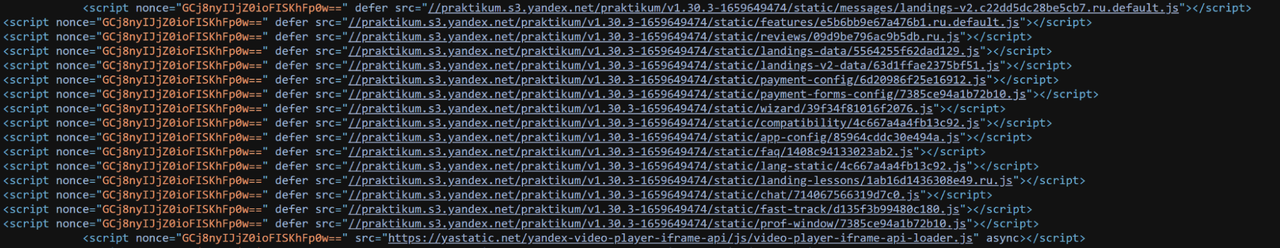
**Лекция номер 1**

**JavaScript**

Объясняем на примерах, почему без JavaScript не будет современных сайтов и что делает этот язык программирования таким востребованным. Расскажем о преимуществах JS и его перспективах.

## Особенности JavaScript

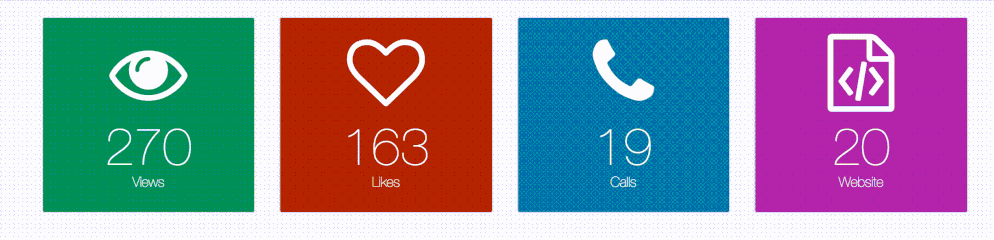
JavaScript — это язык программирования, который в первую очередь применяют в вебе. С его помощью сайты делают интерактивными: добавляют всплывающие окна, анимацию, кнопки лайков и формы для отправки информации. Его ещё называют главным языком фронтенда — «лицевой» стороны сайта, с которой взаимодействуют пользователи.  
  
JavaScript похож на Java, Python, Go, C#, но имеет свои особенности:  
  
**1. Интерпретируемый, а не компилируемый.** Все языки программирования делятся на две группы. Код на компилируемых языках нужно сначала пропустить через специальную программу, компилятор, и только потом его можно использовать. Большинство современных языков как раз компилируемые в отличие от JavaScript.  
  
Код на интерпретируемом языке компилировать не нужно. Его пишут и сразу передают программе-интерпретатору, которая тут же его выполняет. Это ускоряет разработку, но требует запуска только в связке с интерпретатором. Правда, интерпретатор JavaScript встроен во все современные браузеры, так что проблем с запуском кода нет.  
  
**2. Мультипарадигменный**. В программировании существует несколько парадигм:  
  
● Объектная. В ней для каждой сущности в коде, например кнопки заказа, создают объекты. А затем прописывают связи между этими объектами.  
● Функциональная. В ней больше внимания уделяют конкретным инструкциям. Кнопка здесь — не объект, в коде она обозначена как порядок действий, которые происходят после нажатия.  
● Императивная. Здесь код пишут как чёткий набор инструкций, который выполняется строго последовательно. Императивная и функциональная парадигма часто применяются вместе.  
  
Часто языки ориентированы на работу с одной парадигмой. JavaScript не такой — он не диктует, как писать код, и позволяет придерживаться разных стилей и способов.  
  
**3. Без строгой типизации**. Есть языки программирования со статической типизацией. Если создать переменную, нужно сначала обязательно задать её тип, например число. И ничего, кроме числа, хранить в ней нельзя.  
  
У языка JavaScript типизация динамическая — в переменную можно положить что угодно. Это облегчает написание кода, но может вызвать ошибки. Например, JavaScript позволит сравнить строку с числом и даже выдаст какой-то результат. Что больше, «корова» или «81»? JavaScript знает ответ.  
  
**4. Интегрирован с HTML и CSS в браузерах**. HTML и CSS — это языки разметки для сайтов. Они не относятся к языкам программирования, так как не позволяют писать функции и что-то обрабатывать. С их помощью задают внешний вид сайта: расположение блоков, размер шрифта, цвета. Чтобы сайт стал интерактивным, нужен язык программирования, и JavaScript тут подходит идеально. Современные браузеры позволяют просто вписать код на JavaScript прямо в разметку, написанную на HTML и CSS, — и всё заработает.  
  
Часто для удобства скрипты пишут в отдельных файлах, а потом вставляют в код отсылку на сам файл. Это позволяет не перегружать код на странице и использовать одни и те же скрипты в разных местах, не переписывая их заново.  


Так выглядят файлы скриптов на JavaScript, вставленные в код страницы

**5. Подходит для бэкенда**. Код JavaScript можно запускать и в браузере, и на сервере. Это значит, что на нём можно писать не только интерактивные элементы страниц, но и серверную часть сайта: функции обработки данных и вычислений. Правда, для этого нужно освоить отдельный инструмент — [Node.js](https://practicum.yandex.ru/blog/rukovodstvo-po-node-js-dlya-nachinayushchih/). Это движок, который позволяет запускать JavaScript не в браузере, а на сервере.  
  
Иногда из-за похожих названий язык JavaScript путают с Java. На самом деле это абсолютно разные языки для разных задач. Просто когда-то давно Java был очень популярен и его название добавили в язык JavaScript для привлечения внимания.

## Для чего нужен JavaScript

● Сделать сайт интерактивным — чтобы это была не просто статичная страница, а динамичная программа, реагирующая на действие пользователей. Например, если нажать кнопку «Нравится» и число лайков на записи увеличивается сразу, без обновления страницы. Это результат использования JavaScript.

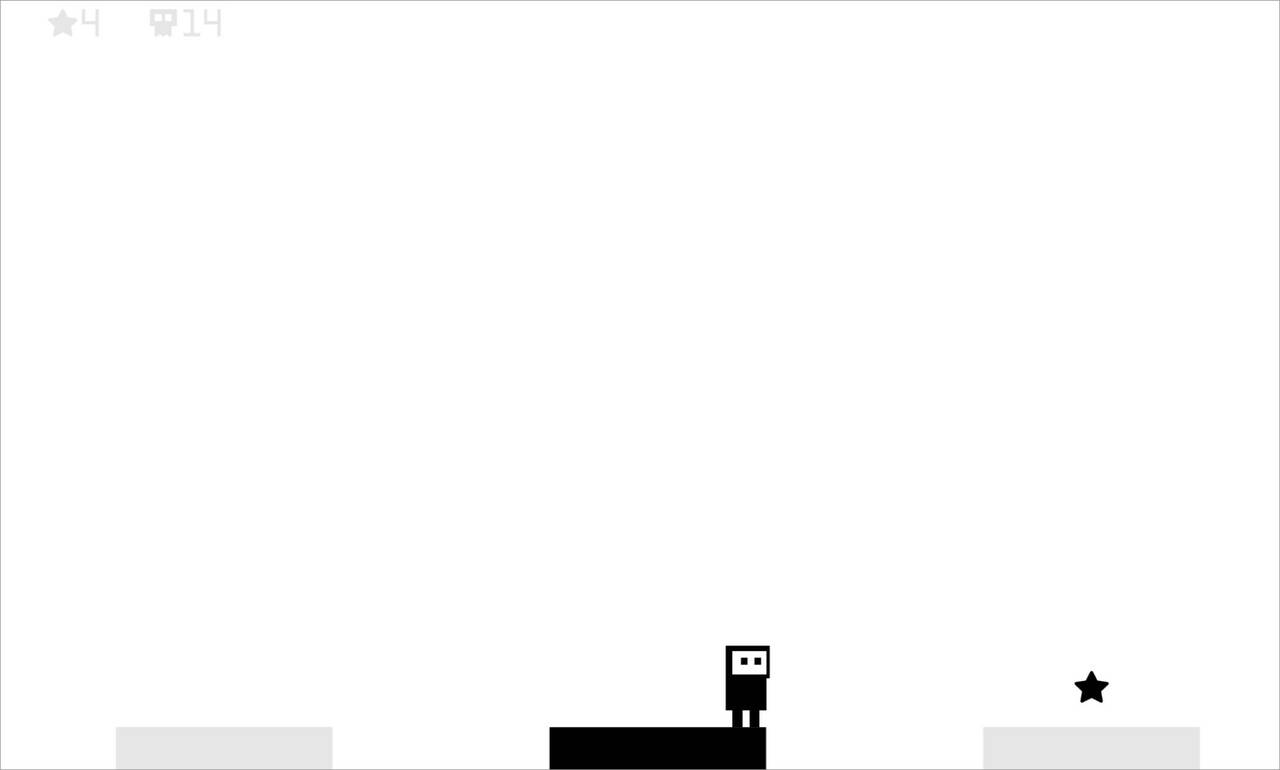


Интерактивные элементы на сайте сделаны с помощью JavaScript

● Передавать информацию между пользователем и сервером. Именно скрипты, написанные на JavaScript, отправляют на сервер информацию, которую ввёл пользователь.   
● Анимировать объекты — добавить скролл до раздела по нажатию кнопки, анимацию фона, движущиеся объекты.  
● Производить вычисления. Например, в онлайн-калькуляторе на сайте. Это можно делать как на стороне сервера, так и в самом браузере с помощью возможностей JavaScript.  
● Писать серверную часть сайта или приложения.

## Области применения JavaScript

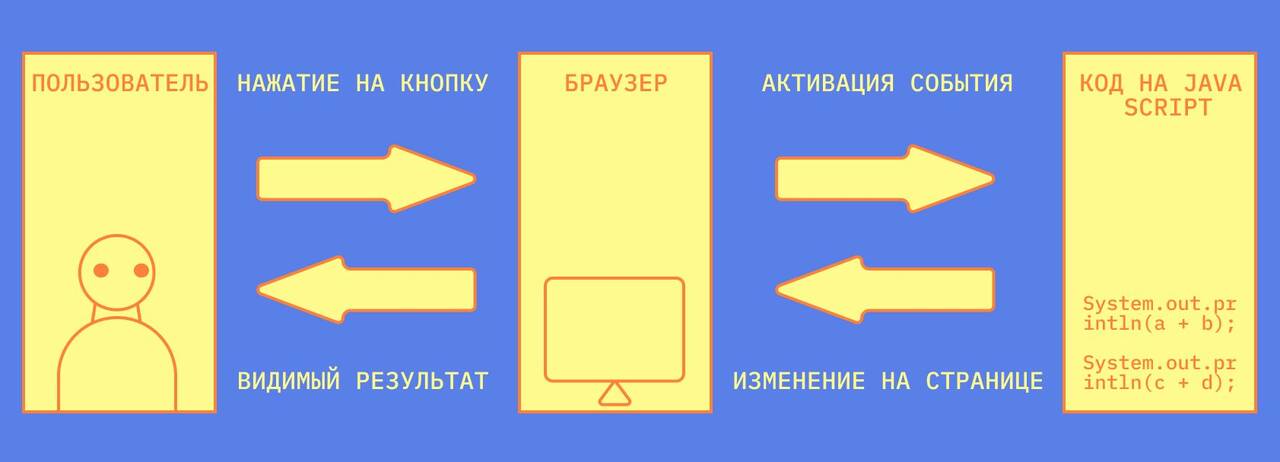
Теперь немного подробнее расскажем, что же можно написать на языке JS, где его используют чаще всего.  
  
**Веб-сайты и веб-приложения**. Самая популярная область применения языка JavaScript — это написание кода для сайта. Практически на каждом современном сайте используют код, написанный на JS.  
  
**Расширения для браузера**. Небольшие простые скрипты, которые добавляют дополнительный функционал — блокируют рекламу, позволяют сохранять аудио, отправляют уведомления о новых письмах или меняют цветовую схему сайта.  
  
**Мобильные приложения**. Их можно писать на специальных языках, например Kotlin. Но если нужно что-то простое, например интерфейс для работы с облачным хранилищем, его можно написать на JavaScript и собрать в приложение с помощью специальных инструментов.  
  
**Серверная часть сайтов и программ.** Язык программирования JavaScript можно использовать для написания любых сервисов: чатов, компьютерных программ и даже нейросетей. Для этого к нему нужно подключить движок Node.js.   
  
**Игры**. На JS можно писать несложные браузерные игры. Вот [интересный платформер с необычной физикой](https://js13kgames.com/games/onoff/index.html).



Простая, но интересная игра-платформер, написанная на JavaScript

## Как работает JS-код на сайте

JavaScript реагирует на действия пользователя на странице. Пользователь нажимает кнопку, это запускает триггер — и тут же срабатывает написанный скрипт.



Когда происходит событие, JavaScript выполняет определённые действия: изменяет содержимое страницы или отправляет данные на сервер

Сам по себе JavaScript не может чего-то уметь или не уметь — это просто язык программирования. Его возможности зависят от того, что именно ему позволяет браузер. В современных браузерах JavaScript-код может:  
  
● Изменить уже существующий на странице [HTML-код](https://practicum.yandex.ru/blog/zachem-nuzhen-html/) или добавить новый, поменять стили. Например, сменить цвет шапки.  
● Среагировать на щелчок мыши, перемещение указателя, нажатие клавиш, прокрутку.  
● Отправить сетевой запрос на сервер, скачать или загрузить файл. Например, взять присланную пользователем картинку и отправить её на сервер для хранения.  
● Задавать пользователю вопросы и показывать сообщения.  
● Запоминать данные и хранить их на стороне клиента, в его браузере. Например, сохранить документ в офлайн-режиме, пока пользователь не очистит кэш или не обновит страницу.

## Чего не умеет делать JavaScript в браузере

JavaScript изначально задумывали как безопасный язык. Браузер не позволяет ему глубоко влезать в работу компьютера пользователя и других веб-сайтов. Поэтому JS в браузере не умеет:  
  
● Читать и записывать файлы на жёсткий диск пользователя, запускать программы, работать с системными функциями ОС. Ему доступно только то, что пользователь сам добровольно согласился загрузить через браузер. В некоторых браузерах, например Mozilla Firefox, функция чтения и записи уже появилась, но пока доступна не везде.  
● Работать с камерой и микрофоном без разрешения пользователя.  
● Взаимодействовать с другими окнами и вкладками. Для этого обе страницы должны «согласиться» с этим и иметь специальный код для обмена данными. Это защищает пользователя и не помогает неизвестному сайту взаимодействовать со вкладкой, где открыты, например, соцсети или почта.  
● Легко получать данные от других сайтов и доменов. Это возможно с согласия другого сайта или сервера.  
  
Некоторые ограничения на использование JS можно снять — для этого браузер запросит у пользователя разрешение.

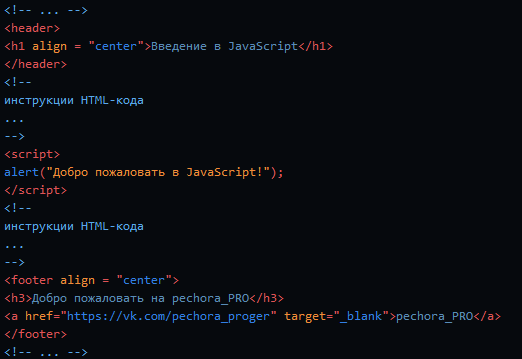
## Стоит ли учить JavaScript

Тем, кто планирует [стать фронтенд‑разработчиком](https://practicum.yandex.ru/blog/professiya-frontend-razrabotchik/), — однозначно стоит. Этот язык будет основным рабочим инструментом. Без JavaScript невозможно разрабатывать ту часть сайтов и веб-приложений, с которой взаимодействуют пользователи.   
  
У JS есть и другие плюсы для изучения:  
  
● **Простой в изучении**. Уже через пять минут можно написать и запустить свой первый скрипт. А через пару дней обучения собрать вполне рабочую программу, например расширение для браузера.  
● **Хорош как база**. На нём можно освоить основы, единые для всех языков программирования. И потом перейти к изучению других. Особенно хорошо после JS идут «С-образные» языки: C++, C#, Java, PHP.  
● **Универсален**. Если освоить Node.js, можно перейти с фронтенда на бэкенд или в другую сферу разработки.  
● **Актуален**. JS был создан больше 20 лет назад, но всё ещё развивается и не сдаёт свои позиции как главный язык для фронтенда.  
● **Имеет множество надстроек**. CoffeeScript, TypeScript и Dart и другие надстройки делают код компактнее, чище и строже. Их можно изучить, чтобы развиваться в сфере фронтенда.

**Лекция номер 2**

**Подключение сценариев**

В этой статье разберёмся, как добавить JavaScript в HTML. Сделать это можно тремя способами: разместить JS-код внутри парного тега <script>; подключить внешний файл; прописать код прямо внутри HTML-тега, используя специальные атрибуты. Рассмотрим каждый способ подробнее, чтобы понять его плюсы и минусы. Вставка кода внутри парного тега <script> Самый простой способ вставить JS-код в HTML-страницу — использовать парный тег <script>. Он сообщает браузеру, что всё его содержимое нужно интерпретировать как исполняемый скрипт, а не как текст или HTML-код. В качестве примера добавим на страницу скрипт, который выводит текущие дату и время.

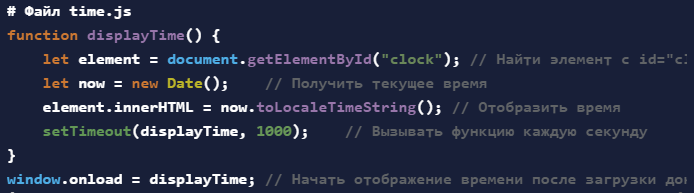


Теперь при загрузке страницы будет появляться уведомление с текущими датой и временем. Внутри парного тега <script> можно написать целую программу. В качестве примера создадим электронные часы.

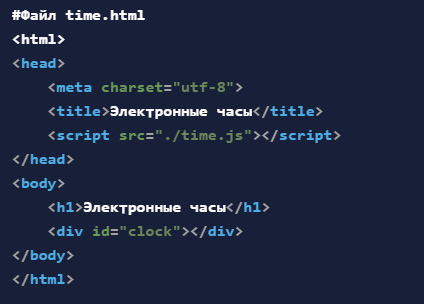


Отличный результат! У вас получилось с помощью JavaScript добавить в элемент HTML информацию о текущем времени, которая обновляется каждую секунду. В комментариях подробно описано, что делает скрипт. Он находит элемент с id=’clock’. Это тег <div>, который находится внутри секции body сразу под заголовком. Затем программа получает текущее время, которое отображается сразу после загрузки документа и обновляется каждую секунду. Выглядит удобно, но у такого подхода есть несколько минусов. Представьте, что вы хотите отображать время на каждой странице сайта. В таком случае придётся дублировать содержимое тега <script>. Другая проблема – чем больше содержимое скрипта, тем хуже читабельность кода. В примере выше только одна функция. А если их требуется 5-10-20, чтобы реализовать нужное поведение? В таком случае удобнее использовать следующий подход — подключать JS-код из внешних файлов.

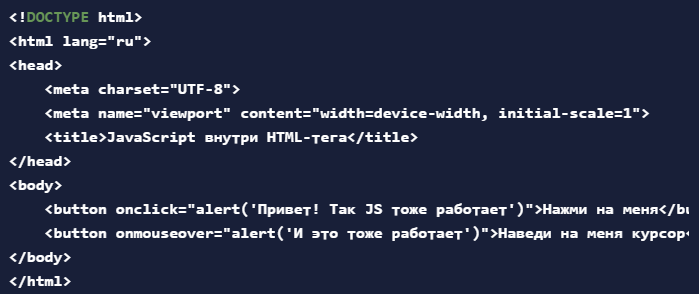
Подключение внешних файлов Для подключения JavaScript из внешних файлов тоже используется тег <script>. Только внутри него появляется атрибут src, в котором вы указываете ссылку на JS-файл. Это может быть файл, который вы создали сами и храните рядом с HTML или в другом каталоге. А ещё это может быть файл JS-библиотеки, которая добавляет на страницу нужное вам поведение. В качестве примера возьмём скрипт, который выводит текущее время. Перенесём всю логику в отдельный файл:



Из файла HTML можно удалить весь код JS. Останется только тег <script>, в котором через атрибут src указана ссылка на файл time.js:



Результат будет таким же — вам снова удалось с помощью JavaScript добавить в HTML div информацию о текущем времени. У такого подхода сразу несколько плюсов: Содержимое файлов HTML становится проще. В них остаются только контент и ссылки на внешние файлы. Весь JS-код можно держать в одном файле и при необходимости подключать его к разным страницам. Если один скрипт используется несколькими страницами, то он будет загружаться полностью только один раз, а затем использоваться повторно из кэша браузера. JS внутри тегов HTML Вы можете разместить JavaScript внутри тега HTML, используя специальные атрибуты — например, onclick, onmouseover, onkeypress, onload. Они добавляют на элементы обработчики событий.



Если вы нажмёте на первую кнопку, то в браузере отобразится уведомление с текстом «Привет! Так JS тоже работает». На вторую кнопку достаточно навести курсор мыши — в ответ тоже появится браузерное уведомление с текстом «И это тоже работает». Это говорит о том, что обработчик событий работает корректно и реагирует на действия пользователя. Вы можете добавить в один тег несколько атрибутов с обработчиками событий, разделив их точкой с запятой. Вот так:



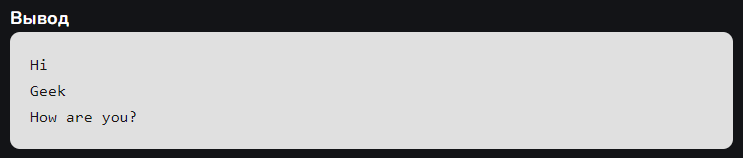
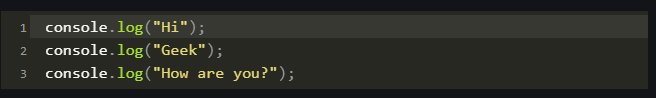
Однако это плохая практика. Большинство разработчиков предпочитает разделять контент и управление поведением по разным файлам. К тому же, как вы видите, использование обработчиков внутри HTML заметно ухудшает читабельность кода.

Где лучше располагать скрипты Технически нет ограничений на вставку тега <script>. Вы можете добавить JavaScript на странице в секциях <head> и <body>, в самом начале и в конце документа. Все эти варианты будут рабочими. Однако обычно скрипты подключают в конце секции body, прямо перед закрывающим тегом. Это ускоряет загрузку страницы. Пользователи быстрее видят контент и понимают, что страница работает. Даже если JS загрузится чуть позже, это не так сильно ухудшает пользовательский опыт, как длительное ожидание отрисовки страницы. Дело в том, что каждый тег <script> блокирует отрисовку страницы до тех пор, пока не будут завершены полная загрузка и выполнение JS. Но подключенные файлы (а иногда и целые библиотеки) могут весить достаточно много. При высокой скорости интернета пауза может быть минимальной. Но стоит пользователю оказаться в месте с плохим покрытием сотовой связи, как он заметит, что сайт очень долго не отображает контент. Размещать скрипты в секции head нужно только в том случае, если на то есть причины. Например, без JS на странице не отображается вообще никакой контент. Но если таких причин нет, и вы беспокоитесь о скорости загрузки страницы, то лучше JavaScript добавить в HTML-код в самом конце документа, перед закрывающим тегом </body>.

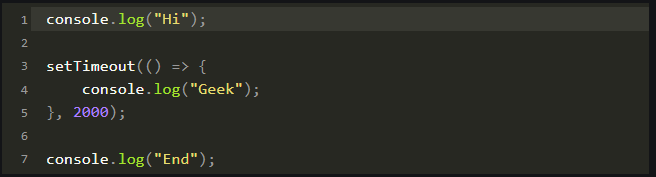
**Лекция номер 3**

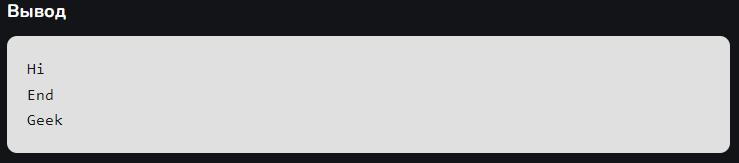
**Синхронные, асинхронные и отложенные сценарии**

JavaScript известен своей способностью обрабатывать и то, и другое синхронный и асинхронный операции. Понимание того, как работают эти две вещи, важно для написания эффективных, отзывчивых и удобных для пользователя приложений. В этой статье мы увидим различия между синхронный и асинхронный JavaScript с наглядными примерами, которые помогут вам. Что такое синхронный JavaScript? В синхронном программировании операции выполняются одна за другой, в определенной последовательности. Таким образом, в основном каждая строка кода ожидает завершения предыдущей, прежде чем перейти к следующей. Это означает, что программа выполняется в предсказуемом линейном порядке, при этом каждая задача завершается до запуска следующей. Пример: В этом примере мы показали синхронную природу JavaScript.



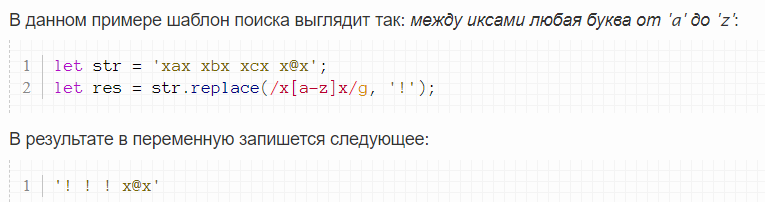
В приведенном выше фрагменте кода первая строка кода Hi будет регистрироваться первой, затем будет регистрироваться вторая строка Geek , а затем, после ее завершения, будет регистрироваться третья строка Как дела. Итак, как мы можем видеть, коды работают последовательно. Каждая строка кода ожидает выполнения предыдущей, а затем выполняется сама. В синхронном коде каждый оператор ожидает завершения предыдущего перед его запуском. Это простой и понятный способ, но у него есть некоторые недостатки, особенно при решении таких трудоемких задач, как извлечение данных с сервера или чтение большого файла. Если такая задача включена в последовательность, она заблокирует выполнение остальной части кода до его завершения, что приведет к потенциальным задержкам и плохому взаимодействию с пользователем. Что такое асинхронный JavaScript? Асинхронное программирование, с другой стороны, позволяет нескольким задачам выполняться независимо друг от друга. В асинхронном коде задача может быть инициирована, и в ожидании ее завершения могут выполняться другие задачи. Такая неблокирующая природа помогает повысить производительность и оперативность реагирования, особенно в веб-приложениях. Пример: В этом примере мы показали асинхронную природу JavaScript.

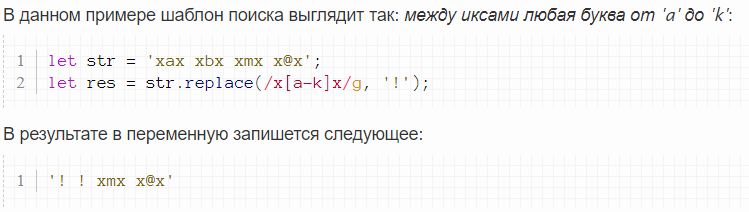


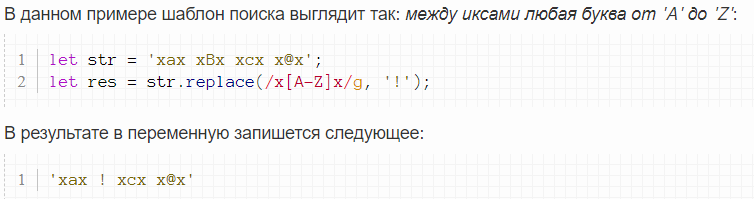


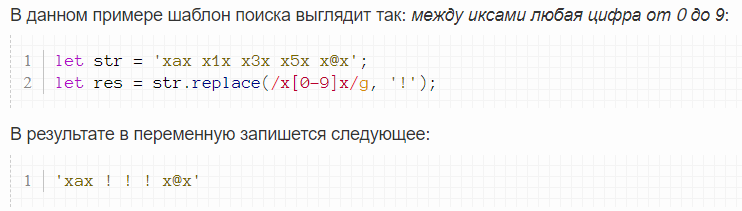
Итак, что делает код, так это сначала он входит в систему Hi , затем вместо выполнения функции setTimeout он входит в систему End и затем запускает функцию setTimeout . Сначала, как обычно, была зарегистрирована инструкция Hi. Поскольку мы используем браузеры для запуска JavaScript, существуют веб-API, которые обрабатывают эти вещи для пользователей. Итак, что делает JavaScript, так это передает функцию setTimeout в таком веб-API, а затем мы продолжаем запускать наш код как обычно. Таким образом, он не блокирует выполнение остальной части кода, и после выполнения всего кода он попадает в стек вызовов и затем, наконец, выполняется. Это то, что происходит в асинхронном JavaScript. Как работает асинхронный JavaScript за кулисами Чтобы лучше понять асинхронное поведение, важно знать о среде выполнения JavaScript, в частности о цикле событий и стеке вызовов: Стек вызовов: Стек вызовов - это место, где функции выполняются в порядке их вызова. При синхронных операциях каждая функция добавляется в стек и выполняется перед переходом к следующей. Веб-API (в браузерах): Такие функции, как setTimeout, HTTP-запросы и прослушиватели событий, обрабатываются веб-API в браузере. Когда вызывается асинхронная функция, такая как setTimeout , она передается этим веб-API, которые управляют синхронизацией, не блокируя основной стек вызовов. Очередь обратного вызова: Как только веб-API завершает свою работу (например, ожидает истечения времени ожидания), он помещает функцию обратного вызова (например, в setTimeout ) в очередь обратного вызова. Цикл событий: Цикл событий непрерывно проверяет стек вызовов. Если он пуст, он помещает функции из очереди обратного вызова в стек для выполнения. Вот почему отложенное сообщение “Geek” регистрируется после завершения другого кода.

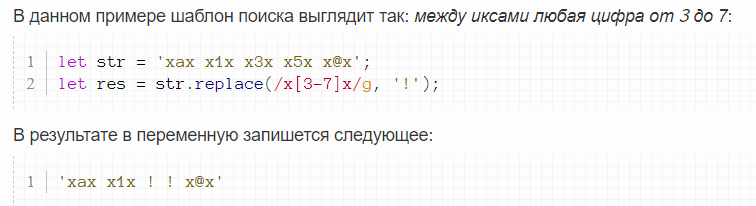
Группы символов \d и \w не очень гибкие. Даже такая простая задача, как найти все буквы, но не цифры - не может быть решена ими. Для таких задач следует использовать квадратные скобки, представляющие собой операцию 'или'. Квадратные скобки заменяют собой один символ, любой из перечисленных внутри. К примеру, вот так: /x[abc]x/ - мы говорим, что по краям должны стоять буквы икс, а внутри - один символ: или 'a', или 'b', или 'c'. После квадратных скобок можно писать операторы повторения. К примеру, вот так: /x[abc]+x/ - мы говорим, что внутри иксов может быть любое количество символов 'a', 'b' и 'c' - в любых комбинациях. Можно не только перечислять символы, но создавать группы символов, записывая между двумя символами дефис. К примеру, вот так: [a-d] - мы получаем все символы от 'a' до 'd'. Давайте посмотрим на примерах.

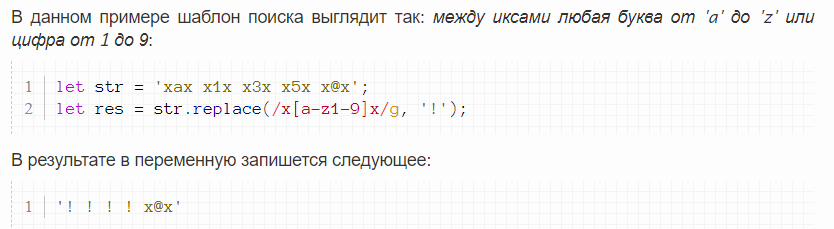












В JavaScript существует несколько типов символов, которые могут вызывать проблемы при написании кода или при работе с данными. Вот некоторые из них:

1. \*\*Специальные символы\*\*:

- Символы, такие как `@`, `#`, `$`, `%`, `^`, `&`, `\*`, `(`, `)`, `-`, `+`, `=`, `{`, `}`, `[`, `]`, `|`, `\`, `:`, `;`, `"`, `'`, `<`, `>`, `,`, `.`, `?`, `/`, могут вызывать синтаксические ошибки, если они используются неправильно.

2. \*\*Пробелы и невидимые символы\*\*:

- Невидимые символы (например, неразрывные пробелы или символы табуляции) могут привести к ошибкам, если они случайно окажутся в коде. Они могут быть трудны для обнаружения.

3. \*\*Кодировка символов\*\*:

- При работе с текстами, особенно в разных кодировках (например, UTF-8, ISO-8859-1), могут возникнуть проблемы с отображением или обработкой символов.

4. \*\*Юникод\*\*:

- Символы, которые выходят за пределы стандартного ASCII, могут вызвать проблемы, если не обрабатываются должным образом. Например, использование юникодных символов в строках или идентификаторах может привести к неожиданному поведению.

5. \*\*Экранирование символов\*\*:

- Некоторые символы, такие как кавычки (`"` или `'`), обратный слеш (`\`), требуют экранирования, если они используются внутри строк. Неправильное экранирование может привести к синтаксическим ошибкам.

6. \*\*Зарезервированные слова\*\*:

- Использование зарезервированных слов JavaScript, таких как `if`, `else`, `function`, `return`, и т.д., в качестве имен переменных или функций может вызвать ошибки.

7. \*\*Символы в регулярных выражениях\*\*:

- Некоторые символы имеют специальное значение в регулярных выражениях (например, `\*`, `+`, `?`, `.` и т.д.), и их необходимо экранировать, если вы хотите использовать их как обычные символы.

Чтобы избежать проблем с символами в JavaScript, рекомендуется:

- Внимательно следить за синтаксисом и структурой кода.

- Использовать редакторы кода, которые помогают выявлять ошибки.

- Проверять код на наличие невидимых символов и неправильной кодировки.

- Обращать внимание на экранирование символов в строках и регулярных выражениях.

**Лекция номер 4**

**ECMAScript**

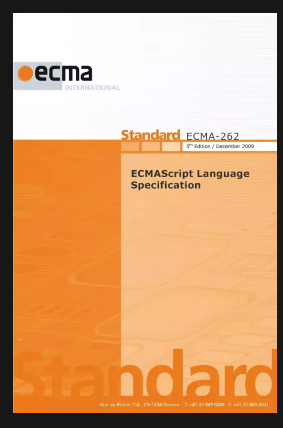
Чем отличаются JavaScript и ECMAScript?

Как-то мы попробовали гуглить «различие между JavaScript и ECMAScript». В итоге пришлось пробираться сквозь море двусмысленных и, казалось бы, противоречивых результатов: «ECMAScript — это стандарт». «JavaScript — это стандарт». «ECMAScript — это спецификация». «JavaScript — это реализация стандарта ECMAScript». «ECMAScript — стандартизованный JavaScript». «ECMAScript — это язык». «JavaScript — это диалект ECMAScript». «ECMAScript — это JavaScript».

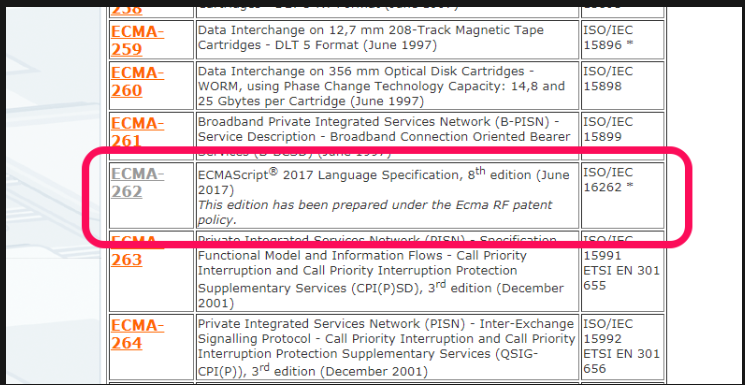
Сдерживая желание заплакать, мы постарались овладеть собой и провели трудоемкое, но продуктивное исследование. В этой статье представлено наше понимание различий между JavaScript и ECMAScript. Статья написана для тех, кто знаком с JavaScript, но хочет получить более четкое представление о его взаимоотношениях с ECMAScript, веб-браузерами, Babel и так далее. Вы также узнаете о скриптовых языках, движках JavaScript и средах выполнения JavaScript. Ну что, погнали. Словарь JavaScript/ECMAScript Ниже приведен список определений, составленный с упором на согласованность и ясность. Определения не совсем полны. Они построены таким образом, чтобы вы хорошо разобрались в связях и родстве между JavaScript и ECMAScript. Ecma International Организация, которая создает стандарты для технологий.

Чтобы проиллюстрировать пример «стандарта» (хотя и созданного не в Ecma): вспомните обо всех клавиатурах, которые вы когда-либо использовали. В подавляющем большинстве случаев раскладка была одинаковой? А пробел, клавиша Enter, курсорные клавиши? Цифры находились в верхнем ряду? Это объясняется тем, что большинство разработчиков используют при создании дизайна клавиатуры стандарт раскладки

ECMA-262 Это стандарт, изданный Ecma International. В нём прописана спецификация скриптового языка общего назначения.



ECMA-262 — это стандарт, подобный QWERTY, только представляющий собой спецификацию скриптового языка, называющегося ECMAScript. ECMA-262 можно считать учётным номером ECMAScript.



Скриптовый язык Язык программирования, созданный для работы с существующим объектом или системой. Чтобы понять, как язык программирования становится скриптовым языком, рассмотрим команды «ходить», «бегать» и «прыгать». Их кто-то должен выполнять, например, человек, собака или персонаж видеоигры. Без выполняющего команды актера все эти «ходить», «бегать» и «прыгать» не имеют смысла. Этот набор действий аналогичен скриптовому языку, который предназначен для управления внешним объектом. ECMAScript Описанная в ECMA-262 спецификация создания скриптового языка общего назначения. Синоним: спецификация ECMAScript.

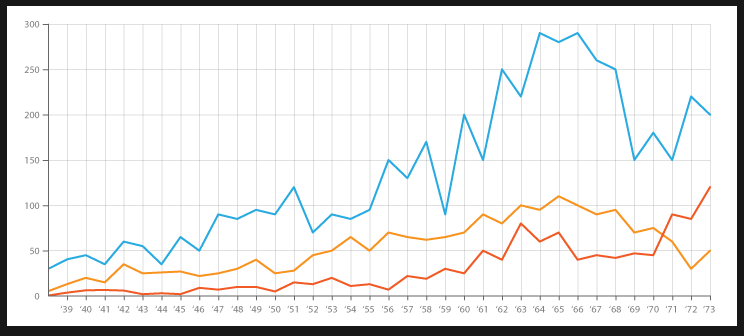
«ECMA-262» — это название и стандарта, и спецификации скриптового языка ECMAScript. ECMAScript содержит правила, сведения и рекомендации, которые должны соблюдаться скриптовым языком, чтобы он считался совместимым с ECMAScript.

JavaScript Скриптовый язык общего назначения, соответствующий спецификации ECMAScript. Это диалект языка ECMAScript.

JavaScript — мой любимый язык программирования, пропитанный ароматом кофе. ECMAScript — это спецификация, на которой он основан. Из спецификации ECMAScript вы узнаете, как создать скриптовый язык, а из документации JavaScript — как использовать скриптовый язык. Когда люди называют JavaScript «диалектом языка ECMAScript», они имеют в виду то же самое, что и при упоминании о диалектах английского, французского или китайского языка. Диалект большую часть своего лексикона и синтаксиса наследует из родительского языка, от которого при этом заметно отличается. Как сказано в ECMA-262, JavaScript в основном реализует спецификацию ECMAScript, но с некоторыми отличиями. Mozilla в общих чертах обрисовала свойства JavaScript, не характерные для ECMAScript:

JavaScript-движок Программа или интерпретатор, способный понимать и выполнять JavaScript-код. Синонимы: интерпретатор JavaScript, реализация JavaScript.

JavaScript-движки обычно используются в веб-браузерах, включая V8 в Chrome, SpiderMonkey в Firefox и Chakra в Edge. Каждый движок подобен языковому модулю, который позволяет приложению поддерживать определенное подмножество языка JavaScript. Для браузера JavaScript-движок — это как для человека способность понимать речь. Если опять обратиться к нашему примеру с «ходить», «бегать» и «прыгать», то JavaScript-движок — часть «объекта», который на самом деле понимает, что означают эти действия. Эта аналогия помогает объяснить некоторые особенности браузера:

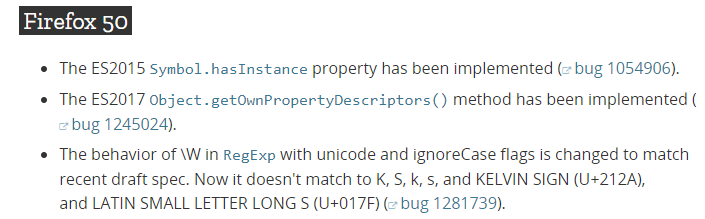


Разное быстродействие браузеров Два человека поймут команду «прыгать», но один из них отреагирует раньше, потому что смог быстрее понять и обработать команду. Аналогично, два браузера могут понимать код JavaScript, но один из них работает быстрее, потому что его JavaScript-движок работает эффективнее.



Разная поддержка в браузерах Рассмотрим большую группу людей, говорящих по-английски. Кто-то среди них может знать определенные слова, выражения и синтаксические правила, не известные другим людям, и наоборот. То же самое с браузерами. Хотя все браузерные JavaScript-движки понимают JavaScript, некоторые понимают его лучше других. Различаются и способы поддержки языка браузерами. Говоря о поддержке в браузерах, обычно упоминают о «совместимости с ECMAScript», а не о «совместимости с JavaScript», хотя JavaScript-движки детально анализируют и выполняют… JavaScript. Звучит запутанно, позвольте объяснить.

Как вы помните, ECMAScript — это спецификация того, как может выглядеть скриптовый язык. Появление новой версии ECMAScript не означает, что у всех движков JavaScript появятся новые функции. Всё зависит от групп или организаций, которые отвечают за обновления JavaScript-движков с учётом новейшей спецификацией ECMAScript. Поэтому разработчики, как правило, задают вопросы вроде «Какую версию ECMAScript поддерживает этот браузер?» или «Какие функции ECMAScript поддерживает этот браузер?» Они хотят знать, удалось ли Google, Mozilla и Microsoft наделить JavaScript-движки (соответственно V8, SpiderMonkey и Chakra) свойствами, описанными в последней версии ECMAScript. Ответы на многие вопросы вы найдёте в Таблице совместимости ECMAScript. Если выйдет новая версия ECMAScript, то в JavaScript-движках не появятся разом все нововведения. Они будут внедряться постепенно, релиз за релизом, как это видно на примере журнала изменений JavaScript в Firefox:



Какие-то части стандартов ES2015 и ES2017 были реализованы в JavaScript-движке SpiderMonkey в браузере Firefox 50. Ещё раньше были реализованы другие части ES2015 и ES2017, и это продолжилось в последующих версиях. Среда выполнения JavaScript В этой среде выполняется JavaScript-код и интерпретируется JavaScript-движком. Среда выполнения предоставляет хост-объекты, на которых и с которыми может работать JavaScript. Синонимы: хост-среда (Host environment).

Среда выполнения JavaScript — это «существующий объект или система», упомянутые в определении скриптового языка. Код проходит через JavaScript-движок, в котором объект или система анализирует код и разбирает его работу, а потом выполняет интерпретированные действия. Собака идет, человек бежит, персонаж видеоигры прыгает (или разрушает, как на картинке). JavaScript-скрипты могут обращаться к приложениям, потому что те предоставляют «хост-объекты» в среде выполнения. На клиентской стороне средой выполнения JavaScript будет веб-браузер, в котором становятся доступными для манипуляций такие хост-объекты, как окна и HTML-документы. Вы когда-нибудь работали с хост-объектами в виде окна или документа? Такие объекты не являются частью базового JavaScript. Это веб-API, объекты, предоставляемые браузером, действующим как хост-среда JavaScript. На серверной стороне среда выполнения JavaScript — это Node.js. В Node.js предоставляются связанные с сервером хост-объекты, такие как файловая система, процессы и запросы. Любопытно, что разные среды выполнения JavaScript могут использовать один и тот же JavaScript-движок. Например, V8 — это движок, используемый в двух совершенно разных средах — в Google Chrome и Node.js. ECMAScript 6 Это шестая редакция стандарта ECMA-262, внёсшая в спецификацию ECMAScript существенные изменения и улучшения. Синонимы: ES6, ES2015 и ECMAScript 2015.

С 2015 года Ecma International перешла на ежегодные релизы ECMAScript, и эту версию ECMAScript переименовали с ES6 на ES2015. Ecma International стала называть новые версии спецификации ECMAScript в соответствии с годом выпуска. То есть ES6 и ES2015 — это одно и то же. Babel Транспилятор, конвертирующий код ES6 в код ES5.

В ES6 разработчикам доступны замечательные новые функции, но нельзя забывать и о межбраузерной совместимости. Во время написания этой статьи Edge и Internet Explorer поддерживают не все функции из спецификации ES6. При желании, разработчики могут с помощью Babel преобразовать — транспилировать — код ES6 в функционально эквивалентную версию, использующую только функции ES5. Все основные браузеры полностью поддерживают ES5, поэтому транспилированный код будет работать без каких-либо проблем. Ещё кое-что Надеюсь, вы узнали что-то для себя новое о JavaScript и ECMAScript. Прежде чем закончить, хотелось бы ещё кое-что разъяснить начинающим веб-разработчикам. Курица или яйцо JavaScript был создан в 1996 году. В 1997 году Ecma International предложила стандартизировать JavaScript, и в результате появился ECMAScript. Но поскольку JavaScript соответствует спецификации ECMAScript, JavaScript является примером реализации ECMAScript. Получается, что ECMAScript основан на JavaScript, а JavaScript основан на ECMAScript. Да, это смахивает на затёртый кинематографический штамп, когда путешественники во времени становятся своими собственными родителями. Забавно, хотя и немного странно. Понимаем, что всё здесь сказанное немного вас позабавило, но надеемся, что вы получили пищу для размышлений. Позвольте откланяться.

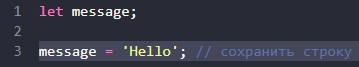
**Лекция номер 5**

**Объявление переменной**

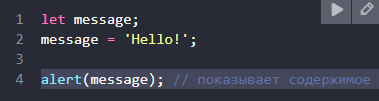
Переменные JavaScript-приложению обычно нужно работать с информацией. Например: Интернет-магазин – информация может включать продаваемые товары и корзину покупок. Чат – информация может включать пользователей, сообщения и многое другое. Переменные используются для хранения этой информации. Переменная 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) переменную с именем «message»:



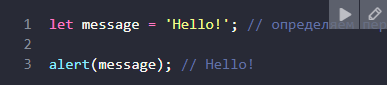
Теперь можно поместить в неё данные (другими словами, определить переменную), используя оператор присваивания =:



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



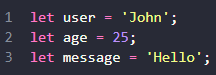
Для краткости можно совместить объявление переменной и запись данных в одну строку:



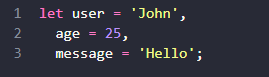
Мы также можем объявить несколько переменных в одной строке:



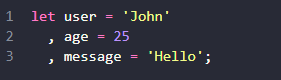
Такой способ может показаться короче, но мы не рекомендуем его. Для лучшей читаемости объявляйте каждую переменную на новой строке. Многострочный вариант немного длиннее, но легче для чтения:



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



…Или даже с запятой в начале строки:



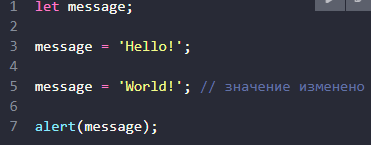
В принципе, все эти варианты работают одинаково. Так что это вопрос личного вкуса и эстетики.



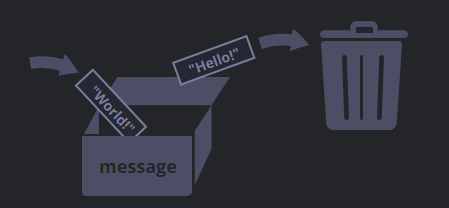
Аналогия из жизни Мы легко поймём концепцию «переменной», если представим её в виде «коробки» для данных с уникальным названием на ней. Например, переменную message можно представить как коробку с названием "message" и значением "Hello!" внутри:



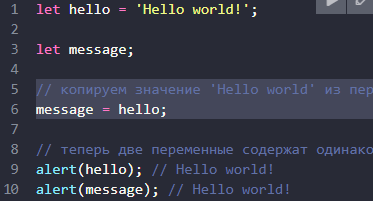
Мы можем положить любое значение в коробку. Мы также можем изменить его столько раз, сколько захотим:



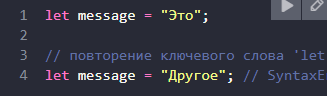
При изменении значения старые данные удаляются из переменной:



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



Повторное объявление вызывает ошибку Переменная может быть объявлена только один раз. Повторное объявление той же переменной является ошибкой:



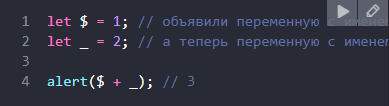
Поэтому следует объявлять переменную только один раз и затем использовать её уже без let.

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.

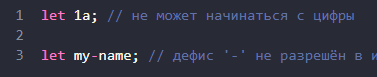
Имена переменных В JavaScript есть два ограничения, касающиеся имён переменных: Имя переменной должно содержать только буквы, цифры или символы $ и \_. Первый символ не должен быть цифрой. Примеры допустимых имён:



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:



Примеры неправильных имён переменных:



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:

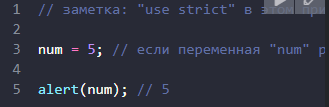


Технически здесь нет ошибки, такие имена разрешены, но есть международная традиция использовать английский язык в именах переменных. Даже если мы пишем небольшой скрипт, у него может быть долгая жизнь впереди. Людям из других стран, возможно, придётся прочесть его не один раз.

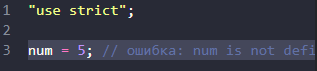
Зарезервированные имена Существует список зарезервированных слов, которые нельзя использовать в качестве имён переменных, потому что они используются самим языком. Например: let, class, return и function зарезервированы. Приведённый ниже код даёт синтаксическую ошибку:



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.



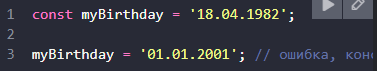
Это плохая практика, которая приводит к ошибке в строгом режиме:



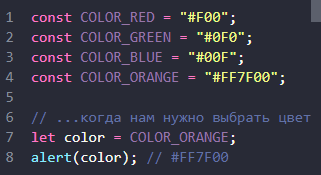
Константы Чтобы объявить константную, то есть, неизменяемую переменную, используйте const вместо let:



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



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»:



Преимущества: COLOR\_ORANGE гораздо легче запомнить, чем "[#FF7F00](tg://search_hashtag?hashtag=FF7F00)". Гораздо легче допустить ошибку при вводе "[#FF7F00](tg://search_hashtag?hashtag=FF7F00)", чем при вводе COLOR\_ORANGE. При чтении кода COLOR\_ORANGE намного понятнее, чем [#FF7F00](tg://search_hashtag?hashtag=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:



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. Именование 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.

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.

**Лекция номер 6**

**Консоль браузера**

Сегодня мы публикуем заметку, посвящённую особенностям использования JavaScript-консоли в браузерах, лежащим за пределами широко известной команды console.log(). Собственно говоря, эта команда представляет собой простейший инструмент для отладки программ, который позволяет выводить что-либо в консоль. Однако знание некоторых особенностей этого инструмента позволит тем, кто им пользуется, повысить эффективность работы.

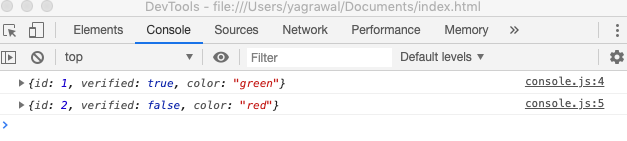
Простейший вариант использования console.log() заключается, например, в выводе некоей строки или объекта. Например, выведем в консоль строку:



Теперь представим себе, что в консоль нужно вывести несколько объектов. Например — таких:

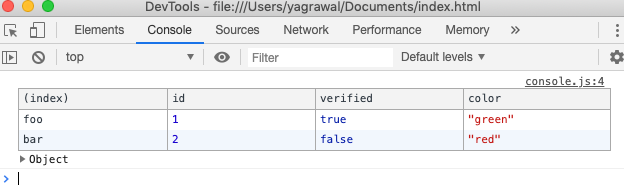


Пожалуй, логичнее всего будет воспользоваться для решения этой задачи несколькими командами вида console.log(variable). Хотя данные в консоль и попадают, при их выводе понятной становится одна проблема. Взглянем на то, что выводится в консоль.

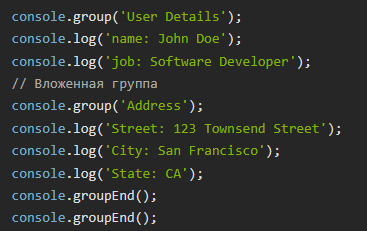


Как можно видеть, имён переменных, foo и bar, здесь нет. Объекты, пользуясь значком в виде стрелки в левых частях строк, можно разворачивать, но даже так, глядя на внутреннюю структуру объектов, понять, какой именно объект выведен в консоль, может быть весьма непросто. В решении этой проблемы нам помогут вычисляемые имена свойств объектов. А именно, эта особенность объектных литералов, появившаяся в ES6, позволяет пользоваться удобной конструкцией следующего вида:

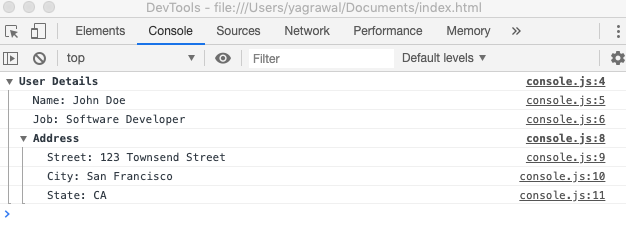
При таком подходе в консоль попадёт объект, имена свойств которого будут представлять собой имена переменных-объектов, которые нужно вывести. Кроме того, это позволяет избавиться от некоторых вызовов console.log(), используемых ранее для вывода объектов по отдельности. Команда console.table() Улучшать внешний вид того, что программа выводит в консоль, можно и дальше, оформив содержимое объектов в виде таблицы. Это хорошо скажется на читаемости информации. А именно, речь идёт о том, что если вы выводите в консоль объекты с одинаковыми именами свойств, или массивы похожих объектов, вы можете воспользоваться командой console.table(). Вот как выглядит результат выполнения команды вида console.table({ foo, bar }).



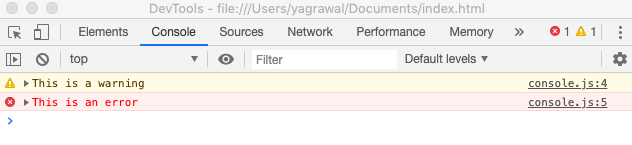
Команда [console.group](https://console.group/)() Эту команду можно использовать в том случае, если нужно сгруппировать некие связанные данные и создать структуры из вложенных групп, которые повышают удобство работы с такими данными. Кроме того, этот подход можно использовать в тех случаях, когда в некоей функции выполняется несколько команд вывода чего-либо в консоль, и нужно, чтобы можно было бы чётко, с одного взгляда, отделить результаты выполнения таких команд от других. Предположим, мы выводим в консоль сведения о неких пользователях:



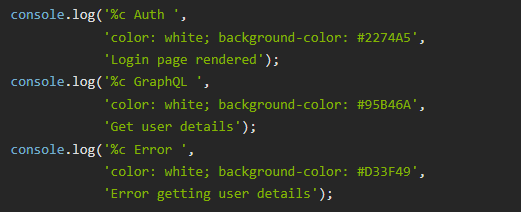
Вот как выглядят результаты работы этого кода.



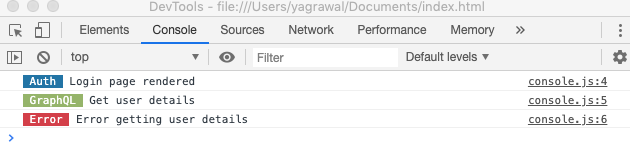
При использовании команды [console.group](https://console.group/)() группы, по умолчанию, выводятся в развёрнутом виде. Для того, чтобы они выводились свёрнутыми, вместо этой команды можно воспользоваться командой console.groupCollapsed(). Для того, чтобы просмотреть содержимое такой группы, её понадобится развернуть с помощью значка, находящегося слева от имени группы. Команды console.warn() и console.error() В зависимости от ситуации, для того чтобы подчеркнуть важность некоторых сообщений, выводимых в консоль, вам могут пригодиться команды console.warn() и console.error(). Они используются, соответственно, для вывода предупреждений и ошибок.



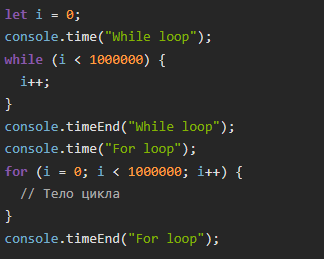
Возможно, вам пригодится и команда [console.info](https://console.info/)(), которая предназначена для вывода информационных сообщений. В настройке внешнего вида сообщений, выводимых в консоль, можно пойти и ещё дальше, самостоятельно их стилизовав. Для стилизации текстов, выводимых в консоль, можно воспользоваться директивой %c. Это может оказаться полезным, например, для организации визуального разделения сведений, поступающих от подсистем выполнения обращений к неким API, от подсистем, ответственных за обработку событий, генерируемых пользователем, и так далее. Главное тут — выработать некие правила стилизации и их придерживаться. Вот пример настройки внешнего вида данных, выводимых в консоль:



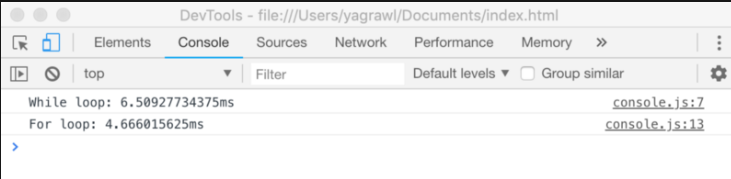
Тут же можно настраивать и другие CSS-свойства текста, наподобие font-size и font-style.



Команда console.trace() Команда console.trace() выводит в консоль результаты трассировки стека и позволяет судить о том, что произошло в определённом месте программы во время её выполнения. Например, существуют некоторые методы, которые, в определённых ситуациях, нужно вызывать лишь один раз, скажем — методы для удаления информации из базы данных. Проверить, действительно ли выполняется лишь однократный вызов подобного метода, можно с помощью console.trace(). Эта команда позволяет вывести в консоль сведения, которые помогают проконтролировать правильность работы внутренних механизмов программ. Команда console.time() Одна из важных задач, встающая перед фронтенд-разработчиком, заключается в том, чтобы обеспечить высокую скорость работы кода. Команда console.time() позволяет замерять время выполнения операций и выводить то, что удалось выяснить, в консоль. Например, исследуем с помощью этой команды пару циклов:



Взглянем на то, что попало в консоль после выполнения этого кода.

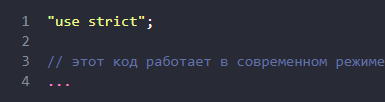


В этом материале мы рассмотрели некоторые полезные мелочи, касающиеся вывода данных в консоли браузеров. Если раньше вы об этих возможностях не знали — надеемся, теперь у вас появились новые полезные JavaScript-инструменты. Уважаемые читатели! Если вы занимаетесь разработкой больших JavaScript-проектов — просим вас рассказать о том, какими средствами вы решаете в них проблемы логирования.

**Лекция номер 7**

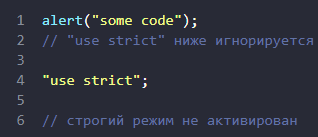
**Строгий режим**

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:



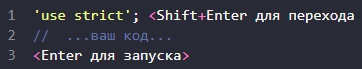
Совсем скоро мы начнём изучать функции (способ группировки команд), поэтому заранее отметим, что в начале большинства видов функций можно поставить "use strict". Это позволяет включить строгий режим только в конкретной функции. Но обычно люди используют его для всего файла.

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:

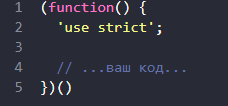


Над "use strict" могут быть записаны только комментарии.

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:



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:



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.

**Лекция номер 8**

**Кэширование файлов**

Кэширование кода (также называемое кэшированием байт-кода) является важным инструментом оптимизации. Оно уменьшает время запуска часто посещаемых сайтов за счёт кэширования результатов парсинга и компиляции. Большинство популярных браузеров реализует кэширование в некоторой форме, и Chrome не исключение. О том, как Chrome и V8 кэшируют скомпилированный код, уже много всего написано и сказано. В этой статье вы найдёте несколько советов JS-разработчикам, которые хотят с помощью кэширования кода улучшить загрузку сайтов. Мы будем говорить о реализации кэширования в Chrome / V8, но большинство советов можно использовать и для кэширования кода других браузеров.

Обзор кэширования кода Кэш оперативной памяти У Chrome есть два уровня кэширования скомпилированного в V8 кода (классических скриптов и скриптов модулей): быстрый и «лучший из возможного» кэш в оперативной памяти, обеспечиваемый средствами V8 (кэш Isolate), а также полный сериализованный кэш на диске. Кэш Isolate работает со скриптами, скомпилированными в том же V8 Isolate (т. е. тот же процесс, грубо говоря «одни и те же страницы сайта при навигации по одной и той же вкладке»). Это «лучшее из возможного» в том смысле, что кэш оперативной памяти, насколько это возможно, быстр и минимален: он использует уже имеющиеся данные за счёт потенциально более низкой частоты обращений и отсутствия кэширования между процессами. Когда V8 компилирует скрипт, скомпилированный байт-код сохраняется в хеш-таблице (в куче V8), ключ которого определяется исходным кодом сценария. Когда Chrome просит V8 скомпилировать другой скрипт, V8 сначала проверяет, соответствует ли исходный код этого скрипта чему-либо в хеш-таблице. Если соответствует, просто возвращается существующий байт-код.

Этот кэш быстрый и эффективный, но на практике у него лишь 80% частоты попаданий. Кэш на диске Кэш кода на диске управляется Chrome (в частности, с помощью Blink) и заполняет пробел, который кэш Isolate не может заполнить: совместное использование кэшей кода между процессами и между несколькими сеансами Chrome. Он использует преимущества существующего кэша HTTP-ресурсов, который управляет кэшированием и очисткой данных с истёкшим сроком действия, полученных по сети. Когда JS-файл запрашивается впервые (т. е. выполняется «холодный» запуск), Chrome загружает его и даёт V8 для компиляции. Он также сохраняет файл в кэше браузера на диске. Когда JS-файл запрашивается во второй раз (т. е. выполняется «тёплый» запуск), Chrome берёт файл из кэша браузера и снова передаёт его в V8 для компиляции. Однако на этот раз скомпилированный код сериализуется и прикрепляется к кэшированному файлу скрипта в качестве метаданных. В третий раз (т. е. «горячий» запуск) Chrome извлекает как файл, так и метаданные файла из кэша и передаёт их в V8. Тот в свою очередь десериализует метаданные и может пропустить компиляцию.

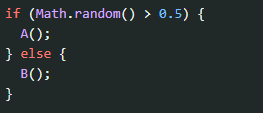
В итоге:



Основываясь на этом описании, можно оптимизировать использование кэшей кода на вашем сайте. Совет 1: не делайте ничего Лучшее, что JS-разработчик может сделать для оптимизации кэширования кода, — ничего не делать. Но ничего не делать можно по-разному: пассивно и активно. Кэширование кода в конце концов является частью реализации браузера. По сути, это увеличение производительности за счёт дополнительных расходов памяти, реализация и эвристика которых могут постоянно меняться. Мы, как разработчики V8, должны делать всё возможное, чтобы эти эвристики работали для каждого. Если чрезмерно оптимизировать кэширование кода, можно очень разочароваться уже после нескольких релизов, когда эти детали изменятся. Кроме того, другие механизмы JavaScript могут иметь различные эвристики для своей реализации кэширования кода. Так что во многих отношениях лучший совет для получения кэшированного кода похож на совет по написанию JS: пишите чистый идиоматический код и сделайте всё возможное, чтобы оптимизировать его кэширование. Помимо пассивного бездействия, вы также должны стараться активно «ничего не делать». Любая форма кэширования по своей природе зависит от того, что ничего не меняется, поэтому бездействие — лучший способ сохранить кэшированные данные. Активно ничего не делать можно разными способами.

Не меняйте код Всякий раз, когда вы отправляете новый код, он ещё не кэширован. Когда браузер делает HTTP-запрос для URL-адреса сценария, он может включать дату последней выборки этого URL-адреса. Если сервер знает, что файл не изменился, он может отправить ответ «304 Not Modified», который сохраняет кэш кода «горячим». В противном случае ответ «200 OK» обновляет кэшированный ресурс и очищает кэш кода, возвращая его обратно в «холодный» режим. Всегда хочется сразу же отправить последние изменения кода, особенно если вы хотите измерить влияние конкретного изменения, но для кэша лучше позволить коду существовать как он есть, по крайней мере, обновлять его как можно реже. Подумайте о том, чтобы установить ограничение «≤ Х развёртываний в неделю», где Х — это слайдер, который вы можете регулировать для баланса между кэшированием и устареванием данных.

Не меняйте URL’ы Кэш кода связан с URL-адресом скрипта, так как это облегчает поиск, ведь нет необходимости читать фактическое содержимое скрипта. Это означает, что изменение URL-адреса (включая любые параметры запроса) создаёт новую запись в кэше ресурсов, а вместе с ним и новую запись «холодного» кэша. Конечно, это также можно использовать для принудительной очистки кэша, хоть это и является частью реализации. Однажды мы можем решить связать кэши с исходным текстом, а не с исходным URL, и этот совет перестанет работать. Не меняйте поведение выполнения Одна из самых свежих оптимизаций кэширования кода заключается в сериализации скомпилированного кода только после его выполнения. Это делается для того, чтобы попытаться поймать лениво скомпилированные функции, которые инициализируются только во время выполнения, а не во время начальной компиляции. Эта оптимизация работает лучше всего, когда каждый запуск скрипта выполняет один и тот же код или хотя бы одинаковые функции. Это может быть проблемой, если у вас есть, например A/B-тесты, которые зависят от выбора времени выполнения:

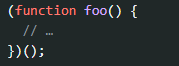


В этом случае только A() или B() компилируются и выполняются при «горячем» запуске и вводятся в кэш кода, но любой из них может быть выполнен и в последующих запусках. Вместо этого надо попытаться сохранить выполнение детерминированным, чтобы сохранить его в кэшированном пути. Совет 2: сделайте что-нибудь Конечно, совет не делать ничего (пассивно или активно) не очень удовлетворит вас. Помимо того что вы ничего не делаете, учитывая текущую эвристику и реализацию, есть несколько вещей, которые вы можете сделать. Эвристика может измениться, сам этот совет может измениться и нет никакой альтернативы для профилирования.

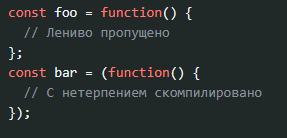
Отделите библиотеки от кода Кэширование кода имеет свои особенности: изменения в любой части сценария делают недействительной кэш-память для всего сценария. Если ваш бандл состоит из стабильных и изменяющихся частей в одном скрипте, например библиотеки и бизнес-логики, то изменения в коде бизнес-логики изменяют кэш кода библиотеки. Вместо этого вы можете отделить код стабильной библиотеки в самостоятельный скрипт и включить его отдельно. Затем код библиотеки может быть кэширован один раз и оставаться в кэше при изменении бизнес-логики. Это даёт дополнительные преимущества, если библиотеки совместно используются на разных страницах вашего сайта, поскольку к сценарию прикреплён кэш кода. Кэш кода для библиотек также разделяется между страницами.

Объедините библиотеки с кодом Кэширование кода выполняется после выполнения каждого скрипта. Это означает, что кэш будет включать в себя именно те функции в этом скрипте, которые были скомпилированы по завершении его выполнения. Это ведёт к нескольким важным последствиям для библиотечного кода: кэш кода не будет включать функции из более ранних скриптов; кэш кода не будет включать в себя лениво скомпилированные функции, вызываемые более поздними скриптами. Если библиотека состоит из полностью лениво скомпилированных функций, эти функции не будут кэшироваться, даже если они будут использоваться позже. Одним из решений этой проблемы является объединение библиотек в единый сценарий, таким образом кэширование кода «видит», какие части библиотеки используются. Это полная противоположность совету выше, универсального решения нет. Конечно, не рекомендуется объединять все ваши JS-скрипты в один большой бандл. В целом разделение его на несколько более мелких скриптов будет полезнее (например, множественные сетевые запросы, потоковая компиляция, интерактивность страниц и т. д.).

Используйте преимущество эвристики IIFE Только те функции, которые скомпилированы к моменту, когда завершится выполнение скрипта, учитываются в кэше, поэтому существует много видов функций, которые не будут кэшироваться, несмотря на выполнение в более поздний момент. Обработчики событий (даже onload()), цепочки промисов, неиспользуемые библиотечные функции и всё, что лениво компилируется без вызова к моменту, когда </script> виден — всё это остаётся ленивым и не кэшируется. Один из способов сделать эти функции кэшированными — заставить их компилироваться. Распространённым способом принудительной компиляции является использование эвристики IIFE. IIFE (immediately-invoked function expressions) — это шаблон, в котором функция вызывается сразу после создания:



Так как IIFE вызываются немедленно, большинство движков JavaScript пытаются обнаружить их и немедленно скомпилировать, чтобы избежать затрат на ленивую компиляцию с последующей полной компиляцией. Существуют различные эвристики для раннего обнаружения IIFE (до анализа функции), наиболее распространённой из которых является символ «(» перед ключевым словом function. Поскольку эта эвристика применяется рано, она запускает компиляцию, даже если функция на самом деле вызывается не сразу:



Функции, которые должны находиться в кэше, можно принудительно ввести в него, заключив их в скобки. Но это может привести к тому, что время загрузки будет страдать, если совет будет применён неправильно. Это своего рода злоупотребление эвристикой, поэтому не следует так делать без необходимости.

Группируйте небольшие файлы вместе В Chrome — минимальный размер для кэшей кода, сейчас это 1 КБ исходного кода. Сценарии меньше не кэшируются, так как затраты будут больше выгоды. Если на вашем сайте много таких небольших сценариев, подсчёт затрат может больше не применяться аналогичным образом. Лучше объединить их так, чтобы они превышали минимальный размер кэширования кода, и получить выгоду от общего сокращения затрат скрипта. Избегайте встроенных скриптов Теги скриптов, исходный код которых встроен в HTML, не имеют внешнего исходного файла, с которым они связаны. Поэтому они не могут быть кэшированы с помощью этого механизма. Chrome пытается кэшировать встроенные скрипты, прикрепляя их к ресурсу документа HTML, но эти кэши становятся зависимыми всего документа HTML — он не должен изменяться и разделяться между страницами. Простые сценарии не стоит встраивать в HTML, лучше выносить их в отдельные файлы.

Используйте кэши сервис-воркера Сервис-воркер — это механизм, позволяющий вашему коду перехватывать сетевые запросы на ресурсы на вашей странице. Они позволяют вам создавать локальный кэш из некоторых ваших ресурсов и обслуживать ресурс из кэша всякий раз, когда их запрашивают. Это особенно полезно для страниц, которые продолжают работать в offline-режиме вроде PWA. Ниже типичный пример сайта, использующего сервисный воркер. Регистрация воркера в основном файле сценария:

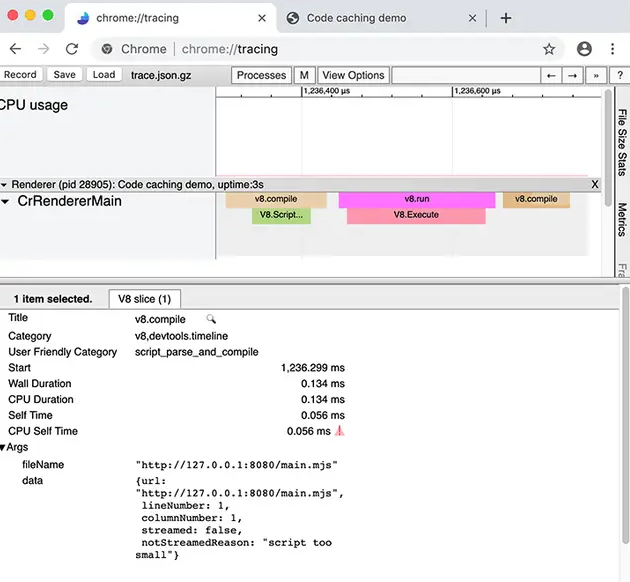


Воркер добавляет обработчики событий для установки (создание кэша) и извлечения (обслуживание ресурсов).

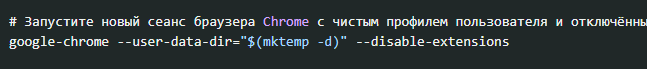


Эти кэши могут включать в себя кэшированные ресурсы JS. Но поскольку ожидается, что кэши воркеров будут преимущественно использоваться для PWA, для них используется немного другая эвристика по сравнению с обычным «автоматическим» кэшированием в Chrome. Во-первых, они сразу же создают кэш кода при каждом добавлении ресурса JS. Это означает, что кэш доступен уже при второй загрузке (а не только при третьей, как в обычном случае). Во-вторых, генерируется «полный» кэш для этих скриптов — функции больше не компилируются лениво. Всё компилируется и помещается в кэш. Преимущество заключается в быстрой и предсказуемой производительности, без каких-либо зависимостей порядка выполнения, хоть и за счёт увеличения использования памяти. Обратите внимание, что такая эвристика применяется только к кэшам сервисных воркеров, а не к другому использованию Cache API. В настоящее время Cache API вообще не выполняет кэширование кода, когда используется вне сервисных воркеров.

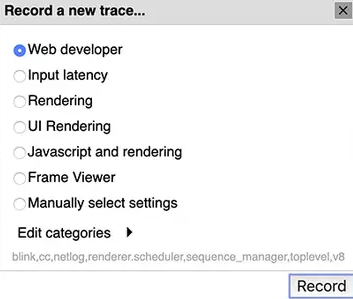
Трассировка Ни один из советов выше не поможет ускорить работу вашего сайта. К сожалению, информация о кэшировании сейчас не предоставляется в DevTools, поэтому наиболее надёжный способ выяснить, какие из сценариев вашего сайта кэшируются, — использовать чуть более низкий уровень chrome://tracing. chrome://tracing записывает инструментальные трассировки Chrome в течение некоторого периода времени с такой визуализацией :

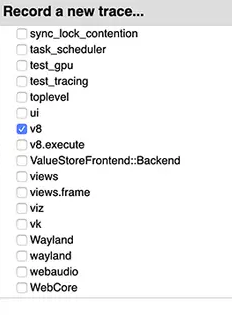


Трассировка записывает поведение всего браузера, включая другие вкладки, окна и расширения, поэтому она лучше всего работает, когда выполняется в чистом профиле пользователя, с отключёнными расширениями и без открытия других вкладок браузера:

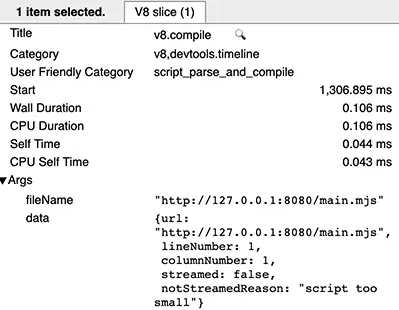


При записи вы должны выбрать, какие категории трассировать. В большинстве случаев вы можете просто выбрать набор категорий «Web developer» (Веб-разработчик), но категории можно выбрать и вручную. Важная категория для кэширования кода — v8.

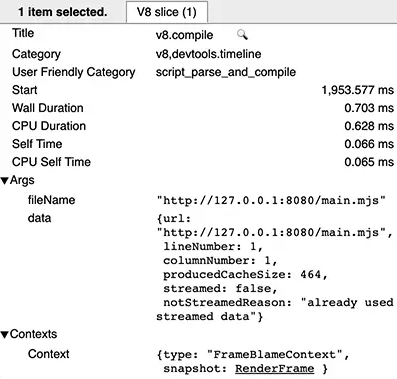




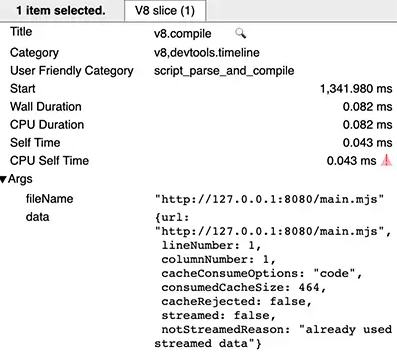
После записи с категорией v8 найдите фрагменты v8.compile в трассировке (в качестве альтернативы вы можете ввести v8.compile в поле поиска интерфейса). Эти компоненты показывают компилируемый файл и некоторые метаданные о компиляции. При холодном запуске сценария информация о кэшировании кода отсутствует. Это означает, что сценарий не участвовал в создании или использовании данных кэша.



При тёплом запуске есть две записи v8.compile на сценарий: одна для фактической компиляции (как указано выше) и одна (после выполнения) для создания кэша. Вы можете узнать последнюю, так как она имеет поля метаданных cacheProduceOptions и producedCacheSize.



При горячем запуске вы увидите запись v8.compile для использования кэша с полями метаданных cacheConsumeOptions и consumedCacheSize. Все размеры выражены в байтах.

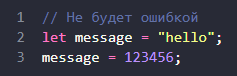


Для большинства разработчиков кэширование кода должно «просто работать». И работает это лучше всего (как и любой кэш), когда всё остаётся неизменным, и с использованием эвристики, которая может меняться между версиями. Тем не менее, кэширование кода имеет особенности, которые можно использовать, и ограничения, которых стоит избегать. Тщательный анализ с использованием chrome://tracing может помочь вам настроить и оптимизировать использование кэша вашим сайтом.

**Лекция номер 9**

**Типы данных**

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:



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



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:



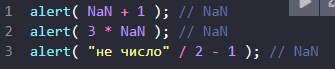
Или задать его явно:



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



Значение NaN «прилипчиво». Любая математическая операция с NaN возвращает 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 (253-1) (т. е. 9007199254740991) или меньшими, чем -(253-1) для отрицательных чисел. Если говорить совсем точно, то, технически, тип number может хранить большие целые числа (до 1.7976931348623157 \* 10308), но за пределами безопасного диапазона целых чисел ±(253-1) будет ошибка точности, так как не все цифры помещаются в фиксированную 64-битную память. Поэтому можно хранить «приблизительное» значение. Например, эти два числа (прямо за пределами безопасного диапазона) совпадают:

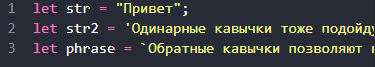


То есть все нечетные целые числа, большие чем (253-1), вообще не могут храниться в типе number. В большинстве случаев безопасного диапазона чисел от -(253-1) до (253-1) вполне достаточно, но иногда нам требуется весь диапазон действительно гигантских целых чисел без каких-либо ограничений или пропущенных 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:

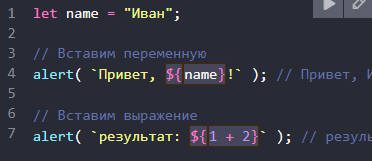


Так как необходимость в использовании BigInt–чисел появляется достаточно редко, мы рассмотрим их в отдельной главе BigInt. Ознакомьтесь с ней, когда вам понадобятся настолько большие числа.

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.



В JavaScript существует три типа кавычек. Двойные кавычки: "Привет". Одинарные кавычки: 'Привет'. Обратные кавычки: Привет. Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript. Обратные же кавычки имеют расширенную функциональность. Они позволяют нам встраивать выражения в строку, заключая их в ${…}. Например:



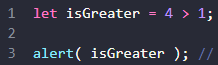
Выражение внутри ${…} вычисляется, и его результат становится частью строки. Мы можем положить туда всё, что угодно: переменную name, или выражение 1 + 2, или что-то более сложное. Обратите внимание, что это можно делать только в обратных кавычках. Другие кавычки не имеют такой функциональности встраивания!



Мы рассмотрим строки более подробно в главе Строки. 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:



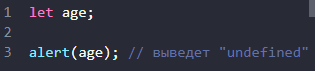
Булевые значения также могут быть результатом сравнений:



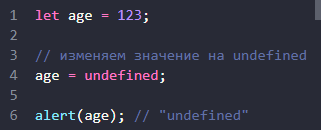
Мы рассмотрим булевые значения более подробно в главе Логические операторы. Значение «null» Специальное значение null не относится ни к одному из типов, описанных выше. Оно формирует отдельный тип, который содержит только значение null:



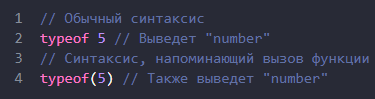
В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно». В приведённом выше коде указано, что значение переменной age неизвестно. Значение «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:



Технически мы можем присвоить значение undefined любой переменной:



…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:



Если передается выражение, то нужно заключать его в скобки, т.к. typeof имеет более высокий приоритет, чем бинарные операторы:



Другими словами, скобки необходимы для определения типа значения, которое получилось в результате выполнения выражения в них. Вызов typeof x возвращает строку с именем типа:



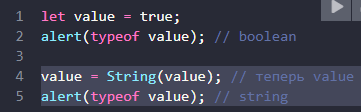
Последние три строки нуждаются в пояснении: Math — это встроенный объект, который предоставляет математические операции и константы. Мы рассмотрим его подробнее в главе Числа. Здесь он служит лишь примером объекта. Результатом вызова typeof null является "object". Это официально признанная ошибка в typeof, ведущая начало с времён создания JavaScript и сохранённая для совместимости. Конечно, null не является объектом. Это специальное значение с отдельным типом. 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.

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 ±(253-1). 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.

**Лекция номер 10**

**Преобразование типов**

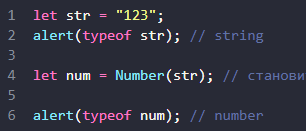
Чаще всего операторы и функции автоматически приводят переданные им значения к нужному типу. Например, alert автоматически преобразует любое значение к строке. Математические операторы преобразуют значения к числам. Есть также случаи, когда нам нужно явно преобразовать значение в ожидаемый тип. Пока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



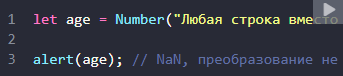
Преобразование происходит очевидным образом. false становится "false", null становится "null" и т.п. Численное преобразование Численное преобразование происходит в математических функциях и выражениях. Например, когда операция деления / применяется не к числу:



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



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:



Правила численного преобразования:

Значение Преобразуется в…

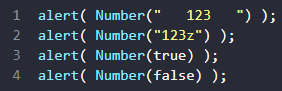
undefined NaN

null 0

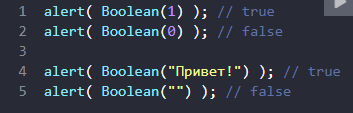
true / false

1 / 0

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



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:



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.

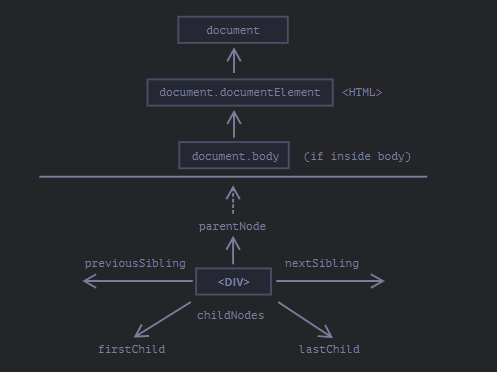


Существует 3 наиболее широко используемых преобразования: строковое, численное и логическое. Строковое – Происходит, когда нам нужно что-то вывести. Может быть вызвано с помощью String(value). Для примитивных значений работает очевидным образом. Численное – Происходит в математических операциях. Может быть вызвано с помощью Number(value). Преобразование подчиняется правилам: Значение Становится… undefined NaN null 0 true / false 1 / 0 string Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то получаем 0, иначе из непустой строки «считывается» число. При ошибке результат NaN. Логическое – Происходит в логических операциях. Может быть вызвано с помощью Boolean(value). Подчиняется правилам: Значение Становится… 0, null, undefined, NaN, "" false любое другое значение true 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.

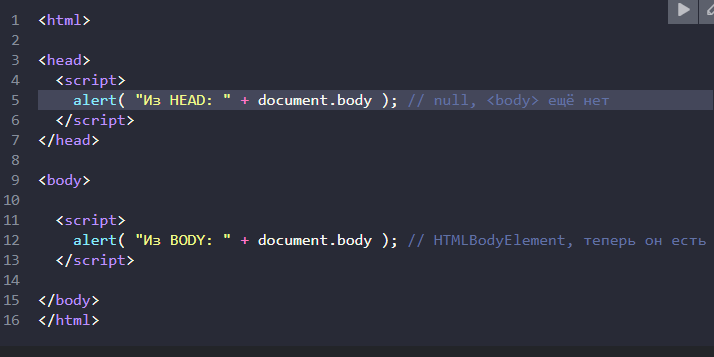
**Лекция номер 11**

**Доступ к Dom**

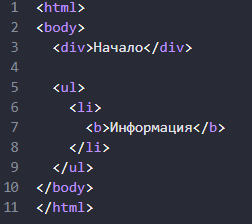
DOM позволяет нам делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект. Все операции с DOM начинаются с объекта document. Это главная «точка входа» в DOM. Из него мы можем получить доступ к любому узлу. Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



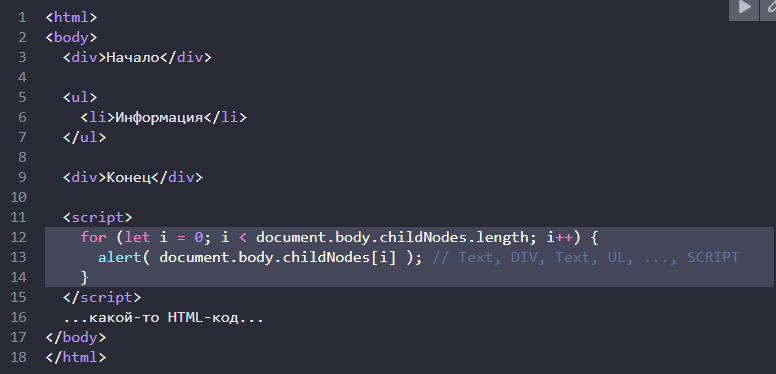
Сверху: documentElement и body Самые верхние элементы дерева доступны как свойства объекта document: <html> = document.documentElement Самый верхний узел документа: document.documentElement. В DOM он соответствует тегу <html>. <body> = document.body Другой часто используемый DOM-узел – узел тега <body>: document.body. <head> = document.head Тег <head> доступен как document.head. Есть одна тонкость: document.body может быть равен null Нельзя получить доступ к элементу, которого ещё не существует в момент выполнения скрипта. В частности, если скрипт находится в <head>, document.body в нём недоступен, потому что браузер его ещё не прочитал. Поэтому, в примере ниже первый alert выведет null:

В мире DOM null означает «не существует» В DOM значение null значит «не существует» или «нет такого узла».

Дети: childNodes, firstChild, lastChild Здесь и далее мы будем использовать два принципиально разных термина: Дочерние узлы (или дети) – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, <head> и <body> являются детьми элемента <html>. Потомки – все элементы, которые лежат внутри данного, включая детей, их детей и т.д. В примере ниже детьми тега <body> являются теги <div> и <ul> (и несколько пустых текстовых узлов):



…А потомки <body>– это и прямые дети <div>, <ul> и вложенные в них: <li> (ребёнок <ul>) и <b> (ребёнок <li>) – в общем, все элементы поддерева. Коллекция childNodes содержит список всех детей, включая текстовые узлы. Пример ниже последовательно выведет детей document.body:



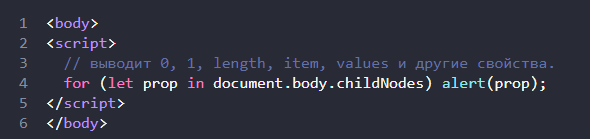
Обратим внимание на маленькую деталь. Если запустить пример выше, то последним будет выведен элемент <script>. На самом деле, в документе есть ещё «какой-то HTML-код», но на момент выполнения скрипта браузер ещё до него не дошёл, поэтому скрипт не видит его. Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему дочернему элементу. Они, по сути, являются всего лишь сокращениями. Если у тега есть дочерние узлы, условие ниже всегда верно:



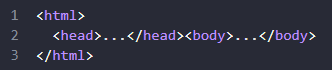
Первый пункт – это хорошо для нас. Второй – бывает неудобен, но можно пережить. Если нам хочется использовать именно методы массива, то мы можем создать настоящий массив из коллекции, используя Array.from:



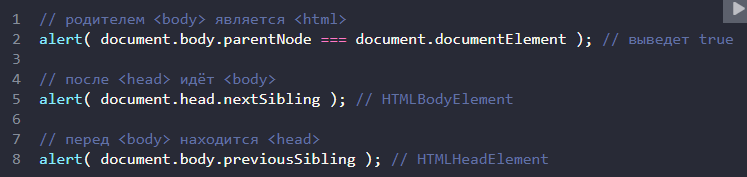
DOM-коллекции – только для чтения DOM-коллекции, и даже более – все навигационные свойства, перечисленные в этой главе, доступны только для чтения. Мы не можем заменить один дочерний узел на другой, просто написав childNodes[i] = .... Для изменения DOM требуются другие методы. Мы увидим их в следующей главе. DOM-коллекции живые Почти все DOM-коллекции, за небольшим исключением, живые. Другими словами, они отражают текущее состояние DOM. Если мы сохраним ссылку на elem.childNodes и добавим/удалим узлы в DOM, то они появятся в сохранённой коллекции автоматически. Не используйте цикл for..in для перебора коллекций Коллекции перебираются циклом for..of. Некоторые начинающие разработчики пытаются использовать для этого цикл for..in. Не делайте так. Цикл for..in перебирает все перечисляемые свойства. А у коллекций есть некоторые «лишние», редко используемые свойства, которые обычно нам не нужны:



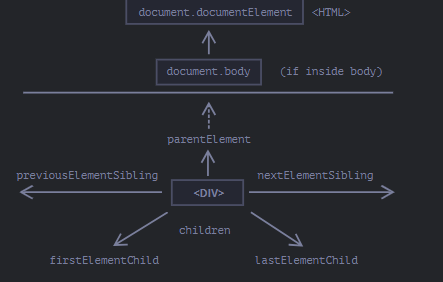
Соседи и родитель Соседи – это узлы, у которых один и тот же родитель. Например, здесь <head> и <body> соседи:



говорят, что <body> – «следующий» или «правый» сосед <head> также можно сказать, что <head> «предыдущий» или «левый» сосед <body>. Следующий узел того же родителя (следующий сосед) – в свойстве nextSibling, а предыдущий – в previousSibling. Родитель доступен через parentNode. Например:



Навигация только по элементам Навигационные свойства, описанные выше, относятся ко всем узлам в документе. В частности, в childNodes находятся и текстовые узлы и узлы-элементы и узлы-комментарии, если они есть. Но для большинства задач текстовые узлы и узлы-комментарии нам не нужны. Мы хотим манипулировать узлами-элементами, которые представляют собой теги и формируют структуру страницы. Поэтому давайте рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы:

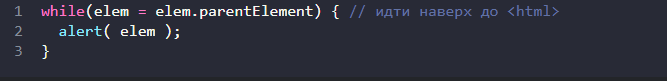


Эти ссылки похожи на те, что раньше, только в ряде мест стоит слово Element: children – коллекция детей, которые являются элементами. firstElementChild, lastElementChild – первый и последний дочерний элемент. previousElementSibling, nextElementSibling – соседи-элементы. parentElement – родитель-элемент.

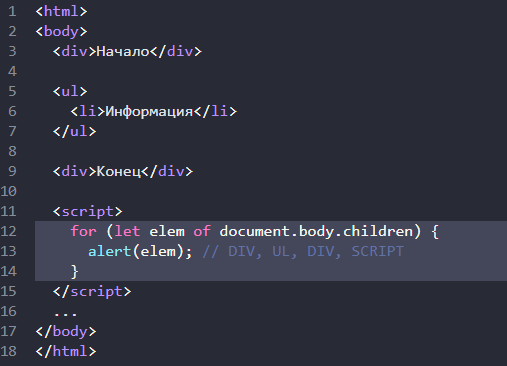
Зачем нужен parentElement? Разве может родитель быть не элементом? Свойство parentElement возвращает родитель-элемент, а parentNode возвращает «любого родителя». Обычно эти свойства одинаковы: они оба получают родителя. За исключением document.documentElement:



Причина в том, что родителем корневого узла document.documentElement (<html>) является document. Но document – это не узел-элемент, так что parentNode вернёт его, а parentElement нет. Эта деталь может быть полезна, если мы хотим пройти вверх по цепочке родителей от произвольного элемента elem к <html>, но не до document:



Изменим один из примеров выше: заменим childNodes на children. Теперь цикл выводит только элементы:



Ещё немного ссылок: таблицы До сих пор мы описывали основные навигационные ссылки. Некоторые типы DOM-элементов предоставляют для удобства дополнительные свойства, специфичные для их типа. Таблицы – отличный пример таких элементов. Элемент <table>, в дополнение к свойствам, о которых речь шла выше, поддерживает следующие: table.rows – коллекция строк <tr> таблицы. table.caption/tHead/tFoot – ссылки на элементы таблицы <caption>, <thead>, <tfoot>. table.tBodies – коллекция элементов таблицы <tbody> (по спецификации их может быть больше одного). <thead>, <tfoot>, <tbody> предоставляют свойство rows: tbody.rows – коллекция строк <tr> секции. <tr>: tr.cells – коллекция <td> и <th> ячеек, находящихся внутри строки <tr>. tr.sectionRowIndex – номер строки <tr> в текущей секции <thead>/<tbody>/<tfoot>. tr.rowIndex – номер строки <tr> в таблице (включая все строки таблицы). <td> and <th>: td.cellIndex – номер ячейки в строке <tr>. Пример использования:



Спецификация: tabular data. Существуют также дополнительные навигационные ссылки для HTML-форм. Мы рассмотрим их позже, когда начнём работать с формами.

Получив DOM-узел, мы можем перейти к его ближайшим соседям используя навигационные ссылки. Есть два основных набора ссылок: Для всех узлов: parentNode, childNodes, firstChild, lastChild, previousSibling, nextSibling. Только для узлов-элементов: parentElement, children, firstElementChild, lastElementChild, previousElementSibling, nextElementSibling. Некоторые виды DOM-элементов, например таблицы, предоставляют дополнительные ссылки и коллекции для доступа к своему содержимому.

**Лекция номер 12**

**система контроля версий git**

Git — это система для управления версиями исходного кода программ. В статье мы познакомимся с её основными возможностями, покажем отличие от GitHub и объясним, зачем Git новичку. Ещё вы узнаете, с чего начать обучение и почему не стоит тратить время на альтернативные программы.

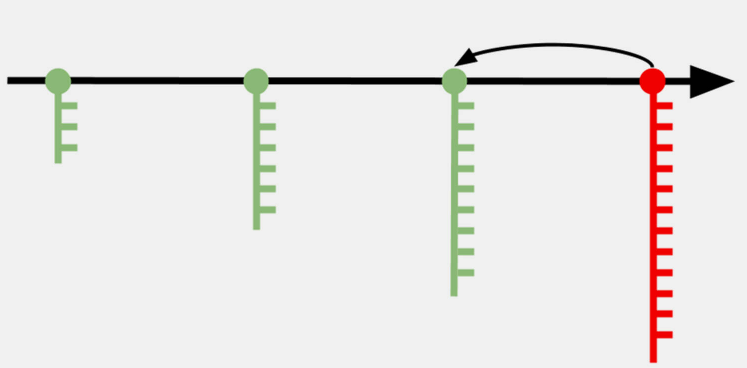
Представьте себя в команде разработчиков. Вы скачиваете на компьютер копию проекта и пишете код. Так же поступают и остальные разработчики. Через некоторое время всей команде нужно объединять код и обмениваться друг с другом изменениями. Вот за это отвечает программа Git и сервис GitHub.

В этом разделе вы познакомитесь с основными возможностями, которые предоставляет Git. Это теория. В последнем разделе мы добавим инструкцию, по которой вы сможете установить Git к себе на компьютер.

Представьте ситуацию: геймер доходит до финала, проигрывает и возвращается к началу уровня — попадает в ближайшую контрольную точку игры, где разработчики разрешили сохраниться. Если мы уберём контрольные точки, после каждого проигрыша придётся начинать игру заново.

В программировании за сохранение кода в контрольных точках отвечает система контроля версий — специальная технология, которую можно подключить к любому проекту. Система контроля версий страхует от ошибок и возвращает код в то состояние, когда всё работало.

Контрольные точки называются коммитами. Один коммит — это пакет изменений, хранящий информацию с добавленными, отредактированными или удалёнными файлами кода. В один коммит принято добавлять не более десяти изменений — так получается длинная история версий, которая позволяет в случае ошибки откатиться с минимальной потерей работоспособного кода.

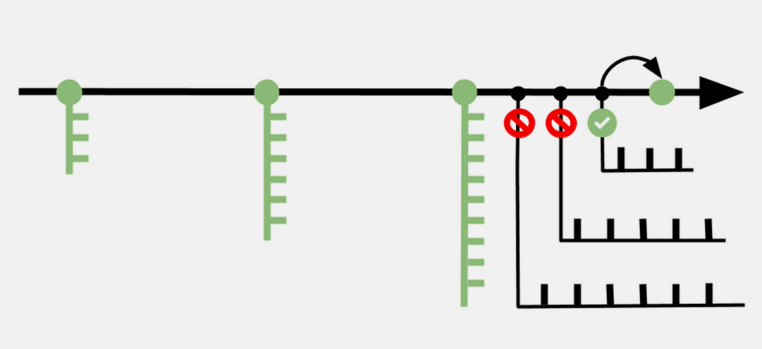


Система контроля версий — это хранилище с файлами всех изменений проекта. Оно позволяет вернуть проект в любое предыдущее состояние, когда программа стабильно работала

Коммиты располагаются на master-ветке — основной версии проекта, которая после завершения работы превратится в продукт.

Система контроля версий позволяет создавать ответвления от master-ветки и экспериментировать с проектом, не мешая другим участника команды.

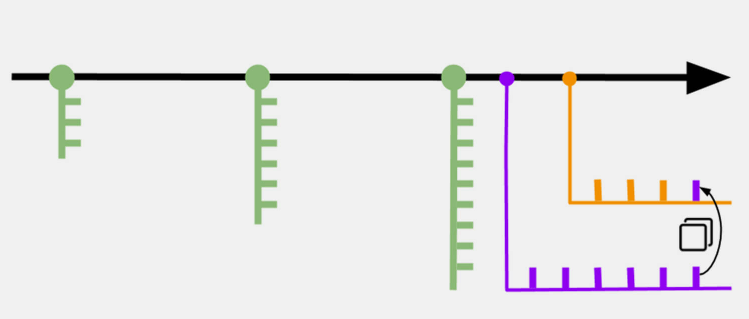
Возьмём предыдущую схему, где мы обнаружили ошибку и откатились на один коммит назад. Чтобы поправить код, создадим несколько дополнительных веток и в каждой протестируем разные варианты решения проблемы. Когда решение найдено, ветку с правильным кодом переносим в master-ветку и сохраняем коммит. Лишние ветки оставляем или удаляем, поскольку они не влияют на проект и скрыты от других разработчиков — это ваш личный черновик.

Система контроля версий помогает разработчикам параллельно работать над проектом, не мешать друг другу и добавлять в master-ветку только качественный код

Часто бывает так: разработчики отделяются от master-ветки и работают над частью проекта самостоятельно — например, чтобы протестировать дополнительные функции. Но не могут продолжить, пока кто-то из команды не допишет код.

Система контроля версий позволяет не ждать обновления master-ветки и разрешает всем участникам команды свободно перемещаться между ветками других разработчиков для копирования нужных фрагментов кода.

Бывают и обратные ситуации, когда несколько разработчиков одновременно дописывают код, заливают его в master-ветку и сталкиваются с конфликтом — один файл получает несколько несогласованных изменений. В этом случае Git попробует автоматически исправить ошибки. Если не получится, разработчики это увидят и смогут поправить код вручную.

Система контроля версий открывает доступ к веткам других разработчиков

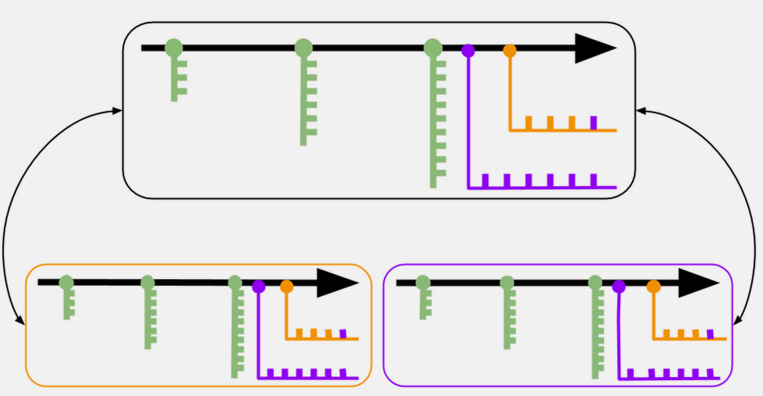
Системы контроля версий бывают локальными, централизованными или распределёнными.

Локальная система хранит файлы на одном устройстве, централизованная использует общий сервер, а распределённая — общее облачное хранилище и локальные устройства участников команды. В локальной системе удобно работать с большими проектами, но сложно взаимодействовать с удалённой командой.

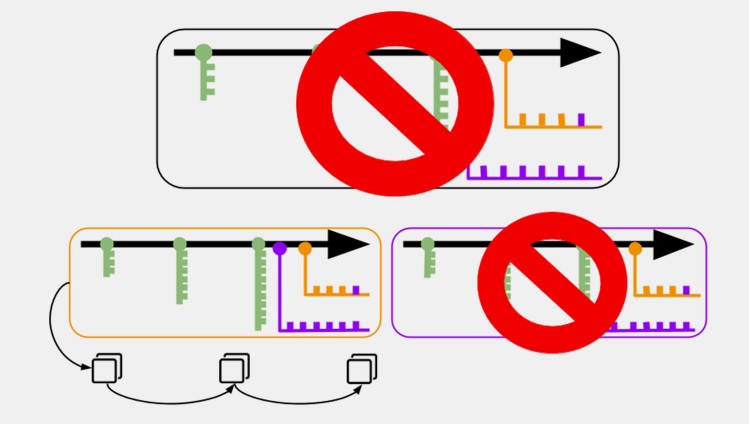
В централизованной системе налажена удалённая работа, но всё привязано к одному серверу. Любой сбой или взлом может повредить файлы проекта.

В распределённой системе налажена удалённая работа. Если с файлами основного репозитория что-то случится — проект легко восстановить из копии любого участника команды.

Из-за удобства и гибкости распределённая система версий Git считается современным форматом. Это стандарт для большинства IT-команд.



Распределённая система контроля версий — это когда у каждого разработчика есть клон проекта



Распределённая система контроля версий — это когда из одной копии можно восстановить проект

Git — это программа, которую нужно установить и подключить к проекту для управления системой контроля версий. GitHub — это сайт-хранилище для историй версий проектов: вы подключаете Git, регистрируетесь на GitHub, создаёте онлайн-репозиторий и переносите файлы из Git на GitHub.

Git — это самая популярная система контроля версий, а GitHub — онлайн-хранилище кода. Git и GitHub настроены на взаимодействие и поэтому часто используются как единый механизм работы с проектом.

Если нужно, Git можно заменить альтернативной программой контроля версий, а GitHub — другим онлайн-хранилищем кода. Большинству работодателей это не нужно, поскольку знакомство с другими сервисами отнимает время и неудобно многим разработчикам.

Git используется в большинстве компаний, где над проектом работают хотя бы два разработчика:

Новый человек приходит в компанию и клонирует репозиторий проекта на ПК.

Получает задачу, создаёт новую ветку и пишет код.

Когда всё готово — отправляет запрос на добавление кода в master-ветку.

Другие разработчики смотрят код, оставляют комментарии и указывают на ошибки.

Новичок дорабатывает код, обновляет master-ветку и переходит к следующей задаче.

Это общая схема того, как проходит командная работа в проекте. В ней не учтены правила использования Git, которые каждая команда пишет под себя. Например, у каждой команды свой порядок проверки кода и свои критерии его готовности для добавления в master-ветку.

Знание Git и знание правил использования Git в команде — это два разных навыка, которые можно сравнить с умением ездить на автомобиле и знанием правил дорожного движения. Если умеете управлять автомобилем — вам проще сконцентрироваться и быстро выучить правила. С Git аналогичная ситуация: если вы умеете управлять системой контроля версий, то можете сразу влиться в проект, не отвлекаться на второстепенные вещи и сосредоточиться на качестве кода.

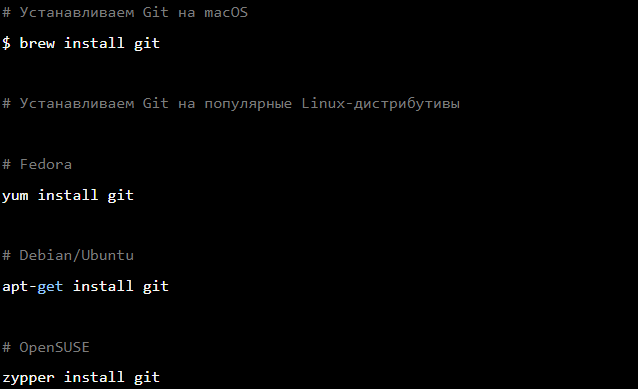
## **С чего начать: 3 шага, чтобы освоить Git**

Это практический раздел. Мы установим Git на компьютер, настроим его для работы, создадим локальный репозиторий и загрузим в него первый файл.

### **Установка**

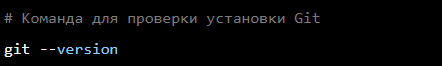
Перейдите на сайт [git-scm.com](https://git-scm.com/downloads) и выберите способ установки Git под свою операционную систему. Для Windows можно скачать загрузочный файл и поставить Git как обычную программу. Базовые настройки можно не менять.

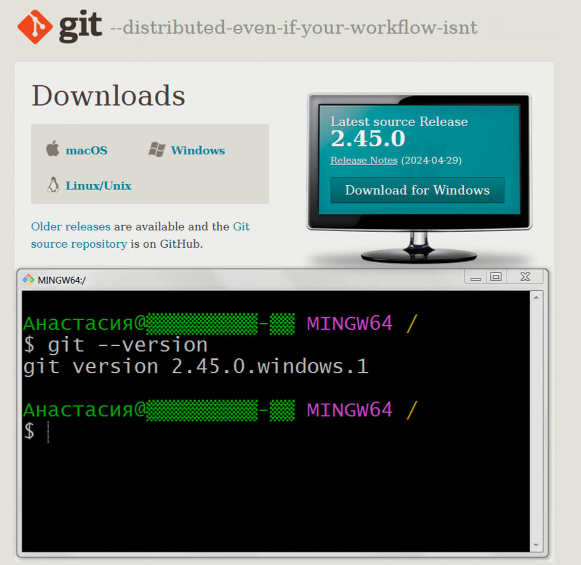
Git на macOS и Linux удобно устанавливать через программу «Терминал». Если у вас Linux — узнайте название своего дистрибутива. Если macOS — поставьте менеджер пакетов [Homebrew](https://brew.sh/ru/). После введите команду загрузки.



Если у вас macOS и Linux — оставайтесь в «Терминале». Если Windows — откройте программу Git Bash, которая установится вместе с системой Git.

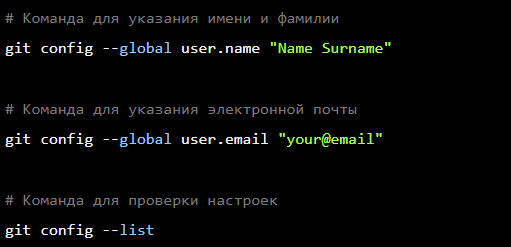
Введите в консоли команду и проверьте установку Git. Вы должны увидеть номер версии, которая загружена в вашу систему. Иногда вместо номера версии выводится сообщение об ошибке: Unsupported command: git. Это означает, что Git установить не получилось и процесс нужно повторить.

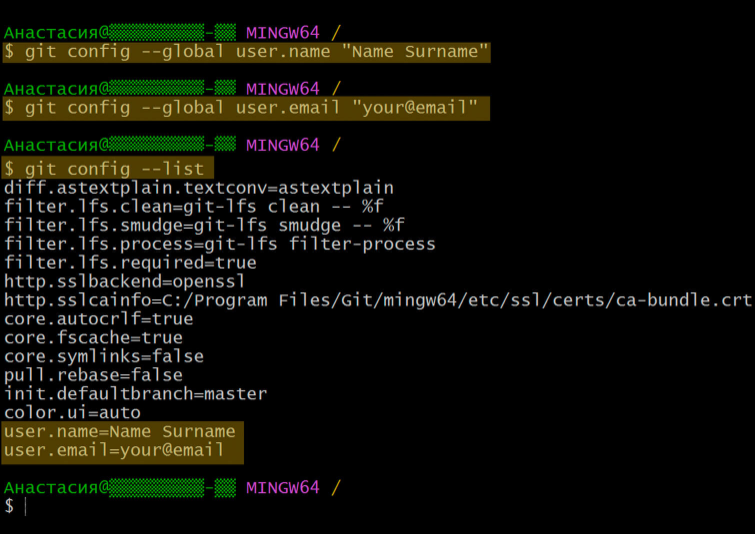




### **Настройка**

После установки Git нужно провести его начальную настройку. Для этого вам нужно указать свои имя, фамилию и почту. Эти данные видны в коммитах, и по ним другие разработчики будут знать, когда вы вносили изменения в проект.

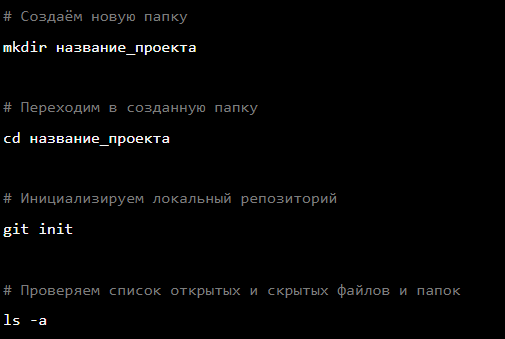


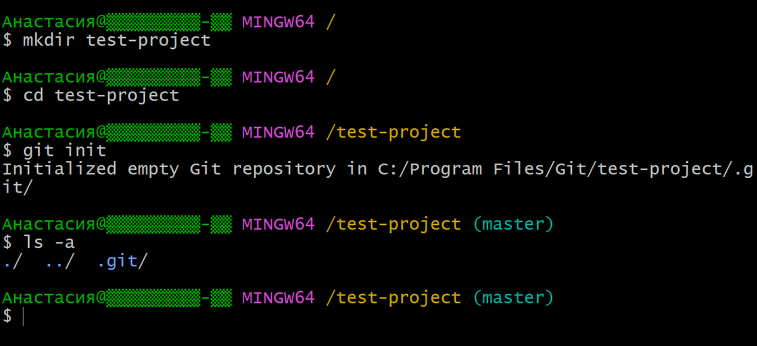


### **Создание репозитория**

Когда Git настроен, вы можете создать проект. Для этого выполним несколько команд: создадим новую папку, зайдём в неё и инициализируем репозиторий.

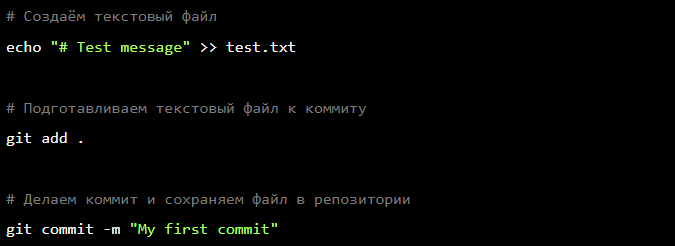
Если репозиторий успешно инициализирован, то у вас в проекте появится скрытая папка .git. В этой папке система Git будет хранить метаинформацию, которая необходима для работы. Это ваш локальный репозиторий.

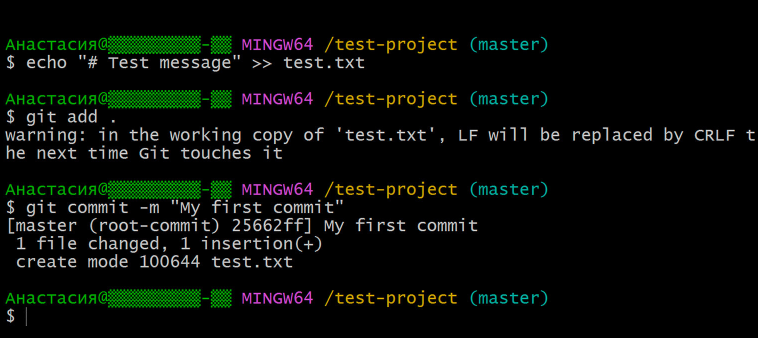




Создаём репозиторий в системе Windows через Git Bash

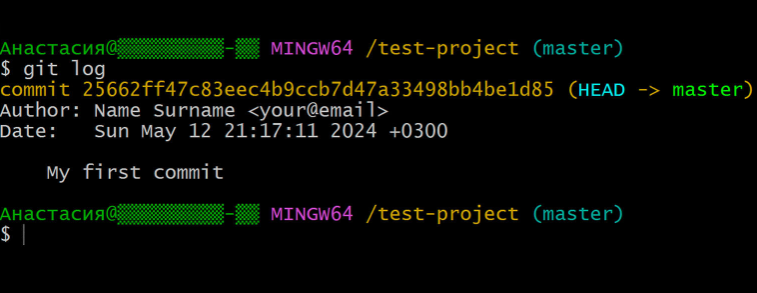
Проверим работу репозитория. Для этого создадим новый файл и выполним коммит — сохраним его в нашем локальном репозитории. Нужные команды:





Сохраняем файл в локальном репозитории через Git Bash

Введём команду git log и посмотрим в журнал. Перед вами должна появиться информация о сделанном коммите. Вы увидите дату внесения изменений, имя и фамилию автора, почту, текст сообщения и [хеш](https://skillbox.ru/media/code/kheshfunktsiya-chto-eto-dlya-chego-nuzhna-i-kak-rabotaet/?utm_source=media&utm_medium=link&utm_campaign=all_all_media_links_links_articles_all_all_skillbox) коммита. Теперь если вы продолжите разработку, то в любой момент сможете вернуться и посмотреть, в каком состоянии находился проект на этапе первого коммита.



Список коммитов в локальном репозитории в программе Git Bash