достаточно безопасного диапазона чисел от $-(2^{53} - 1)$ до $(2^{53} - 1)$. Но иногда нам нужен диапазон действительно гигантских целых чисел без каких-либо ограничений или пропущенных значений внутри него. Например, в криптографии или при использовании метки времени («timestamp») с микросекундами.

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

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

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

Так как `BigInt`-числа нужны достаточно редко, мы рассмотрим их в отдельной главе [BigInt](#). Ознакомьтесь с ней, когда вам понадобятся настолько большие числа.

Поддержка

В данный момент `BigInt` поддерживается только в браузерах Firefox, Chrome, Edge и Safari, но не поддерживается в IE.

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.

1. Двойные кавычки: `"Привет"`.
2. Одинарные кавычки: `'Привет'`.
3. Обратные кавычки: ``Привет``.

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

Обратные же кавычки имеют расширенную функциональность. Они позволяют нам 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. Мы можем положить туда всё, что угодно: переменную `name`, или выражение `1 + 2`, или что-то более сложное.

Обратите внимание, что это можно делать только в обратных кавычках. Другие кавычки не имеют такой функциональности встраивания!

```
alert( "результат: ${1 + 2}" ); // результат: ${1 + 2} (двойные кавычки ничего не делают)
```

Мы рассмотрим строки более подробно в главе [Строки](#).

i Нет отдельного типа данных для одного символа.

В некоторых языках, например C и Java, для хранения одного символа, например `"a"` или `"%"`, существует отдельный тип. В языках C и Java это `char`.

В JavaScript подобного типа нет, есть только тип `string`. Строка может содержать ноль символов (быть пустой), 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")
```

Мы рассмотрим булевы значения более подробно в главе [Логические операторы](#).

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 значение переменной `age` неизвестно.

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

Технически мы можем присвоить значение `undefined` любой переменной:

```
let age = 123;  
  
// изменяем значение на undefined  
age = undefined;  
  
alert(age); // "undefined"
```

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

Объекты и символы

Тип `object` (объект) – особенный.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка, или число, или что-то ещё). В объектах же хранят коллекции данных или более сложные структуры.

Объекты занимают важное место в языке и требуют особого внимания. Мы разберёмся с ними в главе [Объекты](#) после того, как узнаем больше о примитивах.

Тип `symbol` (символ) используется для создания уникальных идентификаторов в объектах. Мы упоминаем здесь о нём для полноты картины, изучим этот тип после объектов.

Оператор typeof

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

У него есть две синтаксические формы:

1. Синтаксис оператора: `typeof x`.
2. Синтаксис функции: `typeof(x)`.

Другими словами, он работает со скобками или без скобок. Результат одинаковый.

Вызов `typeof x` возвращает строку с именем типа:

```
typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

typeof null // "object" (2)

typeof alert // "function" (3)
```

Последние три строки нуждаются в пояснении:

1. `Math` — это встроенный объект, который предоставляет математические операции и константы. Мы рассмотрим его подробнее в главе [Числа](#). Здесь он служит лишь примером объекта.
2. Результатом вызова `typeof null` является `"object"`. Это официально признанная ошибка в `typeof`, ведущая начало с времён создания JavaScript и сохранённая для совместимости. Конечно, `null` не является объектом. Это специальное значение с отдельным типом.
3. Вызов `typeof alert` возвращает `"function"`, потому что `alert` является функцией. Мы изучим функции в следующих главах, где заодно увидим, что в JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но `typeof` обрабатывает их особым образом, возвращая `"function"`. Так тоже повелось от создания JavaScript. Формально это неверно, но может быть удобным на практике.

Итого

В JavaScript есть 8 основных типов данных.

- Семь из них называют «примитивными» типами данных:
 - `number` для любых чисел: целочисленных или чисел с плавающей точкой; целочисленные значения ограничены диапазоном $\pm(2^{53}-1)$.
 - `bigint` для целых чисел произвольной длины.
 - `string` для строк. Строка может содержать ноль или больше символов, нет отдельного символьного типа.

- `boolean` для `true/false`.
- `null` для неизвестных значений – отдельный тип, имеющий одно значение `null`.
- `undefined` для неприсвоенных значений – отдельный тип, имеющий одно значение `undefined`.
- `symbol` для уникальных идентификаторов.
- И один не является «примитивным» и стоит особняком:
 - `object` для более сложных структур данных.

Оператор `typeof` позволяет нам увидеть, какой тип данных сохранён в переменной.

- Имеет две формы: `typeof x` или `typeof(x)`.
- Возвращает строку с именем типа. Например, `"string"`.
- Для `null` возвращается `"object"` – это ошибка в языке, на самом деле это не объект.

В следующих главах мы сконцентрируемся на примитивных значениях, а когда познакомимся с ними, перейдём к объектам.

✓ Задачи

Шаблонные строки

важность: 5

Что выведет этот скрипт?

```
let name = "Ilya";

alert( `hello ${1}` ); // ?

alert( `hello ${"name"}` ); // ?

alert( `hello ${name}` ); // ?
```

[К решению](#)

Взаимодействие: alert, prompt, confirm

Так как мы будем использовать браузер как демо-среду, нам нужно познакомиться с несколькими функциями его интерфейса, а именно: `alert`, `prompt` и `confirm`.

alert

С этой функцией мы уже знакомы. Она показывает сообщение и ждёт, пока пользователь нажмёт кнопку «ОК».

Например:

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

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

prompt

Функция `prompt` принимает два аргумента:

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

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

title

Текст для отображения в окне.

default

Необязательный второй параметр, который устанавливает начальное значение в поле для текста в окне.

Квадратные скобки в синтаксисе [...]

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

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

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

Например:

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

⚠ Для IE: всегда устанавливайте значение по умолчанию

Второй параметр является необязательным, но если не указать его, то Internet Explorer вставит строку `"undefined"` в поле для ввода.

Запустите код в Internet Explorer и посмотрите на результат:

```
let test = prompt("Test");
```

Чтобы `prompt` хорошо выглядел в IE, рекомендуется всегда указывать второй параметр:

```
let test = prompt("Test", ''); // <-- для IE
```

confirm

Синтаксис:

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

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

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

Например:

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

Итого

Мы рассмотрели 3 функции браузера для взаимодействия с пользователем:

alert

показывает сообщение.

prompt

показывает сообщение и запрашивает ввод текста от пользователя. Возвращает напечатанный в поле ввода текст или `null`, если была нажата кнопка «Отмена» или `Esc` с клавиатуры.

confirm

показывает сообщение и ждёт, пока пользователь нажмёт ОК или Отмена. Возвращает `true`, если нажата ОК, и `false`, если нажата кнопка «Отмена» или `Esc` с клавиатуры.

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

На все указанные методы распространяются два ограничения:

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

Такова цена простоты. Есть другие способы показать более приятные глазу окна с богатой функциональностью для взаимодействия с пользователем, но если «навороты» не имеют значения, то данные методы работают отлично.

✓ Задачи

Простая страница

важность: 4

Создайте страницу, которая спрашивает имя у пользователя и выводит его.

[Запустить демо](#)

[К решению](#)

Преобразование типов

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

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

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

i Пока что мы не говорим об объектах

В этой главе мы не касаемся объектов. Сначала мы разберём преобразование примитивных значений.

Мы разберём преобразование объектов позже, в главе [Преобразование объектов в примитивы](#).

Строковое преобразование

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

Например, `alert(value)` преобразует значение к строке.

Также мы можем использовать функцию `String(value)`, чтобы преобразовать значение к строке:

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

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

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

Численное преобразование

Численное преобразование происходит в математических функциях и выражениях.

Например, когда операция деления `/` применяется не к числу:

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

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

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // становится числом 123

alert(typeof num); // number
```

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

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

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

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

Правила численного преобразования:

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

Примеры:

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

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

Большинство математических операторов также производит данное преобразование, как мы увидим в следующей главе.

Логическое преобразование

Логическое преобразование самое простое.

Происходит в логических операциях (позже мы познакомимся с условными проверками и подобными конструкциями), но также может быть выполнено явно с помощью функции `Boolean(value)`.

Правило преобразования:

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

Например:

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

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

 **Заметим, что строка с нулём `"0"` — это `true`**

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

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

Итого

Существует 3 наиболее широко используемых преобразования: строковое, численное и логическое.

Строковое – Происходит, когда нам нужно что-то вывести. Может быть вызвано с помощью `String(value)`. Для примитивных значений работает очевидным образом.

Численное – Происходит в математических операциях. Может быть вызвано с помощью `Number(value)`.

Преобразование подчиняется правилам:

Значение	Становится...
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true / false</code>	<code>1 / 0</code>
<code>string</code>	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то получаем <code>0</code> , иначе из непустой строки «считывается» число. При ошибке результат <code>NaN</code> .

Логическое – Происходит в логических операциях. Может быть вызвано с помощью `Boolean(value)`.

Подчиняется правилам:

Значение	Становится...
<code>0</code> , <code>null</code> , <code>undefined</code> , <code>NaN</code> , <code>""</code>	<code>false</code>
любое другое значение	<code>true</code>

Большую часть из этих правил легко понять и запомнить. Особые случаи, в которых часто допускаются ошибки:

- `undefined` при численном преобразовании становится `NaN`, не `0`.
- `"0"` и строки из одних пробелов типа `" "` при логическом преобразовании всегда `true`.

В этой главе мы не говорили об объектах. Мы вернёмся к ним позже, в главе [Преобразование объектов в примитивы](#), посвящённой только объектам, сразу после того, как узнаем больше про основы JavaScript.

Базовые операторы, математика

Многие операторы знакомы нам ещё со школы: сложение `+`, умножение `*`, вычитание `-` и так далее.

В этой главе мы начнём с простых операторов, а потом сконцентрируемся на специфических для JavaScript аспектах, которые не проходят в школьном курсе арифметики.

Термины: «унарный», «бинарный», «операнд»

Прежде, чем мы двинемся дальше, давайте разберёмся с терминологией.

- *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» вместо «операндов».
- *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 вычитает значения
```

Формально, в последних примерах мы говорим о двух разных операторах, использующих один символ: оператор отрицания (унарный оператор, который обращает знак) и оператор вычитания (бинарный оператор, который вычитает одно число из другого).

Математика

Поддерживаются следующие математические операторы:

- Сложение `+`,
- Вычитание `-`,
- Умножение `*`,
- Деление `/`,
- Взятие остатка от деления `%`,
- Возведение в степень `**`.

Первые четыре оператора очевидны, а про `%` и `**` стоит сказать несколько слов.

Взятие остатка %

Оператор взятия остатка `%`, несмотря на обозначение, никакого отношения к процентам не имеет.

Результат `a % b` – это [остаток](#) от целочисленного 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
```

Возведение в степень **

В выражении `a ** b` оператор возведения в степень умножает `a` на само себя `b` раз.

Например:

```
alert( 2 ** 2 ); // 4 (2 умножено на себя 2 раза)
alert( 2 ** 3 ); // 8 (2 * 2 * 2, 3 раза)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2, 4 раза)
```

Математически, оператор работает и для нецелых чисел. Например, квадратный корень является возведением в степень $1/2$:

```
alert( 4 ** (1/2) ); // 2 (степень 1/2 эквивалентна взятию квадратного корня)
alert( 8 ** (1/3) ); // 2 (степень 1/3 эквивалентна взятию кубического корня)
```

Сложение строк при помощи бинарного +

Давайте рассмотрим специальные возможности операторов JavaScript, которые выходят за рамки школьной арифметики.

Обычно при помощи плюса '+' складывают числа.

Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну:

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

Обратите внимание, если хотя бы один операнд является строкой, то второй будет также преобразован в строку.

Например:

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

Как видите, не важно, первый или второй операнд является строкой.

Вот пример посложнее:

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

Здесь операторы работают один за другим. Первый + складывает два числа и возвращает 4, затем следующий + объединяет результат со строкой, производя действие `4 + '1' = '41'`.

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( 6 - '2' ); // 4, '2' приводится к числу
alert( '6' / '2' ); // 3, оба операнда приводятся к числам
```

Приведение к числу, унарный +

Плюс `+` существует в двух формах: бинарной, которую мы использовали выше, и унарной.

Унарный, то есть применённый к одному значению, плюс `+` ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число.

Например:

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

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

// Преобразует не числа в числа
alert( +true ); // 1
alert( +"" ); // 0
```

На самом деле это то же самое, что и `Number(...)`, только короче.

Необходимость преобразовывать строки в числа возникает очень часто. Например, обычно значения полей HTML-формы — это строки. А что, если их нужно, к примеру, сложить?

Бинарный плюс сложит их как строки:

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

alert( apples + oranges ); // "23", так как бинарный плюс объединяет строки
```

Поэтому используем 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
```

С точки зрения математика, такое изобилие плюсов выглядит странным. Но с точки зрения программиста тут нет ничего особенного: сначала выполнятся унарные плюсы, которые приведут строки к числам, а затем бинарный `+` их сложит.

Почему унарные плюсы выполнились до бинарного сложения? Как мы сейчас увидим, дело в их приоритете.

Приоритет операторов

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

Из школы мы знаем, что умножение в выражении `1 + 2 * 2` выполнится раньше сложения. Это как раз и есть «приоритет». Говорят, что умножение имеет более высокий приоритет, чем сложение.

Скобки важнее, чем приоритет, так что, если мы не удовлетворены порядком по умолчанию, мы можем 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.

Отрывок из [таблицы приоритетов](#) (нет необходимости всё запоминать, обратите внимание, что приоритет унарных операторов выше, чем соответствующих бинарных):

Приоритет	Название	Обозначение
...
15	унарный плюс	+
15	унарный минус	-
14	возведение в степень	**
13	умножение	*
13	деление	/
12	сложение	+
12	вычитание	-
...
2	присваивание	=
...

Так как «унарный плюс» имеет приоритет `15`, который выше, чем `12` у «сложения» (бинарный плюс), то в выражении `" +apples + +oranges"` сначала выполнятся унарные плюсы, а затем сложение.

Присваивание

Давайте отметим, что в таблице приоритетов также есть оператор присваивания `=`. У него один из самых низких приоритетов: `2`.

Именно поэтому, когда переменной что-либо присваивают, например, `x = 2 * 2 + 1`, то сначала выполнится арифметика, а уже затем произойдёт присваивание `=` с сохранением результата в `x`.

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



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

Присваивание = возвращает значение

Тот факт, что `=` является оператором, а не «магической» конструкцией языка, имеет интересные последствия.

Большинство операторов в JavaScript возвращают значение. Для некоторых это очевидно, например сложение `+` или умножение `*`. Но и оператор присваивания не является исключением.

Вызов `x = value` записывает `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 переменной `a` (то есть `3`). Потом оно используется для дальнейших вычислений.

Забавное применение присваивания, не так ли? Нам нужно понимать, как это работает, потому что иногда это можно увидеть в JavaScript-библиотеках.

Однако писать таким в таком стиле не рекомендуется. Такие трюки не сделают ваш код более понятным или читабельным.

Присваивание по цепочке

Рассмотрим ещё одну интересную возможность: цепочку присваиваний.

```
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 результат присваивается переменным слева: `c`, `b` и `a`. В конце у всех переменных будет одно значение.

Опять-таки, чтобы код читался легче, лучше разделять подобные конструкции на несколько строчек:

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