Web Technologies

▼ Что такое прогрессивный рендеринг?

Прогрессивный рендеринг - это обобщенное название технологий, которые используются для ускорения отрисовки веб страниц.

Основная идея - это как можно раньше показать пользователю визуальный контент.

Цель - уменьшить время загрузки веб-страницы.

Примеры технологий:

- Ленивая загрузка картинок. Картинки на странице не загружаются все разом. JavaScript подгрузит картинки тогда, когда пользователь доскроллит до той части страницы, на которой они расположены.
- Приоритизация видимого контента. Только минимум CSS, контента, скриптов, необходимых для отрисовки той части страницы, которую пользователь увидит первой, т.е. отвечающую за ключевой layout и первую порцию конента для отображения. Вы можете использовать отложенные скрипты или слушать события `DOMContentLoaded` или `load`, чтобы загрузить остальные ресурсы и контент.
- Асинхронные фрагменты HTML. Отправка в браузер частей HTML-страницы, созданной на бэкенде.

▼ Что такое прогрессивный SSR?

Подробнее тут

Прогрессивный рендеринг на стороне сервера - основан на концепции потоковой передачи HTML, он разбивает страницы на осмысленные самостоятельные компоненты с помощью разделения кода. Эти части страницы управляются разными скриптами, в результате чего появляется возможность сделать гидрацию независимо.

Основные этапы:

- Браузер запрашивает у сервера HTML код.
- Сервер делает API запросы и сначала рендерит критический контент, а затем отправляет его клиенту.
- Браузер анализирует HTML и отображает его на экране.
- Сервер рендерит некритический контент и передает его браузеру.
- Браузер анализирует и отображает некритичный контент.
- Параллельно JS бандлы загружаются и выполняются в фоновом режиме, а браузер передает интерактивность элементов в DOM.

Прогрессивный рендеринг повышает производительность веб-приложения извлекая и визуализируя компоненты страницы параллельно.

▼ Что такое Progressive Web Application

Progressive Web Application (PWA) - это веб сайты, которые ведут себя подобно нативным веб приложениям, они могут быть установлены на телефон или компьютер, а также как правило работают в оффлайн.

Для последнего используются Service Worker'ы и интерфейс кэширования

Преимущества: размер и относительная легкость разработки. Не нужно разрабатывать отдельно мобильную версию, можно веб версию обернуть в PWA.

Нужно:

- Service worker - технология позволяюшая запускать JS в браузере в фоновом режиме
- HTTPS - т.к. Service Worker требует этого по соображениям безопасности.

- Manifest file - обычный JSON-файл, который содержит мета-информацию о веб-приложении. Здесь есть данные о значках приложения (один из них пользователь видит на главном экране после установки приложения), о фоновом цвете приложения, о его полном и сокращённом названии, и так далее.

Стоит отметить, что спецификация упоминает три типа веб-воркеров:

- Выделенные воркеры (Dedicated Workers)
- Разделяемые воркеры (Shared Workers)
- Сервис-воркеры (Service Workers)

▼ Что такое кроссбраузерность?

Кроссбраузерность - это корректная адаптивная верстка для правильного отображения сайта или приложения и сохранения функциональности в разных браузерах и на разных устройствах.

Подходы для создания кроссбраузерного сайта:

- семантическая верстка
- использование reset или normalize css
- добавление вендорных префиксов, использование медиазапросов
- применение Progressive Enhancement и Graceful Degradation
- для сохранения функциональности применяют полифилы и транспайлеры по типу Babel.

Вендорные префиксы

Часто разработчики топовых браузеров внедряют новые свойства css, которые еще не стандартизированы. Эти свойства предваряются специальными приставками, которые называются «вендорные префиксы». Каждый браузер имеет свой префикс:

- `o-` — префикс для браузера Опера
- `moz-` — префикс для браузера Mozilla
- `ms-` — префикс для Internet Explorer
- `webkit-` — префикс для браузеров, построенных на движке Webkit, таких, как Safari и Chrome

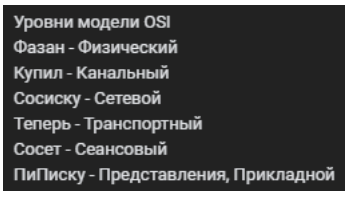
▼ Что такое OSI модель?

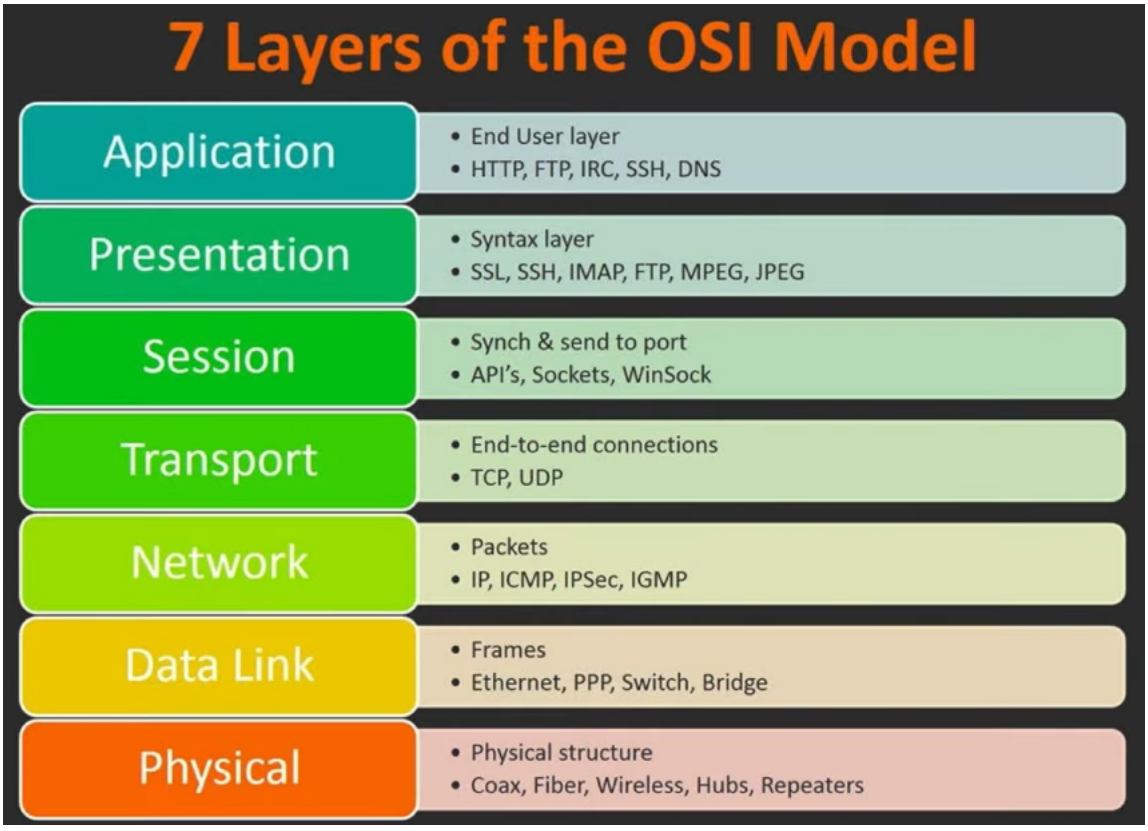
Модель OSI (Open System Interconnection), или эталонная модель взаимодействия открытых систем описывает, как устройства в локальных и глобальных сетях обмениваются данными и что происходит с этими данными.

При этом сама по себе эталонная модель — не стандарт интернета, как, например, TCP/IP; её можно сравнить с фреймворками в мире языков программирования: в OSI «из коробки» доступны разные веб-стандарты — UDP, HTTP, FTP, Telnet и другие. Всего таких протоколов — более 100 штук.

Модель OSI включает семь слоёв, или уровней, — причём каждый из них выполняет определённую функцию: например, передать данные или представить их в понятном для человека виде на компьютере. Кстати, у каждого слоя — свой набор протоколов.

Слои ничего не знают о том, как устроены другие слои. Это называется абстракцией.





Самый нижний слой отвечает за физическое представление данных, то есть за то, как данные передаются по проводам или с помощью радиоволн, а самый верхний — за то, как приложения взаимодействуют с сетью.

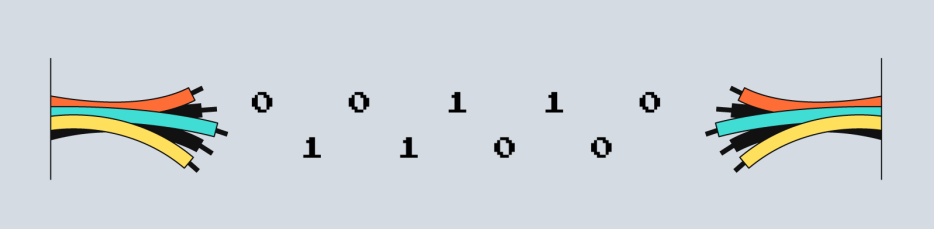
Нижний слой оперирует такими понятиями, как «тип кабеля» или «тип коннектора», а верхний — такими, как HTTP или API.

Рассмотрим каждый слой подробнее.

1-й уровень OSI — физический(L1, physical layer)

На самом нижнем уровне модели OSI данные представляют собой физические объекты — ток, свет или радиоволны. Они передаются по проводам или с помощью беспроводных сигналов.

Этот слой работает с кабелями, контактами в разъёмах, модуляцией сигнала, кодированием единиц и нулей и другими низкоуровневыми штуками. По сути, первый уровень — это уровень проводов и физических способов передачи сигнала. Минимальная абстракция.



Данные в виде сигналов передаются между устройствами*Изображение: Skillbox Media*

Самый известный протокол на физическом уровне — Ethernet. Он описывает, как сигналы кодируются и передаются по проводам. Кроме него есть Bluetooth, Wi-Fi и ИК-порт, которые также содержат инструкции для передачи данных.

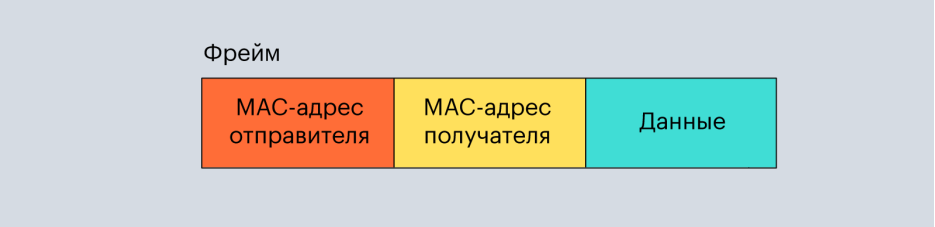
Устройства физического уровня — концентраторы и репитеры. Они работают с физическим сигналом «втупую» и не вникают в его логику: получили данные — передали их дальше по проводу.

Концентратор Netgear*Фото: Wikimedia Commons*

2-й уровень OSI — канальный(L2, data link layer)

Над физическим уровнем располагается канальный. Его задача — проверить целостность полученных данных и исправить ошибки. Этот уровень «поумнее» предыдущего: он уже понимает, что разные амплитуды напряжений отвечают разным битам — нулям и единицам. А ещё канальный уровень умеет кодировать сигналы в биты и передавать их дальше.

Полученные с нижнего уровня данные делятся на фреймы, или кадры. Каждый фрейм состоит из служебной информации — например, адреса отправителя и адреса получателя, — а также самих данных.



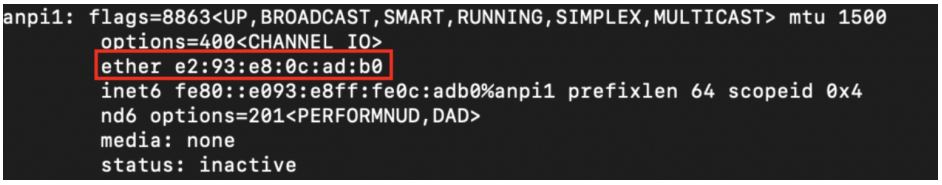
Структура фрейма*Изображение: Skillbox Media*

Получается что-то вроде почтового конверта. На лицевой стороне у него написано, от кого пришло письмо, а внутри находится само письмо (в нашем случае данные).

Лицевая сторона конверта — это MAC-адрес устройства, которое отправило нам информацию. Он нужен, чтобы идентифицировать устройства в локальной сети, состоит из 48 или 64 бит и выглядит примерно так:

Запись MAC-адреса в шестнадцатеричной системе счисления*Изображение: Skillbox Media*

Ещё один важный факт о MAC-адресах: когда на заводе собирают ноутбук или смартфон, ему сразу же присваивают определённый MAC-адрес, который потом уже никак нельзя поменять. MAC-адрес настольных ПК зашит в сетевую карту, поэтому его можно изменить, только заменив эту самую карту.



Изображение: Skillbox Media

С помощью команды ifconfig можно узнать MAC-адрес вашего Macbook или компьютера на Linux. В Windows нужно ввести команду ipconfig.

Канальный уровень не так прост — он делится ещё на два подуровня:

- уровень управления логическим каналом — LLC (logical link control);
- уровень управления доступом к среде — тот самый MAC (media access control).

Первый подуровень нужен для взаимодействия с верхним уровнем, сетевым, а второй — для взаимодействия с нижним, физическим.

Устройства канального уровня — коммутаторы и мосты. Они нужны, чтобы передавать фреймы нужному адресату. Протоколы канального уровня — PPP, CDP.

3-й уровень OSI — сетевой(L3, network layer)

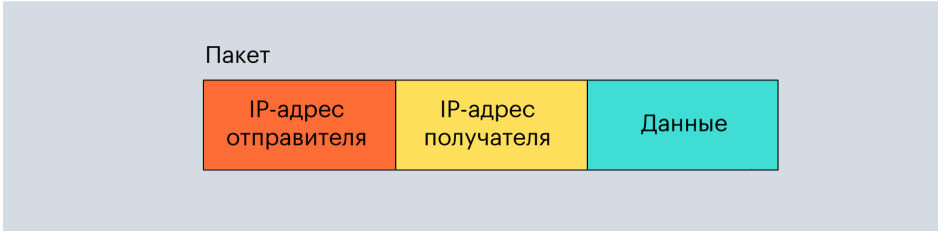
Этот уровень отвечает за маршрутизацию данных внутри сети между компьютерами. Здесь уже появляются такие термины, как «маршрутизаторы» и «IP-адреса».

Маршрутизатор, который используют интернет-провайдеры. Обычно маршрутизатор — это Wi-Fi-роутер*Фото: Wikimedia Commons*

Маршрутизаторы позволяют разным сетям общаться друг с другом: они используют MAC-адреса, чтобы построить путь от одного устройства к другому.

Данные на сетевом уровне представляются в виде пакетов. Такие пакеты похожи на фреймы из канального уровня, но используют другие адреса получателя и отправителя — IP-адреса.

Чтобы получить IP-адрес обоих устройств (отправителя и получателя), существует протокол ARP (address resolution protocol). Он умеет конвертировать MAC- в IP-адрес и наоборот.



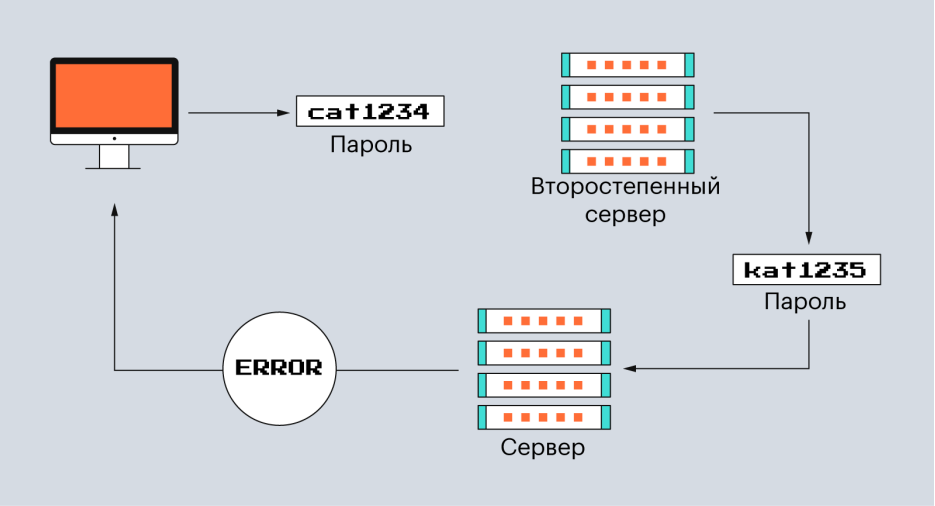
Примерно так выглядят пакеты

4-й уровень OSI — транспортный(L4, transport layer)

Из названия понятно, что на этом уровне происходит передача данных по сети. Так и есть. Два главных протокола здесь — TCP и UDP. Они как раз и отвечают за то, как именно будут передаваться данные.

TCP (Transmission Control Protocol) — это протокол, который гарантирует доставку данных в корректном виде. Он жёстко следит за каждым битом информации, но работает гораздо медленнее UDP.

Например, когда вы вводите логин и пароль при входе в социальную сеть, очень важно, чтобы все символы отправились в определённой последовательности. Если какие-то потеряются или изменятся, вы просто не сможете авторизоваться. Поэтому протокол TCP использует разные методы проверок — например, контрольные суммы.

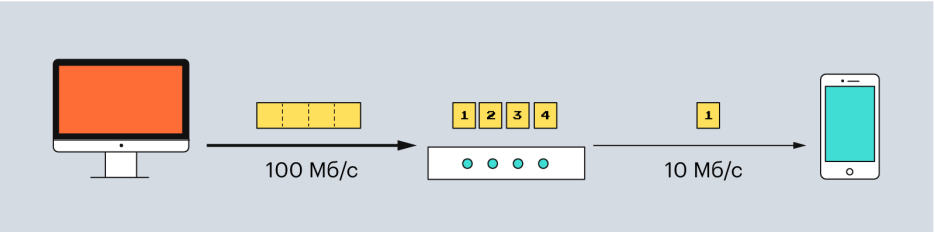


Для этого и нужен TCP — чтобы данные доходили в правильном виде*Изображение: Skillbox Media*

А вот в видео или аудио небольшие потери некритичны, зато важна скорость передачи данных. Для таких задач как раз и придумали протокол UDP (user datagram protocol). Он уже не проверяет цельность битов, его задача — как можно быстрее передать данные с одного устройства на другое.

В протоколе TCP данные делятся на сегменты. Каждый сегмент — часть пакета. Сегменты нужны, чтобы передавать информацию по сети, учитывая её пропускную способность.

Например, если вы передаёте данные с компьютера, у которого пропускная способность 100 Мб/с, на смартфон с пропускной способность 10 Мб/с, то данные разделятся так, чтобы не застревать в самом медленном устройстве.



Вот так данные разделяются на несколько сегментов, чтобы протиснуться в сеть с пропускной способностью 10 Мб/с*Изображение: Skillbox Media*

Ещё сегментация важна для надёжности. Один большой пакет может потеряться или направиться не тому адресату. А маленькие пакеты снижают риск подобных ошибок и даже позволяют проверять их количество. Если какой-то сегмент не получилось доставить, протокол TCP может запросить его у отправителя снова. Так обеспечивается надёжность.

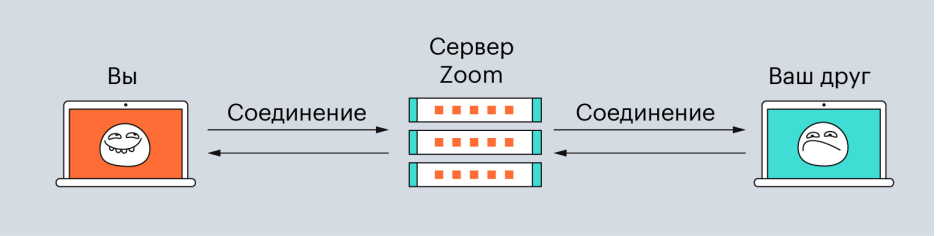
В UDP данные делятся на датаграммы — это примерно то же, что и пакет, только датаграммы автономны. Каждая датаграмма имеет всё необходимое, чтобы дойти до получателя. Поэтому они не зависят от сети и могут доставляться по разным маршрутам и в произвольном порядке.

5-й уровень OSI — сеансовый(L5, session layer)

Начиная с этого уровня и выше, данные имеют уже нормальный вид — например, привычных нам JPEG- или MP3-файлов. Задача сети на этих уровнях — представить информацию в понятном для человека виде и сделать так, чтобы пользователь мог её как-то «потрогать».

Сеансовый уровень управляет соединениями, или сессиями. Типичный пример — звонок по Skype или Zoom. Когда вы звоните другому человеку, между вашими компьютерами устанавливается соединение, по которому передаются аудио и видео. Если такое соединение разорвать, то и ваш звонок прервётся.

На сеансовом уровне очень важно, чтобы соединение правильно установилось и поддерживалось. То есть механизмы протоколов должны проверить, что у обоих собеседников есть нужные кодеки и сигнал между устройствами присутствует.

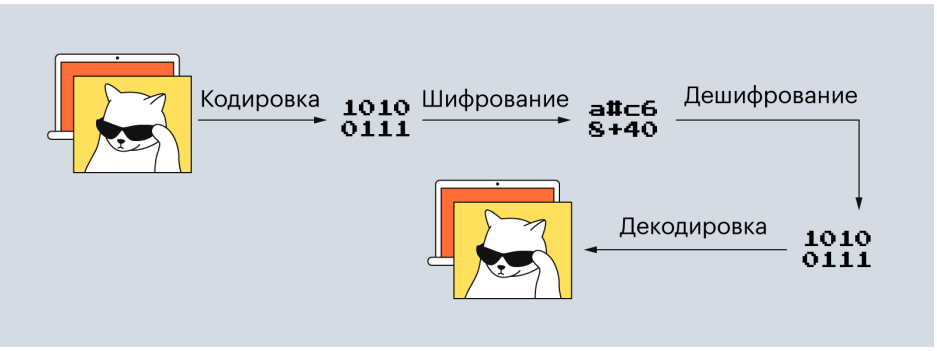


Сеанс звонка в Zoom*Изображение: Skillbox Media*

6-й уровень OSI — уровень представления данных(L6, presentation layer)

На этом уровне происходит преобразование форматов данных — их кодирование и сжатие. Например, полученные данные могут превратиться в GIF- или MP4-файл. То же самое происходит и в обратном порядке: когда пользователь отправляет файл другому человеку, данные сначала конвертируются в биты и сжимаются, а потом уже передаются на транспортный уровень.

Помимо кодировки и сжатия на уровне представления, данные могут шифроваться — если, конечно, это необходимо.



Обычный процесс отправки данных с одного устройства на другое*Изображение: Skillbox Media*

7-й уровень OSI — прикладной(L7, application layer)

Последний уровень модели OSI — прикладной. На нём находятся сетевые службы, которые помогают без проблем сёрфить в интернете.

Прикладной уровень похож на некий графический интерфейс для всей модели OSI — с его помощью пользователь взаимодействует с другими уровнями, даже не подозревая об этом. Этот интерфейс называется сетевым.

Самые популярные из сетевых интерфейсов — это HTTP, HTTPS, FTP и SMTP. А «устройства» здесь — это уже программы: Zoom, Telegram, браузеры.

Например, по HTTP браузеры запрашивают веб-страницы и получают в ответ HTML-страницы*Изображение: Skillbox Media*

Как на практике работает сетевая модель OSI

В начале статьи мы задались вопросом: а как передаются сообщения в Telegram? Настало время на него ответить — и показать весь процесс передачи данных по модели OSI.

Мы хотим отправить сообщение нашему другу. Печатаем текст и нажимает кнопку «Отправить», а дальше перемещаемся внутрь компьютера.

Прикладной уровень. Приложение Telegram работает на прикладном уровне модели OSI. Когда мы печатаем текст сообщения и нажимаем кнопку «Отправить», эти данные передаются на сервер мессенджера, а оттуда — нашему другу.

Весь процесс проходит через API разных библиотек — например, для HTTP-запросов. Интерфейсы позволяют без лишних проблем обмениваться данными и не погружаться в то, как они представлены на низком уровне. Всё, что нужно знать, — это какую функцию вызвать и какие переменные туда передать.

Уровень представления. Здесь данные должны преобразоваться в унифицированный формат, чтобы их можно было передавать на разные устройства и операционные системы. Например, если мы отправляем сообщение с Windows на macOS, данные должны быть в читаемом для компьютеров Apple виде. Такая же ситуация и с другими устройствами.

Раз мы собираемся передать данные на другой компьютер, их нужно перевести в бинарный формат. После этого начнётся сам процесс передачи по сети.

Сеансовый уровень. Чтобы данные успешно передались сначала на сервер Telegram, а затем к нашему другу, приложению нужно установить соединение, или сеанс. Он обеспечивает синхронизацию между устройствами и восстанавливает связь, если она прервалась.

Благодаря сеансам вы можете видеть, что собеседник что-то печатает или отправляет вам картинки или видео. Но главная задача этого соединения — обеспечить стабильное соединение для передачи данных.

Транспортный уровень. Когда соединение установлено и данные унифицированы, пора передавать их. Этим занимается транспортный уровень.

Здесь данные разбиваются на сегменты и к ним добавляется дополнительная информация — например, номер порта и контрольные суммы. Всё это нужно, чтобы данные дошли до пользователя в целостности.

Сетевой уровень. Теперь данным нужно найти маршрут к устройству нашего друга, а затем отправить их по нему. Поэтому данные упаковываются в пакеты и к ним добавляются IP-адреса.

Чтобы получить IP-адрес устройств, которым нужно отправить пакеты, маршрутизаторы (устройства сетевого уровня) обращаются к ARP. Этот протокол быстро найдёт адрес получателя и отдаст его нам.

Канальный уровень. Здесь данные передаются от одного MAC-адреса к другому. Изначальный текст делится на фреймы — с заголовками и контрольными суммами для проверки целостности данных.

Физический уровень. И на самом нижнем уровне данные в виде электрических сигналов передаются по проводам, кабелям или по радиоволнам. Тут только одна задача — как можно быстрее откликаться на сигналы свыше.

После прохождения всех уровней модели OSI сообщение успешно доставляется на устройство нашего друга. Правда, в реальности это занимает всего миллисекунды.

Что запомнить

Модель OSI описывает, как работает весь интернет: как электрические сигналы преобразуются в картинки с котиками и как устройства обмениваются этими данными.

Модель включает семь уровней:

- физический;
- канальный;
- сетевой;
- транспортный;
- сеансный;
- представления;
- прикладной.

На каждом уровне находятся определённые протоколы, которые помогают данным перемещаться или превращаться в удобный для пользователей формат.

▼ **Что такое поток документа?**

Поток - это принцип организации элементов на странице.

Даже если мы не стилизуем страницу при помощи CSS, то отображение элементов в браузере будет предсказуемо, так как у тегов есть стили по умолчанию, зашитые в движок браузера. Благодаря ним заголовок **h1** больше заголовка **h2**, а ссылка синие и подчеркнутые.

Элементы на странице можно разделить на 2 категории: блочные(div) и строчные(span) кторые формируют стандартный поток документа. Изменить стандартное поведение можно при помощи CSS свойства display - absolute, fixed, sticky, flex, grid.

Также в CSS есть свойства которое вырывают элементы из нормального потока документа, к таким свойствам можно отнести float и position.

▼ Разница между адаптивным (adaptive) и отзывчивым (responsive) дизайнами?

Фиксированный (Fixed) (или статический) макет имеет строгую ширину в пикселях. Все элементы в фиксированных величинах и их ширина остается неизменной независимо от размера экрана.

Адаптивный дизайн (Adaptive) - Все элементы в фиксированных величинах и адаптация через медиа запросы.

Резиновый (Rubber) или жидкостный (Fluid) отличается тем, что размер контента у него указывается в процентах. **Какие минусы? -** Например, на телефонах размер контента может сильно уменьшиться и будет неудобно с ним взаимодействовать или при добавлении изображения/видео их соотношение сторон будет меняться и изображение будет искажаться.

Отзывчивый дизайн (Responsive) - Сочетает в себе адаптивный и резиновый, мы задаем размеры контента в % и пишем медиа запросы.

▼ Разница между Progressive Enhancement и Graceful Degradation?

Эти оба подхода используются для создания кроссплатформенных и кроссбраузерных приложений.

Progressive Enhancement - предполагает поэтапное создание веб интерфейсов от простого к сложному, на каждом из этапов создается законченный веб интерфейс как улучшенная версия предыдущего, обеспечивается в четыре этапа.

- базовое отображение без стилей
- добвляются базовые стили
- стили CSS3 типа прозрачностей и градиентов
- добавляется JS.

Graceful Degradation - функционал сайта деградирует от сложного к более простому. Сначала создается максимально функциональная версия веб страницы в современном браузере, а затем идет деградация до более ранних версий с помощью вендорных префиксов и директивы `supports`

Предпочтительнее использовать Progressive Enhancement т.к. он обеспечивает более предсказуемый результат.

▼ Что такое Веб-компоненты и какие технологии в них используются?

Веб-компоненты - это технология, которая позволяет создавать многократно используемые компоненты в веб документах и приложениях, они поддерживаются браузерами напрямую и не требуют библиотек для работы.

▼ html templates

- Содержимым `<template>` может быть любой синтаксически корректный HTML.
- Содержимое `<template>` считается находящимся «вне документа», поэтому оно ни на что не влияет.
- Мы можем получить доступ к `template.content` из JavaScript, клонировать его и переиспользовать в новом компоненте.

Элемент `<template>` уникальный по следующим причинам:

- Браузер проверяет правильность HTML-синтаксиса в нём (в отличие от строк в скриптах).
- ...При этом позволяет использовать любые HTML-теги, даже те, которые без соответствующей обёртки не используются (например `<tr>`).

- Его содержимое оживает (скрипты выполняются, `<video autoplay>` проигрывается и т. д.), когда помещается в документ.

Элемент `<template>` не поддерживает итерацию, связывания данных или подстановки переменных. Однако эти возможности можно реализовать поверх него.

▼ html import

Устарел и removed

Тег `<link rel="import">` позволяет подключить любой документ к странице, причём:

- Скриптовое пространство и стили со страницей будут общие.
- Документ DOM – отдельный, он доступен как `link.import` снаружи, а из внутреннего скрипта – через `document.currentScript.ownerDocument`. Можно без проблем переносить элементы из главного документа в импорт и наоборот.
- Импорты могут содержать другие импорты.
- Если какой-то URL импортируется повторно – подключается уже готовый документ, без повторного выполнения скриптов в нём. Это позволяет избежать дублирования при использовании одной библиотеки во множестве мест.

▼ custom elements

Есть два типа пользовательских элементов:

1. «Автономные» – новые теги, расширяющие `HTMLElement`.

   Схема определения:

```
class MyElement extends HTMLElement {
  constructor() {
    super();
    // элемент создан
  }

  connectedCallback() {
    // браузер вызывает этот метод при добавлении элемента в документ
    // (может вызываться много раз, если элемент многократно добавляется/удаляется)
  }

  disconnectedCallback() {
    // браузер вызывает этот метод при удалении элемента из документа
    // (может вызываться много раз, если элемент многократно добавляется/удаляется)
  }

  static get observedAttributes() {
    return [/* массив имён атрибутов для отслеживания их изменений */];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    // вызывается при изменении одного из перечисленных выше атрибутов
  }

  adoptedCallback() {
    // вызывается, когда элемент перемещается в новый документ
    // (происходит в document.adoptNode, используется очень редко)
  }

  // у элемента могут быть ещё другие методы и свойства
}

// сообщим браузеру, что <my-element> обслуживается нашим новым классом
```

```
customElements.define("my-element", MyElement);

// Теперь для любых HTML-элементов с тегом <my-element>
// создаётся экземпляр MyElement и вызываются вышеупомянутые методы.
// Также мы можем использовать document.createElement('my-element')
// в JavaScript.
```

`connectedCallback` срабатывает, когда элемент добавляется в документ. Не просто добавляется к другому элементу как дочерний, но фактически становится частью страницы. Таким образом, мы можем построить отдельный DOM, создать элементы и подготовить их для последующего использования. Они будут рендериться только тогда, когда попадут на страницу.

Имя пользовательского элемента должно содержать дефис `-`, например, `my-element` и `super-button` – валидные имена, а `myelement` – нет.

Это чтобы гарантировать отсутствие конфликтов имён между встроенными и пользовательскими элементами HTML.

2. «Модифицированные встроенные элементы» – расширения существующих элементов, что обеспечивает понимание тега поисковыми роботами и читалками, но при этом имеющего расширенную функциональность.

   Требуют ещё один аргумент в `.define` и атрибут `is="..."` в HTML:

```
class MyButton extends HTMLButtonElement { /*...*/ }
customElements.define('my-button', MyElement, {extends: 'button'});
/* <button is="my-button"> */
```

▼ shadow dom

▼ **Shadow DOM**

Теневой DOM – это способ создать свой, изолированный, DOM для компонента.

1. `shadowRoot = elem.attachShadow({mode: open|closed})` – создаёт теневой DOM для `elem`. Если `mode="open"`, он доступен через свойство `elem.shadowRoot`.
2. Мы можем создать подэлементы внутри `shadowRoot` с помощью `innerHTML` или других методов DOM.

Элементы теневого DOM:

- Обладают собственной областью видимости идентификаторов
- Невидимы JavaScript селекторам из главного документа, таким как `querySelector`,
- Стилизуются своими стилями из теневого дерева, не из главного документа.

Теневой DOM, если имеется, отрисовывается браузером вместо обычных потомков (light DOM).

▼ **Слоты теневого DOM, композиция**

Обычно, если у элемента есть теневое дерево, то содержимое обычного, светлого DOM не показывается. Слоты позволяют показать элементы светлого DOM на заданных местах в теневом DOM.

Существует два вида слотов:

- Именованные слоты: `<slot name="X">...</slot>` – получают элементы светлого DOM с `slot="X"`.
- Слот по умолчанию: первый `<slot>` без имени (последующие неименованные слоты игнорируются) – показывает элементы элементов светлого дерева, которые не находятся в других слотах.
- Если одному слоту назначено несколько элементов, они добавляются один за другим.
- Содержимое элемента `<slot>` используется как резервное. Оно отображается, если в слоте нет элементов из светлого дерева.

Процесс отображения элементов внутри слота называется «композицией». В результате композиции строится «развёрнутый DOM».

При композиции не происходит перемещения узлов – с точки зрения JavaScript, DOM остаётся прежним.

JavaScript может получить доступ к слотам с помощью следующих методов:

- `slot.assignedNodes/Elements()` – возвращает узлы/элементы, которые находятся внутри `slot`.
- `node.assignedSlot` – обратный метод, возвращает слот по узлу.

Если мы хотим знать, что показываем, мы можем отследить контент слота следующими способами:

- событие `slotchange` – запускается, когда слот наполняется контентом в первый раз, и при каждой операции добавления/удаления/замещения элемента в слоте, за исключением его потомков. Сам слот будет `event.target`.
- MutationObserver для более глубокого просмотра содержимого элемента в слоте и отслеживания изменений в нём.

Теперь, когда мы научились показывать элементы светлого DOM в теневом DOM, давайте посмотрим, как их правильно стилизовать. Основное правило звучит так: теневые элементы стилизуются внутри, а обычные элементы – снаружи; однако есть заметные исключения.

▼ **Настройка стилей теневого DOM**

Теневой DOM может включать в себя стили, такие как `<style>` или `<link rel="stylesheet">`.

Локальные стили могут влиять на:

- теневое дерево,
- элемент-хозяин, при помощи псевдоклассов `:host` и `:host()`,
- слотовые элементы (из светлого DOM), `::slotted(селектор)` позволяет стилизовать сами слотовые элементы, но не их дочерние элементы.

Стили документов могут влиять на:

- элемент-хозяин (так как он находится во внешнем документе)
- слотовые элементы и их содержимое (так как они также физически присутствуют во внешнем документе)

Когда свойства CSS конфликтуют, обычно стили документа имеют приоритет, если только свойство не помечено как `!important`. Тогда предпочтение отдаётся локальным стилям.

Пользовательские свойства CSS проникают через теневой DOM. Они используются как «хуки» для придания элементам стиля:

1. Компонент использует пользовательское CSS-свойство для стилизации ключевых элементов, например `var(--component-name-title, <значение по умолчанию>)`.
2. Автор компонента публикует эти свойства для разработчиков, они так же важны, как и другие общедоступные методы компонента.
3. Когда разработчик хочет стилизовать заголовок, он назначает CSS-свойство `--component-name-title` для элемента-хозяина или выше.
4. Profit!

▼ **Теневой DOM и события**

Только те события пересекают границы теневого DOM, у которых флаг `composed` установлен в значение `true`.

У большинства встроенных событий стоит `composed: true`, это описано в соответствующих спецификациях:

- UI Events https://www.w3.org/TR/uievents.
- Touch Events https://w3c.github.io/touch-events.
- Pointer Events https://www.w3.org/TR/pointerevents.
- ...И так далее.

У некоторых встроенных событий всё же стоит `composed: false`:

- `mouseenter`, `mouseleave` (вообще не всплывают),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

Эти события могут быть пойманы только на элементах, принадлежащих тому же DOM-дереву. Т.е. оно может быть поймано лишь внутри теневого DOM.

Если мы генерируем своё событие `CustomEvent`, то должны явно поставить флаг `composed: true`.

Обратите внимание, что в случае вложенных компонентов теневые DOM могут быть вложены друг в друга. События с флагом `composed` всплывают через границы всех теневых DOM. Поэтому, если событие предназначено только для ближайшего внешнего компонента-родителя, мы можем инициировать его на элементе-хозяине и установить флаг `composed: false`. Тогда оно будет уже вне теневого DOM компонента, но не выплывает наружу в «ещё более внешний» DOM.

▼ Особенности разработки мультиязычных сайтов?

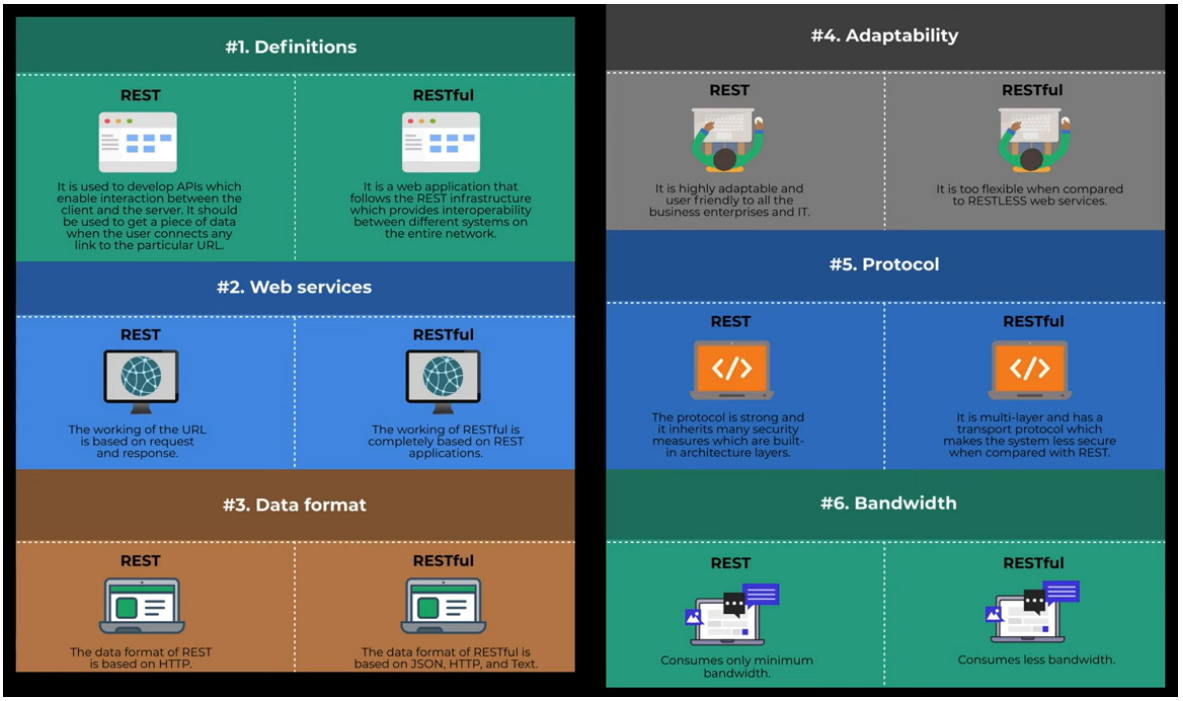
- Использование lang в HTML для правильного рендеринга текста и синтезирования речи.
- Редирект пользователя на нужный язык, проверяя его текущий язык через браузер или дать ему выбрать язык на стартовой странице или во всплывающем окне.
- Направление чтение текста, с помощью css
- Форматирование даты и валюты, с помощью js
- Использование ограничений для текста, например правильные переносы строк или сокращения текста многоточием.
- Порядок слов в предложении, больше относится к грамматике при написнии текста
- text-rendering для отрисовки или запрета лигатрур

▼ Что такое REST? Что такое REST и RESTful api? Принципы REST-архитектуры?

REST - это архитектурный стиль API.

Он не ограничивается никакими протоколами и не имеет собственных методов. Но обычно в RESTful-сервисах используют стандарт HTTP, а файлы передают в формате JSON или XML.

Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.



https://medium.com/@andr.ivas12/rest-простым-языком-90a0bca0bc78

REST (Representational state transfer) — общие принципы организации взаимодействия приложения/сайта с сервером по `HTTP`. Сервер не запоминает состояние пользователя между запросами. В запросах передаётся информация, идентифицирующая пользователя (например *токен*) и параметры для выполнения операции.

Разница между REST и RESTful

REST — архитектурный стиль, а **RESTful** это сервис который следует этому стилю. Т.е если у вас на сервере *REST API*, а на клиенте вы делаете запросы к этому *API*, то у вас **RESTful** приложение/сайт.

Как это работает

Взаимодействие с сервером сводится к 4 операциям (4 — необходимый и достаточный минимум):

1. Получение данных (`JSON` или `XML`)
2. Добавление новых данных
3. Модификация данных
4. Удаление данных

Получение данных не меняет состояния сервера. Каждый тип операции пользуется своим HTTP методом:

1. **GET** — Получение
2. **POST** — Добавление
3. **PUT** — Модификация
4. **DELETE** — Удаление

Пример REST API

В качества примера возьмем стандартное API для списка пользователей. Мы запрашиваем список пользователей, получаем информацию о пользователе, добавляем нового пользователя, изменяем информацию и удаляем пользователя. Если у нас на сервере REST API, то нам понадобятся следующие запросы:

1. **GET** `/api/users` — получить список пользователей
2. **GET** `/api/users/7` — получить информацию о пользователе с `ID=7`
3. **POST** `/api/users` — добавить нового пользователя
4. **PUT** `/api/users/7` — изменить информацию о пользователе с `ID=7`
5. **DELETE** `/api/users/7` — удалить пользователя с `ID=7`

```
GET http://andreyolegovich.ru:8080 /resource1/status
GET http://andreyolegovich.ru:8080 /resource1/getserviceInfo
PUT http://andreyolegovich.ru:8080 /resource1/putID
GET http://andreyolegovich.ru:8080 /resource1/eventslist
POST http://andreyolegovich.ru:8080 /resource1/eventslist
PUT http://andreyolegovich.ru:8080 /resource2/putID
```

У эндпойнтов (Enpoints) различные окончания. Такое окончание в Endpoint называются **Resource**, а начало **Base URL**. Такое определение Endpoint и Resource используется, например, в SOAP UI для RESTful интерфейсов

```
https://andreyolegovich.ru:8080 - это Base URL
/resource1/status - это Resource

Endpoint = Base URL + Resource
```

```
Request = Method + Endpoint
```

После того как все эти API созданы, их необходимо описать. Нужен документ из которого будет понятно

1. Какие методы можно использовать, посылая запросы на каждый Endpoint
2. Должны ли передаваться какие-то данные
3. Если нужно передавать данные в теле запроса, то какие
4. Какие ответы мы ожидаем в случае успешного запроса
5. Какие ответы мы ожидаем когда с запросом или его обработкой на сервере что-то не так

Принципы REST

- **Ресурсы** позволяют легко понять структуру каталогов URI
- **Представления** передают JSON или XML в качестве представления данных объекта и атрибутов
- **Сообщения** используют HTTP методы явно(например, GET, POST, PUT и DELETE)
- **Отсутствие состояния** взаимодействий не сохраняет контекст клиента на сервере между запросами

HTTP методы

Используются HTTP методы для определения CRUD (create, read, update, delete) операций HTTP запросов.

GET

Получение информации. GET запросы должны быть безопасны и идемпотентны, т.е. независимо от того, как много времени они повторяются с теми же самыми параметрами, результат останется тот же. Они могут иметь побочные эффекты, но пользователь не ожидает их, поэтому они не могут критичными для функционирования системы. Запросы могут быть также частичными или условными.

Получение адреса по ID, равным 1:

`GET /addresses/1`

POST

Запрос что-то сделать с ресурсом по URI с предоставлением сущности. Часто POST используется для создания новой сущности, но также возможно использовать и для обновления существующей.

Создание нового адреса:

`POST /addresses`

PUT

Сохраняет сущность по URI. PUT может создавать новую сущность или обновлять существующую. PUT запрос идемпотентен. Идемпотентность - главное отличие в поведении между PUT и POST запросом.

Изменение адреса по ID, равным 1:

`PUT /addresses/1`

PUT заменяет существующую сущность. Те элементы сущности, которые не представлены в запросе, будут очищены или заменены на null.

PATCH

Обновляет только определенные поля сущности по URI. PATCH запрос идемпотентен. Идемпотентность - главное отличие в поведении между PUT и POST запросом.

`PATCH /addresses/1`

DELETE

Запрос, который удаляет ресурс; кроме того, ресурс не должен быть удален немедленно. Он может быть асинхронным или "долгоиграющим" запросом.

Удаления адреса по ID, равным 1:

```
DELETE /addresses/1
```

Идемпотентность

С точки зрения RESTful-сервиса, операция (или вызов сервиса) идемпотентна тогда, когда клиенты могут делать один и тот же вызов неоднократно при одном и том же результате на сервере. Другими словами, создание большого количества идентичных запросов имеет такой же эффект, как и один запрос. Заметьте, что в то время, как идемпотентные операции производят один и тот же результат на сервере, ответ сам по себе может не быть тем же самым (например, состояние ресурса может измениться между запросами).

Методы **PUT** и **DELETE** по определению идемпотентны. Тем не менее, есть один нюанс с методом DELETE. Проблема в том, что успешный DELETE-запрос возвращает статус 200 (OK) или 204 (No Content), но для последующих запросов будет все время возвращать 404 (Not Found), Состояние на сервере после каждого вызова DELETE то же самое, но ответы разные.

Методы **GET**, **HEAD**, **OPTIONS** и **TRACE** определены как безопасные. Это означает, что они предназначены только для получения информации и не должны изменять состояние сервера. Они не должны иметь побочных эффектов, за исключением безобидных эффектов, таких как: логирование, кеширование, показ баннерной рекламы или увеличение веб-счетчика.

По определению, **безопасные операции идемпотентны**, так как они приводят к одному и тому же результату на сервере. Безопасные методы реализованы как операции только для чтения. Однако безопасность не означает, что сервер должен возвращать тот же самый результат каждый раз.

Чтобы система считалась RESTful, она должна “вписываться” в шесть REST ограничений:

1. Приведение архитектуры к модели клиент-сервер

В основе данного ограничения лежит разграничение потребностей. Необходимо отделять потребности клиентского интерфейса от потребностей сервера, хранящего данные. Данное ограничение повышает переносимость клиентского кода на другие платформы, а упрощение серверной части улучшает масштабируемость системы. Само разграничение на “клиент” и “сервер” позволяет им развиваться независимо друг от друга.

2. Отсутствие состояния

Архитектура REST требует соблюдения следующего условия. В период между запросами серверу не нужно хранить информацию о состоянии клиента и наоборот. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Таким образом и сервер, и клиент могут «понимать» любое принятое сообщение, не опираясь при этом на предыдущие сообщения.

3. Кэширование

Клиенты могут выполнять кэширование ответов сервера. У тех, в свою очередь, должно быть явное или неявное обозначение как кэшируемых или некэшируемых, чтобы клиенты в ответ на последующие запросы не получали устаревшие или неверные данные. Правильное использование кэширования помогает полностью или частично устранить некоторые клиент-серверные взаимодействия, ещё больше повышая производительность и расширяемость системы.

4. Единообразие интерфейса

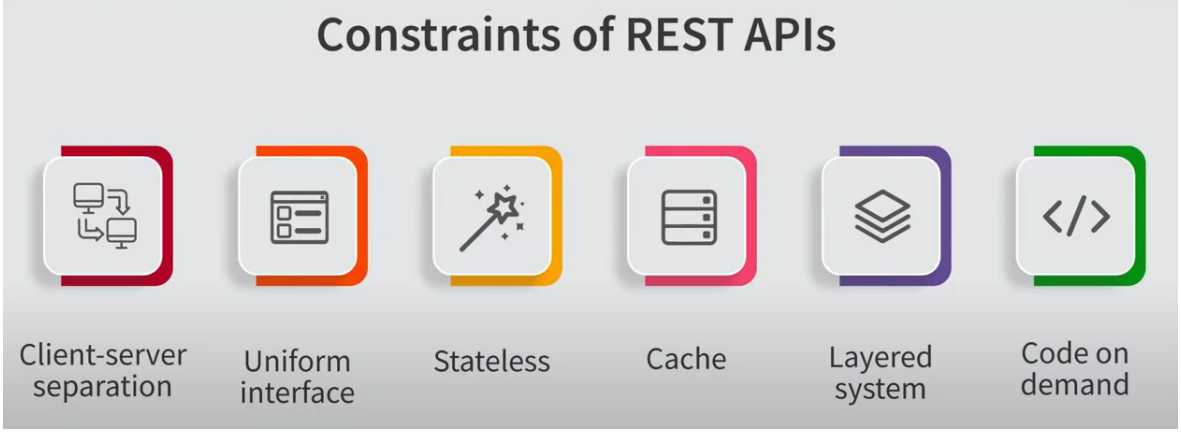
К фундаментальным требованиям REST архитектуры относится и унифицированный, единообразный интерфейс. Клиент должен всегда понимать, в каком формате и на какие адреса ему нужно слать запрос, а сервер, в свою очередь, также должен понимать, в каком формате ему следует отвечать на запросы клиента. Этот единый формат клиент-серверного взаимодействия, который описывает, что, куда, в каком виде и как отсылать и является унифицированным интерфейсом

5. Слои

Под слоями подразумевается иерархическая структура сетей. Иногда клиент может общаться напрямую с сервером, а иногда — просто с промежуточным узлом. Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределённого кэширования. Приведем пример. Представим себе некоторое мобильное приложение, которое пользуется популярностью во всем мире. Его неотъемлемая часть — загрузка картинок. Так как пользователей — миллионы человек, один сервер не смог бы выдержать такой большой нагрузки. Разграничение системы на слои решит эту проблему. Клиент запросит картинку у промежуточного узла, промежуточный узел запросит картинку у сервера, который наименее загружен в данный момент, и вернет картинку клиенту. Если здесь на каждом уровне иерархии правильно применить кэширование, то можно добиться хорошей масштабируемости системы.

6. Код по требованию (необязательное ограничение)

Данное ограничение подразумевает, что клиент может расширять свою функциональность, за счет загрузки кода с сервера в виде апплетов или сценариев.



Что такое RESTful:

Чтобы распределенная система считалась сконструированной по REST архитектуре (Restful), необходимо, чтобы она удовлетворяла следующим критериям:

1. **Client-Server.** Система должна быть разделена на клиентов и на серверов. Разделение интерфейсов означает, что, например, клиенты не связаны с хранением данных, которое остается внутри каждого сервера, так что мобильность кода клиента улучшается. Серверы не связаны с интерфейсом пользователя или состоянием, так что серверы могут быть проще и масштабируемы. Серверы и клиенты могут быть заменяемы и разрабатываться независимо, пока интерфейс не изменяется.
2. **Stateless.** Сервер не должен хранить какой-либо информации о клиентах. В запросе должна храниться вся необходимая информация для обработки запроса и если необходимо, идентификации клиента.
3. **Cache.** Каждый ответ должен быть отмечен является ли он кэшируемым или нет, для предотвращения повторного использования клиентами устаревших или некорректных данных в ответ на дальнейшие запросы.
4. **Uniform Interface.** Единый интерфейс определяет интерфейс между клиентами и серверами. Это упрощает и отделяет архитектуру, которая позволяет каждой части развиваться самостоятельно.

Четыре принципа единого интерфейса:

- *Identification of resources (основан на ресурсах).* **В REST ресурсом является все то, чему можно дать имя**. Например,пользователь, изображение, предмет (майка, голодная собака, текущая погода) и т.д. Каждый ресурс в REST должен быть идентифицирован посредством стабильного идентификатора, который не меняется при изменении состояния ресурса. **Идентификатором в REST является URI.**
- *Manipulation of resources through representations. (Манипуляции над ресурсами через представления).* Представление в REST используется для выполнения действий над ресурсами. Представление ресурса представляет собой текущее или желаемое состояние ресурса. Например, если

ресурсом является пользователь, то представлением может являться XML или HTML описание этого пользователя.

- *Self-descriptive messages (само-документируемые сообщения)*. Под само-описательностью имеется ввиду, что запрос и ответ должны хранить в себе всю необходимую информацию для их обработки. Не должны быть дополнительные сообщения или кэши для обработки одного запроса. Другими словами отсутствие состояния, сохраняемого между запросами к ресурсам. Это очень важно для масштабирования системы.
- *HATEOAS (hypermedia as the engine of application state).* Статус ресурса передается через содержимое body, параметры строки запроса, заголовки запросов и запрашиваемый URI (имя ресурса). Это называется гипермедиа (или гиперссылки с гипертекстом). HATEOAS также означает, что, в случае необходимости ссылки могут содержатся в теле ответа (или заголовках) для поддержки URI , извлечения самого объекта или запрошенных объектов.

5 .**Layered System**. В REST допускается разделить систему на иерархию слоев но с условием, что каждый компонент может видеть компоненты только непосредственно следующего слоя. Например, если вы вызывайте службу PayPal а он в свою очередь вызывает службу Visa, вы о вызове службы Visa ничего не должны знать.

6. **Code-On-Demand (опционально)**. В REST позволяется загрузка и выполнение кода или программы на стороне клиента.

Серверы могут временно расширять или кастомизировать функционал клиента, передавая ему логику, которую он может исполнять. Например, это могут быть скомпилированные Java-апплеты или клиентские скрипты на Javascript

Важно ! Сама архитектура REST не привязана к конкретным технологиям и протоколам, но в реалиях современного Веб, построение RESTful API почти всегда подразумевает использование HTTP и каких-либо распространенных форматов представления ресурсов, например JSON, или, менее популярного сегодня, XML.

▼ Что такое модель зрелости Ричардсона? Основные уровни модели зрелости Ричардсона?

Описывает прогрессивное развитие веб сервисов от простой интеграции до полноценных REST архитектур. Помогает оценить степень зрелости веб сервиса.

Состоит из четырех уровней, каждый из которых представляет собой шаг вперед в использовании основных принципов REST.

Эти уровни включают в себя использование ресурсов, http методов, гипермедиа и независимость клиента.

▼ Еще одно описание

Модель зрелости REST сервисов Леонарда Ричардсона (Richardson Maturity Model)

Уровень 0

Сервисы имеют единственный URI, который использует единственный http-метод (как правило, метод POST).Примерами таких сервисов могут служить большинство веб-сервисов - они используют единственный URI для идентификации точки входа и HTTP POST для передачи SOAP-пакета с данными. Службы XML-RPC и POX(Plain Old XML) используют похожие вызовы, при которых по одному URI передаётся XML, как часть http-запроса.

Уровень 1 (Resources(Ресурсы))

Такие сервисы используют несколько URI, но единственный http-метод. Разница между L1 и L0, заключается в том, что L1 выставляет много логических ресурсов, а сервис уровня L0 объединяет все взаимодействия через единственный (большой и сложный) ресурс. В сервисах уровня L1, имена операций и параметров находятся в самом URI и передаются удалённому сервису, как правило через HTTP GET.

Уровень 2 (HTTP-verbs(http-методы/глаголы))

Сервисы этого уровня имеют много URI-адресуемых ресурсов и используют несколько http-методов для каждого ресурса. Это, как правило CRUD(Create Read Update Delete)-сервисы, где состояние ресурса представляет бизнес-сущность, которой можно управлять через сеть.

Уровень 3 (Hypermedia Controls)

Смысл таких сервисов в том, что при вызове сервиса, они возвращают не только состояние(изменённое или неизменённое), но и ссылки(слоты) на то что мы можем делать дальше с этим сервисом. Фактически, сервисы этого уровня пытаются обеспечить этакую само-документацию, т.е. дать пользователю возможность выбрать дальнейшие действия из доступного списка.

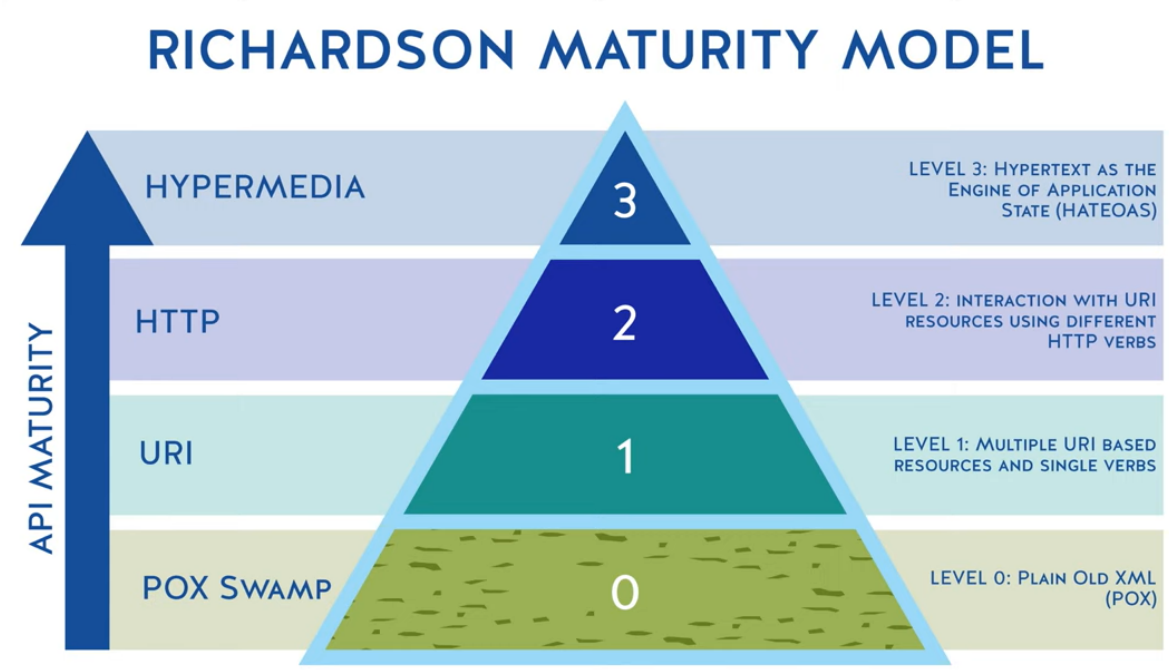
Пример запроса (уровень 3):

```
POST /slots/1234 HTTP/1.1[various other headers]<appointmentRequest>  <patient id = "js
```

и ответа (уровень 3):

```
HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]<appointment>  <slot id = "1234" doctor = "mjones" start = "1400" end
```

Есть 4 уровня зрелости.



- Уровень 1 решает проблему сложности методом «разделяй и властвуй», разбивая конечную точку большого сервиса на множество ресурсов.
- Уровень 2 вводит стандартный набор HTTP методов так, что мы можем обрабатывать подобные ситуации одинаковым образом, устраняя несущественные различия.
- Уровень 3 вносит способность к обнаружению, предоставляя способ сделать протокол более само-документируемым.

УРОВЕНЬ 0: БОЛОТО

- Использование HTTP как простого способа передачи данных
- API предоставляет только **один** эндпоинт (или конечную точку, или URI), который используется **всеми типами** операций.
- использует только метод, например **POST**

УРОВЕНЬ 1: РЕСУРСЫ

Для повышения производительности связи используется несколько **URI** (уникальный идентификатор ресурса).

- использование нескольких эндпоинтов, каждый для своей цели(один **специализированный URI** относится к одному типу операций)
- только один метод HTTP, например POST используется для всех типов операций

УРОВЕНЬ 2: МЕТОДЫ HTTP

Для повышения производительности связи используется несколько **URI** (уникальный идентификатор ресурса) к которым добавляется использование различных HTTP методов.

- один выделенный URI(эндпоинт) относится к одному типу операции
- используются разные HTTP-методы: GET, POST, PUT, PATCH, DELETE.

› GET: Получить предмет или коллекцию

› POST: создание элемента или коллекции

› PUT: обновить элемент (PUT является **идемпотентным**, что означает, что если элемент уже существует, PUT заменит его)

› PATCH: Частичное обновление предмета

› DELETE: удалить элемент или коллекцию

УРОВЕНЬ 3: КОНТРОЛЬ ГИПЕРМЕДИА

Последний уровень знакомит с представлением гипермедиа. Также называемые HATEOAS (Гипермедиа как механизм состояния приложения), это элементы, встроенные в ответные сообщения ресурсов, которые позволяют устанавливать связь между отдельными объектами данных, возвращаемыми из API и передаваемыми в них. Например, запрос GET к системе бронирования отелей может возвращать количество доступных номеров вместе с гиперссылками (на ранних этапах модели это были элементы управления гиперссылками в формате html), позволяющие клиенту забронировать определенные номера.

Это последний уровень модели зрелости Ричардсона.

Запрос:

```
GET/room/?CustomerID=1&date= 10-11-2020&hotelCode=ASTORIA HTTP/1.1
```

Ответ:

```
{
 customerId: "1",
 reservations: [{room: "102", checkin: "10-11-2020", checkout: "11-14-2020", price: "10
0", href: "https://localhost:8080/room/102"}]
}
```

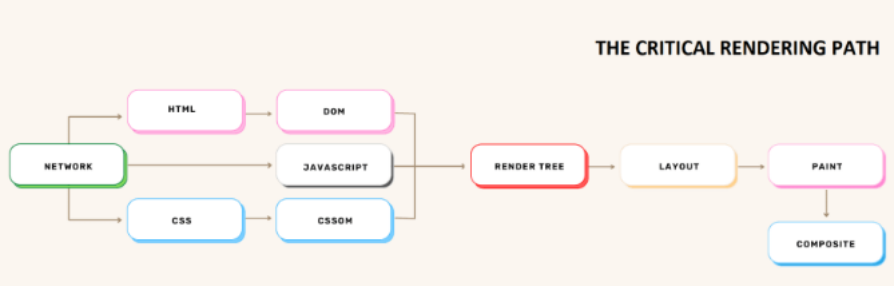
▼ Назовите критические этапы рендеринга?

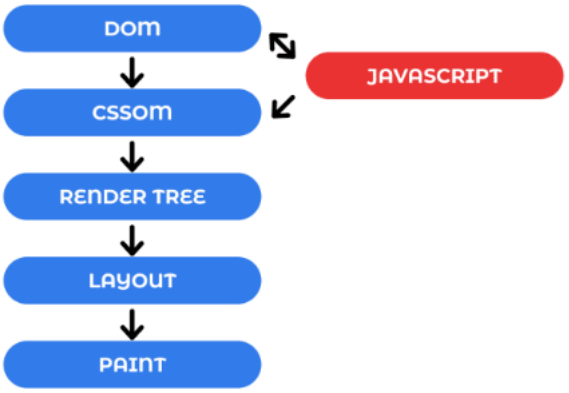
Как браузер рисует страницы

Понимание критического пути рендеринга

Устраните ресурсы, блокирующие рендеринг

Critical Rendering Path - это последовательность шагов, которые выполняет браузер, когда HTML, CSS и JS преобразуются в пиксели, которые видны на экране.





- **Парсинг HTML и создание DOM.**
- **DOM** ответы в виде HTML превращаются в токены, которые в свою очередь превращаются в узлы и в последующем формируют DOM дерево.
- **CSSOM** все данные о том как стилизовать DOM.
- **JavaScript** загрузка всех скриптов.
- **Accessibility Tree** при парсинге HTML, анализируются специальные атрибуты по типу role и aria.
- **Render Tree** Соединение DOM и CSSOM.
- **Layout/Reflow** - это процесс определения размеров и позиций всех элементов на странице, а также их отношений друг к другу. Это происходит на основе CSS-правил и содержимого страницы.
- **Paint/Repaint** это процесс рендеринга элементов на странице. Он включает в себя применение цветов, текстур, градиентов и других стилей к элементам.
- **Compositing** техника разделения частей страницы на слои, их раздельной отрисовки и компоновки в виде страницы в отдельном потоке, называемом потоком композитора. Когда части документа рисуются в разных слоях, накладываясь друг на друга, композитинг необходим для того, чтобы они выводились на экран в правильном порядке и содержимое отображалось корректно. При этом используются мощности GPU.

▼ Разница между layout, painting и compositing?

Все 3 этапа относятся к отрисовке веб-страницы.

layout - это расчет места для объекта на основе CSS правил.

painting - это рисование пикселей для отображения визуальных элементов.

compositing - это отрисовка слоев в определенном порядке, то есть правильно наложение и т.д.

▼ Что такое Flash Of Unstyled Content (FOUC)? Как его избежать?

Flash Of Unstyled Content (Вспышка не стилизованного контента) - появление неоформленного контента при загрузке.

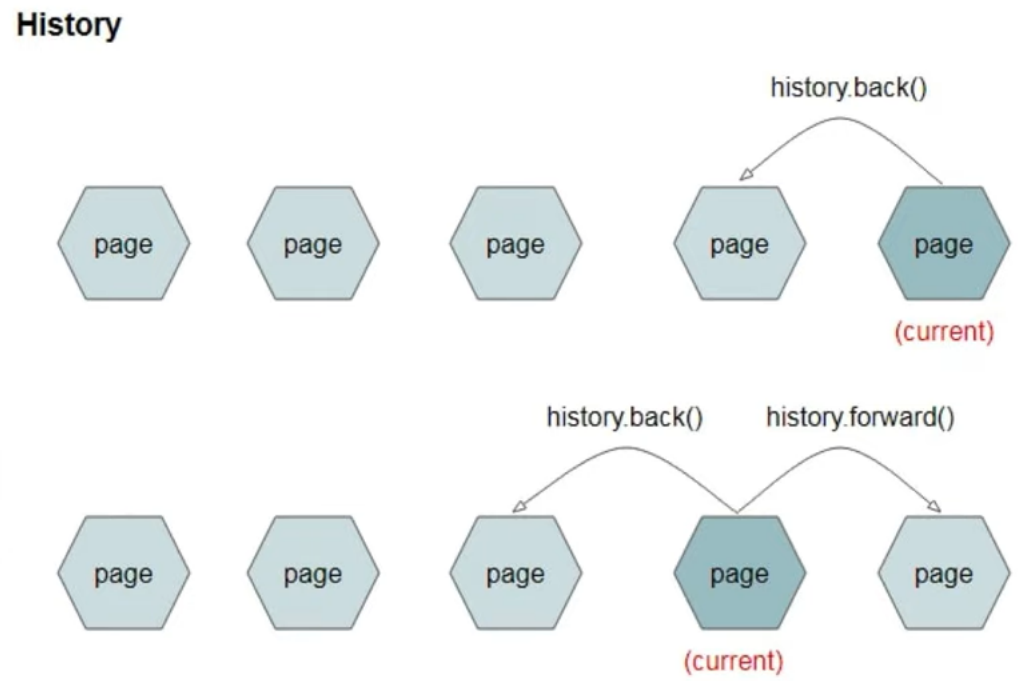
Как избежать - вставить критичный CSS в <head> страницы или показывать прелоадер или скелетон до полной загрузки всего контента.

Критичный CSS это минимальный набор блокирующего CSS требуемого для первого экрана с контентом, которую в первую очередь видит пользователь.

▼ Что такое History API в браузере?

History API - даёт доступ к управлению историей браузера в рамках текущей сессии.

Браузер создает новую сессию, когда пользователь открывает новую вкладку или новое окно браузера.



С помощью History API можно переходить по истории вперёд, назад и управлять содержимым истории. Доступ к API осуществляется с помощью объекта window.history.

Основные методы:

- **back()** перемещает пользователя по истории на страницу назад;
- **forward()** перемещает пользователя по истории на страницу вперёд;
- **go()** универсальный метод для перемещения по истории вперёд или назад;
- **pushState()** добавляет новую запись в истории сессии;
- **replaceState()** изменяет текущую запись в истории сессии.

window.history

▼ Что такое веб-хранилище (web storage)?

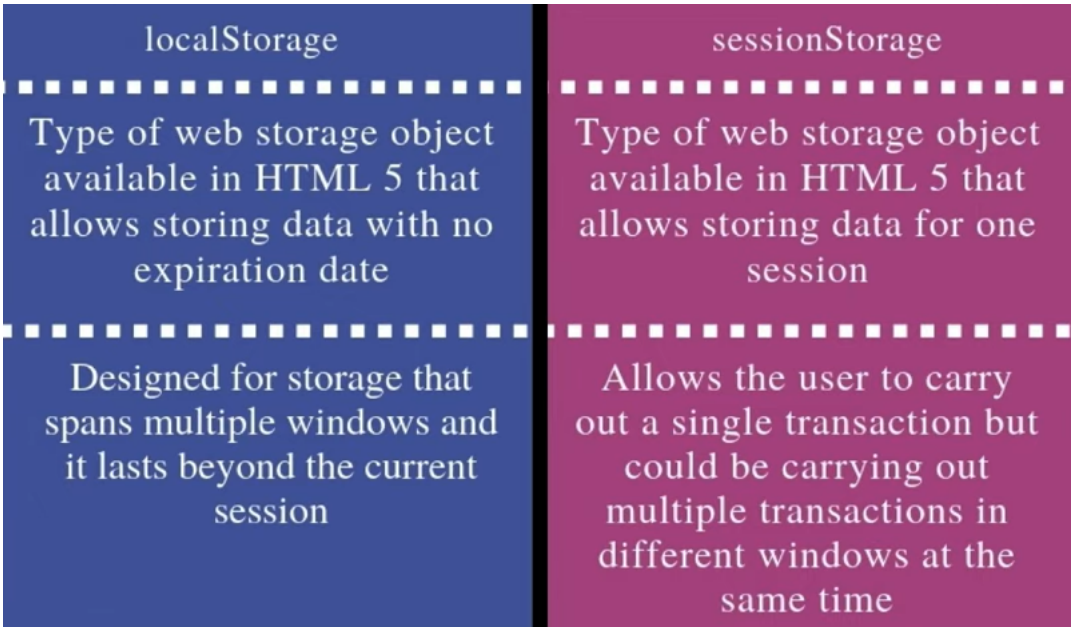
Позволяет хранить данные локально в раузере пользователя.

Хранение данных в браузере

`cookie`

`localStorage`

`sessionStorage`



▼ Разница между `cookie`, `sessionStorage` и `localStorage`?

	cookie	sessionStorage	localStorage
Инициатор	Клиент/Сервер (Set-Cookie)	Клиент	Клиент
Срок хранения	Установка вручную	До закрытия вкладки	Не ограничено
Связь с доменом	Да	Нет	Нет
Емкость	4 Kb	5 Mb	5 Mb
Доступность	Любое окно	Только вкладка	Любое окно

▼ Способы уменьшения времени загрузки веб-страницы?

1. Минификация и конкатенация JS/CSS файлов.
2. Оптимизация/сжатие изображений.
3. Использование CDN. Content Delivery Network
4. Использование gzip метода сжатия
5. Использование кэширования.
6. https://www.cdnvideo.ru/solutions/optimizatsiya-izobrazhenij/?utm_source=yandex&utm_medium=cpc&utm_campaign=gzip&utm_content=11981913057&utm_term=gzip&yclid=

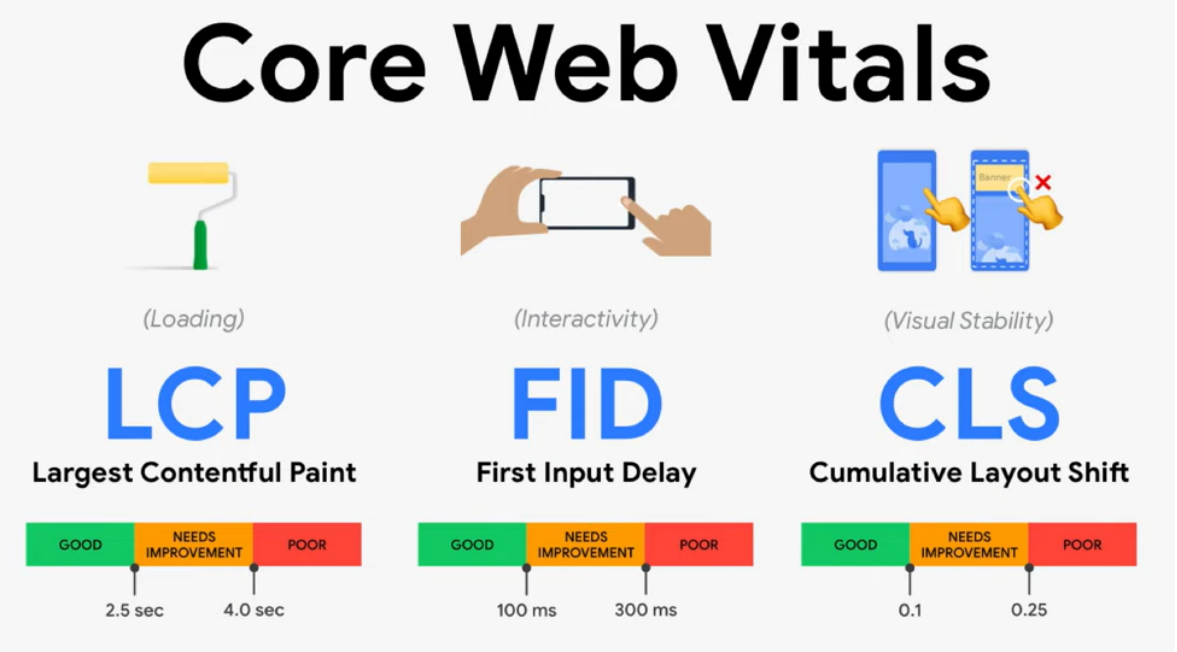
▼ Что такое Core Web Vitals? Какие основные метрики туда входят? Расскажите о метриках Core Web Vitals?

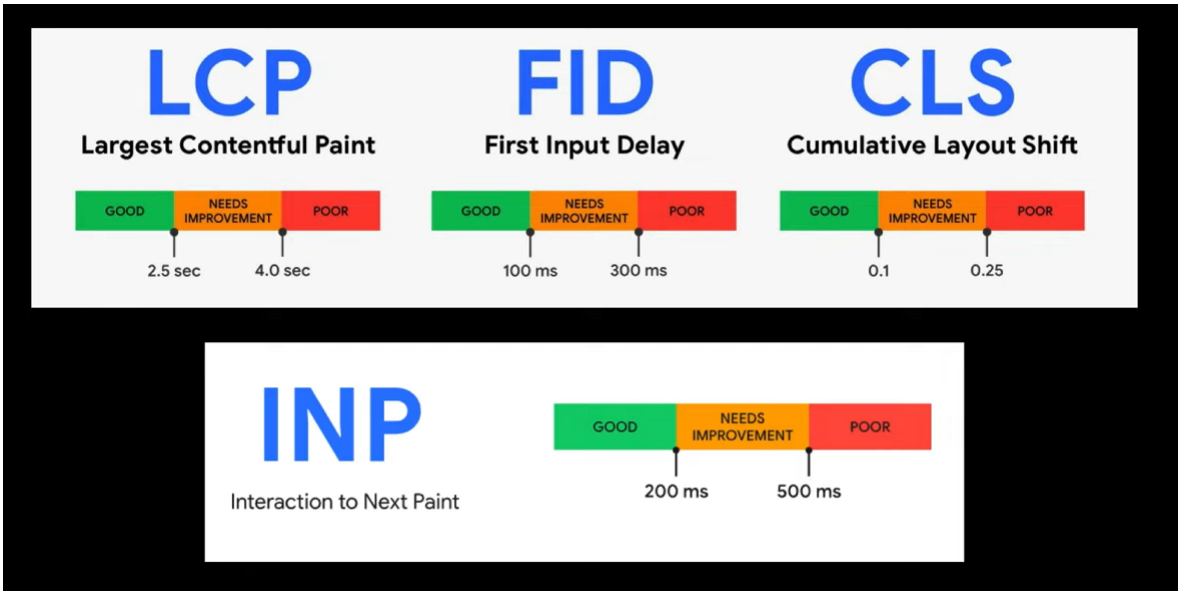
Core Web Vitals - это набор метрик, которые используются Google для оценки качества пользовательского опыта на веб-сайтах. Эти метрики включают в себя:

1. **Largest Contentful Paint (LCP)** время загрузки самого большого контентного элемента на странице, такого как изображение или текст.
2. **First Input Delay (FID)** время задержки между первым взаимодействием пользователя с сайтом и реакцией на это действие.
3. **Cumulative Layout Shift (CLS)** - мера стабильности макета сайта и отсутствия неожиданных перемещений элементов при загрузке страницы.
4. **Interaction to Next Paint (INP) -** интерактивность до следующей отрисовки, пока что это экспериментальный показатель. Она предназначена для измерения общей скорости ответной реакции на действия пользователя на странице. Как считают эксперты, метрика сможет стать альтернативой FID (First Input Delay), которая учитывается при оценке Core Web Vitals. Представители компании Google объяснили введение показателя INP в связи с тем, что FID имеет несколько «больших слепых зон». INP

при этом учитывает жизненный цикл страницы полностью, со всеми взаимодействиями с пользователем, а не только с первым.

Google рекомендует, чтобы все три метрики Core Web Vitals были удовлетворены на сайте, чтобы улучшить пользовательский опыт и рейтинг в поисковой выдаче.





▼ Разница между `preload`, `prefetch`, `preconnect` и `prerender`?

Все это значения атрибута `rel`. Используются для оптимизации загрузки ресурсов.

- **preload** говорит браузеру как можно скорее загрузить и кэшировать ресурс, напрмер скрипт или таблицу стилей, загрузка происходит с высоким приоритетом. Это полезно когда ресурс понадобится через несколько секунд после загрузки страницы и желательно ускорить этот процесс. Браузер ничего не делает с ресурсом после загрузки, ресурс просто кешируется и предоставляется по запросу.
- **prefetch** анаогичен preload**,** просит браузер загрузить и кэшировать ресурс (например, скрипт или таблицу стилей) в фоновом режиме, но загрузка происходит с низким приоритетом, поэтому не мешает более важным ресурсам например preload. Полезно когда ресурс может понадобиться на следующей странице или в каком то подменю приложения.

- **preconnect** просит браузер заранее подключиться к домену, когда вы хотите ускорить установку соединения в будущем. Установка соединения занимает несколько сотен милисекунд, она происходит один раз, но все равно занимает время, которое можно сэкнонмить установив его заранее что позволит быстрее начать загрузку ресурсов с домена.
- **prerender** просит браузер загрузить URL-адрес и отобразить его на невидимой вкладке. Когда пользователь нажимает на ссылку, страница должна отобразиться немедленно. Полезно когда мы уверены что пользоваель посетит эту страницу и хотим ускорить ее отображение.

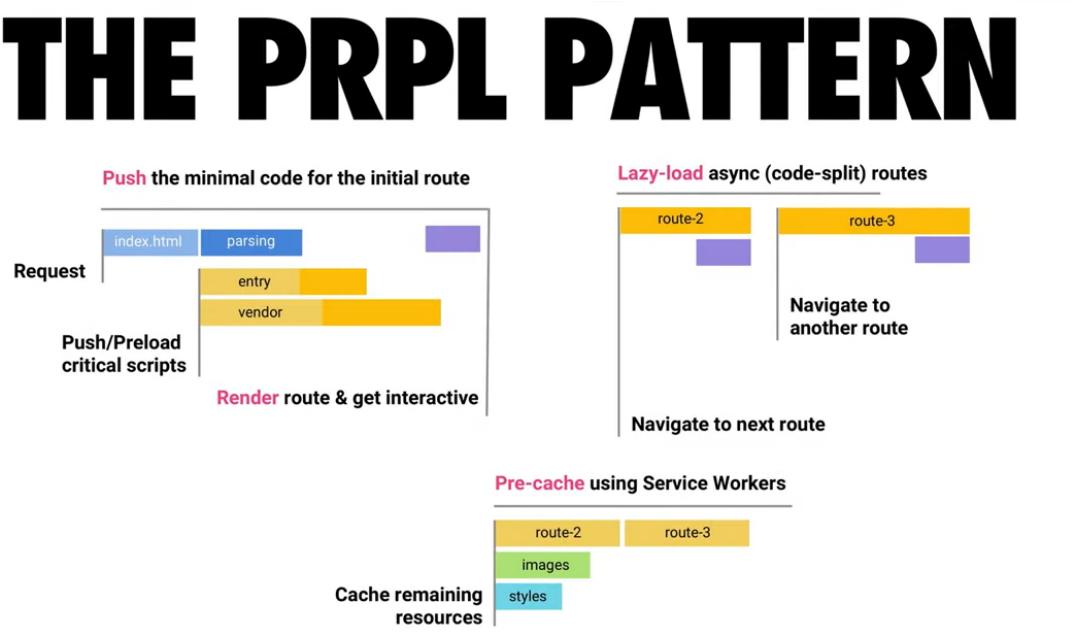
Technique	Code	Where?	What?
DNS-Prefetch	<link rel="dns-prefetch" href="">	(Sub-)Domain, URL	Conveys he DNS info of a domain
Preconnect	<link rel="preconnect" href="">	(Sub-)Domain, URL	Conveys the DNS, TCP, TLS (SSL) - infos of a domain
Prefetch	<link rel="prefetch" href="">	CSS/JS/Image	Stores a resource with low prio in browser cache
Subresource	<link rel=subresource" href="">	CSS/JS/Image	Stores a resource with high prio in browser cache
Preload	<link rel=preload" href="">	URL/CSS/JS/Image	Stores a resource with high prio in browser cache (experimental/working draft)
Prerender	<link rel=prerender" href="">	URL	Stores a URL plus all associated resources in the browser cache and renders the complete page at the same time

▼ Для чего нужен паттерн PRPL?

Паттерн для струтурирования и улучшения производительности веб приложений, например SPA или PWA.

Описывает четыре этапа жизненного цикла приложения, от этапа доставки кода до этапа его отрисовки.

- **Push** — отправка, или **preload** — предварительная загрузка, наиболее важных ресурсов при открытии приложении.
- **Render** — отрисовка первого экрана приложения с использованием минимльного набора необходимых ресурсов.
- **Pre-cache** — предварительная подгрузка и кеширование оставшихся ресурсов для тех страниц приложения на которые вероятнее всего перейдет пользователь.
- **Lazy load** — отложенная загрузка других маршрутов и некритических ресурсов по мере необходимости в процессе работы приложения. Реализуется с помощью lazy loading, code splitting и динамических импортов.



▼ Разница между SSR и SSG и CSR?

SSR (Server-Side Rendering) и SSG (Static Site Generation) - это два подхода к генерации веб-страниц.

SSR означает, что HTML страницы на каждый запрос генерируются на сервере и отправляются на клиентский браузер уже готовыми. Это позволяет улучшить SEO, так как поисковые роботы видят полный контент страницы, а также ускорить первоначальную загрузку страницы. Фреймворки, которые реализуют server rendering - *Next.js*

Реализуется с помощью getServerSideProps, которые потом передаются в пропс компоненту.

Плюсы: Отрисовка быстрая

```
function Page({ data }) {
  return (
    <ul>
      {data.map((post) => (
        <li key={post.id}>
          <Link href={`/server-side-rendering/${post.id}`}>
            <a>{post.title}</a>
          </Link>
        </li>
      ))}
    </ul>
  )
}

// This gets called on every request
export async function getServerSideProps() {
  // Fetch data from external API
  const res = await fetch('https://jsonplaceholder.typicode.com/posts')
  const data = await res.json()

  // Pass data to the page via props
  return {
    props: {
      data: getByIdsRange(data, 1, 50),
    },
  }
}
```

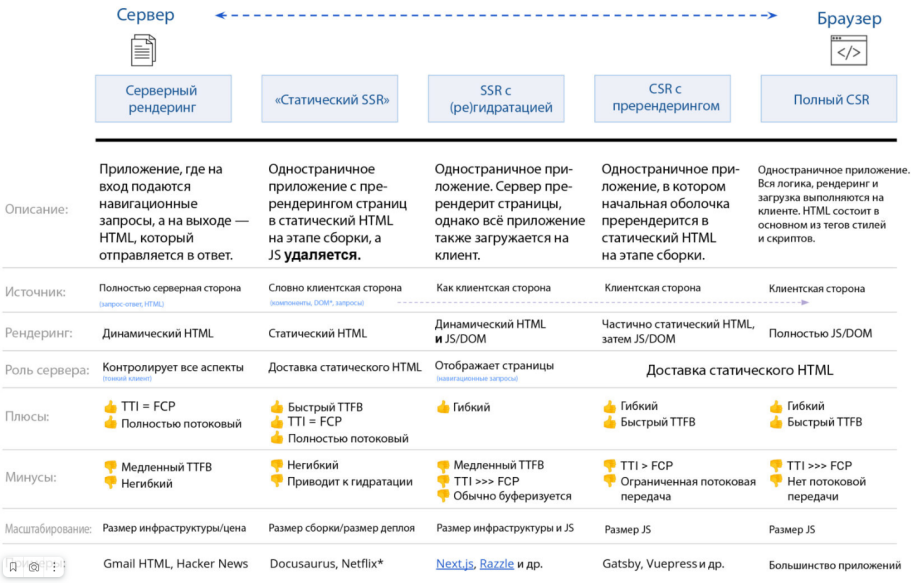
SSG означает, что HTML страницы генерируются на стадии сборки проекта и сохраняются в виде статических файлов. Это позволяет ускорить загрузку страницы, так как серверу не нужно генерировать страницу каждый раз при запросе, а также уменьшить нагрузку на сервер.

Основное отличие между SSR и SSG заключается в том, что SSR генерирует страницы динамически на каждый запрос, а SSG генерирует страницы статически на этапе сборки проекта.

Также в SSR все запросы происходят на сервере, а у SSG на клиенте.

CSR - на стороне клиента - используются со всеми популярными фреймворками. Ресурсы загружаются единожды, весь бандл, html css. Весь рендер происходит в браузере. Минусы - долгое время первичной загрузки, изначально пустой html, слабое SEO.

Плюсы - весь процесс работы с API на стороне клиента. Может быть закешировано. Экономия ресурсов сервера.



▼ Что такое Babel и для чего он используется?

Babel - это траспайлер, который переписывает код с современного стандарта JavaScript на более ранний, программа позволяющая менять исходный код одной программы на эквивалентный код на другом языке.

Основная идея: Ecma International каждый год выпускает обновления для JS, однако эти обновления внедряются в браузер постепенно, чтобы не терять время и сразу начать использовать новые фичи, которые зачастую упрощают разработку используется Babel, которые транслирует язык на более ранний для лучшей кроссбраузерности.

▼ Разница между feature detection, feature inference и анализом строки user-agent?

Feature detection (определение возможностей браузера) - заключается в определении поддерживает ли браузер определенный блок кода, и если блок не поддерживается, то будет выполнен аналог, либо полифил, такой подход помогает обеспечить работоспособность, и предотвратить сбои и ошибки.

```
// Feature detection
if ('geolocation' in navigator) {
  // use navigator.geolocation
} else {
  // another code
}
```

Feature inference (определение возможностей) - это проверка на наличие определенных возможностей, подход применяет функцию, которая предполагает, что определенная возможность уже существует. Если фича X работает, то и фича У должна работать.

```
// Feature inference
if (document.getElementsByTagName) {
  element = document.getElementById(id);
}
```

User-agent - строка сообщаемая браузером, которая позволяет определить: тип приложения, операционную систему, поставщика ПО. Доступ к ней можно получить через navigator.userAgent. Иногда выдает некорреткную информацию о браузере.

Рекомендуется использовать первый вариант.

```
// User Agent
console.log(navigator.userAgent);
// "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.106
```

▼ Что такое блокирующие и неблокирующие ресурсы?

Блокирующие ресурсы - это ресурсы, которые необходимы для отображения страницы и которые не могут быть загружены асинхронно. Примерами блокирующих ресурсов являются файлы CSS, JavaScript и шрифты.

Неблокирующие ресурсы - это ресурсы, которые могут быть загружены асинхронно, без блокирования парсера. Примерами неблокирующих ресурсов являются изображения и видео.

▼ Что такое DOM?

DOM (Document Object Model) - это абстрактное представление HTML-документа с помощью которого браузер может получать доступ к его элементам, изменять его структуру и оформление.

▼ Когда происходит Reflow и Repaint?

Reflow происходит, когда происходят изменения в размерах, позициях или содержимом элементов на странице. Это может произойти при добавлении или удалении элементов, изменении размеров окна браузера или изменении содержимого элементов.

Repaint происходит, когда изменяются стили элементов на странице, но не их размеры или позиции. Это может произойти при изменении цвета фона, шрифта или других стилей элементов.

▼ Что такое GraphQL?

GraphQL - это язык запросов для API, который позволяет клиентским приложениям запрашивать только те данные, которые им нужны, и получать их в удобном формате.

GraphQL работает следующим образом:

1. Определение схемы: разработчик определяет схему, которая описывает типы данных и операции, которые могут быть выполнены.
2. Создание запроса: клиентское приложение создает запрос, указывая необходимые данные и операции.
3. Обработка запроса: сервер GraphQL обрабатывает запрос и возвращает только запрошенные данные.
4. Обновление данных: если данные на сервере изменились, клиент может подписаться на определенные данные и получать уведомления об изменениях.

В GraphQL используется одна ссылка(эндпоинт) для всех запросов. Всегда идет POST запрос, даже если мы просто хотим получить какие-то данные.

GraphQL позволяет улучшить производительность и эффективность API, уменьшив количество запросов и объем передаваемых данных.

mutations

query

overfetching

underfetching

https://habr.com/ru/articles/326986/

GraphQL - это язык запросов для API, разработанный Facebook. Он позволяет клиентам запрашивать только те данные, которые им нужны, и получать их в оптимизированном формате.

Преимущества GraphQL:

- Гибкость: клиенты могут запрашивать только те данные, которые им нужны, а не весь объект целиком, что уменьшает нагрузку на сервер и ускоряет работу приложения.
- Единый интерфейс: GraphQL предоставляет единый интерфейс для всех клиентов, что упрощает разработку и поддержку приложения.
- Автодокументирование: GraphQL автоматически документирует API на основе схемы данных, что упрощает работу с ним.

Недостатки GraphQL:

- Сложность: GraphQL может быть сложным для понимания и использования, особенно для новичков.
- Кэширование: из-за гибкости GraphQL сложнее кэшировать запросы, что может привести к проблемам с производительностью.
- Уязвимость к DoS-атакам: из-за возможности запрашивать большое количество данных, GraphQL может быть уязвим к DoS-атакам.

▼ Что такое JWT?

Видео

JWT расшифровывается как **JSON Web Token**. Это стандарт для создания токенов доступа, которые могут передаваться между двумя сторонами в формате **JSON**.

```
           // Заголовок                                        // Полезная нагрузка
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwia
```

JWT состоит из трех частей:

- **заголовка** (header)
- **полезной нагрузки** (payload)
- **подписи** (signature)

1. **Заголовок (header)**: содержит алгоритм хеширования (обязательно), используемый для подписи токена, и тип токена (не обязательно).

   Обычно это JSON-объект, который выглядит примерно так:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. **Полезная нагрузка (payload)**: это основная часть токена, которая содержит утверждения (claims). Чаще всего мы прописываем их сами, обычно это информация о пользователе, который проходит авторизацию.

   Утверждения бывают трех типов:

   - утверждения о пользователе (registered claims)
   - утверждения о претензиях (public claims)
   - приватные утверждения (private claims).

   Обычно это JSON-объект, который выглядит примерно так:

```
{
  "id": "1234567890",
  "userName": "John Doe",
```

```
  "admin": true
}
```

3. **Подпись (signature)**: используется для проверки подлинности данных в токене. Она вычисляется на основе заголовка, полезной нагрузки и секретного ключа, который известен только серверу, который создал токен.

Заголовок и полезная нагрузка хранятся в “открытом” виде, их может раскодировать каждый, поэтому не стоит там хранить важную информацию (пароли и т.д.)

Web API

▼ Что такое `HTTP` ?

HTTP (HyperText Transfer Protocol) - это протокол, позволяющий получать различные ресурсы, например HTML-документы или любые другие произвольные данные.

Используется в основном для связи между веб браузерами и веб серверами.

Это протокол без сохранения состояния, сервер не запомниает никаких данных между парами запрос/ответ.

Следует классической клиент серверной моедли где клиент окрывает соединение для создания запроса и затем ждет ответа от сервера.

Все ПО для работы протокола разделяется на три категории:

- Клиент: потребител услуг
- Сервер: поставщик услуг
- Прокси: посредник, используется для выполенния транспортных служб

Протокол HTTP лежит в основе обмена данными в Интернете. HTTP является протоколом клиент-серверного взаимодействия, что означает инициирование запросов к серверу самим получателем, обычно веб-браузером.

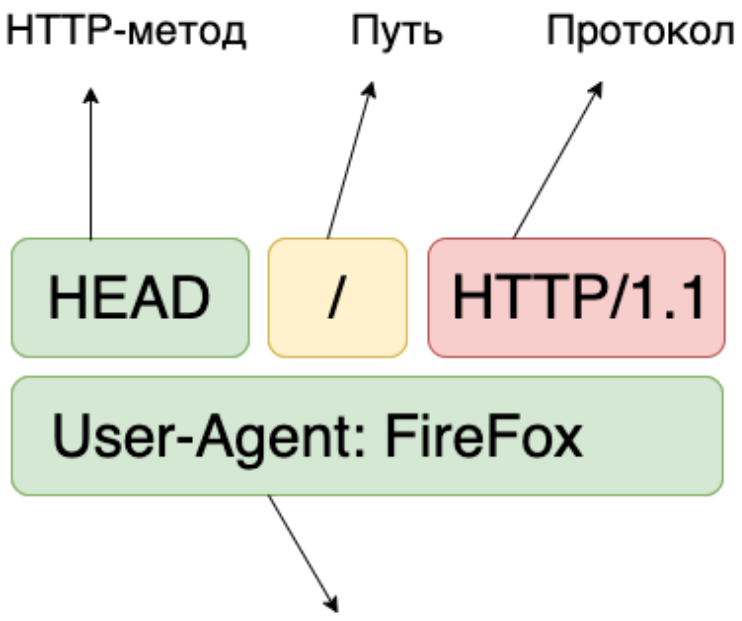
▼ Из чего состоит `HTTP` запрос?

▼ `HTTP`

- `HyperText Transfer Protocol` - протокол передачи гипертекста
- При обращении к сайту по символьному имени осуществялется запрос в `DNS` сервер который на основании имени возвращает `ip` адрес. Этот адрес используется для установления соединения по протоколу `TCP` на порт 80 для `HTTP` и на порт 443 для `HTTPS`

- `HTTPS` - `HyperText Transfer Protocol Secure` расширение обычного HTTP, в нем соединение шифруется с помощью `SSL` и более современного `TLS`. Для работы HTTPS необходим сертификат открытого и закрытого ключа, который можно получить в центре сертификации. Сертификат открытого ключа подтверждает принадлежность данного открытого ключа владельцу сайта. Сертификат открытого ключа и сам открытый ключ посылаются клиенту при установлении соединения, а закрытый ключ используется для расшифровки сообщений от клиента
- После установления соединения браузер отправляет `HTTP` запрос на сервер, и получает `HTTP` ответ

▼ `HTTP` запрос состоит из строки запроса(стартовая строка), заголовков и тела запроса.



Заголовки. Ключ от значения отделяются двоеточием.

- Основные методы `GET`, `PUT`, `POST`, `DELETE`
- Строка запроса `HEAD /posts HTTP/1.1` где `HEAD` - метод, `/posts` - путь, адрес ресурса, `HTTP/1.1` - версия протокола
- Заголовки передают служебную информацию, просто уточняют запрос.

- Заголовки передаются ниже строки запроса и представляют собой пары `ключ: значение` например `User-Agent: Google Chrome` где `User-Agent` указывает на браузер или программу сформировавшую запрос. Заголовки `Content-Type: application/json` и `Content-Length: 88` указывают на тип содержимого и его длинну в символах(байтах). Основные типы заголовков определены спецификацией, но можно создавать и свои, указывая X перед именем заголовка, например `X-Token: secret token` и передавая необходиму информацию на сервер или от сервера на клиент.
- Заголовок `Host: jsonplaceholder.typicode.com` указывает именно на ресурс указанный в его значении, даже если на данном сервере расположено множество сайтов, обращение будет к указанному в `Host`. Если не указать `Host` и на сервере расположено множество сайтов, то обращение будет к сайту по умолчанию настроенному для данного сервера

▼ Тело запроса содержит полезные данные, например данные введенные в форму пользователем

```
POST /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
Content-Type: application/json
Content-Length: 88
{
"title": "Новая публикация",
"body": "Текст публикации",
"userId": 31337
}
```

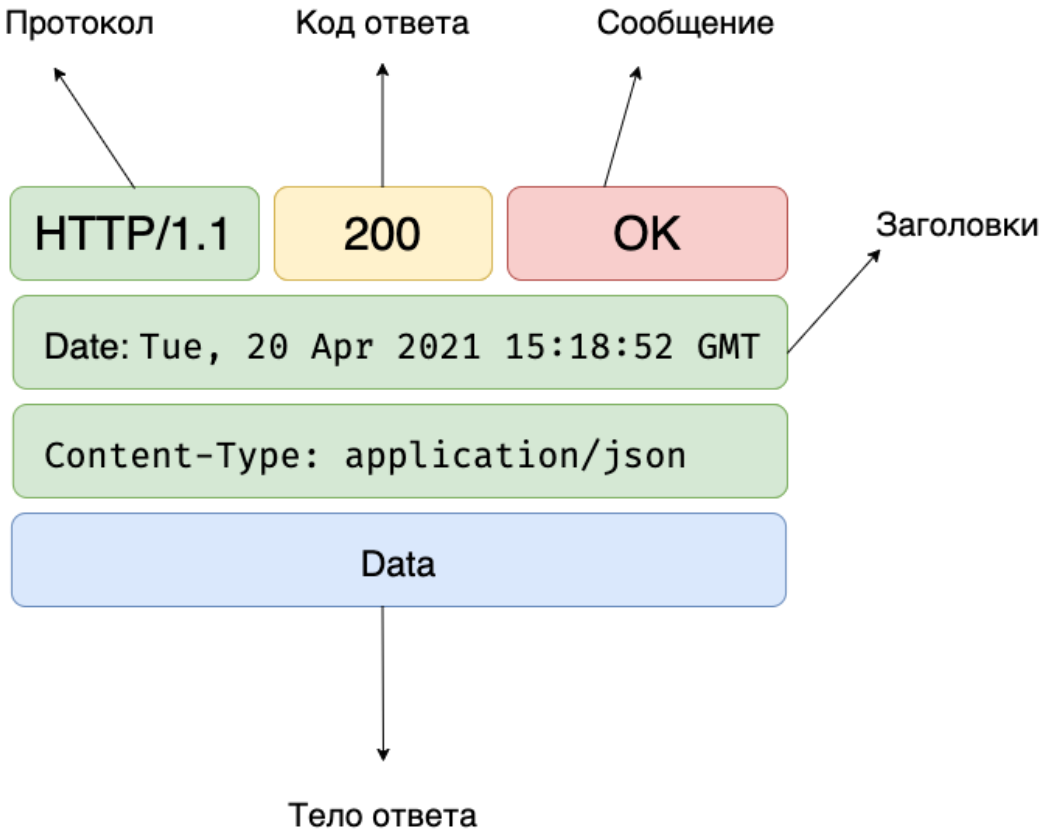
или

```
POST /auth HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 26
login=keks&password=secret
```

- В теле запроса передача служебных символов таких как `=` или `&` в качестве данных пользователя чтобы не было ошибок

при передаче решается с помощью кодирования данных, этим браузер занимается самостоятельно.

▼ `HTTP` ответ



▼ Первой строкой идет строка состояния которая содержит версию протокола, код ответа и расшифровку ответа. Первая цифра кода ответа определяет класс состояния, остальные две его дополняют.

- 1 — информационные (informational)
- 2 — успешно (success)
- 3 — перенаправление (redirection)
- 4 — ошибка клиента (client error)
- 5 — ошибка сервера (server error)

- Далее идут заголовки, принцип такой же как и у `HTTP` запросов

- В теле ответа содержатся данные, например `HTML` код запрашиваемой страницы сайта.

▼ Базовая аутентификация и авторизация

- Basic authentication позволяет ограничить доступ к определённым страницам без программирования
- При запросе к серверу с закрытм доступом, ответ будет содержать

```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Protected server"
```

- Затем браузер покажет модальное окно в которое необходимо ввести логин и пароль, после чего браузер отправит запрос с закодированным логином и паролем. И если данные совпадут доступ будет предоставлен

```
GET /secret-page HTTP/1.1
Host: myserver.local
Authorization: Basic YWRtaW46MTIzNDU2
```

Для принятия решения о том, как обрабатывать URL, браузеры используют MIME типы, *а не расширения файлов*, так что серверам необходимо отправлять правильные MIME типы в `Content-Type` заголовке ответа. При неточном задавании этого заголовка, браузеры с большой вероятностью будут неправильно интерпретировать и обрабатывать содержание файлов, из-за чего сайт будет работать неверно.

Простейший MIME тип состоит из *типа* и *подтипа* — двух строк разделённых наклонной чертой (`/`), без использования пробелов.

```
тип/подтип
```

Тип представляет общую категорию, в которой находится тип данных, например `video` или `text`. **Подтип** же строго отождествляется с отдельным типом данных, представляемых данным MIME типом. Например, для MIME типа `text`, подтипы могут быть `plain` (простой текст), `html` (HTML source code) или `calendar` (для iCalendar `.ics`).

Необязательный **параметр** может быть добавлен для указания дополнительных деталей

```
тип/подтип;параметр=значение
```

Например, для MIME типов категории `text`, необязательный параметр `charset` может быть задан для уточнения кодировки, используемой в документе. Для объявления, что пересылаемый файл имеет кодировку UTF-8, необходимо использовать MIME тип `text/plain;charset=UTF-8`. При не указании параметра `charset`, его значение автоматически будет задано, как ASCII (`US-ASCII`), если в настройках браузера не будет определено иначе.

MIME типы являются нечувствительными к регистру, но традиционно их пишут строчными буквами, за исключением значений параметров.

Типы

https://ru.wikipedia.org/wiki/Список_MIME-типов

Все типы можно разделить на два класса: **дискретные** и **многокомпонентные.** Дискретные типы представляют одиночные файлы, например, одиночный текстовый, музыкальный или видео файл. Многокомпонентные типы представляют документы, составленные из нескольких частей, каждая из которых может иметь свой отдельный MIME тип, или они могут заключать в себе несколько отдельных файлов, передаваемых в одном сообщении. Например, многокомпонентные MIME типы используются для передачи нескольких изображений в одном email.

Сообщение типа `multipart/form-data` состоит из нескольких частей, каждая из которых представляет содержимое некоторого элемента формы. Части отправляются обрабатывающему агенту в том же порядке, в котором соответствующие управляющие элементы представлены в потоке документа с формой. Каждая часть должна содержать:

1. Заголовочное поле `Content-Disposition`, имеющее значение `form-data`.
2. Атрибут `name` определяет имя соответствующего управляющего элемента. Имена управляющих элементов, изначально закодированные с использованием наборов символов, отличных от ASCII, могут кодироваться с помощью метода, описанного в RFC 2047.[1]

https://doka.guide/js/form-data/

Существует несколько самых популярных способов кодирования данных для отправки на
сервер: `'application/xwwwformurlencoded'`, `'multipart/formdata'` и `'application/json'`. Иногда бывает так, что сервер поддерживает только какой-то определённый способ. Тогда выбирать не приходится. Но чаще всего современные решения на бэкенде поддерживают несколько способов, поэтому выбирать нужно в зависимости от задачи.

`'application/xwwwformurlencoded'` — способ, который используют HTML-формы по умолчанию. Из-за особенностей преобразования, этот способ плохо подходит для больших объёмов данных, в особенности, файлов или строк с большим количеством символов не из ASCII таблицы (например, символы русского алфавита).

`'application/json'` — достаточно популярный формат из-за широкого распространения JSON как формата обмена данными. Из плюсов - поддерживает вложенные структуры, поэтому можно в одном запросе отправить, например, целый объект с данными. Однако, чтобы отправить файл при помощи этого формата, необходимо файл дополнительно закодировать в строку каким-нибудь алгоритмом, например Base64. Причём на сервере нужно декодировать эти данные обратно.

`'multipart/formdata'` — удобный способ для загрузки файлов, оптимален с точки зрения размера закодированных данных, но в качестве значений может хранить только строки или файлы.

Поэтому, лучше всего использовать `FormData` для отправки файлов на сервер или когда поддержка только строковых данных не является проблемой. Дополнительно, при создании `FormData` можно передать DOM-элемент формы (будет рассмотрено ниже), и коллекция вытащит из этой формы все данные. Поэтому, если стоит задача отравить данные какой-либо формы, `FormData` позволит сделать это с минимумом кода.

▼ Какие методы может иметь `HTTP` запрос?

Указывает на действие которое хочет выполнить пользователь

- **GET** - получение ресурса.
- **POST** создание нового ресурса.
- **PUT** обновление существующего ресурса.
- **PATCH** обновление части существующего ресурса.

- **DELETE** удаление ресурса.
- **HEAD** получение метаданных ресурса без тела ответа.
- **OPTIONS** получение списка поддерживаемых методов и параметров для ресурса.
- **CONNECT** установление сетевого соединения с ресурсом.
- **TRACE** получение диагностической информации о запросе и ответе на него от сервера.

Method	Used*	Safe	Idempotent	Description
`GET`	Y	Y	Y	Retrieves a representation of the resource at the given URI.
`HEAD`	Y	Y	Y	Retrieves a representation of resource at the given URI without the body.
`POST`	Y	N	N	Creates the provided resource as a child of the one identified by the given URI.
`PUT`	Y	N	Y	Stores (i.e creates or updates) the provided resource at the given URI.
`PATCH`	Y	N	N	Submits a partial modification to the resource at the given URI.
`DELETE`	Y	N	Y	Deletes the resource at the given URI.
`OPTIONS`	Y	Y	Y	Returns the methods the server supports for the given URI.
`TRACE`	N	Y	Y	Echoes the request, so that the client can trace any modifications made to it.
`CONNECT`	N	N/D	N/D	Converts the connection to a TCP/IP tunnel, useful for upgrading it to SSL.

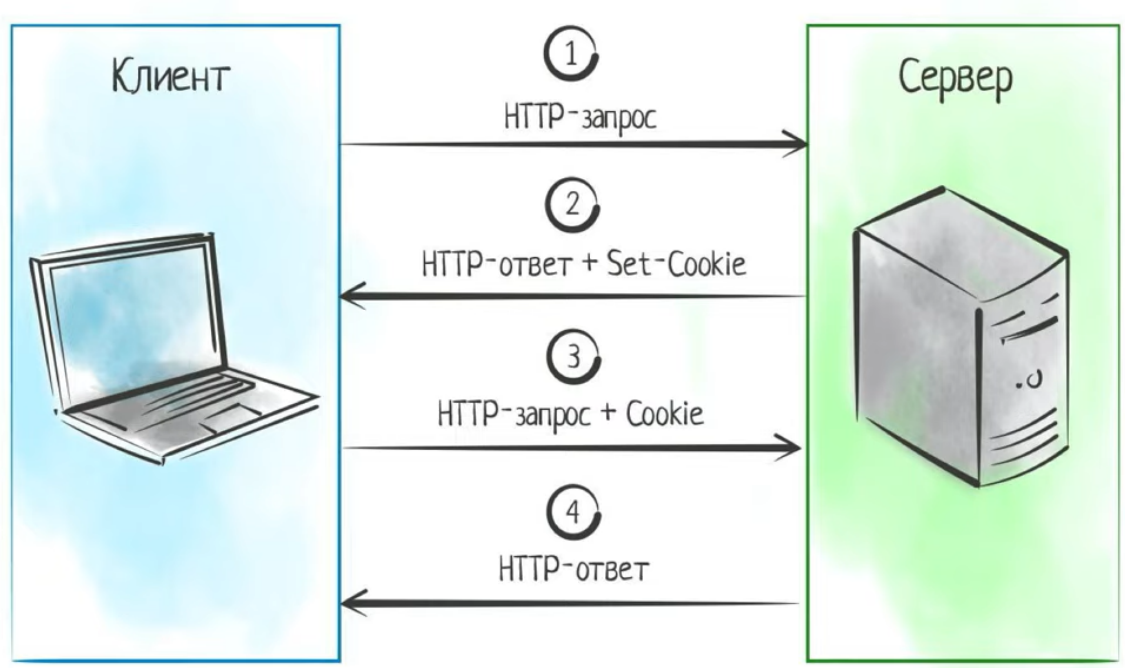
▼ Что такое `HTTP` cookie? Для чего они используются?

HTTP Cookie - это небольшой фрагмент данных отправляемых сервером на браузер пользователя, который тот может сохранить и отсылать обратно с новым запросом к данному серверу.

При обращении к серверу он возвращает куки, которые потом при каждом запросе на сервер передаются вместе с запросом, так сервер может идентифицировать пользователя и например не запрашивать авторизацию повторно.

Используются для:

- Управление сеансом (логины, токены)
- Мониторинга, отслеживания поведения пользователя
- Персонализации (пользовательские предпочтения)



`document.cookie` предоставляет доступ к куки.

- Операция записи изменяет только то куки, которое было указано.
- Имя и значение куки должны быть закодированы.
- Одно куки вмещает до 4kb данных, разрешается не менее 20 куки на сайт (зависит от браузера) и не менее 300 в общем, если они помещаются в выделенные 4 килобайта.

Настройки куки:

- `path=/`, по умолчанию устанавливается текущий путь, делает куки видимым только по указанному пути и ниже.
- `domain=site.com`, по умолчанию куки видно только на текущем домене, если явно указан домен, то куки видно и на поддоменах.
- `expires` или `max-age` устанавливает дату истечения срока действия, без них куки умрёт при закрытии браузера.
- `secure` делает куки доступным только при использовании HTTPS.

- `samesite` запрещает браузеру отправлять куки с запросами, поступающими из других доменов, помогает предотвратить XSRF-атаки.

Дополнительно:

- Сторонние куки могут быть запрещены браузером, например Safari делает это по умолчанию.
- Установка отслеживающих куки пользователям из стран ЕС требует их явного согласия на это в соответствии с законодательством GDPR.

Чтение из document.cookie

Хранит ли ваш браузер какие-то куки с этого сайта? Посмотрим:

```
// На javascript.info мы используем сервис Google Analytics
// поэтому какие-то куки должны быть
alert( document.cookie ); // cookie1=value1; cookie2=value2;
```

Значение `document.cookie` состоит из пар `ключ=значение`, разделённых `;`. Каждая пара представляет собой отдельное куки.

Чтобы найти определённое куки, достаточно разбить строку из `document.cookie` по `;`, и затем найти нужный ключ. Для этого мы можем использовать как регулярные выражения, так и функции для обработки массивов.

Запись в document.cookie

Мы можем писать в `document.cookie`. Но это не просто свойство данных, а аксессор (геттер/сеттер). Присваивание к нему обрабатывается особым образом.

Запись в `document.cookie` обновит только упомянутые в ней куки, но при этом не затронет все остальные.

Например, этот вызов установит куки с именем `user` и значением `John`:

```
document.cookie = "user=John"; // обновляем только куки с им
alert(document.cookie); // показываем все куки
```

Если вы запустите этот код, то, скорее всего, увидите множество куки. Это происходит, потому что операция `document.cookie=` перезапишет не все куки, а лишь куки с вышеупомянутым именем `user`.

Технически, и имя и значение куки могут состоять из любых символов, для правильного форматирования следует использовать встроенную функцию `encodeURIComponent`:

```
// специальные символы (пробелы), требуется кодирование
let name = "my name";
let value = "John Smith"

// кодирует в my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURI

alert(document.cookie); // ...; my%20name=John%20Smith
```

При установке кук можно указывать не только её название и значение, но и другие параметры. Все они являются необязательными и разделяются точкой с запятой `;`

Эти настройки указываются после пары `ключ=значение` и отделены друг от друга разделителем `;`, вот так:

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 20
```

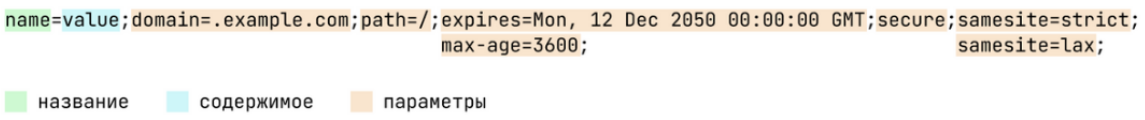
- `path` – определяет путь, по которому будет доступна кука. Он должен быть абсолютным, то есть начинаться с `/`. Если параметр не передан, то кука будет доступна на всех страницах сайта.
- `domain` - определяет домен, для которого указана кука. Если не указано, то будет использоваться текущий домен.
- `maxage` и `expires` - определяет время жизни куки. `maxage` указывает, через сколько секунд, а `expires` указывает точное время, когда кука станет недействительна. Время для `expires` можно отформатировать с помощью встроенного метода даты `Date.toUTCString()`
- `secure` - указывает, что данная кука может быть передана только при запросах по защищённому протоколу HTTPS.
- `samesite` - определяет, может ли данная кука быть отправлена при кросс-доменном запросе. Значение параметра `strict` будет

предотвращать отправку на другие домены, а `lax` разрешит отправлять куки с GET-запросами.

Есть пара ограничений при специфичных названиях кук. Если название куки начинается с `__Secure`, то обязательно должен быть передан параметр `secure`. При этом мы должны находиться на странице, которая была получена по HTTPS-протоколу. Если название куки начинается с `__Host`, то обязательно должны быть переданы параметры `path=/` и `secure` (страница также должна быть открыта по HTTPS-протоколу), а атрибут `domain` должен отсутствовать для снижения кроссдоменных уязвимостей.

Запись куки с разрешением передавать её только по HTTPS и только для текущего домена, со временем жизни в 1 час будет выглядеть так:

```
document.cookie = 'sidebar=true;secure;samesite=strict;max-a
```



Для установки куки, которая будет доступна на текущем домене и всех его поддоменах, используйте название текущего домена и поставьте точку в начале

```
.${window.location.hostname}
```

Cookie — это просто пара имя-значение, которую (точнее, которые) сервер может оставить у клиента (браузера). Наглядно:

1. Приходит клиент, спрашивает у сервера страницу.
2. Сервер в заголовках ответе может установить cookies. Например, выдав два заголовка: `Set-Cookie: foo=123` и `Set-Cookie: bar=baz` — по-русски — «запомни, foo — 123, а bar — baz».
3. При следующем обращении клиент, если он решил запомнить, говорит серверу «Cookie: bar=baz; foo=123».

Упрощенно, не затрагивая тонкости — все, вот все, что представляют собой cookies.

Сессии — более эфемерное понятие, которое не привязано к какой-то конкретной реальной технологии. Это просто некая методика, которая позволяет отличить одного клиента от другого, и, как правило, где-то хранить связанные с каждым клиентом данные.

Как правило, сессии реализуются используя cookies и идентификаторы сессий. Т.е. сервер со своей стороны создает уникальный идентификатор, например, «1a2b3c» (`session_id` про который вы спрашивали), а клиента просит его запомнить. Обычно — при помощи cookies, говоря что-то в духе `Set-Cookie: PHPSESSID=1a2b3c` (где «`PHPSESSID`» — **имя сессии**, обычно, оно только одно, вести параллельно несколько сессий нужно редко). Со своей стороны сервер где-то (зависит от реализации, иногда это файл, например, `/tmp/1a2b3c`, иногда запись в БД, иногда еще что-то) хранит различные данные, которые ему приказано связывать с этой сессией. Например, имя пользователя.

Еще момент. Сами по себе сессии не дают авторизации — это только хранилище. Авторизацию делаете Вы сами (или фреймворк, которым Вы воспользовались). Вы пользуетесь тем, что данные, связанные с сессией (в типичной PHP'шной реализации) хранятся на сервере, и клиент их там подменить не может.

Соответственно, когда клиент аутентифицируется — в сессии сохраняется кто он. Обратно — если у нас в сессии записано кто он — можно этому верить. Вся дальнейшая логика — пускать ли клиента, давать ли ему какие-то возможности и т.д. — на Вас.

`Secure` ("безопасные") и `HttpOnly` куки

"Безопасные" (secure) куки отсылаются на сервер только тогда, когда запрос отправляется по протоколу SSL и HTTPS. Однако важные данные никогда не следует передавать или хранить в куках, поскольку сам их механизм весьма уязвим в отношении безопасности, а флаг `secure` никакого дополнительного шифрования или средств защиты не обеспечивает. Начиная с Chrome 52 и Firefox 52, незащищённые сайты (http:) не могут создавать куки с флагом `Secure`.

Куки HTTPonly не доступны из JavaScript через свойства `Document.cookie` API, что помогает избежать межсайтового скриптинга (XSS (en-US)). Устанавливайте этот флаг для тех кук, к которым не требуется обращаться через JavaScript. В частности, если

куки используются только для поддержки сеанса, то в JavaScript они не нужны, так что в этом случае следует устанавливать флаг `HttpOnly`.

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly
```

Способы предотвращения атак, использующих куки:

- Используйте атрибут `HttpOnly` для предотвращения доступа к кукам из JavaScript.
- Куки, которые используются для хранения чувствительной информации, такой как аутентификационный токен, должны иметь короткое время жизни и атрибут `SameSite`, установленный в `Strict` или `Lax`. Для того чтобы узнать больше, смотрите раздел SameSite. В браузерах с поддержкой SameSite это гарантирует предотвращение отправки кук аутентификации с межсайтовыми запросами, фактически такие запросы с точки зрения бэкенда становятся неаутентифицированными.

https://proglib.io/p/veb-autentifikaciya-fayly-cookies-ili-tokeny-2021-08-14

Cookie Based Authentication

Pros

1. *HttpOnly Flag*: Session cookies can be created with the *HttpOnly* flag which secures the cookies from malicious JavaScript (XSS-Cross-Site Scripting).
2. *Secure flag*: Session cookies can be created with *Secure* flag that prevents the cookies transmission over an unencrypted channel.

Cons

1. *CSRF*: Cookies are vulnerable/susceptible to CSRF attacks since the third party cookies are sent by default to the third-party domain that causes the exploitation of CSRF vulnerability.
2. *Performance and Scalability*: Cookie based authentication is a stateful authentication such that server has to store the cookies in a file/DB in order to maintain the state of all the users. As the user base increases the

backend server has to maintain a separate system so as to store session cookies.

Token Based Authentication:

Pros

1. *Performance and Scalability*: Tokens contains the metadata and its signed value(for tamper protection). They are *self-contained* and hence there is no need of maintaining state at the server. This improves the performance and thus scalability when the expansion is required.
2. *CSRF*: Unlike cookie-based authentication, token-based authentication is not susceptible to Cross-Site Request Forgery since the tokens are not sent to third party web applications by default.

Cons

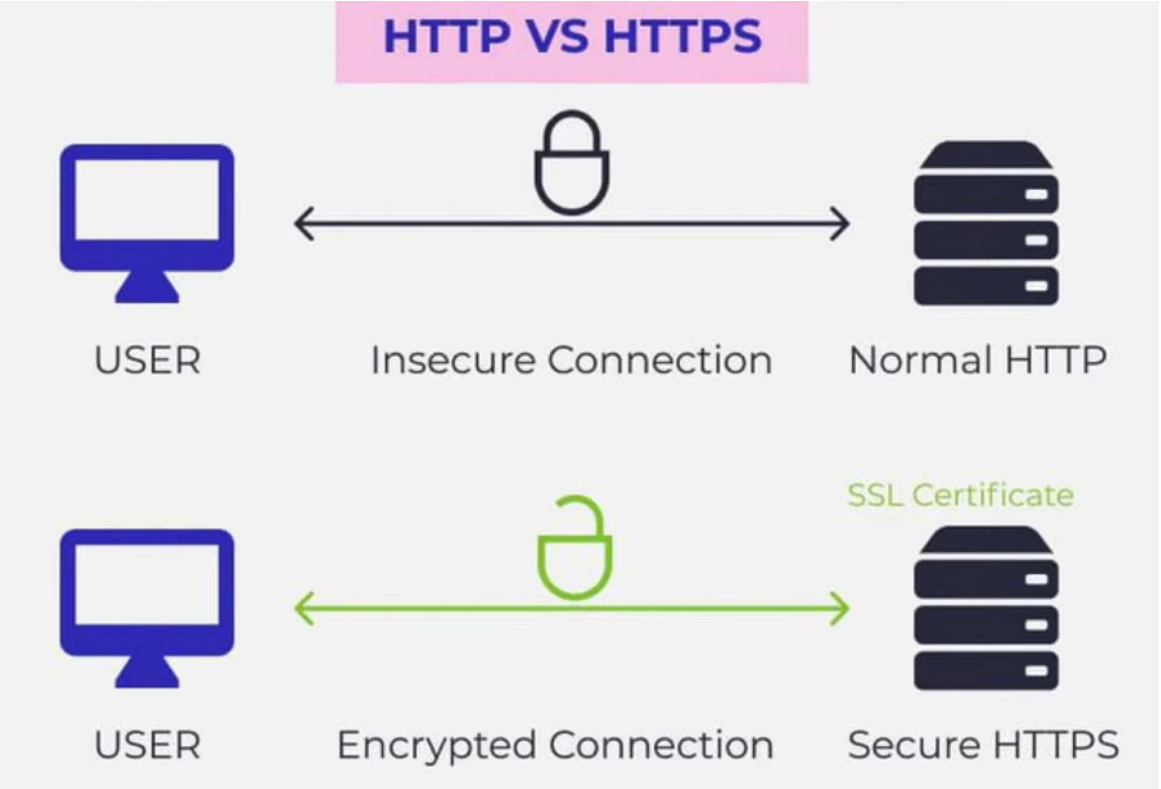
1. *XSS*: Since the session tokens are stored in the local data storage of the browser and it is accessible to the JS of the same domain. Hence there is no option to secure session identifier from XSS attacks unlike HTTPOnly security flag which is available in the cookie-based authentication.

▼ Разница между `HTTP` и `HTTPS`?

HTTP - не зашифрованный запрос.

HTTPS- зашифрованный запрос.

Задача HTTPS проверить подлинность сайта и установить безопасное соединение шифруя запрос, в то время как HTTP просто устанавливает соединение не шифруя запрос.



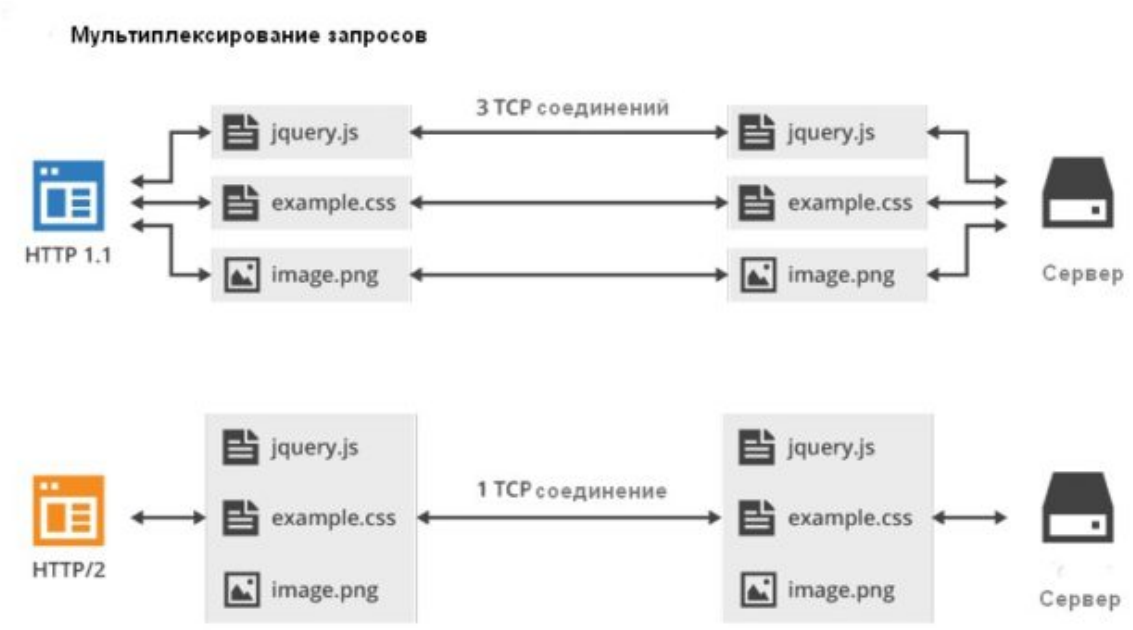
▼ Разница между `HTTP/1` и `HTTP/2`?

HTTP/2 обеспечивает более эффективное использование сети и повышенную производительность в сравнеии с HTTP/1

Это достигается с помощью:

- **Мультиплексирование** - запросы и ответы могут быть разделены на фрагмены или кадры, и отправлять параллельно по одному соединению, что снизижает время ожидания на окончание завершения предыдущего запроса и эффективнее использовать сеть. Один из основных плюсов второй версии HTTP. В предыдущей версии для одного запроса была необходима установка отдельного TCP-соединения. И чем больше было запросов, тем медленней работал браузер. Благодаря мультиплексированию браузер отправляет сразу несколько запросов через одно TCP-соединение. Современные браузеры задействуют ограниченное число TCP-соединений, что не позволяет им быстро загружать «тяжелые» страницы. А http/2, где внедрена технология мультиплексирования, дает возможность загружать большое количество статического контента одновременно, существенно повышая производительность.
- **Компрессия заголовков** - использует алгоритм сжатия что позволяет сократить объем передаваемых данных и снизить нагрузку на сеть

- **server push** - серверные уведомления, сервер может активно отправлять ресурсы клиенту которые клинет может потенциально запросить, даже до того как этот запрос выполнен явно.
- **Приоретизация** - позволяет установить приоритеты для различных запросов. что помогает оптимизировать поток данных и улучшает производительность при обработке параллельных запросов. Суть заключается в том, что браузер запрашивает у сервера отдельную загрузку некоторых элементов контента, к примеру, сначала скриптов JavaScript, а затем изображений. Приоритезация в протоколе не обязательна, но рекомендована, потому что мультиплексирование не способно полноценно работать без приоритетов. Чревато это медленной загрузкой, даже хуже, чем в http 1.1 версии. Ресурсы вторичного приоритета, занимая полосу, негативно скажутся на производительности.
- **Безопасность** - обязательно требует шифрование с помощью протокола TLS что обеспечивает повышенную безопасность.

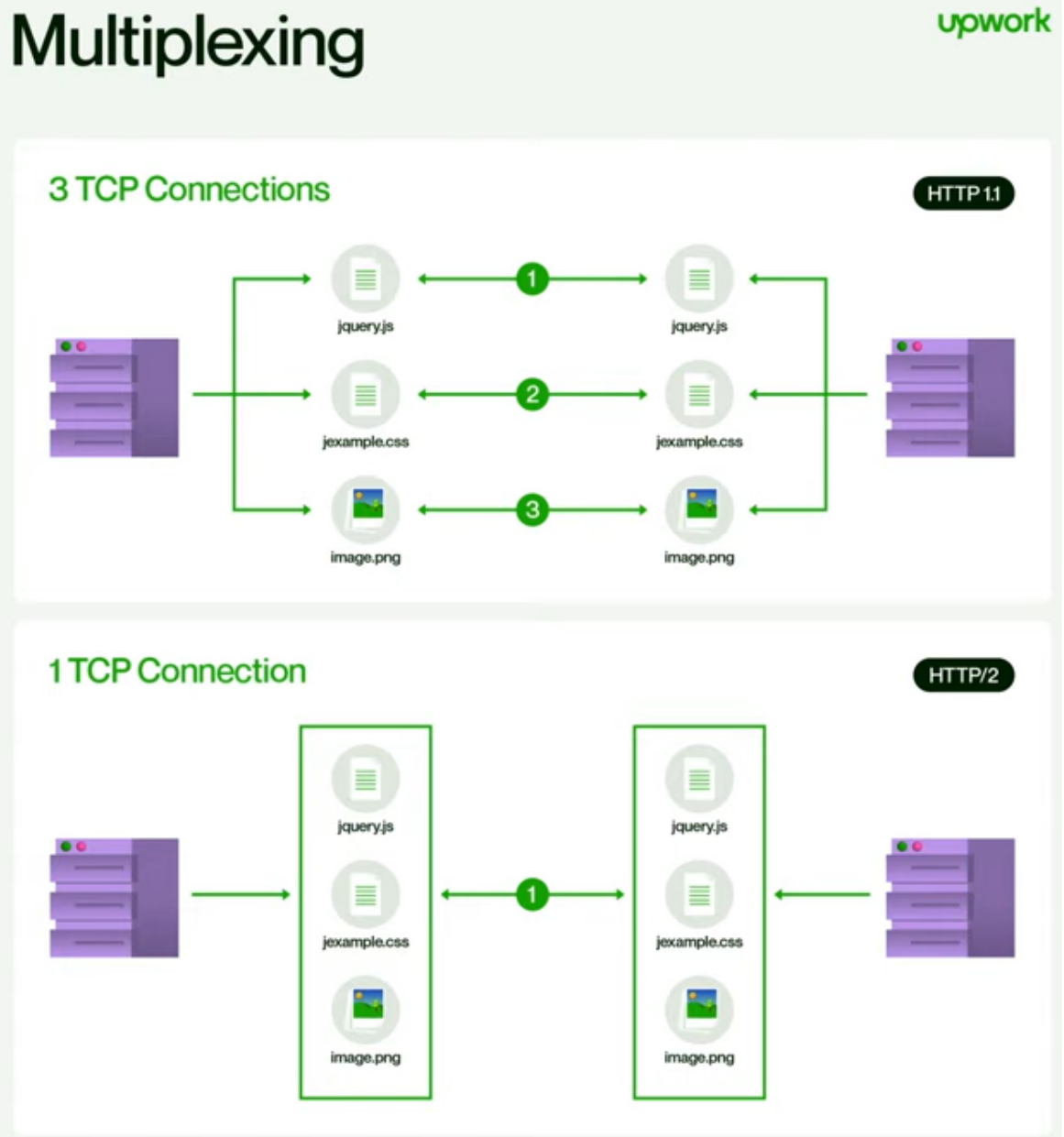


▼ Как работает мультиплексирование в `HTTP/2` ?

Позволяет отправлять несколько запросов и ответов параллельно по одному соединению. Это достигается путем разделения запросов на маленькие фрагменты называемые кадрами и передачей их независимо друг от друга.

Выполняется в 4 шага:

- Установление соединения
- Разделение на кадры, которые могут передаваться в любом порядке, каждый кадр сожержит определенную часть данных: заголовков, тела запроса или ответа
- Идентификация кадров - каждый кадр имеет уникальный идентификатор который позволяет получателю собрать запросы и ответы из кадров в правильном порядке
- Параллельная передача - клиент и сервер могут параллельно отправлять и принимать несколько кадров, увеличивая тем самым пропускную способность и эффективно использовать сетевые ресурсы, что полезно при загрузке больших ресурсов и выполнении множества запросов одновременно.



▼ Что такое “трехстороннее рукопожатие” (Triple handshake)?

Чтобы установить надежное соединение, TCP использует процесс, называемый термином “трехстороннее рукопожатие” (TCP three-way/triple handshake). Установленное соединение будет полнодуплексным, то есть оба канала могут передавать информацию одновременно, а также они синхронизируют (SYN) и подтверждают (ACK) друг друга. Обмен выполняется следующим образом:

1. Клиент отправляет сегмент с установленным флагом SYN. При этом сегменту присваивается произвольный порядковый номер (sequence number) в интервале от 1 до 232 (т.н. initial sequence number),

относительно которого будет вестись дальнейший отсчет последовательности сегментов в соединении.

2. Сервер получает запрос и отправляет ответный сегмент с одновременно установленными флагами SYN+ACK, при этом записывает в поле «номер подтверждения» (acknowledgement number), полученный порядковый номер, увеличенный на 1 (что подтверждает получение первого сегмента), а также устанавливает свой порядковый номер, который, как и в SYN-сегменте, выбирается произвольно.
3. После получения клиентом сегмента с флагами SYN+ACK соединение считается установленным, клиент, в свою очередь, отправляет в ответ сегмент с флагом ACK, обновленными номерами последовательности, и не содержащий полезной нагрузки.
4. Начинается передача данных.

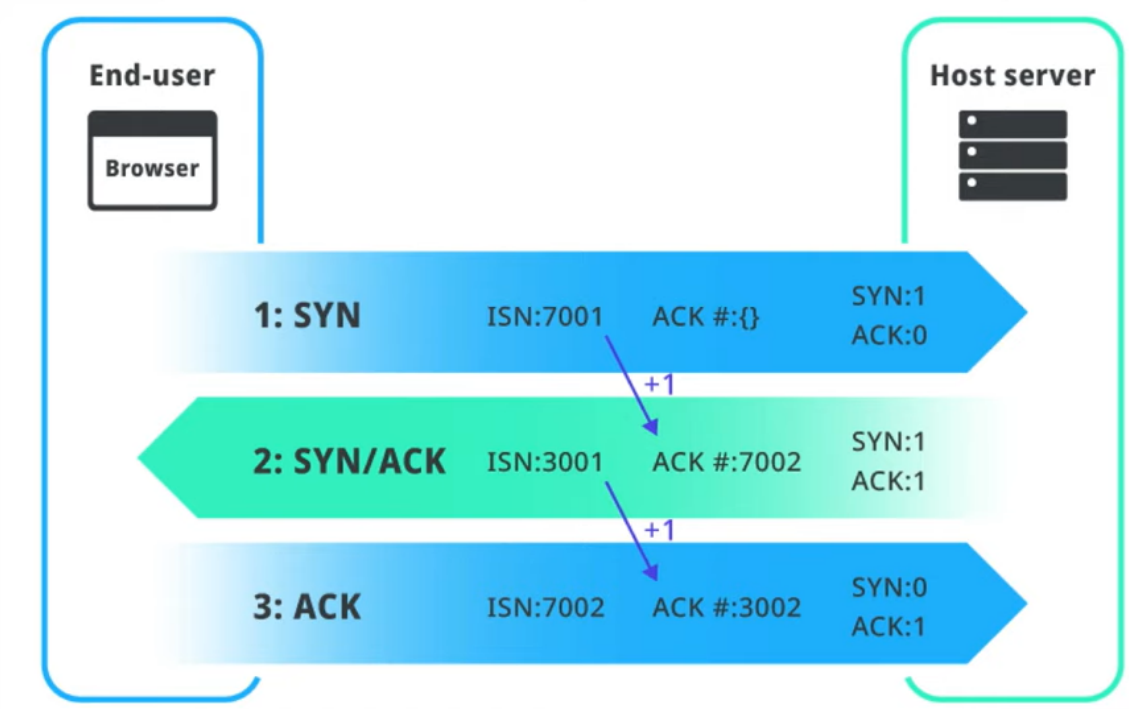
Принцип работы TCP рукопожатия

В процессе передачи данных клиент и сервер обмениваются сегментами с установленными флагами ACK или PSH+ASK, поочередно подтверждая число полученных пакетов путем увеличения счетчиков порядкового номера (sequence number) и подтверждения (acknowledgement number).

Этапы 1, 2 определяют параметр подключения (порядковый номер) для одного направления, и подтверждают его. Этапы 2, 3 определяют параметр подключения для другого направления, и также подтверждают его. С их помощью устанавливается полнодуплексная связь.

Интересно, что начальные параметры SYN выбираются случайным образом при установлении соединений между клиентом и сервером.

В заключение стоит добавить, что проверка номеров последовательности приходящих пакетов и принципов, используемых при 3-way handshake, позволяют реализовать простейшие методы фильтрации DDoS-атак.



▼ Разница между `PUT` и `POST` запросами?

PUT - приводят к замене данных т.е. замена целевого ресурса на данные передаваемые в запросе, его можно использовать для замены содержимого целевого ресурса или для создания ного ресурса. Можно выыполнять многократно полчучая один и тот де результат не изменяя входные данные.

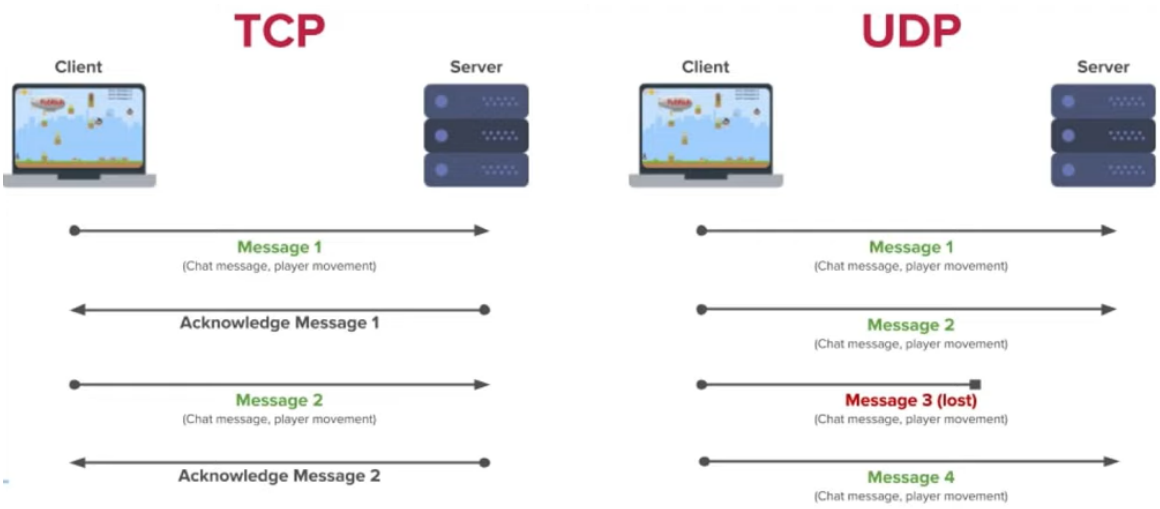
POST - добавление новых данных т.е. создание новых ресурсов, выгрузки новых данных на сервер, отправка форм. На каждый запрос создаем новые данные т.е. несколько запросов = несколько новых объектов на сервере.

HTTP Method	CRUD	Action
PUT	Update	Updates an existing record
POST	Create	Creates a new record

▼ Разница между протоколами `TCP` и `UDP` ?

TCP - это основной транспортный протокол интернета, который обеспечивает доставку данных через соединение, предварительно установленное между двумя компьютерами. Соединение работает на протяжении всего сеанса связи. Гарантирует целостность и порядок приходящих данных в ущерб скорости их передачи, это важно при обмене файлами.

UDP - предназначен для быстрой передачи порции данных без гарантии доставки и без предварительной установки соединения. Используется когда скорость выше гарантии доставки и порядка получения данных, например при передаче потокового видео/аудио.



▼ Что такое `WebSocket` ? В чем принцип его работы?

Подробнее тут

WebSocket - это протокол, который обеспечивает возможность обмена данными между браузером и сервером через постоянное соединение. Данные передаются по этому соединению в обоих направлениях в виде пакетов, без разрыва соединения и дополнительных HTTP запросов.

Чтобы открыть соединение нужно создать объект new WebSocket указав в `url` специальный протокол `ws` и `wss`. Как только объект вебсокета создан, на его события можно подписаться и прослушивать их.

WebSocket сам по себе не содержит такие функции, как переподключение при обрыве соединения, аутентификацию пользователей и другие механизмы высокого уровня. Для этого есть клиентские и серверные библиотеки, а также можно реализовать это вручную.

Методы:

- `socket.send(data)`,
- `socket.close([code], [reason])`.

События:

- `open`,

- `message`,
- `error`,
- `close`.

```
const socket = new WebSocket("ws://just-test.com");

socket.onopen = function() {
  alert("[open] Connected!");
  socket.send("Hello world!");
};

socket.onmessage = function(e) {
  alert(`[message] Data is received from server: ${e.data}`);
};

socket.onclose = function(e) {
  event.wasClean
    ? alert(`[close] Connection closed cleanly, code=${e.code} reason=${e.reason}`);
    : alert('[close] Connection interrupted');
};

socket.onerror = function(e) {
  alert(`[error] ${e.message}`);
};
```

▼ Разница между Long-Polling, Websockets и Server-Sent Events?

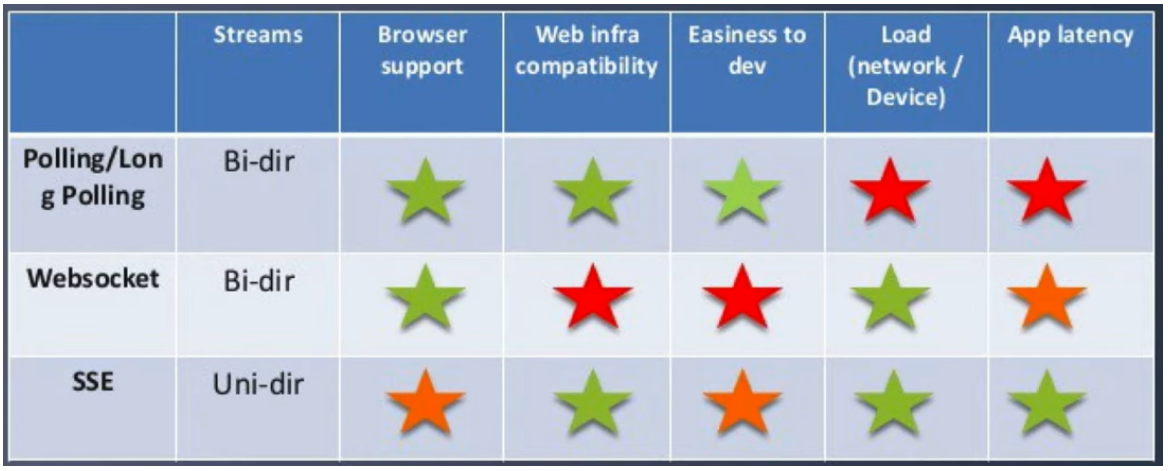
Начинается все одинково, запрос страницы по HTTP.

Polling - отправление запросов раз в пару секунд.

Long polling - просто отправляет постоянные запросы и когда получает ответ снова отправляет запрос или когда кончится время ожидания тоже отправляет запрос.

Websockets - самый мощный инструмент, который обеспечивает постоянное соединение между клиентом и сервисом.

Service-Sent Events - обеспечивает лишь одностороннее соединение.



Polling

По сути, опрос - это техника запроса информации с сервера через регулярные промежутки времени. Такое соединение происходит по протоколу HTTP.

Существует два типа опроса:

Short Polling

При коротком опросе клиент запрашивает информацию у сервера. Сервер обрабатывает запрос. Если данные для запроса доступны, сервер отвечает на запрос требуемой информацией. Если же на сервере нет данных для клиента, то сервер возвращает пустой ответ. В обоих случаях после получения ответа соединение будет закрыто. Клиенты продолжают отправлять новые запросы даже после того, как сервер посылает пустые ответы. Такой механизм увеличивает сетевые затраты сервера.

Long polling

При длинном опросе клиенты могут запрашивать у сервера информацию с расчетом на то, что сервер может ответить не сразу. Получив запрос, сервер, если у него нет свежих данных для клиента, вместо того чтобы вернуть пустой ответ, оставляет запрос открытым и ожидает поступления данных. Когда сервер получает новые данные, он сразу же доставляет ответ клиенту, завершая открытый запрос. После получения ответа от сервера клиент может отправить еще один запрос на получение новых данных. Длительный опрос позволяет снизить затраты за счет уменьшения количества пустых ответов.

WebSocket

WebSocket - это протокол, обеспечивающий двусторонние (bi-directional) каналы связи через одно TCP-соединение. Websocket обеспечивает

постоянное соединение между клиентом и сервером, позволяя обеим сторонам начать передачу данных в любой момент. WebSocket handshake - это процедура, с помощью которой клиент создает WebSocket-соединение. Если операция проходит успешно, то сервер и клиент могут отправлять и получать данные в любой момент времени. В основном используется в веб-приложениях реального времени, таких как WhatsApp, Uber.

Server-sent event (SSE)
В отличие от WebSockets, с помощью SSE мы не можем передавать запросы от клиента к серверу, поскольку это одностороннее соединение. Если нам требуется передача данных от сервера к клиенту в режиме, близком к реальному времени, или если сервер генерирует данные в цикле, SSE - идеальный выбор.

▼ Как работает `JSONP`?

Способ используемый для обхода политики ограничения доменов в браузера, т.к. AJAX запросы с текущей старницы к серверу находящемуся в другом домене запрещены. Это политика называется same origin policy.

https://learn.javascript.ru/ajax-jsonp

JSON - это широко используемый формат обмена данными, который прост, широко поддерживается и безопасен.

JSONP - Это технология для выполнения запроса к другому домену (через обычный XHR это невозможно).

JSONP - это метод для выполнения запросов из разных источников, который менее безопасен и официально не поддерживается JSON. JSONP часто используется для загрузки внешних данных на веб-сайт или веб-приложение, также нужно быть уверенным в безопасности поставщика данных.

Это технология для выполнения запроса к другому домену (через обчный XHR это не возможно).

В двух словах работает это следующим образом:

В head страницы добавляется новый тег script с src установленным в адрес запроса и параметры (например "`www.example.com/?id=1&jsonp=myCallback`").

Как вы понимаете этот скрипт будет загружен браузером несмотря на домен на котором мы находимся.

При этом ответная сторона (example.com) в тело этого скрипта вернет не просто JSON, а вызов javascript функции указанной в jsonp параметре (мы разумеется эту функцию должны заблаговременно объявить).

Пример ответа от example.com: `myCallback({ "user": "Rrooom", "message": "test" });`

Это не плохой способ обойти проблему кросс доменных запросов, но возможен только GET.

GET-параметр может называться по-разному. Вот так выглядит JSONP для Google карт и вариант названия нашей функции

```
http://maps.googleapis.com/maps/api/js?sensor=true&language=ru&callback=Test.method
```

А вот и метод нашего объекта

```
var Test = {
  method: function () {
    console.log(arguments);
    //Пришли аргументы от maps.googleapis.com
  }
};
```

То есть не обязательно должна быть функция в глобальном контексте, но метод обязан быть доступен из глобального контекста.

JSONP - **это костыль**, используемый в данный момент только за счет такой штуки как IE, который до 10-ой версии не поддерживает CORS.

Проблема состоит в том что политикой безопасности браузера не разрешается делать кросдоменные XHR запросы (тобиш AJAX в простонародьи). Учитывая что angular-based (да и вооообще любое приложение на клиенте) приложения должны получать данные с REST API, и это API может находиться на другом сервере, то вполне логично что нужно придумать какой-то способ получать эти данные и что бы это можно было делать не взирая на ограничения браузера.

Нормальные ребята для нормальных браузеров придумали и используют

CORS

(Cross-origin resource sharing), который стандартизирован, хорошо и няшно работает и легко прикручивается к проекту. Но если у вас заявлена поддержка IE9 или более старых версий, то там все это **работать не будет** и приходится опять ваять кастыли с jsonp.

▼ Что такое IndexedDB в браузере? Преимущества IndexedDB?

https://learn.javascript.ru/indexeddb#itogo

IndexedDB - это способ постоянного хранения данных внутри клиентского браузера, другими словами это NOSQL хранилище на стороне клиента. Что позволяет создавать веб-приложения с богатыми возможностями обращения к данным независимо от доступности сети, ваши приложения могут работать как онлайн, так и офлайн.

IndexedDB был разработан в 2010 году и является стандартом W3C. Он представляет собой объектную базу данных, которая хранится в браузере пользователя. IndexedDB использует JavaScript API для создания, чтения, обновления и удаления данных в локальной базе данных. Он поддерживается не всеми браузерами, но его поддержка растет.

IndexedDB - это низкоуровневый API, для хранения на клиенте значительного кол-ва структурированных данных. Мини база данных в браузере.

Объем до 50% объема жесткого диска.

Данные хранятся на стороне клиента, и получается что пользователь ограничен браузером в котором работает.

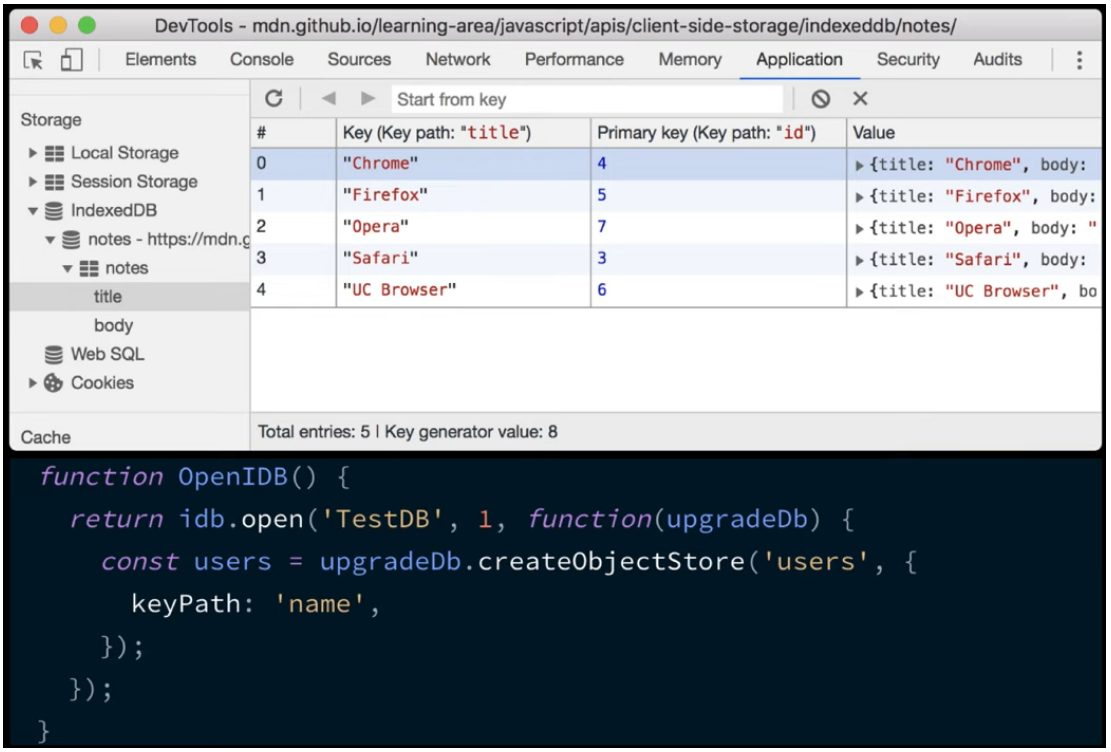
Используется часто для development разработки, если приложение не нужно публиковать в прод и достаточно того что оно будет работать с привязкой к браузеру и машине.

Используется для разработки расширений для браузера.

Преимущества:

- Могут обрабатывать более сложные структуры данных, строки, числа, объекты, даты, файлы
- Могу работать с разными базами данных и таблицами внутри каждой базы.
- Больший объем хранения.

- let openRequest = indexedDB.open(name, version);



Типичная схема работы с базой

Обычная последовательность шагов при работе с IndexedDB :

1. Открыть базу данных.
2. Создать хранилище объектов в базе данных, над которой будут выполняться наши операции.
3. Запустить транзакцию и выдать запрос на выполнение какой-либо операции с базой данных, например, добавление или извлечение данных.
4. Ждать завершения операции, обрабатывая событие DOM, на которое должен быть установлен наш обработчик.
5. Сделать что-то с результатами (которые могут быть найдены в возвращаемом по нашему запросу объекте).

▼ Что такое Service Workers?

Service Worker - фактически выполняет роль прокси-сервера, находящегося между веб приложением и браузером, а также сетью.

Service Worker - если пропадает сетевое соединение, то сервис воркер позволяет приложению продолжить работать с сохраненными ранее в

кэше файлами

По сути действует как прокси сервер который находится между веб приложениями, браузером и сетью если они доступны.

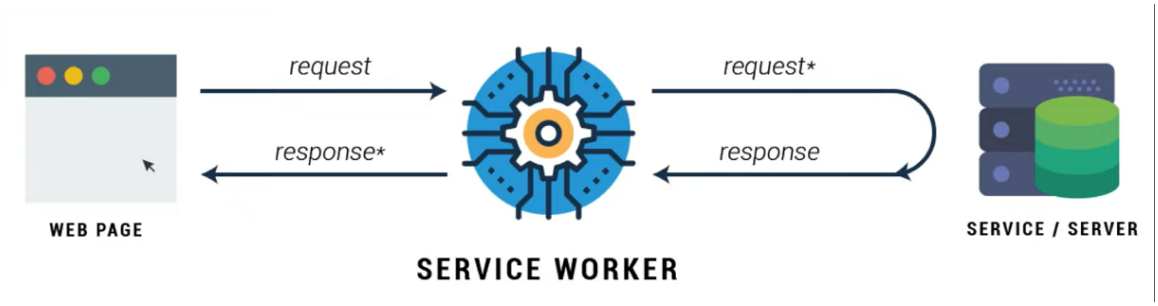
Можно сказать это сценарий, который запускается браузером в фоновом режиме, он никак не связан с веб страницей или DOM. Дает такие функции как обработка сетевых запросов, пуш уведомления и фоновая синхронизация.

Предназначены для эффективного взаимодействия в автономном режиме, перехвата сетевых запросов и принятия соответствующих мер в зависимости от доступности сети и налчиия обновенных данных на сервере. Грубо говоря дает возможность работать оффлайн. Являются посредником между клиентом и сервером, и можно перехватывать все запросы.

Запускается в контексте worker и не имеет доступа к DOM,и работает в отдельном потоке от основного потока JS на котором работает приложение, поэтому они являются неблокирующими и полностью асинхронными.

Работают только по HTTPS из соображений безопасности.

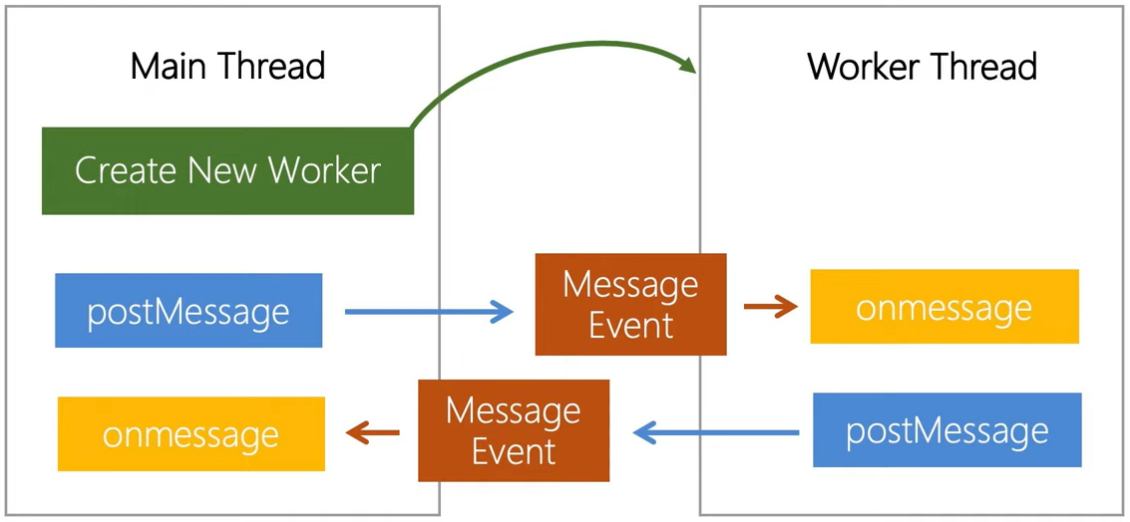
Он позволяет (кроме прочего) описывать корректное поведение веб-приложения в режиме офлайн, перехватывать запросы сети и принимать соответствующие меры, основываясь на доступности сети, и обновлять данные, находящиеся на сервере при доступе к нему. Также они имеют доступ к push-уведомлениям и API для фоновой синхронизации.



▼ Что такое Web Workers?

Web worker — это скрипты общего назначения позволяющие разгрузить работу основного потока. Позволяет веб-страницам выполнять задачи в фоновом режиме независимо от основного потока и пользовательского интерфейса веб-сайта.

Это изолированная среда, которая изолирована от объекта window, объекта document, прямого доступа в Интернет и лучше всего подходит для длительных или сложных вычислительных задач.



Для длительных тяжёлых вычислений, которые не должны блокировать событийный цикл, мы можем использовать Web Workers.

Это способ исполнить код в другом, параллельном потоке.

Web Workers могут обмениваться сообщениями с основным процессом, но они имеют свои переменные и свой событийный цикл.

Web Workers не имеют доступа к DOM, поэтому основное их применение – вычисления. Они позволяют задействовать несколько ядер процессора одновременно.

Worker - это объект, создаваемый конструктором (например, `Worker()`) и запускающий именной JavaScript файл — этот файл содержит код, который будет выполнен в потоке Worker'a; объекты же Workers запускаются в другом глобальном контексте, отличающемся от текущего, - `window`.

Вы можете запускать любой код внутри потока worker-a, за некоторыми исключениями. Например, вы не можете прямо манипулировать DOM внутри worker-a, или использовать некоторые методы по умолчанию и свойства объекта `window`. Но вы можете использовать большой набор опций, доступный под `Window`, включая WebSockets, и механизмы хранения данных, таких как IndexedDB и относящихся только к Firefox OS Data Store API.

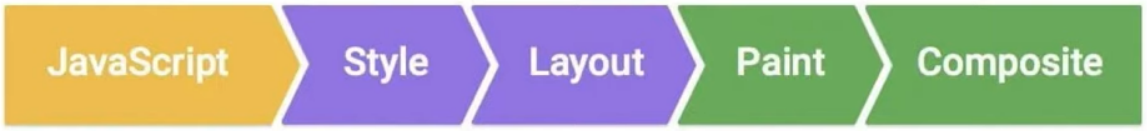
▼ Что такое Web Worklet?

Легкие и специфичные воркеры, позволяют подключаться к различным частям процесса рендеринга браузером в критическом пути рендеринга, а конкретно на этапах style, layout, paint, composite.

Интерфейс Worklet представляет собой облегченную версию Web Workers и предоставляет разработчикам доступ к низкоуровневым частям критического пути рендеринга.

С помощью Worklets можно запускать код JavaScript и WebAssembly для рендеринга графики или обработки звука, где требуется высокая производительность.

Можно вынести ресурсозатратные операции рендеринга в отдельный поток, уменьшив нагрузку на страницу и тем самым увелчитить скорость её отрисовки.




```
/* myWorklet.js */
registerPaint('myGradient', class {
  paint(ctx, size, properties) {
    let gradient = ctx.createLinearGradient(0, 0, 0, size.height / 3); gradient.addColorStop(0, "black");
    gradient.addColorStop(0.7, "rgb(210, 210, 210)");
    gradient.addColorStop(0.8, "rgb(230, 230, 230)");
    gradient.addColorStop(1, "white");
    ctx.fillStyle = gradient;
    ctx.fillRect(0, 0, size.width, size.height / 3);
  }
});

/* main.js */
CSS.paintWorklet.addModule('myWorklet.js');
```

Worklet types

Ворклеты ограничены конкретными случаями использования; они не могут применяться для произвольных вычислений, как Web Workers. Интерфейс Worklet абстрагирует свойства и методы, общие для всех

видов Worklet, и не может быть создан напрямую. Вместо этого можно использовать один из следующих классов:

Name	Description	Location	Specification
`AudioWorklet`	For audio processing with custom AudioNodes.	Web Audio render thread	Web Audio API
`AnimationWorklet`	For creating scroll-linked and other high performance procedural animations.	Compositor thread	CSS Animation Worklet API
`LayoutWorklet`	For defining the positioning and dimensions of custom elements.		CSS Layout API

▼ Что такое `SSL` / `TLS` ? Зачем они используются в веб-разработке?

https://aws.amazon.com/ru/what-is/ssl-certificate/

TLS на самом деле является всего лишь более поздней версией SSL. Он устраняет некоторые уязвимости в более ранних протоколах SSL.

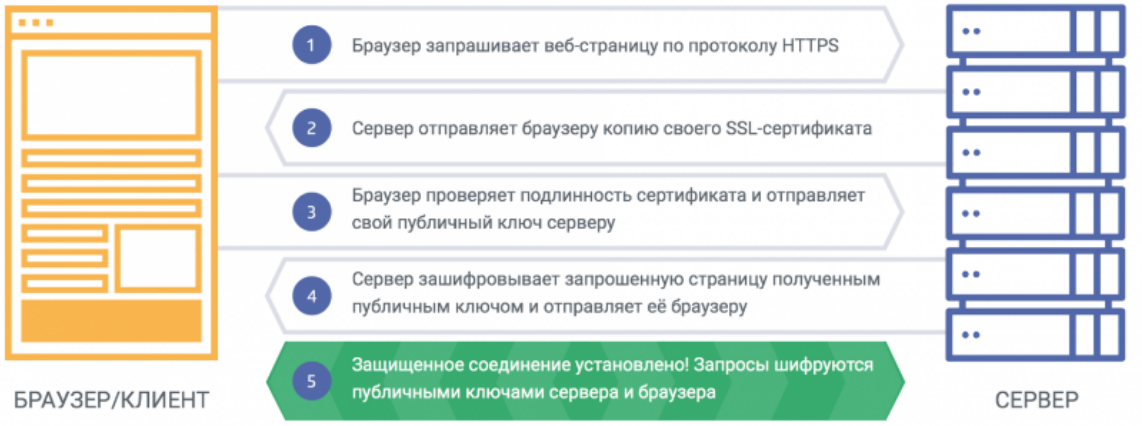
SSL расшифровывается как Secure Socket Layer, что означает «уровень защищенных сокетов». TLS же обозначается как Transport Layer Security, «безопасность транспортного уровня». По своей сути обе технологии занимаются одним делом – защитой пользовательской информации от злоумышленников.

Их отличие состоит лишь только в том, что TLS основан на уже действующей спецификации SSL 3.0. А сам SSL уже давно устарел, разработчики редко его используют как единственную защиту. Чаще всего можно увидеть связку двух сертификатов SSL/TLS. Такая поддержка обеспечивает работу как с новыми, так и со старыми устройствами.

Без подключения SSL/TLS контакт между пользователем и веб-сайтом происходит через канал HTTP. А это означает, что вся передаваемая информация находится в открытом виде: доступ к данным лежит на поверхности. То есть, когда происходит связь между пользователем и сайтом, например, при оплате билетов на самолет, вся информация, включая паспортные данные, может быть получена злоумышленником. Такое происходит, если на сайте не используются сертификаты защиты.

При подключении SSL/TLS, пользователь устанавливает соединение с веб-сервером HTTPS, который защищает все конфиденциальные данные при передаче. Кроме того, срабатывает привязка криптографического ключа к передаваемой информации и выполняется шифровка данных, которую никто не сможет перехватить.

SSL/TLS используют ассиметричное шифрование для аутентификации пользователя и симметричное для сохранения целостности личной информации.



▼ Механизм установки сеанса между клиентом и сервером?

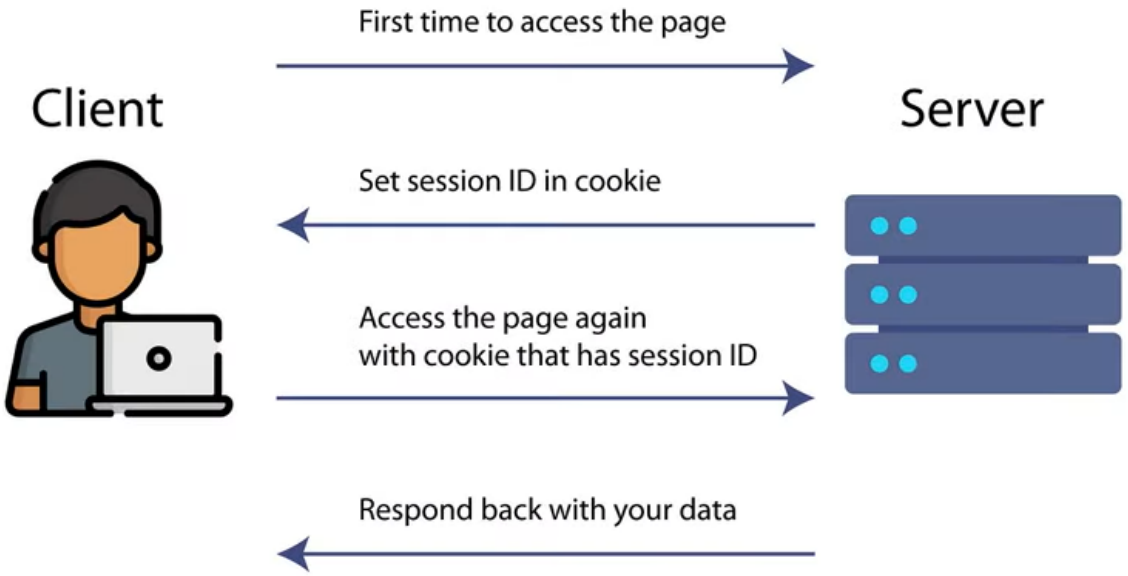
Клиент отправляет запрос на сервер

Сервер получает запрос и генерирует ID сеанса, затем отправляет пользователю ID

Пользователь сохраняет ID. Это могут быть куки, заголовки запроса или параметры запроса.

Далее на каждый запрос ID передается на сервер для идентификации клиента и поддерживать состояние сессии, сохраняя состояние, например информация о предыдущих действиях, аутентификация, товары в корзине

По окончании работы с приложением клиент может завершить сессию отправив соответствующий запрос, либо сессия автоматически завершится по истечении определенного времени.



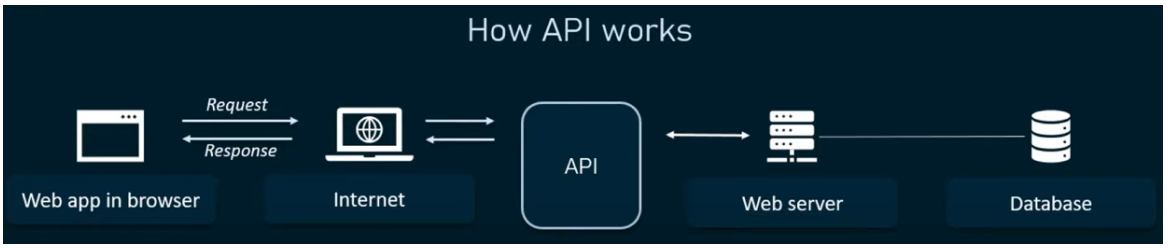
▼ Что Такое API?

API (Application Programming Interface) - набор методов с помощью которых интерфейс может взаимодействовать с бекендом, с базой данных. Описание способов взаимодействия между программами. Как они могут общаться и передавать данные друг другу.

В целом это способ общения программ друг с другом с помощью подготовленных эндпоинтов, в том числе бекенды с другими бекендами.

Это взаимодействие осуществляется с помощью стека CRUD операций.

Для каждой из этих операций существует свой запрос, который идет на определенный в API url адрес и как-то манипулирует данными.



▼ Что такое CDN?

CDN (Content Delivery Network) - это группа серверов расположенных во многих местах, эти серверы хранят дублированные копии данных, чтобы ресурсы могли выполнять запросы за ними, в зависимости от того, какие серверы находятся ближе всего к соответствующим конечным пользователям.

В результате, если сайт использует CDN ресурсы, то при обращении к нему, эти ресурсы будут возвращены пользователю от ближайшего сервера.

▼ Что такое IP-адрес?

IP-адрес – это уникальный адрес, идентифицирующий устройство в интернете или локальной сети. IP означает «Интернет-протокол» – набор правил, регулирующих формат данных, отправляемых через интернет или локальную сеть. Построен на стеке протоколов TCP/IP.

По сути, IP-адрес – это идентификатор, позволяющий передавать информацию между устройствами в сети: он содержит информацию о местоположении устройства и обеспечивает его доступность для связи.

IP-адрес – это строка чисел, разделенных точками. IP-адреса представляют собой набор из четырех чисел, например, 192.158.1.38. Каждое число в этом наборе принадлежит интервалу от 0 до 255.

▼ Разница между host и domain?

Host - это сервер, который содержит сайт, у него есть уникальный IP адрес в среде сервисов TCP/IP.

Domain - это адрес сайта, который имеет свое уникальное символьное имя в системе доменных имен, т.е. доменное имя.

Получается что когда в браузер вводится доменное имя, то под капотом идет запрос к IP адресу хоста ассоциированному с этим именем.

▼ Разница между URI и URL?

Подробнее тут

- **URI (Uniform Resource Identifier)** - имя и адрес ресурса в сети, включает в себя URL и URN.
- **URL (Uniform Resource Location)** - адрес ресурса в сети, определяет местонахождение и способ обращения к нему.
- **URN (Uniform Resource Name)** - имя ресурса в сети, определяет только название ресурса, но не говорит как к нему подключиться.

Например:

- URI – https://wiki.merionet.ru/images/vse-chto-vam-nuzhno-znat-pro-devops/1.png
- URL - https://wiki.merionet.ru
- URN - images/vse-chto-vam-nuzhno-znat-pro-devops/1.png

▼ Почему очищать кэш важно? Как это можно сделать?

Почему - если веб сайт был изменен или обновлен, а кэш пользователя все еще ссылается на старые файлы, это может оставить их со старой функциональностью, либо сломать веб сайт если кешированные CSS и JS ссылаются на элементы которые удалены или переименованы.

Как это сделать - присвоить новым файлам другое имя, отличное от прошлого. Например сделать версионирование файлов, добавляя в конец имени файла новое число.

Кэширование

Application cache

https://developer.mozilla.org/ru/docs/Web/HTTP/Caching

https://www.internet-technologies.ru/articles/povyshenie-proizvoditelnosti-prilozheniy-s-pomoschyu-http-zagolovkov-keshirovaniya.html

Различные виды кеширования

Техника кеширования заключается в сохранении копии полученного ресурса для возврата этой копии в ответ на дальнейшие запросы. Запрос на ресурс, уже имеющийся в веб-кеше, перехватывается, и вместо обращения к исходному серверу выполняется загрузка копии из кеша. Таким образом снижается нагрузка на сервер, которому не приходится самому обслуживать всех клиентов, и повышается производительность — кеш ближе к клиенту и ресурс передаётся быстрее. Кеширование является основным источником повышения производительности веб-сайтов. Однако, кеш надо правильно сконфигурировать: ресурсы редко остаются неизменными, так что копию требуется хранить только до того момента, как ресурс изменился, но не дольше.

Приватный (private) кеш браузера

Приватный кеш предназначен для отдельного пользователя. Вы, возможно, уже видели параметры кеширования в настройках своего браузера. Кеш браузера содержит все документы, загруженные пользователем по HTTP. Он используется для доступа к ранее загруженным страницам при навигации назад/вперёд, позволяет сохранять страницы, или просматривать их код, не обращаясь лишний раз к серверу. Кроме того, кеш полезен при отключении от сети.

Общий (shared) прокси-кеш

Кеш совместного использования — это кеш, который сохраняет ответы, чтобы их потом могли использовать разные пользователи. Например, в локальной сети вашего провайдера или компании, может быть установлен прокси, обслуживающий множество пользователей, чтобы можно было повторно использовать популярные ресурсы, сокращая тем самым сетевой трафик и время ожидания.

Основные заголовки

1. cache-control

Самый главный из всех. Обычно вы задаёте в строке его параметры, нечто вроде:

```
cache-control: private, max-age=0, no-cache
```

Эти настройки называются директивы ответа кэша, и они бывают следующие:

```
private | public
```

Сообщает, не является ли контент предназначенным для конкретного пользователя. Если это так, кэшировать его не нужно.

`no-cache`

Сама по себе директива говорит, что этот запрос нужно каждый раз делать заново. Обычно используется заголовок etag, о котором ниже. Веселье начинается, когда вы задаёте имя поля после этой директивы. Тогда кэширующие сервера понимают, что ответ можно кешировать, но при этом надо удалять заданные поля. Это, например, полезно для правильной работы куков. Однако, некоторые старые программы не умеют работать с этим трюком.

`no-store`

Сообщает, что этот ответ не нужно хранить. Удивительно, но факт. Если кэш работает по правилам, он убедится, что никакая из частей запроса не будет храниться. Это нужно для того, чтобы обезопасить всякую чувствительную информацию.

`max-age`

Обычно время жизни ресурса задаётся через expires, но если вам надо быть более конкретным, можно задать max-age в секундах. И эта директива имеет преимущество над expires.

`s-maxage`

Немного похоже на предыдущую, однако s здесь означает shared cache, и нужна для CDN. Эта директива имеет преимущество над max-age и expires, когда речь идёт о CDN-серверах.

`must-revalidate`

Говорит, что каждый запрос нужно делать заново, и ни при каких условиях не предоставлять пользователю закешированный контент. Имеет преимущество над всеми другими директивами, которые разрешают кэширование. В основном используется в некоторых особенных протоколах (к примеру, денежные переводы).

`no-transform`

Некоторые прокси умеют сжимать и конвертировать контент для ускорения работы. Эта директива запрещает подобное поведение.

`proxy-revalidate`

Примерно то же, что must-revalidate, но для промежуточных CDN-серверов. Почему её не назвали s-mustrevalidate? Кто его знает. Смысл в

том, что проверять, не обновился ли контент, нужно для каждого нового пользователя только один раз.

2. expires

Изначально это был стандартный метод определения того, когда устаревает ресурс. Сегодня max-age и s-maxage имеют над ним преимущество, но всегда полезно задавать этот заголовок в целях обратной совместимости.

Задав дату, отстоящую более, чем на год, вы нарушите спецификацию заголовка.

3. etag

Сокращение от entity-tag. Это уникальный идентификатор запрашиваемого ресурса – обычно, некий хэш его содержимого, или хэш времени его обновления. В общем, способ клиента запросить у CDN «дай мне ресурс X, если у него etag отличается от моего».

4. vary

Очень мощная штука. IE в прошлом обрабатывал его неправильно, да и сейчас не совсем корректно справляется. В какой-то момент даже Chrome с ним глючил. По сути, заголовок говорит системам кэширования, какие из заголовков можно использовать для определения того, допустимый ли у них в кэше лежит контент. Если рассматривать кэш как хранилище данных вида ключ-значение, то использование vary добавляет эти значения к ключам.

Часто можно встретить заголовок типа Accept-Encoding, который удостоверяется, что ваши ресурсы, сжатые gzip, будут приняты клиентом. Это здорово сберегает трафик. Кроме этого, настройка

`vary: User-Agent`

сделает ваш сайт более дружественным к SEO, если вы раздаёте разные HTML/CSS в зависимости от User-Agent. Google заметит эту штучку и Googlebot будет обрабатывать и ваш мобильный контент.

5. pragma

Довольно старая директива, которая умеет делать много чего, что, однако, уже обрабатывается при помощи более современных. Нам более всего интересна форма

`pragma: no-cache`

что в современных клиентах превращается в

`cache-control: no-cache`

- `Expires`: заголовок `Expires` очень простой, хотя и довольно ограниченный по объему. По сути, он устанавливает время в будущем, когда срок действия контента истекает. На этом этапе любые запросы на тот же контент должны быть возвращены на исходный сервер. Этот заголовок, вероятно, лучше всего использовать только как запасной вариант.
- `Cache-Control`: это более современная замена заголовка `Expires`. Он хорошо поддерживается и реализует гораздо более гибкий дизайн. Почти во всех случаях это предпочтительнее, чем `Expires`, но может не помешать установить оба значения. Мы обсудим особенности параметров, которые вы можете установить с помощью `Cache-Control`, немного позже.
- `Etag`: заголовок `Etag` используется при проверке кеша. Источник может предоставить уникальный `Etag` для элемента, когда он изначально обслуживает контент. Когда кэшу необходимо проверить содержимое, которое у него есть, по истечении срока его действия, он может отправить обратно `Etag`, которые он имеет для содержимого. Источник либо сообщит кешу, что содержимое такое же, либо отправит обновленное содержимое (с новым `Etag`).
- `Last-Modified`: этот заголовок указывает, когда в последний раз элемент был изменен. Это может использоваться как часть стратегии проверки для обеспечения свежего контента.
- `Content-Length`: заголовок `Content-Length`, хотя он специально не участвует в кэшировании, важно установить при определении политик кэширования. Определенное программное обеспечение откажется кэшировать содержимое, если оно не знает заранее размер содержимого, для которого ему потребуется резервировать место.
- `Vary`: кэш обычно использует запрошенный хост и путь к ресурсу в качестве ключа для хранения элемента кеша. Заголовок `Vary` может использоваться для указания кешам обращать внимание на дополнительный заголовок при принятии решения о том, относится ли запрос к тому же элементу. Это чаще всего используется, чтобы

указать кешам на ключ с помощью заголовка `Accept-Encoding`, чтобы кеш знал, как различать сжатый и несжатый контент.

В стороне о различных заголовках

Заголовок `Vary` дает вам возможность хранить разные версии одного и того же контента за счет разбавления записей в кэше.

В случае `Accept-Encoding` установка заголовка `Vary` позволяет провести критическое различие между сжатым и несжатым содержимым. Это необходимо для правильной передачи этих элементов браузерам, которые не могут обрабатывать сжатый контент, и необходимо для обеспечения простоты использования. Одна характеристика, которая говорит вам, что `Accept-Encoding` может быть хорошим кандидатом на `Vary`, заключается в том, что он имеет только два или три возможных значения.

Такие элементы, как `User-Agent`, на первый взгляд могут показаться хорошим способом различить мобильные и настольные браузеры для обслуживания разных версий вашего сайта. Однако, поскольку строки `User-Agent` нестандартны, результатом, скорее всего, будет много версий одного и того же содержимого в промежуточных кэшах с очень низким коэффициентом попадания в кеш. Заголовок `Vary` следует использовать с осторожностью, особенно если у вас нет возможности нормализовать запросы в промежуточных кэшах, которые вы контролируете (что может быть возможно, например, если вы используете сеть доставки контента).

Как флаги Cache-Control влияют на кэширование

Выше мы упоминали, как заголовок `Cache-Control` используется для современной спецификации политики кеширования. С помощью этого заголовка можно задать ряд различных инструкций политики, а несколько инструкций разделить запятыми.

Вот некоторые из параметров `Cache-Control`, которые вы можете использовать для определения политики кэширования вашего контента:

- `no-cache`: эта инструкция указывает, что любой кэшированный контент должен повторно проверяться при каждом запросе перед тем, как он будет передан клиенту. По сути, это немедленно отмечает содержимое как устаревшее, но позволяет использовать методы

повторной проверки, чтобы избежать повторной загрузки всего элемента снова.

- `no-store`: эта инструкция указывает, что содержимое не может быть кэшировано каким-либо образом. Это целесообразно установить, если ответ представляет конфиденциальные данные.
- `public`: это помечает контент как общедоступный, что означает, что он может кэшироваться браузером и любыми промежуточными кешами. Для запросов, использующих HTTP-аутентификацию, ответы по умолчанию помечаются `private`. Этот заголовок переопределяет этот параметр.
- `private`: помечает содержимое как `private`. Частный контент может храниться в браузере пользователя, но не должны кэшироваться любыми промежуточными сторонами. Это часто используется для пользовательских данных.
- `max-age`: этот параметр настраивает максимальный возраст, в течение которого контент может быть кэширован, прежде чем он должен будет повторно проверить или повторно загрузить контент с исходного сервера. По сути, он заменяет заголовок `Expires` для современного просмотра и является основой для определения свежести фрагмента контента. Эта опция принимает значение в секундах с максимальным допустимым временем обновления в один год (31536000 секунд).
- `s-maxage`: это очень похоже на настройку `max-age`, поскольку указывает количество времени, в течение которого контент может быть кэширован. Разница в том, что эта опция применяется только к промежуточным кешам. Сочетание этого с вышеизложенным позволяет более гибко строить политику.
- `must-revalidate`: это указывает, что информация о свежести, указанная в `max-age`, `s-maxage` или заголовке `Expires`, должна строго соблюдаться. Просроченное содержание не может быть подано ни при каких обстоятельствах. Это предотвращает использование кэшированного содержимого в случае прерывания работы сети и подобных сценариев.
- `proxy-revalidate`: действует так же, как и вышеупомянутый параметр, но применяется только к промежуточным прокси. В этом случае браузер пользователя может потенциально использоваться для

обслуживания устаревшего контента в случае прерывания сети, но промежуточные кэши не могут использоваться для этой цели.

- `no-transform`: эта опция сообщает кешам, что им не разрешено изменять полученный контент по причинам производительности ни при каких обстоятельствах. Это означает, например, что кэш не может отправлять сжатые версии контента, которые он не получил от сжатого исходного сервера и не разрешен.

Их можно комбинировать различными способами для достижения различного поведения кэширования. Некоторые взаимоисключающие значения:

- `no-cache`, `no-store` и обычное поведение кэширования, на которое указывает отсутствие
- `public` и `private`

Параметр `no-store` заменяет `no-cache`, если оба присутствуют. Для ответов на неаутентифицированные запросы подразумевается `public`. Для ответов на аутентифицированные запросы подразумевается `private`. Их можно изменить, включив противоположную опцию в заголовок `Cache-Control`.

▼ Разница между идентификацией, аутентификацией, авторизацией?

Идентификация - процедура установления личности.

Аутентификация - подтверждение личности, например сравнение введенного логина и пароля с существующими в базе данных.

Авторизация - определение прав и доступов.

▼ Виды аутентификации?

Form-based - преоставление данных через форму на странице

Token-based - предоставление специального токена после успешной аутентификации

OAuth - с помощью социальных сетей

Two-factor - двухфакторная

Biometric - на основе отпечатков пальцев или сканирование лица

Certificate-based - пользователь предоставлят сертификат который ему выдается

OpenID - открытый стандарт который позволяет использовать одну учётную запись для доступа к нескольким веб приложениям

One-time password - одноразовый пароль действующий только в пределеах текущей сесии или на короткий период времени

▼ Что такое безопасные (Secure) и HttpOnly cookies?

Безопасные куки - отправляются на сервер только если запрос выполняется по протоколу SSL и HTTPS. Но важные данные все равно не стоит передавать или хранить в cookies, т.к. они сами по себе уязвимы и флаг secure не обеспечивает никакого шифрования или средств защиты. Сайты с протоколом HTTP не могут создавать cookie с флагом secure.

HttpOnly Cookie - недоступны из JavaScript, что обеспечиватет защиту от XSS. Используется для тех cookie к которым не требуется обращаться через JS, например если используются только для поддержки сессии.

▼ Что такое Content Security Policy (CSP)?

CSP - это HTTP заголовок, который позволяет операторам сайта детально контролировать откуда могут быть загружены ресурсы на их сайт.

Использование этого заголовка это лучший способ предотвратить XSS.

Является обязательным для всех новых сайтов и рекомендуется к внедрению несуществующих сайтах с высоким уровнем риска.

Настройка заголовка в соответствии со списком доверенных источников из которых можно получать контент.

▼ Что такое CORS?

CORS (Cross-Origin Resource Sharing — «совместное использование ресурсов между разными источниками») — это механизм, который позволяет веб-страницам запрашивать ресурсы с другого домена, отличного от домена, с которого была загружена сама страница.

По умолчанию, в целях безопасности, веб-браузеры ограничивают кросс-доменные запросы с использованием политики одного источника (**Same-Origin Policy**). Он предоставляет веб-серверам возможность явно разрешить некоторые кросс-доменные запросы, сохраняя при этом безопасность.

Как он работает

Когда веб-приложение пытается сделать запрос к ресурсу, который находится на другом домене (кросс-доменный запрос), браузер автоматически добавляет к запросу заголовок **Origin**.

Этот заголовок содержит домен, с которого был сделан запрос. Веб-сервер, к которому направлен запрос, затем проверяет этот заголовок и решает, разрешить ли запрос. Если сервер разрешает запросы из этого источника, он отвечает с соответствующими CORS заголовками, указывающими, какие действия разрешены.

Один из таких заголовков — `Access-Control-Allow-Origin`, который может быть установлен в `*` (что означает разрешение для всех доменов) или в конкретный домен.

Примеры:

- `Access-Control-Allow-Origin`: Указывает, какие домены могут получать доступ к ресурсу. Может быть установлен в конкретный домен или `*` для разрешения всех доменов.

- `Access-Control-Allow-Methods`: Указывает, какие HTTP методы разрешены при доступе к ресурсу.
- `Access-Control-Allow-Headers`: Указывает, какие HTTP заголовки могут быть использованы во время запроса.
- `Access-Control-Allow-Credentials`: указывает, разрешено ли передавать куки и авторизационные заголовки при доступе к ресурсам.

Почему CORS важен

Решает важную проблему безопасности, позволяя контролировать, какие веб-сайты могут использовать ресурсы вашего веб-сайта. Это предотвращает множество видов атак, таких как CSRF (Cross-Site Request Forgery — подделка межсайтовых запросов), позволяя при этом легитимным сайтам запрашивать данные через браузер.

Проблемы с CORS

Хотя он повышает безопасность, неправильная настройка CORS может привести к уязвимостям. Например, слишком широкое использование

```
Access-Control-Allow-Origin: *
```

может случайно разрешить небезопасные кросс-доменные запросы. Разработчики должны тщательно настраивать политики CORS, чтобы избежать потенциальных проблем с безопасностью.

CORS является ключевым элементом современной веб-разработки, позволяя безопасно реализовывать кросс-доменные запросы и взаимодействия между веб-приложениями. Правильное понимание и настройка CORS необходимы для обеспечения безопасности и гибкости веб-приложений.

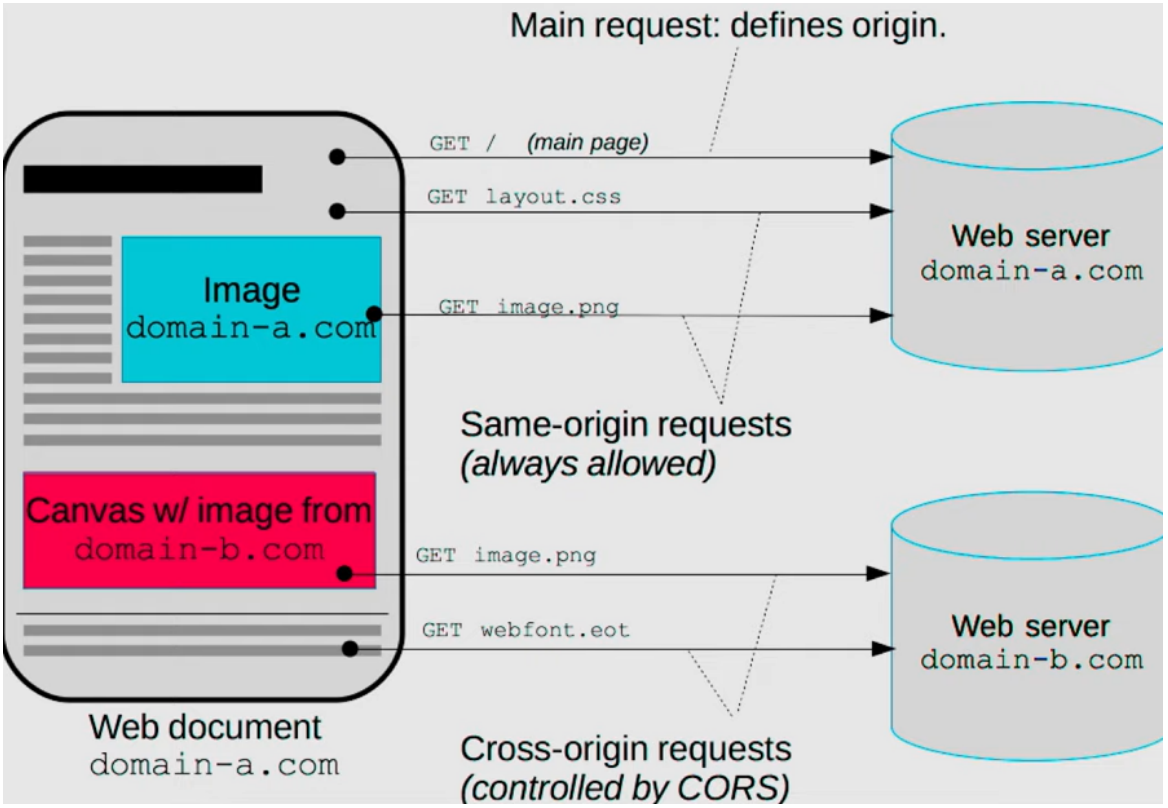
CORS позволяет защитить веб-приложения от несанкционированного доступа к их ресурсам, но при этом позволяет браузеру безопасно загружать данные с других доменов.

Два URL имеют «одинаковый источник» в том случае, если они имеют совпадающие протокол, домен и порт.

Эти URL имеют одинаковый источник:

- http://site.com

- http://site.com/
- http://site.com/my/page.html



Проблему CORS можно решить с помощью:

1. Отключение CORS-проверок в браузере (самый удобный и простой способ)
2. Использование браузерных расширений, таких как CORS Everywhere, которые позволяют обойти ограничения CORS.
3. Можно запустить локальный proxy сервер, который будет пересылать данные между нашим приложением и сервером, добавляя необходимые заголовки
4. Использование JSONP (JSON with Padding) для получения данных с другого домена. JSONP позволяет получить данные с другого домена, обернув их в функцию, которая вызывается на веб-странице.

CORS позволяет сайту A дать разрешение сайту B на чтение (потенциально конфиденциальных) данных с сайта A (используя браузер и учетные данные посетителя).

CSP позволяет сайту на котором установлен этот заголовок предотвратить загрузку (потенциально вредоносного) содержимого из неожиданных источников (например, в качестве защиты от XSS).

▼ Что такое межсайтовый скриптинг (XSS)?

ОЧЕНЬ подробно

Межсайтовый скриптинг (XSS) - это довольно распространенная уязвимость, которую можно обнаружить на множестве веб приложений

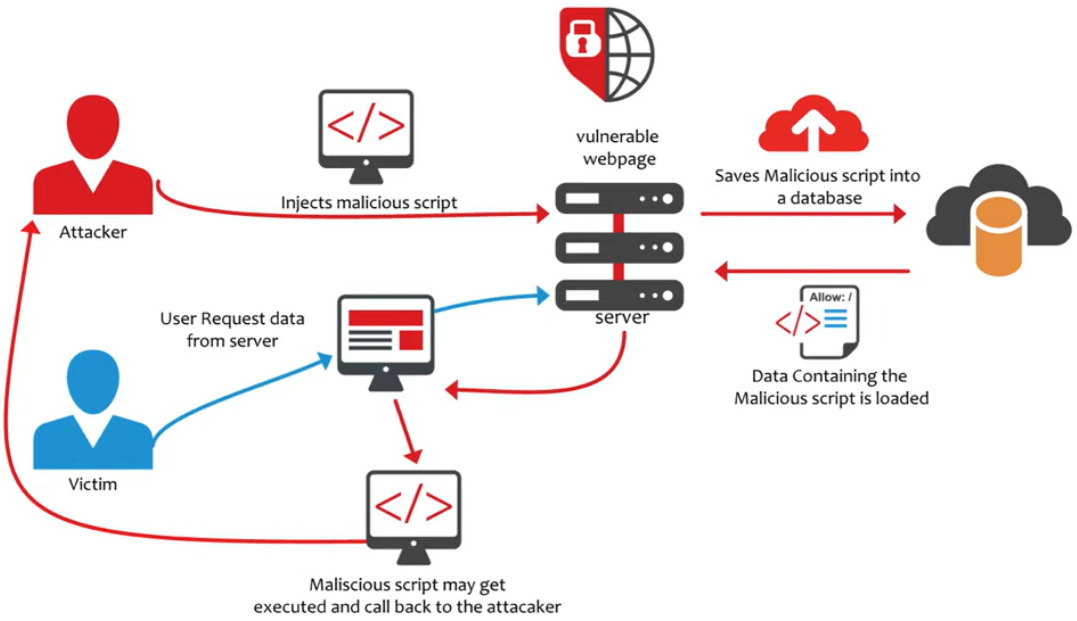
Суть - злоумышленнику удается внедрить на страницу JS код, который не был предусмотрен разработчиками, и этот код будет выполняться каждый раз, когда пользователи будут заходить на страницу веб приложения куда этот код был добавлен.

Например злоумышленнику удастся заполучить авторизационные данные пользователя и войти в его аккаунт.

Или злоумышленник может незаметно для жертвы перенаправить его на другую страницу-клон. Эта страница может выглядеть совершенно идентично той, на которой пользователь рассчитывал оказаться. Но вот принадлежать она будет злоумышленнику. Если пользователь не заметит подмены и на этой странице введет какие-то sensitive data, то есть личные данные, они окажутся у злоумышленника.

Например, можно добавить JavaScript-код в поле ввода, текст из которого сохраняется и в дальнейшем отображается на странице для всех пользователей. Это может быть поле для ввода информации о себе на странице профиля социальной сети или комментарии на форуме.

Злоумышленник вводит текст (и за одно вредоносный код), который сохраняется на странице. Когда другие пользователи зайдут на эту же страницу, вместе с текстом они загрузят и JavaScript-код злоумышленника. Именно в момент загрузки этот код отработает.

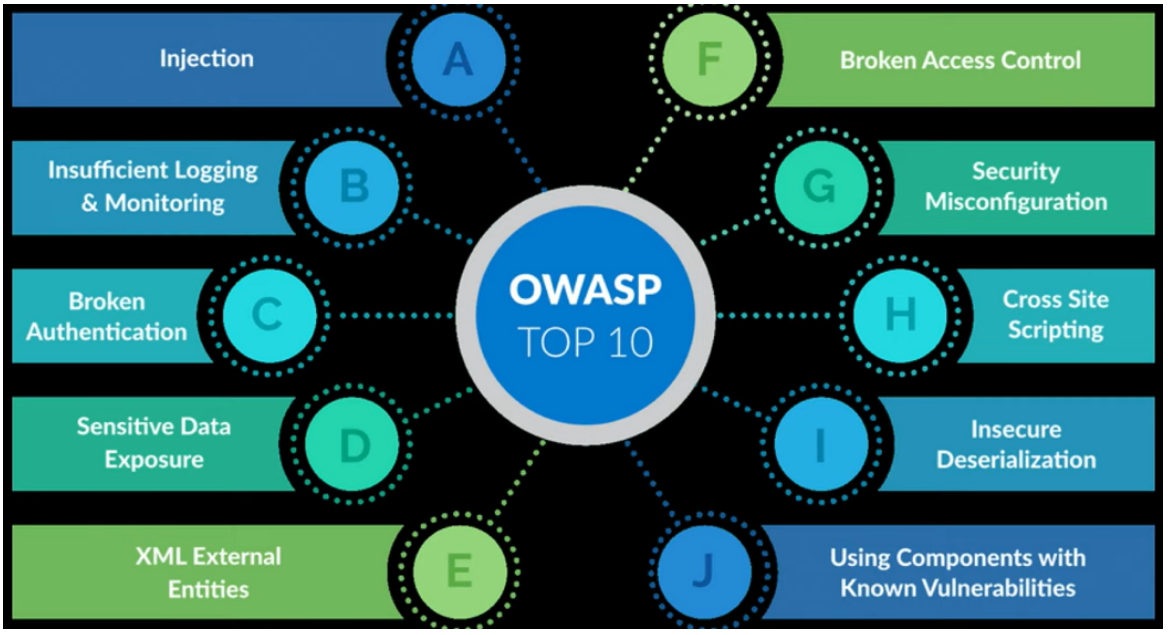


▼ Методы повышения безопасности веб-приложений?

- Валидация входных данных
- Использование HTTPS
- Использование CORS
- Использование CSP
- Использование хэширования
- Предотвращение редиректов
- Проверка входящих JSON-схем
- Регулярное обновление библиотек
- Тестирование безопасности

▼ Что такое OWASP Top 10?

Список наиболее критических и актуальных уязвимостей веб приложений, обновляется ежегодно.



▼ Что такое безопасный и идемпотентный метод?

Безопасный метод HTTP - это метод, который не изменяет состояние ресурса на сервере и не имеет побочных эффектов на другие ресурсы. Такие методы могут быть выполнены безопасно, без опасности повреждения данных или нарушения безопасности.

Идемпотентный метод HTTP - это метод, который может быть выполнен несколько раз без изменения состояния ресурса на сервере. Такие методы не приводят к изменению данных при повторном выполнении запроса с теми же параметрами.

▼ Безопасные и идемпотентные методы?

Безопасные методы HTTP:

- **GET** используется для получения ресурса без его изменения
- **HEAD** аналогичен методу GET, но возвращает только заголовки без тела ответа
- **OPTIONS** используется для получения информации о возможностях сервера и поддерживаемых методах запросов

Идемпотентные методы HTTP:

- **GET** каждый запрос на получение ресурса будет возвращать одинаковый результат
- **PUT** используется для обновления или создания ресурса, при этом повторное выполнение запроса не приведет к изменению состояния ресурса

- **DELETE** используется для удаления ресурса, повторное выполнение запроса не приведет к ошибке, так как ресурс уже удален
- **HEAD** также является идемпотентным, так как не изменяет состояние ресурса

▼ Какие есть HTTP коды?

- **Информационные** 100 - 199
- **Успешные** 200 - 299
- **Перенаправления** 300 - 399
- **Клиентские ошибки** 400 - 499
- **Серверные ошибки** 500 - 599

▼ Что такое безопасные (Secure) и HttpOnly Cookies?

Безопасные куки - отправляются на сервер только если запрос выполняется по протоколу SSL и HTTPS.

HttpOnly Cookie - недоступны JavaScript.

▼ Что такое WebSQL и IndexedDB, а также их различия?

WebSQL и IndexedDB - это две технологии, которые позволяют веб-приложениям хранить и обрабатывать данные локально на клиентской стороне.

WebSQL был разработан в 2009 году и представляет собой SQL-базу данных, которая хранится в браузере пользователя. Он поддерживается большинством браузеров, но не является стандартом W3C. WebSQL использует SQL-запросы для создания, чтения, обновления и удаления данных в локальной базе данных.

IndexedDB был разработан в 2010 году и является стандартом W3C. Он представляет собой объектную базу данных, которая хранится в браузере пользователя. IndexedDB использует JavaScript API для создания, чтения, обновления и удаления данных в локальной базе данных. Он поддерживается не всеми браузерами, но его поддержка растет.

Основное различие между WebSQL и IndexedDB заключается в том, как они хранят данные. WebSQL хранит данные в таблицах, как в SQL-базе данных, а IndexedDB хранит данные в объектах JavaScript. Кроме того, IndexedDB предоставляет более мощный механизм индексации данных, чем WebSQL.

В целом, IndexedDB является более современной и гибкой технологией, чем WebSQL, но WebSQL все еще может быть полезным для простых приложений, которые не требуют сложной логики хранения данных.

▼ Что такое BOM?

BOM (Browser Object Model) - это модель, которая позволяет JS взаимодействовать с браузером.

В него входит:

- screen
- navigator
- location
- history
- document

▼ Разница между cookie, sessionStorage и localStorage?

	cookie	**sessionStorage**	**localStorage**
Инициатор	Клиент/Сервер (Set-Cookie)	Клиент	Клиент
Срок хранения	Установка вручную	До закрытия вкладки	Не ограничено
Связь с доменом	Да	Нет	Нет
Емкость	4 Kb	5 Mb	5 Mb
Доступность	Любое окно	Только вкладка	Любое окно

▼ Опишите весь процесс

Подробнее тут

1. Вы вводите адрес
2. Браузер ищет в своем кэше запись о DNS. Он проверяет 4 кэша: браузера, операционной системы, роутера, провайдера
3. DNS запрос ищет нужный IP на разных DNS серверах, сперва в Local DNS resolver, который поставляется провайдером. Он в свою очередь сначала идет к Root Server, который дает ему TLD Top Level Domain(например .ru) сервер и так по цепочке.
4. Браузер инициализирует TCP соединение с сервером

5. Браузер посылает HTTP запрос на получение нужной страницы
6. Сервер обрабатывает запрос и высылает ответ. Вместе с этим высылает куки файл, которым ПК и сервер обмениваются при каждом запросе.
7. Рендерится HTML разметка
8. Проверяются HTML теги и отсылаются GET запросы на получение дополнительных элементов. Файлы кэшируются
9. Отображается страница в браузере

Архитектуры + методологии

▼ Модульная простая

Модульная архитектура веб-приложения — это подход к организации кода, при котором приложение разбивается на небольшие, независимые компоненты или модули. Каждый модуль отвечает за выполнение конкретной функции или решение определенной задачи.

Основные принципы включают:

- **Разделение на независимые модули:** Приложение разбивается на отдельные компоненты, которые могут быть разработаны, тестированы и поддерживаться независимо друг от друга. Это упрощает разработку и сопровождение приложения.
- **Модульная связь:** Модули могут взаимодействовать друг с другом через строго определенные интерфейсы или API. Это позволяет снизить связанность между модулями и делает приложение более гибким и масштабируемым.
- **Переиспользование кода:** Модули могут быть переиспользованы в различных частях приложения или даже в других проектах. Это позволяет сократить объем кода, улучшить его качество и упростить его сопровождение.
- **Изоляция функциональности:** Каждый модуль отвечает за конкретную функцию или часть функциональности приложения. Это упрощает отладку и тестирование кода, так как изменения в одном модуле не должны затрагивать другие части приложения.

Слои:

Pages – в идеале это максимально тонкий компонент, который содержит только перечисление модулей и не содержит логики

Modules – максимально самостоятельные компоненты со своей зоной ответственности, со своим стейтом, хелперами, константами, api (бесконечный список товаров, который использует компоненты из components и свою логику бесконечной подгрузки товаров). Важный

момент – мы должны изолировать все эти внутренности и запретить отдавать их наружу, нужно использовать public api (index.ts) в котором прописаны экспорты только необходимого.

Components – могут содержать в себе компоненты из слоя **UI**, но не могут содержать компоненты из слоя **modules** (могут обладать логикой, но очень простой) (карточка юзера, рейтинг товара)

UI – содержит только переиспользуемые компоненты без логики (спиннеры, кнопки, формы и т.д.)

Нижележащие слои не могут использовать вышележащие!

Преимущества:

- Изоляция внутренностей модуля за счет **public api**
- Однонаправленный поток данных **pages – modules – components – ui**
- Переиспользуемость за счет слоев
- Модули легко удалять из кода

Недостатки:

- Не всегда понятно, когда делать **module**, а когда **components**
- Что делать если надо использовать один модуль внутри другого
- Где хранить бизнесовые сущности (типы)? Например – товары, статьи, пользователи
- Неявные связи образуются в глобальных функциях (корневой стор, хелперы, константы)

▼ Atomic design

Atomic Design — это методология разработки интерфейсов, предложенная Брэдом Фростом. Он предлагает представить интерфейсы в виде иерархии компонентов, начиная с самых простых элементов и заканчивая сложными шаблонами страниц.

Основные концепции:

Атомы (Atoms): это самые базовые элементы интерфейса, такие как кнопки, поля ввода, заголовки, иконки и т. д. Они обычно не разбиваются на более мелкие элементы. Это как **UI** слой.

Молекулы (Molecules): это комбинации атомов, которые образуют более сложные компоненты. Например, форма для ввода данных может

быть молекулой, состоящей из текстовых полей, кнопок и меток. Это как **components.**

Организмы (Organisms): это компоненты, состоящие из молекул и атомов, которые работают вместе для выполнения определенной функции. Например, блок с боковым меню и формой поиска может быть организмом. Это как **modules**.

Шаблоны (Templates): это структуры, определяющие расположение и взаимодействие различных организмов в рамках страницы. Например, шаблон для страницы блога может включать в себя заголовок, боковое меню, контент и подвал.

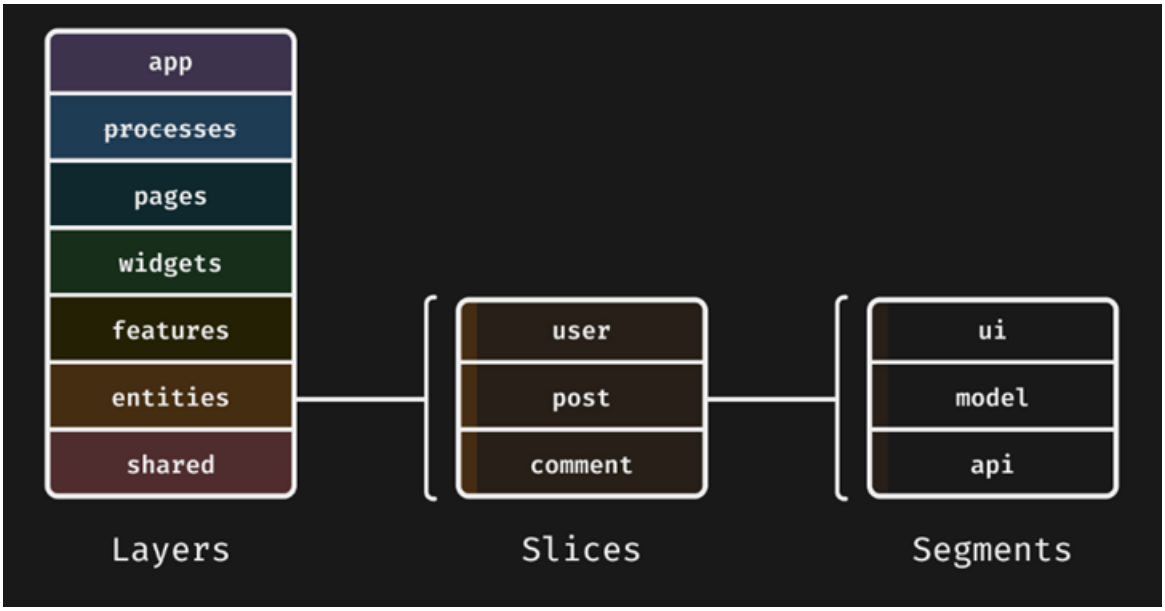
Страницы (Pages): это конечные результаты, которые пользователь видит и с которыми взаимодействует. Они создаются путем объединения шаблонов с конкретным контентом.

Преимущества и **недостатки** такие же как и в модульной архитектуре.

▼ FSD (Feature Slice Design)

FSD — это методология разработки веб-приложений, разработанная исключительно для фронтенда.

Проект на FSD состоит из **слоев (layers)**, каждый слой состоит из **слайсов (slices)** и каждый слайс состоит из **сегментов (segments**).



Слои:

Слои стандартизированы во всех проектах и расположены вертикально. Модули на одном слое могут взаимодействовать лишь с модулями, находящимися на слоях **строго** ниже. На данный момент слоев семь (снизу вверх):

shared — переиспользуемый код, не имеющий отношения к специфике приложения/бизнеса. (например, UIKit, libs, API)

entities (сущности, опциональный) — бизнес-сущности. (например, User, Product, Order)

features (фичи, опциональный) — взаимодействия с пользователем, действия, которые несут бизнес-ценность для пользователя. (например, SendComment, AddToCart, UsersSearch)

widgets (виджеты) — композиционный слой для соединения **сущностей** и **фич** в самостоятельные блоки (например, Header, Navbar, Footer). Максимально самостоятельный компонент.

pages (страницы) — композиционный слой для сборки полноценных страниц из **сущностей**, **фич** и **виджетов**.

processes (процессы, устаревший слой, опциональный) — сложные сценарии, покрывающие несколько страниц. (например, авторизация)

app — настройки, стили, роутинг и провайдеры для всего приложения.

Затем есть **слайсы**, разделяющие код по предметной области. Они группируют логически связанные модули, что облегчает навигацию по кодовой базе. Слайсы не могут использовать другие слайсы на том же слое, что обеспечивает высокий уровень **связности** (cohesion) при низком уровне **зацепления** (coupling).

Сегменты:

В свою очередь, каждый слайс состоит из **сегментов**. Это маленькие модули, главная задача которых — разделить код внутри слайса по техническому назначению.

Самые распространенные сегменты:

- UI
- Model – взаимодействие со стейтом, селекторы, экшены
- Lib – вспомогательные функции

- Config – редкий, конфигурация модуля
- Api – запросы на сервер, которые требуются для данного модуля
- Consts – константы, которые нужны в данном модуле

В большинстве случаев рекомендуется располагать api и config только в shared-слое

Рассмотрим пример приложения социальной сети.

- **app/** содержит настройку роутера, глобальные хранилища и стили.
- **pages/** содержит компоненты роутов на каждую страницу в приложении, преимущественно композирующие, по возможности, без собственной логики.

В рамках этого приложения рассмотрим карточку поста в ленте новостей.

- **widgets/** содержит "собранную" карточку поста, с содержимым и интерактивными кнопками, в которые вшиты запросы к бэкенду.
- **features/** содержит всю интерактивность карточки (например, кнопку лайка) и логику обработки этой интерактивности.
- **entities/** содержит скелет карточки со слотами под интерактивные элементы. Компонент, демонстрирующий автора поста, также находится в этой папке, но в другом слайсе.

Преимущества:

- **Единообразие**

  Код распределяется согласно области влияния (слой), предметной области (слайс) и техническому назначению (сегмент).

  Благодаря этому архитектура стандартизируется и становится более простой для ознакомления.

- **Контролируемое переиспользование логики**

  Каждый компонент архитектуры имеет свое назначение и предсказуемый список зависимостей.

  Благодаря этому сохраняется баланс между соблюдением принципа **DRY** и возможностью адаптировать модуль под разные цели.

- **Устойчивость к изменениям и рефакторингу**

  Один модуль не может использовать другой модуль, расположенный на том же слое или на слоях выше. Благодаря этому приложение можно изолированно модифицировать под новые требования без непредвиденных последствий.

- **Ориентированность на потребности бизнеса и пользователей**

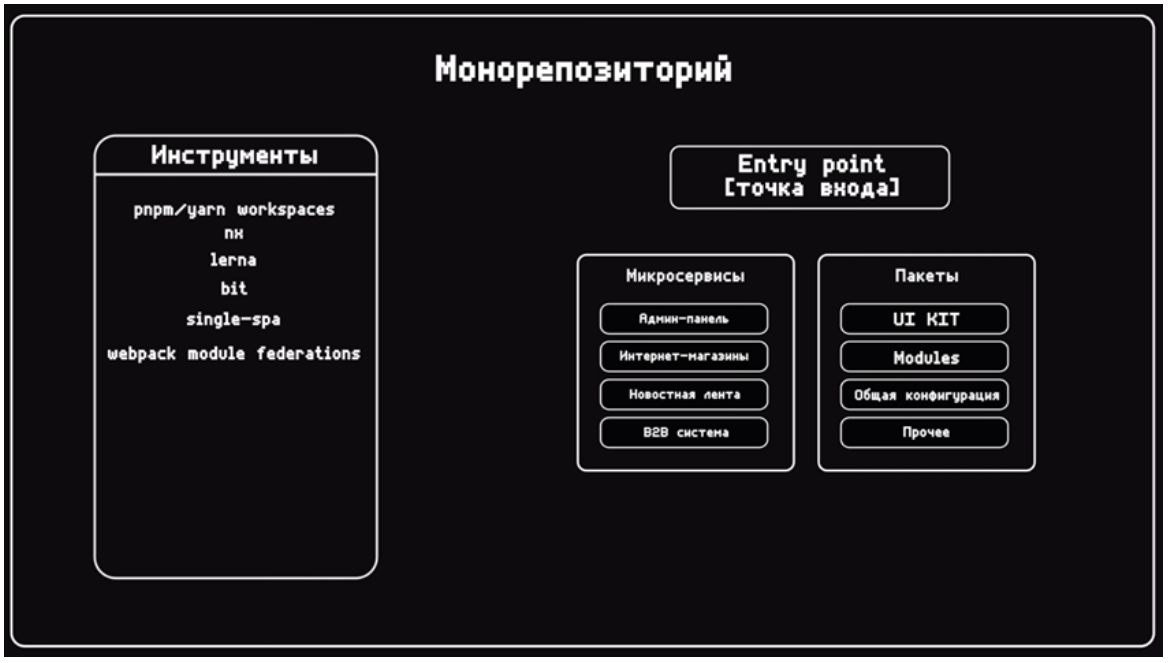
  Разбиение приложения по бизнес-доменам помогает глубже понимать, структурировать и находить фичи проекта.

▼ Микросервисная

Это подход к разработке программного обеспечения, при котором большое приложение разбивается на маленькие, независимые сервисы, которые работают вместе. Каждый микросервис отвечает за выполнение определенной функции или задачи, и они могут взаимодействовать друг с другом через API. Это позволяет упростить разработку, развертывание и масштабирование приложения, так как каждый сервис можно разрабатывать, тестировать и обновлять независимо.

Т.е. мы разделяем наше большое приложение на несколько маленьких независимых и создаем одну точку входа (**Entry point**), которая содержит роуты и перенаправляет нас в нужный микросервис.

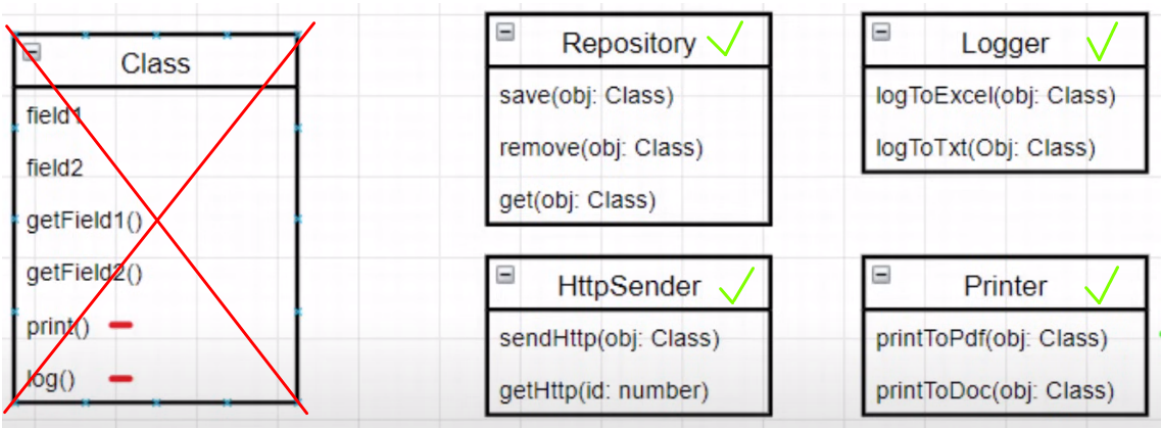
При этом мы выносим в отдельные пакеты UI kit, Modules, Config, которые могут независимо использоваться в каждом сервисе.



▼ SOLID

1. Single Responsobility

Принцип единственной ответственности, который подразумевает 1 сущность = 1 задача. Мы не делаем сущности, которые делают запросы, логируют что-то, сохраняют и т.д., это анти паттерн, такие сущности называют God object. В таком случае нужно декомпозировать сущность на несколько разных, которые выполняют одну задачу.




```
// Плохо:класс отвечает за хранение информации о пользовате
class User {
    constructor(name, email) {
        this.name = name;
        this.email = email;
    }
    saveToDatabase() {
        // код для сохранения пользователя в базе данных
    }
    sendEmail() {
        // код для отправки электронной почты
    }
}
```

```
// Хорошо: разделение классов
class User {
    constructor(name, email) {
        this.name = name;
```

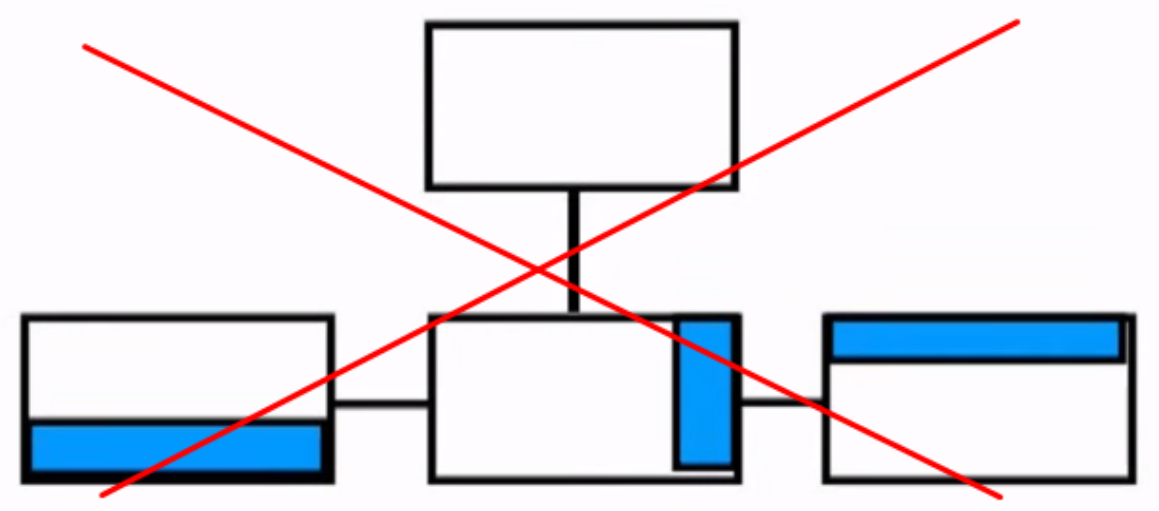
```
        this.email = email;
    }
}

class UserRepository {
    save(user) {
        // код для сохранения пользователя в базе данных
    }
}

class EmailService {
    sendEmail(user) {
        // код для отправки электронной почты
    }
}
```

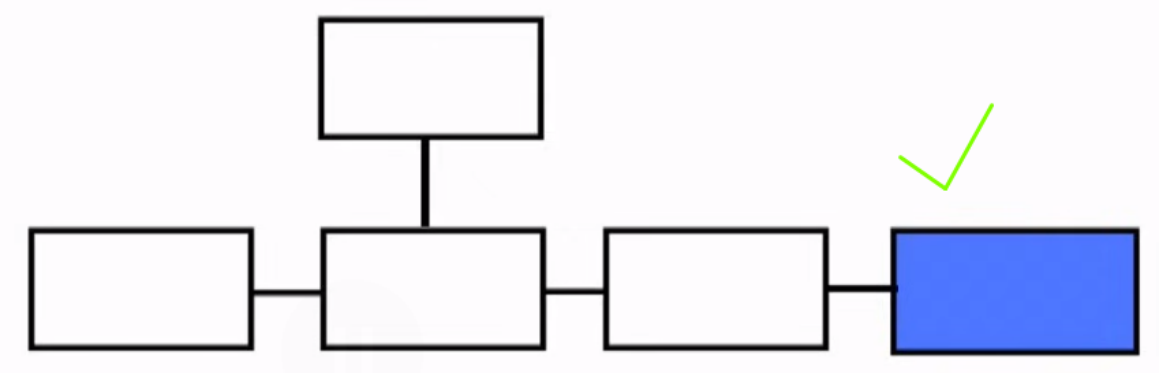
В такой код проще вносить изменения.

Каждая сущность инкапсулирует решение одной задачи.

2. Open-closed Principle

Принцип открытости/закрытости - сущности должны быть открыты для расширения, но закрыты для изменения. Т.е. надо стараться добавлять новый функционал не за счет изменения сущностей, а за счет добавления новых сущностей.





В примере мы представим систему аутентификации, которая имеет два типа аутентификации: по **паролю** и по **отпечатку пальца**.

```
class AuthenticationService {
  authenticateWithPassword(username, password) {
    // Аутентификация по паролю
  }

  authenticateWithFingerprint(username, fingerprint) {
    // Аутентификация по отпечатку пальца
  }
}
```

Представим, что нам нужно добавить третий метод аутентификации, например, аутентификацию по смарт-карте.
Мы могли бы добавить его прямо в класс **AuthenticationService**, но это нарушало бы принцип OCP.

Плохой способ:

```
class AuthenticationService {
  authenticateWithPassword(username, password) {
    // Аутентификация по паролю
  }

  authenticateWithFingerprint(username, fingerprint) {
    // Аутентификация по отпечатку пальца
  }

  authenticateWithSmartCard(username, smartCard) {
```

```
    // Аутентификация по смарт-карте
  }
}
```

Хороший способ:

```
// Интерфейс аутентификации
class Authenticator {
  authenticate(username, credentials) {
    throw new Error('This method should be overridden');
  }
}

// Аутентификация по паролю
class PasswordAuthenticator extends Authenticator {
  authenticate(username, password) {
    // Аутентификация по паролю
  }
}

// Аутентификация по отпечатку пальца
class FingerprintAuthenticator extends Authenticator {
  authenticate(username, fingerprint) {
    // Аутентификация по отпечатку пальца
  }
}

// Аутентификация по смарт-карте
class SmartCardAuthenticator extends Authenticator {
  authenticate(username, smartCard) {
    // Аутентификация по смарт-карте
  }
}
```

В этом примере мы создали интерфейс **Authenticator**, который будет представлять общий интерфейс для всех методов аутентификации. Затем каждый метод аутентификации будет реализовывать этот интерфейс.

3. Liskov Substitution Principle

Принцип подстановки Барбары Лисков - функции/сущности, которые используют родительский тип, должны точно так же работать и с дочерними.

Т.е. объекты в программе могут быть заменены экземплярами их подтипов без изменения правильности выполнения программы.

Пример: Если у нас есть классы "Кот" и "Собака", которые наследуются от класса "Животное", то мы должны иметь возможность использовать объекты класса "Кот" везде, где ожидается объект класса "Животное", не нарушая работу программы.

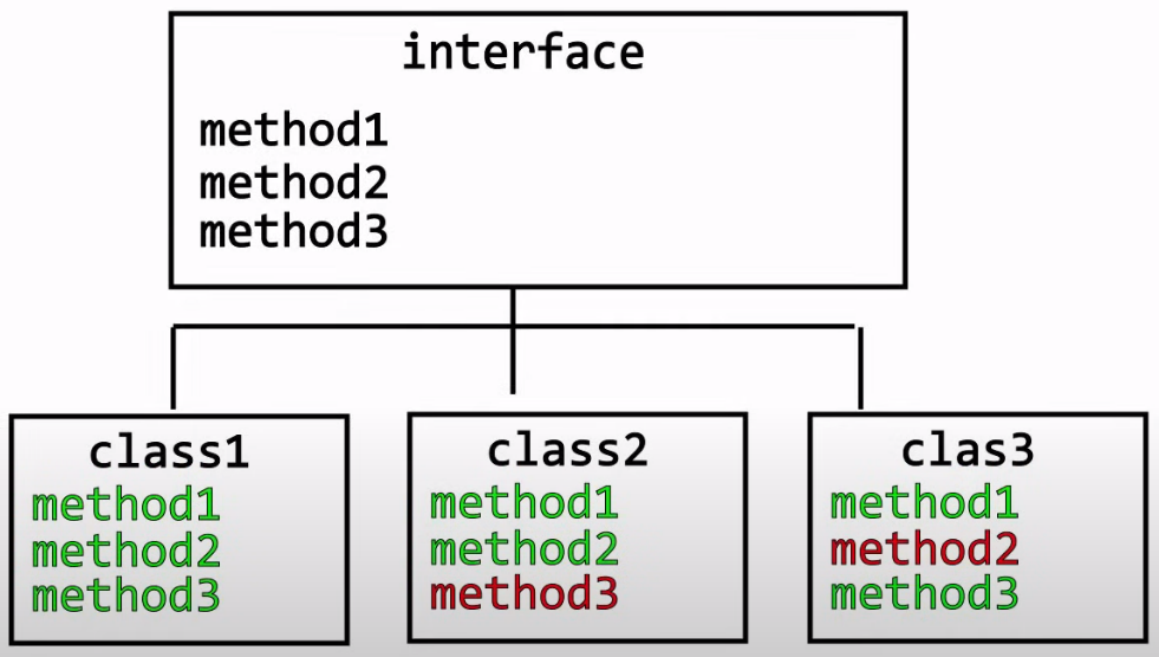
```
class Animal {
  constructor(name) {
    this.name = name;
  }

  makeSound() {
    throw new Error('This method should be overridden');
  }
}

class Cat extends Animal {
  constructor(name) {
    super(name);
  }

  makeSound() {
    return 'Meow!';
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name);
  }

  makeSound() {
```

```
    return 'Woof!';
  }
}

function makeAnimalSound(animal) {
  console.log(`${animal.name} says ${animal.makeSound()}`);
}

const cat = new Cat('Whiskers');
const dog = new Dog('Buddy');

makeAnimalSound(cat); // Вывод: Whiskers says Meow!
makeAnimalSound(dog); // Вывод: Buddy says Woof!
```

4. Interface Segregation Principle

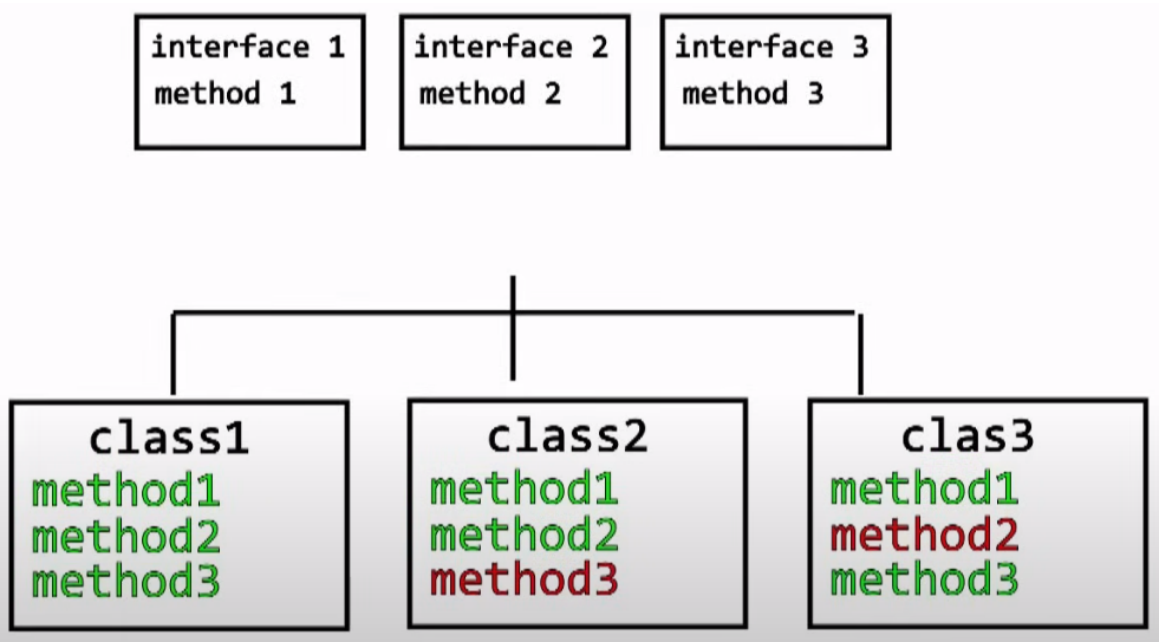
Принцип разделения интерфейса - программные сущности не должны зависеть от методов, которые они не используют.

Например у нас есть интерфейс с тремя методами, и нам нужно создать классы, которые используют эти методы. Первый класс - использует все три метода, второй класс - не использует третий метод, третий класс - не использует второй метод.



Если из методов во 2 и 3 классе возвращать null, это будет нарушение принципа.

Лучше разделить интерфейс на 3 маленьких и в нужный классах использовать только нужные методы.



Плохо: один большой интерфейс

```
class Worker {
  work() {}
  rest() {}
}

class Programmer extends Worker {
  work() {
    // работа программиста
  }
  rest() {
    // не должен тут использоваться
  }
}

class SecurityGuard extends Worker {
```

```
  work() {
    // не должен тут использоваться
  }
  rest() {
    // отдых охранника
  }
}
```

Хорошо: маленькие интерфейсы

```
class Workable {
  work() {}
}

class Restable {
  rest() {}
}

class Programmer extends Workable {
  work() {
    // работа программиста
  }
}

class SecurityGuard extends Restable {
  rest() {
    // отдых охранника
  }
}
```

Избавляем программные сущности от методов, которые они не использую.

Код становится менее связанный.

Получаем более предсказуемую работу.

Dependency Invertion Pronciple

Принцип инверсии зависимости - модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Проще говоря, это означает, что вместо того, чтобы модуль верхнего уровня напрямую зависел от модулей нижнего уровня, оба модуля должны зависеть от общего интерфейса или абстракции. Таким образом, изменения в модуле нижнего уровня не должны затрагивать модуль верхнего уровня, а наоборот.

Пример:

```
// Модуль нижнего уровня, который предоставляет функциональ
class NotificationService {
  sendNotification(message) {
    console.log(`Sending notification: ${message}`);
  }
}

// Модуль верхнего уровня, который использует NotificationS
class UserManager {
  constructor(notificationService) {
    this.notificationService = notificationService;
  }

  notifyUser(user, message) {
    this.notificationService.sendNotification(`${user}: ${m
  }
}

// Создание объектов
const notificationService = new NotificationService();
const userManager = new UserManager(notificationService);

// Использование
userManager.notifyUser('John', 'Your order has been process
```

В этом примере модуль `UserManager` верхнего уровня зависит от модуля `NotificationService` нижнего уровня. Это нарушает принцип инверсии

зависимостей, так как модуль верхнего уровня напрямую зависит от модуля нижнего уровня.

Как правильно:

```
// Абстракция для отправки уведомлений
class NotificationProvider {
  sendNotification(message) {
    throw new Error('sendNotification method not implemente
  }
}

// Модуль нижнего уровня, реализующий NotificationProvider
class NotificationService extends NotificationProvider {
  sendNotification(message) {
    console.log(`Sending notification: ${message}`);
  }
}

// Модуль верхнего уровня, который теперь зависит от абстра
class UserManager {
  constructor(notificationProvider) {
    this.notificationProvider = notificationProvider;
  }

  notifyUser(user, message) {
    this.notificationProvider.sendNotification(`${user}: ${
  }
}

// Создание объектов
const notificationService = new NotificationService();
const userManager = new UserManager(notificationService);

// Использование
userManager.notifyUser('John', 'Your order has been process
```

▼ MVC

Model-View-Controller (Модель-Представление-Контроллер) - популярная архитектурная модель для разработки программного обеспечения, особенно для веб-приложений.

В рамках этой модели:

1. **Модель (Model)** представляет данные и бизнес-логику приложения. Она отвечает за хранение данных и их обработку.
2. **Представление (View)** представляет пользовательский интерфейс приложения. Это то, что пользователь видит и с чем взаимодействует. Представление отображает данные из модели для пользователя.
3. **Контроллер (Controller)** является посредником между моделью и представлением. Он обрабатывает входные данные пользователя, взаимодействует с моделью для получения необходимых данных и передает эти данные представлению для отображения. Контроллер также отвечает за обработку пользовательских действий и управление потоком приложения.

Использование модели MVC помогает разделить логику приложения, пользовательский интерфейс и данные, что делает приложение более модульным, легким для понимания и изменения.

▼ MVP

Минимально жизнеспособный продукт - версия продукта, включающая в себя минимальный набор функций, необходимых для его развертывания и тестирования на рынке. Цель MVP - собрать обратную связь от пользователей и изучить их поведение при использовании с минимальными затратами времени и ресурсов. После сбора обратной связи продукт может быть итеративно улучшен и дополнен в соответствии с потребностями и предпочтениями пользователей. Этот итеративный процесс помогает снизить риск разработки продукта, который не соответствует запросам или ожиданиям рынка.

▼ Tree shaking это?

Подробнее

Это техника оптимизации кода в среде JavaScript, направленная на удаление неиспользуемого кода из бандлов при сборке приложения. Эта техника особенно полезна в современных фронтенд-разработках,

где используются модульные системы (например, CommonJS или ES Modules) и сборщики модулей (например, Webpack или Rollup).

Идея заключается в том, чтобы анализировать зависимости модулей и определять, какие части кода фактически используются приложением. После этого неиспользуемый код (или неиспользуемые части кода) удаляются из итогового бандла. Это позволяет существенно сократить размер итогового JavaScript-кода, ускорить загрузку приложения в браузере и уменьшить потребление ресурсов.

Tree shaking в значительной степени зависит от системы модулей и инструментов сборки. Например, сборщики модулей, такие как Webpack и Rollup, предоставляют возможности для tree shaking с использованием статического анализа кода и определения зависимостей модулей на этапе сборки.

Важно! - чтобы модули были написаны таким образом, чтобы они были подходящими для **tree shaking**, то есть не содержали побочных эффектов при импорте, таких как изменение глобального состояния или вызовы побочных функций.

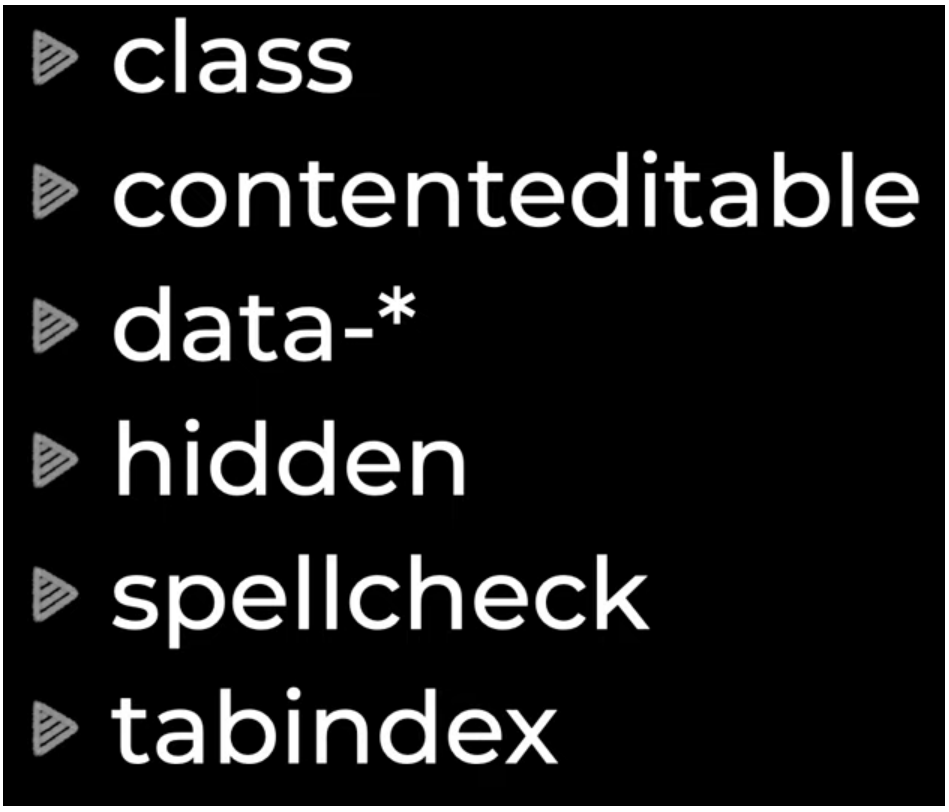
HTML

▼ Что такое HTML и для чего он используется?

HTML - это язык гипертекстовой разметки, который используется для структурирования и отображения веб-страницы и её контента.

▼ Какие глобальные атрибуты есть в HTML?

Глобальные атрибуты - это атрибуты, которые могут использоваться с любым тегом, например:



1. **class** принимает значение в виде списка классов разделенных пробелом, классы позволяют получать доступ к конкретным элементам.
2. **data-* атрибут** - этот атрибут позволяет хранить какие-то данные в элементе и возможность взаимодействовать с ними в JS.

Обращение к свойству dataset вернёт объект со всеми дата-атрибутами, которые есть на элементе.

1. **hidden** - этот логический атрибут, который скрывает контент
2. **id** - этот атрибут определяет идентификатор, который должен быть уникальным для всего документа.
3. **style** - этот атрибут используется для описание стилей, которые должны будут применяться к элементу.представляют собой различные типы элементов, которые используются для структурирования содержимого веб-страниц. HTML5 включает в себя несколько категорий

контента, таких как , , , , , и . Каждая из этих категорий имеет свои особенности и предназначена для определенного типа содержимого

▼ Что такое категории контента в HTML5?

https://developer.mozilla.org/ru/docs/Web/HTML/Content_categories#основные_категории_контента

Категории контента в HTML5 представляют собой различные типы элементов, которые используются для структурирования содержимого веб-страниц. HTML5 включает в себя несколько категорий контента, таких как:

Метаданные(`<base>`, `<command>`, `<link>`, `<meta>`, `<noscript>`, `<script>`, `<style>` и `<title>`).

Основной поток

Секционный контент

Заголовочный контент

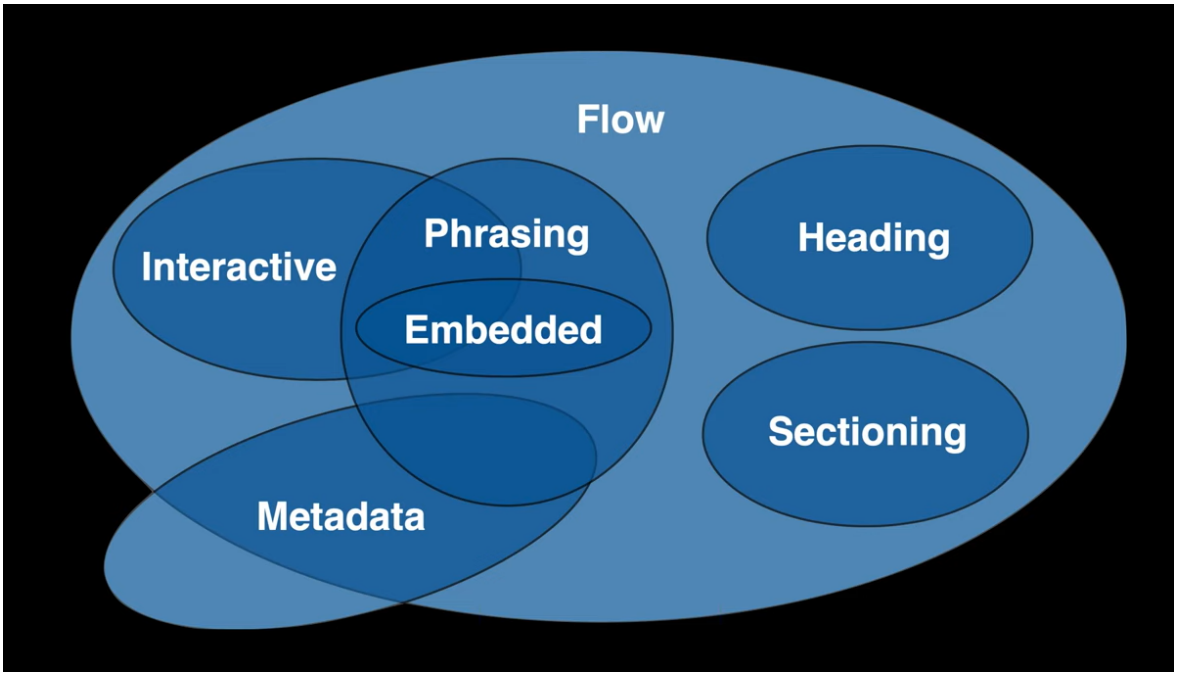
Фразовый контент

Встроенный контент

Интерактивный контент

Контент форм

Каждая из этих категорий имеет свои особенности и предназначена для определенного типа содержимог



▼ Какие категории считаются основными категориями контента?

Категории контента в HTML5 представляют собой различные типы элементов, которые используются для структурирования содержимого веб-страниц. HTML5 включает в себя несколько категорий контента, таких как

- **Метаданные (Meta Data).** Элементы, которые содержат дополнительную информацию об HTML-документе, связывают его с другими документами (например, таблицами стилей css), определяют внешний вид или поведение контента на странице.
- **Потоковый контент (Flow).** К потоковому содержимому относят большинство элементов, которые используются в теле документа (между тегами <body></body>) или веб-

приложения относятся к потоковому контенту. Должен включать в себя хотя бы один явный элемент.

- **Секционный контент (Sectioning).** К секционному контенту относятся элементы, которые создают отдельные секции в структуре документа, существующие независимо друг от друга, такие как заголовочная часть (header), нижний колонтитул (footer) и т.д.
- **Заголовочный контент (Heading).** Заголовочный контент содержит элементы, определяющие заголовки разделов, которые размечены элементами секционного контента или же подразумеваются исходя из содержания заголовка.
- **Текстовой контент (Phrasing).** К текстовому контенту относятся элементы, определяющие текст и его формат. Должен включать в себя хотя бы один явный элемент.
- **Встроенный контент (Embedded).** Встроенным называется контент, импортируемый в HTML-документ из других ресурсов.
- **Интерактивный контент (Interactive).** Интерактивными считаются элементы, разработанные для взаимодействия с пользователем.
- **Явный контент (Palpable).** Элементы контентных моделей, которые позволяют содержать в себе любой потоковый или текстовой контент, должны иметь как минимум один дочерний блок, который будет явным контентом и у которого не будет атрибута со значением hidden.
- **Контент форм.** Включает в себя элементы расположенные в теге form

▼ Что такое doctype? И для чего он используется?

Это инструкция, которая говорит браузеру, какой тип текущего документа используется и на каком языке разметки он сверстан.

Служит для того чтобы браузер мог понять как ему интерпретировать страницу, по какому стандарту осуществлять парсинг документа и исходя из этого будет определять валидность тегов.

Простыми словами, прописывая DOCTYPE, мы говорим браузерам/валидаторам, каким языком разметки мы писали свой документ.

Если сравнивать HTML с устройством человека, то `<!DOCTYPE>` — это его вид. Важно знать, что было раньше, но не надо быть австралопитеком. Единственный вариант, который можно использовать — это `<!DOCTYPE html>`, все остальные устарели.

▼ Опишите базовую структуру HTML-страницы?

- **doctype -** тип документа
- **html -** корневой тег
- **head -** служебная информация о документе, заголовок, сео инфа, подключение стилей и скриптов, описание мета информации. Все что внутри этого тега не отрисовывается на странице
- **meta -** мета инфа, например описание и ключевые слова
- **title -** заголовок
- **body -** соедержит в себе всю разметку.

▼ Что такое валидация? И какие типы проверок HTML документа вы знаете?

Валидация - это проверка кода на соответствия стандартов W3C с помощью валидаторов. Сначала определяется тип документа, затем проверяется html код на правильность и отсутсвие ошибок, проверяется вложенность.

Плюсы: seo, кросс-браузерность, чистота кода.

Типы проверок: проверка синтаксиса, вложенности, Document Type Definition, проверка на наличие посторонних тегов и элементов

▼ Основные этапы проверок валидности HTML-документа?

Валидным считается тот документ, который проходит следующие этапы:

1. Валидация Document Type Definition.
2. Валидация синтаксиса.
3. Проверка вложенности.
4. Проверка на посторонние элементы.

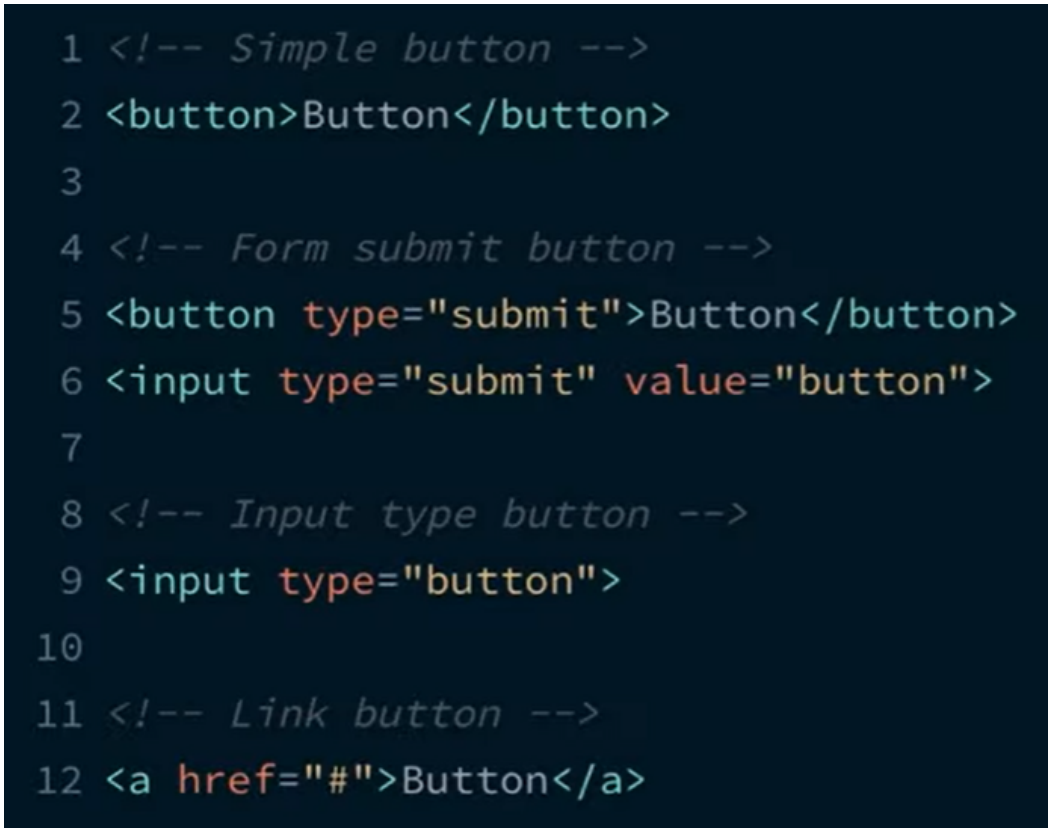
Если хотя бы одна проверка не проходит, то html документ считается невалидным.

▼ Если представить HTML5 как открытую веб-платформу, из каких блоков он состоит?

- **Семантика**. Позволяет более точно описать из чего состоит контент.
- **Стилизация.** Позволяет создавать более сложные темы оформления.
- **Доступ к устройствам.** Позволяет взаимодействовать с различными устройствами ввода и вывода.
- **Связанность**. Позволяет общаться с сервером новыми и инновационными способами.
- **Офлайн и хранилище.** Позволяют страницам хранить данные локально на клиентской стороне и более эффективно работать в офлайне.
- **Мультимедиа**. Создание и подключение видео и аудио.
- **2D- и 3D-графика и эффекты.** Позволяет расширить возможности презентации.
- **Производительность и интеграция.** Обеспечивает большую скорость оптимизации и лучшее использование аппаратных средств.

▼ Какой тэг использовать для того, что бы сверстать кнопку?

Зависит от контекста.



```
<!-- Simple button -->
<button>Button</button>

<!-- Form submit button -->
<button type="submit">Button</button>
<input type="submit" value="button">

<!-- Input type button -->
<input type="button">

<!-- Link button -->
<a href="#">Button</a>
```

▼ Что такое инлайновый стиль? Можно ли его переопределить?

Это способ стилизовать тег прямо внутри HTML документа. Для этого тегу нужно приписать style="" и в кавычках перечислить все стили, которые вы хотите ему присвоить.

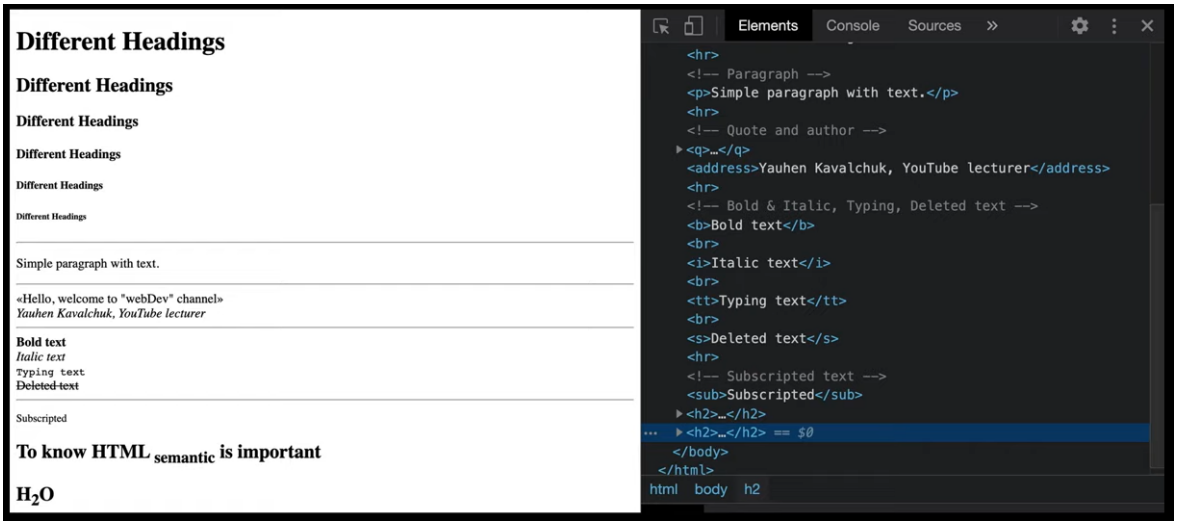
Единственный способ переопределить такие стили это !important

```
<!-- Inline styles -->
<h1 style="color:red;">Heading</h1>

<!-- Override -->
<style>
    h1 {
        color: green !important;
    }
<style>
```

▼ Есть ли у HTML элементов свои дефолтные специфичные стили?

Под капотом у практически каждого элемента есть собственный набор стилей.

У каждого браузера они могут быть разными. Поэтому часто используют normalize.css чтобы привести стили в разных браузерах к общему виду.



▼ Что такое семантика? Какие семантичные тэги вы знаете?

Семантика - это использование правильных тегов HTML5 описывающие содержимое контента. header, main, footer, section, article, nav, aside. Такие теги имею семантическое имя, описывающее свое назначение. Т.е. по тегу можно понять какой контент в нем находится.

```
<!-- global -->
<header></header>
<footer></footer>
<aside></aside>
<nav></nav>
<main></main>
<!-- locals -->
<p></p>
<h1></h1>
<em></em>
<strong></strong>
<ul></ul>
```

▼ Как семантически правильно сверстать картинку с подписью?

```
<figure>
    <img src="" alt="" />
    <img src="" alt="" />
    <figcaption>
        Описание
```

```
    <figcaption>
</figure>
```

Типы списков в HTML?

- **ul** маркированный списки
- **ol** пронумерованные списки
- **dl** список определений (dt - определение, dd - поясняющий текст)

Для какого тэга используется атрибут `alt` и зачем он нужен?

Атрибут alt используется для тега img и предназначен для отображение альтернативного текста на случай того что изображение не прогрузится. Этот атрибут обязательный, и он улучшает доступность.

Какая разница между тэгами `<strong><em>` и `<b><i>` ?

- **** - это сематические теги, они не просто изменяют набор стилей вложенного контента, но и придает логическое выделение. Используется для читалок и поисковых роботов, это текст на котором сделан акцент.
- **<b><i>** - изменяет элемент только визуально

Типы `<input>` элементов в HTML?

Существует много типов этого контейнера:

- text
- image
- submit
- reset
- tel
- email
- date
- hidden
- password
- file
- checkbox
- radio
- button

На экранных клавиатурах может меняться раскладка.

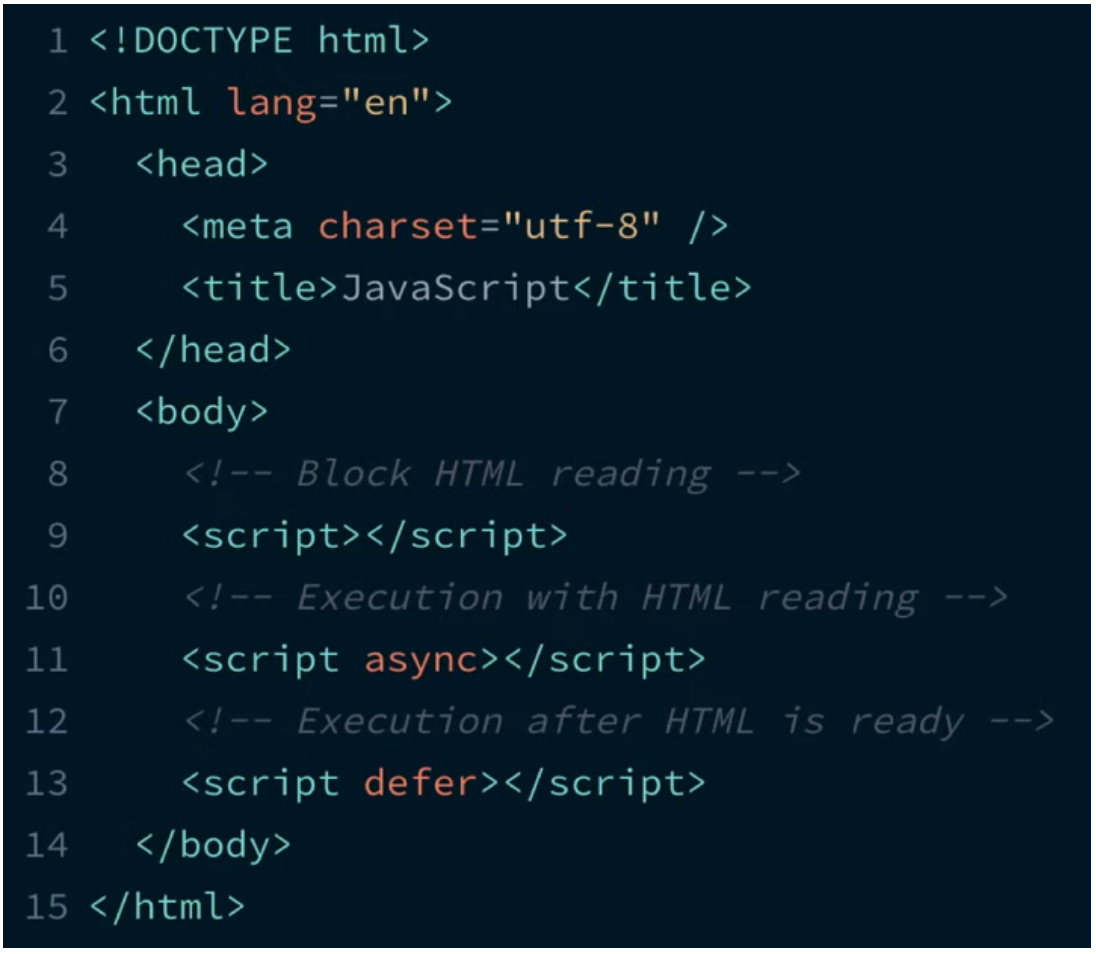
▼ Для чего используют `data-` атрибуты?

date-атрибут - это атрибут, который хранит в себе определенные данные, которые можно использовать в js и css, манипулируя ими.

Не используют из-за легкой доступности из инструментов разработчика и соображений безопасности т.к. его можно легко изменить.

▼ Разница между `<script>`, `<script async>` и `<script defer>`?

- **defer** загружает в фоне и выполняется после загрузки всех элементов DOM. Выполняется в том же порядке в котором объявлен, а не в котором будет скачан. **<script defer>** - выполняется после того как весь body прогрузится.
- **async** загружает в фоне и выполняется сразу же по факту загрузки, приостанавливая остальные операции на странице. Используется для подключения служебных скриптов не влияющих на работу пользователя со страницей, которые можно выполнить до загрузки DOM. Выполняется как только будет скачан, порядок объявления при этом не имеет значения. **<script async>** - выполняется асинхронно, параллельно вне зависимости от прогрузки body, (реклама, счетчики).
- обычный скрипт без этих атрибутов сразу как обнаруживается анализатором приостанавливает остальные операции на странице, скачивается и выполняется. **<script>** - выполняется синхронно, по мере появления в html.



```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>JavaScript</title>
  </head>
  <body>
    <!-- Block HTML reading -->
    <script></script>
    <!-- Execution with HTML reading -->
    <script async></script>
    <!-- Execution after HTML is ready -->
    <script defer></script>
  </body>
</html>
```

▼ Для чего используется элемент `<datalist>` ?

Элемент <datalist> — это источник предложений, то есть такой элемент, который содержит предопределенные варианты выбора для пользователя. В основном за варианты выбора опций выступают элементы <option>.

Кроме функции автодополнения элемента `<datalist>`, основное отличие между `<select>` и этим тегом в том, что в элементе `<select>` нельзя выбрать или указать значение не из списка предлагаемых.

В интерактивных элементах с вводом значений можете указать своё значение или выбрать что-то из предложенного в списке.

```html
<input list="fruits">
<datalist id="fruits">
  <option value="Банан">
  <option value="Арбуз">
  <option value="Киви">
</datalist>
```

Элемент `<datalist>` не сработает, если между элементами `<input>` и `<datalist>` будет любой другой элемент с `id`. Даже если это сам `<input>` со значением, которое равно значению атрибута `list` элемента `<input>`:

```
<form>
  <label for="fruits">Выберите фрукт из неработающего списка:</label>
  <input list="fruitsList" id="fruits" name="fruits">
  <div id="fruitsList">
    <datalist id="fruitsList">
      <option value="Банан">
      <option value="Арбуз">
      <option value="Киви">
    </datalist>
  </div>
</form>
```

▼ Почему хорошей практикой считается располагать `<link>` для подключения CSS стилей внутри тэга `<head>`, а `<script>` для подключения JS ставить перед закрывающимся тэгом `</body>`?

CSS стили нужно подключить как можно раньше, чтобы запросить их как можно ранше ускорив отображение контента вместо белой старницы. Если их подключить в конце то страница будет перерисовываться дважды, т.к. изменяется набор стилей у элементов, сначала встроенные браузерные, а затем скачанные.

A js нужно подключать в конце потому что он останавливает процесс загрузки страницы, пока js не выполнится.

▼ Что такое мета-тэги?

Мета-теги нужны для краткого описания страницы для поисковых систем (кодировка, сео информация, типа, автора, ключевых слов, красткое описание), а также мета тег содержит доп. информацию, которая нужна для роботов и корректной работы страницы на разных девайсах.

SEO-теги нужны как раз для того, чтобы поисковик мог подобрать нужный пользователю контент. Есть три основных SEO-тега: title, description, keywords.

```html
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <!-- SEO -->
    <meta name="description" content="Test web page">
    <meta name="keywords" content="html, webdev, Yauhen">
    <meta name="author" content="Yauhen Kavalchuk">
    <!-- For mobile devices -->
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- Use latest version of IE -->
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
  </head>
  <body>
  </body>
</html>
```

▼ **Что описывается в тэге `<head>` ?**

- **title** заголовок
- **meta теги** - кодировка, сео информация
- **link** подключение стилей, шрифтов
- **script src** для подключение скриптов

```html
<html lang="en">
<head>
  <meta charset="UTF-8">
    <!-- Document title in browser tabs -->
    <title>HTML Basics</title>
    <!-- Document Favicon in browser tabs -->
    <link rel="shortcut icon" href="icon.ico" type="image/x-icon">
    <!-- Additional approach to add favicon -->
    <link rel="apple-touch-icon" href="apple-touch-icon.png">
    <link rel="apple-touch-icon" sizes="72x72" href="apple-touch-icon-72x72.png">
    <!-- SEO -->
    <meta name="description" content="Test web page">
    <meta name="keywords" content="html, webdev, Yauhen">
    <meta name="author" content="Yauhen Kavalchuk">
    <!-- For mobile & no resize -->
    <meta name="viewport" content="width=device-width, initial-scale=1 user-scalable=no">
    <!-- Use latest version of IE -->
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <!-- For external styles -->
    <link rel="stylesheet" href="styles.css">
    <!-- Local styles -->
    <style>div { background-color: red; }</style>
    <!-- Local styles fonts -->
    <link rel="stylesheet" href="styles.css">
    <!-- Google fonts -->
    <link href="https://fonts.googleapis.com/css?family=Roboto&display=swap" rel="stylesheet">
  </head>
  <body>
  </body>
</html>
```

▼ **Для чего используются тэги `<tr>`, `<th>`, `<td>`?**

Это не самостоятельные теги,они должны располагаться внктри тега table

Подробнее тут

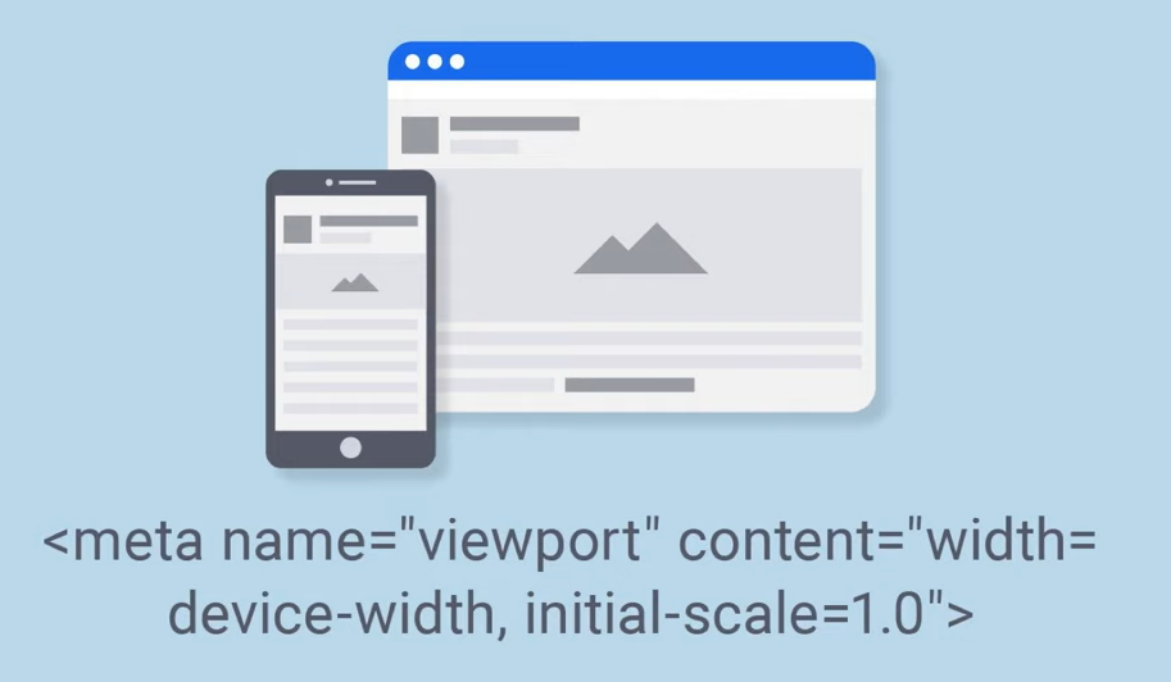
- **th(table header)** предназначен для создания одной ячейки таблицы, которая обозначается, как заголовочная
- **td(table data)** предназначен для создания одной ячейки таблицы
- **tr(table row)** - служит контейнером для создания строки таблицы

```
<table>
  <tr>
    <th>Speciality</th><th>Quantity</th>
  </tr>
  <tr>
    <td>Front-end</td><td>1 Person</td>
  </tr>
  <tr>
    <td>Back-end</td><td>7 Persons</td>
  </tr>
  <tr>
    <td>Full-stack</td><td>2 Persons</td>
  </tr>
  <tr>
    <td>QA</td><td>5 Persons</td>
  </tr>
  <tr>
    <td>4</td><td>15</td>
  </tr>
</table>
```

Speciality	Quantity
Front-end	1 Person
Back-end	7 Persons
Full-stack	2 Persons
QA	5 Persons
4	15

▼ Расскажите о meta-теге с `name="viewport"` ?

Этот тег нужен для создания отзывчивой страницы, для того чтобы определять юзера который зашёл с телефона или планшета и адаптировать веб страницу под его разрешение. max-scale и min-scale задают макисмальное и минимальное допустимое масштабирование страницы, а scalable=”no” запрещает масштабирование.

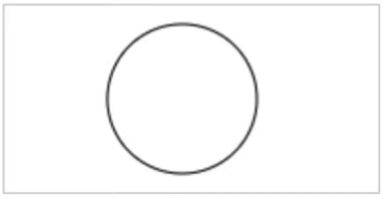


▼ Что такое элемент `<canvas>` ? И для чего он используется?

Тег `<canvas>` добавляет растровый холст на страницу. Этот холст можно использовать для отрисовки 2D- или 3D-графики, анимаций, видео.

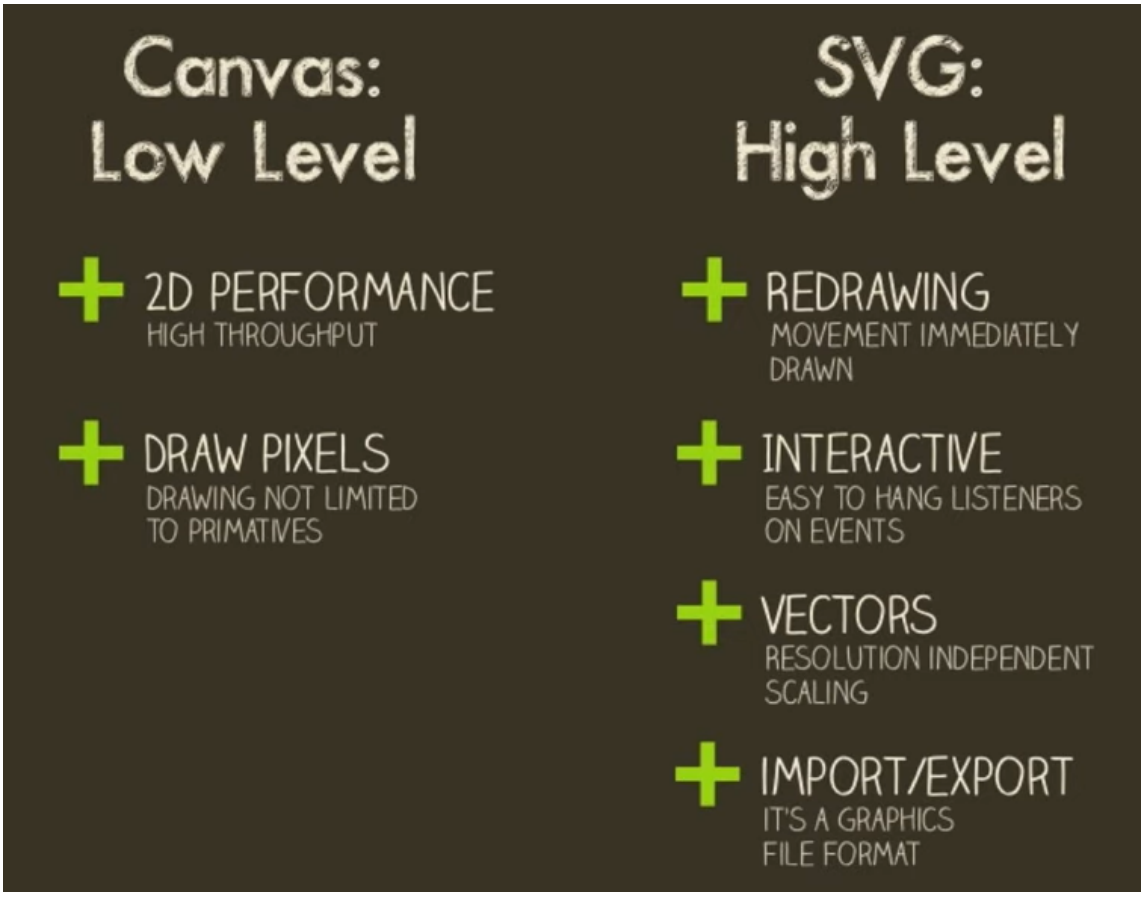
Сам по себе не несёт никакого смысла. Это просто холст для отрисовки растровой графики. Но он предоставляет нам доступ к Canvas API и WebGL API в JavaScript. С помощью этих API вы можете получить доступ к каждому отдельному пикселю в пределах холста и управлять его цветом в отдельности от других пикселей.

```
<canvas
  id="myCanvas"
  width="200"
  height="100"
  style="border:1px solid #d3d3d3;"
>Your browser does not support the HTML5 canvas tag.</canvas>
<script>
  var c = document.getElementById("myCanvas");
  var ctx = c.getContext("2d");
  ctx.beginPath();
  ctx.arc(95,50,40,0,2*Math.PI);
  ctx.stroke();
</script>
```



▼ Что такое `<svg>` и `<canvas>`?

SVG и canvas — это технологии, которые можно использовать для рисования чего-либо на веб-страницах.



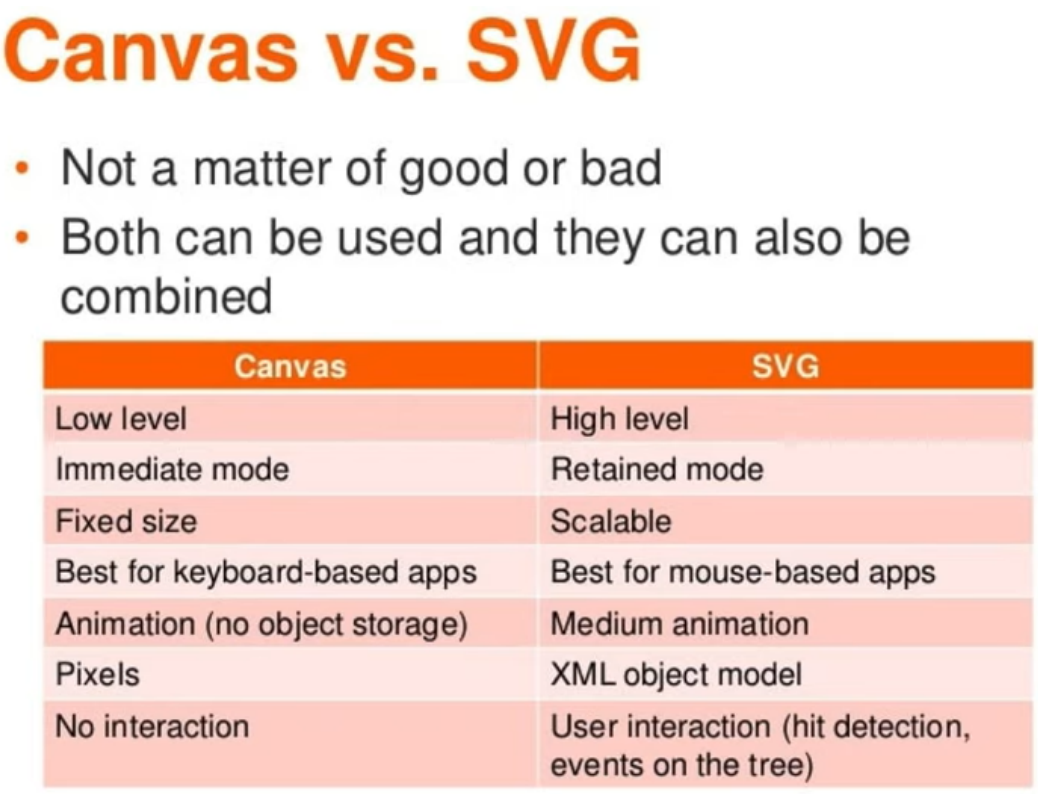
▼ **Разница между `<canvas>` и `<svg>`?**

- **canvas** растровая графика, скриптовый язык, нет отдельного формата, поддерживает обработчики события.
- **svg** векторная, язык разметки XML, формат svg, не поддерживает обработки событий.

Canvas	SVG
Single DOM element	Multiple DOM elements
Script language	Markup language
Raster graphics	Vector graphics
No file format	.svg file format
Must reset canvas to change drawn shape	Can edit shapes after drawing
Not accessible	Accessible
Doesn't support event handlers	Supports event handlers
Non-searchable, can't index	Searchable, can index
Non-compressible	Compressible
Can use hardware acceleration	No hardware acceleration

▼ **В каких случаях лучше использовать `<canvas>`, а в каких `<svg>`?**

- **canvas** следует использовать для редактирования изображения, визуализации данных, графиков, гистограмм, создание спрайтов и фонов.
- **svg** следует использовать для создания пользовательских интерфейсов веб приложениях независимых от разрешения экрана. Высокоинтерактивных анимированных пользовательских интерфейсов, графиков и даиграмм, редактирование векторных изображений.



▼ Плюсы и минусы `<canvas>` и `<svg>`?

canvas лучше всего подходит для создания растровой графики, редактирования изображений и операций требующих манипуляций на уровне пикселей

Плюсы:

- высокая производительность при отрисовки любых 2d эффектов
- стабильная производительность, которая падает только в случае увеличения изображения
- изображения можно сохранить в png или jpg, подходит для создания растровой графики.

Минусы:

- отрисовка основанная на пикселях
- не существует API для анимации. приходится использовать таймеры и разные события для отрисовки
- слабая возможность по рендерингу текста
- плохо с доступностью, т.к. это просто холст, без контекста. Но можно указать альтренативный текст который удет показан в боаузере если не получится использовать canvas

svg

Плюсы:

- нет зависимости от разрешения
- хорошо поддерживает анимацию
- контроль над каждый элементом с помошью svg dom api
- хранится в формате XML, таким образом можно получить доступ к внутренностям

Минусы:

- низкая скорость рендеринга при увеличении сложности т.к. используется DOM модель
- плохо подходит для игр.

▼ Для чего нужен атрибут `autocomplete`?

Этот атрибут помогает заполнять поля форм текстом, который был ведён ранее, т.е. нативная поддержка автозаполнения. Это дополнение можно отключать в настройках браузера.

▼ Что такое элемент `<output>` в HTML5?

HTML-элемент вывода (<output>) - является контейнерным элементом, в котором сайт или приложение могут выводить результаты вычислений или действий пользователя.

```
<form onsubmit="return false" oninput="o.value = parseInt(a.value) + parseInt(b.value)">
    <input name="a" type="number"> +
    <input name="b" type="number"> =
    <output name="o" />
</form>

// Using for
<output name="o" for="a b"></output>
```

▼ Что такое свойство `valueAsNumber`?

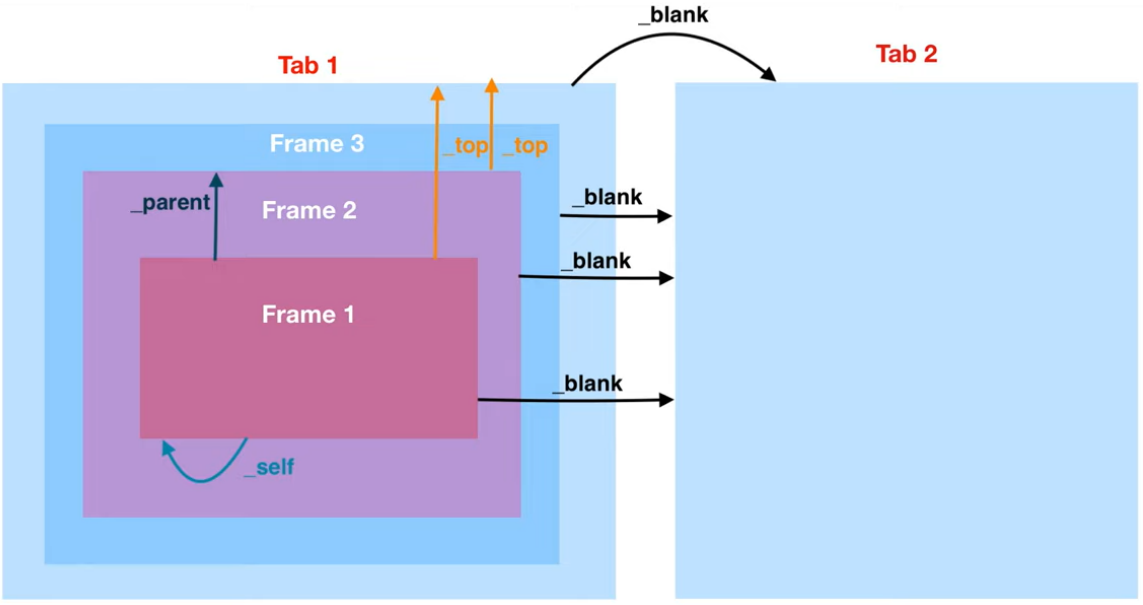
Это свойство возвращает значение в виде числа, а не строки, и данные отлично складываются или вычитаются без необходимости приведения к числовому типу.

```
<form onsubmit="return false" oninput="o.value = a.valueAsNumber + b.valueAsNumber">
    <input name="a" id="a" type="number" step="any"> +
    <input name="b" id="b" type="number" step="any"> =
    <output name="o" for="a b"></output>
</form>
```

▼ Что такое атрибут `target`? Какие значения он принимает?

Атрибут работает в связки с тегом ссылкой. По умолчанию при переходе по ссылке документ открывается в текущем окне, при необходимости это поведение можно изменить с помощью атрибута target.

- **target="_blank"** загружает страницу в новое окно браузера.
- **target="_self"** загружает страницу в текущем окне, значение по умолчанию.
- **target="_parent"** - загружает страницу во фрейм родителя, если фреймов нет то ведет себя как self.
- **target="_top"** - отменяет все фреймы и загружает страницу в текущем окне.



`target` — определяет, где откроется ссылка: в том же окне, в новой вкладке или в новом окне браузера. Без этого атрибута содержимое ссылки откроется в той же вкладке. Вот все варианты, где можно открыть URL-ссылку:

- `_self`: на той же странице. Значение по умолчанию, если именно оно вам нужно, то можно не указывать этот атрибут.
- `_blank`: в новой вкладке или в новом окне браузера — это зависит от настроек браузера, но чаще всего это именно вкладка.
- `_parent`: на родительской странице от текущей, то есть уровнем вложенности выше. Например, если на страницу вставлен фрейм, а внутри него такая ссылка, то она откроется не внутри фрейма, а на той странице, куда вставлен этот фрейм. Если родительской страницы нет, то ссылка откроется вместо текущей страницы.
- `_top`: в самой высокой «корневой» странице. Например, если есть страница, куда вставлен фрейм, в который вставлен фрейм, в котором ссылка с `target="_top"`, то ссылка откроется в самой-самой верхней странице, насколько глубоко она бы ни находилась.

▼ Что такое ApplicationCache в HTML5?

HTML5 вводит этот интерфейс, через него можно указать какие файлы надо кешировать и сделать доступными для офлайн пользователей, а это означает, что веб-приложение может сохранить данные в кэше и даже после перезагрузки страницы без подключения к интернету приложение сохранит функционал или по крайней мере его часть.

Кэш приложений дает приложению три преимущества:

1. **Автономный просмотр** - пользователи могут использовать приложение офлайн
2. **Скорость -** ресурсы кешируются локально, поэтому загружаются быстрее
3. **Снижение нагрузки на сервер** - браузер будет загрузжать с сервера только обновленные/измененные ресурсы

Затем при последующих посещениях, а также когда соединение с Интернет теряется, используются заранее сохраненные данные в специальном «хранилище»

«Хранилище» — условное название, используемое в браузерах для обозначения места хранения Application Cache. Механизмы работы Application Cache и стандартного кэша браузера

различны, поэтому требуют отдельного размещения. Отличия механизмов работы между двумя видами кэширования следующие:

- Данные помещённые в стандартный кэш могут быть автоматически, без команды со стороны пользователя или сервера, удаленны при его заполнении или истечения стока действия, указанного в заголовках файлов. Данные помещённые в хранилище Application Cache могут быть удалены только по команде пользователя или сервера.
- В стандартный кэш попадают только файлы, загруженные в процессе просмотра страницы. В Application Cache можно поместить любые файлы, загружаемые с сервера согласно инструкции manifest.

▼ Для чего используется элемент `<picture>` ?

Тег picture используется, когда для разных устройств или вариантов отображения, нужны разные картинки, он служит контейнером для одного или более элементов source и одного тега img, для обеспечения оптимальной версии изображения для разных версий экрана.

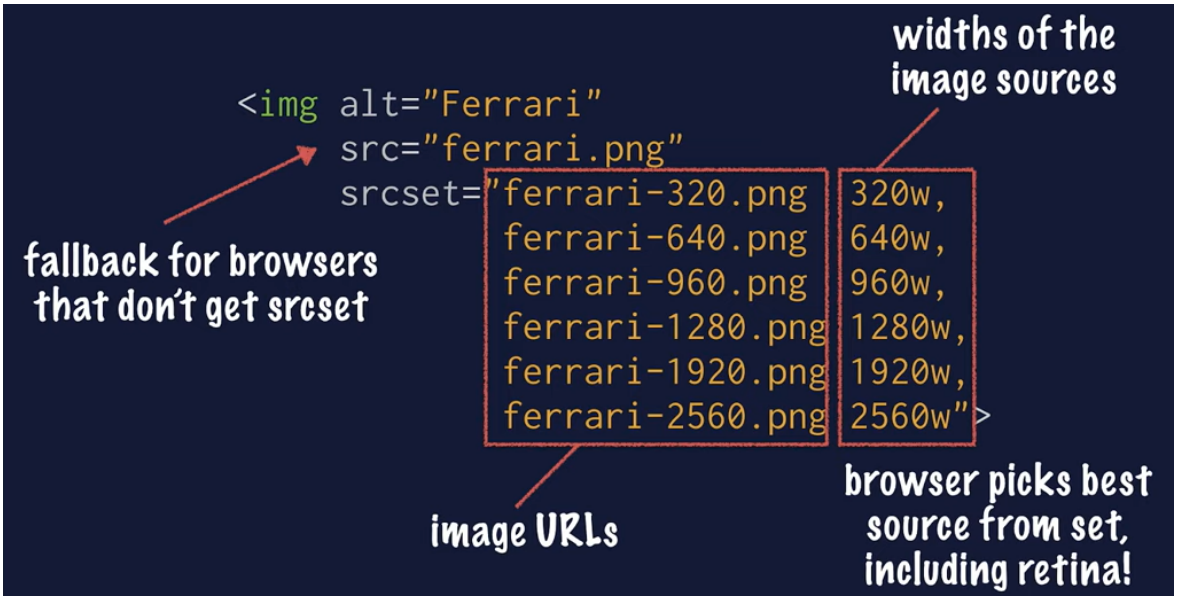
Браузер анализирует каждый тег <source> по порядку, останавливается на первом подходящем под текущие условия и отображает картинку из атрибута srcset. Другие теги <source> не анализируются. Если тег <picture> не поддерживается браузером или ни один из тегов <source> не подходит под условия, то отображается картинка из тега <img>.

```
<picture>
  <source media="(min-width: 1024px)" srcset="https://picsum.photos/600/600">
  <source media="(min-width: 768px)" srcset="https://picsum.photos/300/300">
  <source media="(min-width: 360px)" srcset="https://picsum.photos/100/100">
  <img src="https://picsum.photos/300/300" alt="Just an image">
</picture>
```

▼ Что такое `srcset` ? Как работает `srcset` ?

srcset - это атрибут тега img, который нужен для отображения разных картинок для разных устройств, пары url картинки и ширина разделены запятыми. Может работать в паре с атрибутом sizes. Также это атрибут тега source тега picture.

Подробнее тут



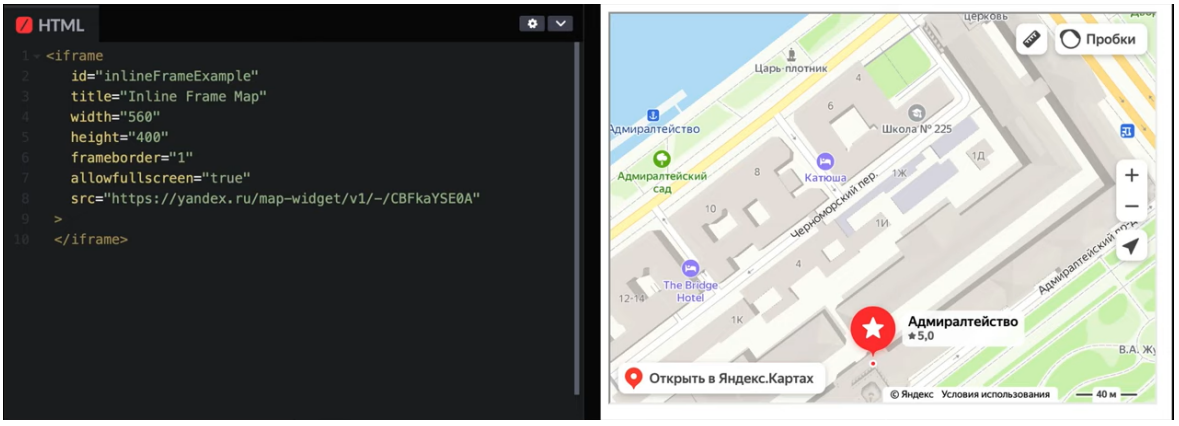
▼ Как семантически верно сверстать навигационное меню?

```
<nav>
    <ul>
        <li><a href=”#”>Главная</li>
        <li><a href=”#”>О нас</li>
        <li><a href=”#”>Контакты</li>
    </ul>
</nav>
```

▼ Что такое `<iframe>`?

iframe (inline frame) - это контейнер, который находится внутри обычного документа и позволяет вставить любой html документ из другого источника.

Часто это интерактивный элемент типа карты или видео. Содержимое внутри области существует независимо от остальной старницы. Стилями можно указать расположение фрейма на странице, но его содержимое стилизовать не получится.



▼ Для чего используются тэги `<sub>` и `<sup>`?

Эти теги относятся к текстовым тегам, они используется для выделения символов, а не слов, их применяют для указания единиц измерений и формул, а также для выделения специфических элементов.

▼ Как можно скрыть элемент разметки не используя CSS и JS?

Атрибут hidden для любого тега. Он становится невидим для пользователя и для читалок, вырывается из потока. Считается не лучшей практикой, лучше все стили задавать через CSS.

Или инлайновые стили т.к. технически мы не подключаем внешний CSS и все происходит в пределах HTML документа.

▼ Разница между `<meter>` и `<progress>`?

meter - числовое, возможно дробное значение в известном диапазоне, этот элемент не подходит для изменения чего-то вроде температуры. Браузеры, поддерживающие этот тег, отрисовывают его в виде прогресс-бара, заполненного в зависимости от значения атрибута `value` и раскрашенного в системные цвета.

Например сообщить о количестве свободного места на диске;вывести допустимые пределы громкости;показать уровень загруженности интернет-канала.

Возможные атрибуты:

- `value` — текущее числовое значение. По умолчанию равно `0`.
- `min` — нижняя граница диапазона. По умолчанию равно `0`.
- `max` — верхняя граница диапазона. По умолчанию равно `1`.
- `low` — определяет, что считать «низким значением». По умолчанию равно значению `min`.
- `high` — определяет, что считать «высоким значением». По умолчанию равно значению `max`.
- `optimum` — определяет оптимальное значение.

progress - используется для отображения хода выполнения задачи.

Тег `<progress>` стоит использовать для вывода информации о процессе, который выполняется и должен завершиться: прогресс загрузки файла, процесс соединения абонентов, длительность таймера.

💡Если пользователю надо показать числовое значение в заданном диапазоне, лучше использовать тег `<meter>`.

`max` — максимальное значение. Должно быть положительным, допускаются дробные значения. По умолчанию равно 1.

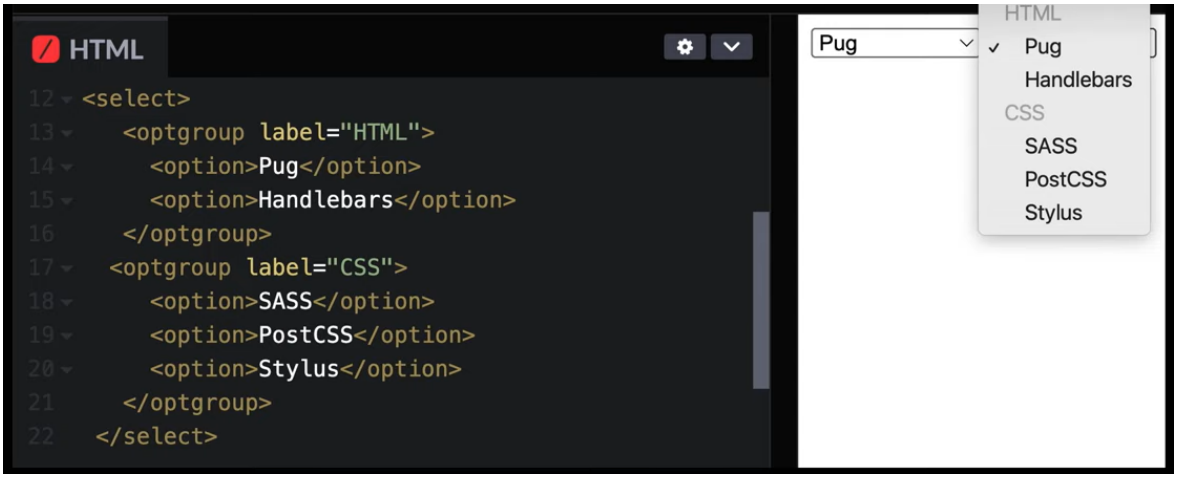
`value` — текущее значение. Положительное число, допускаются дробные значения. Должно находится в пределах между 0 и значением атрибута `max`. Также его можно менять при помощи JavaScript. Если атрибут не прописан, то линия внутри прогресс-бара будет перемещаться от одного края к другому, показывая, что задача выполняется, но не известно, сколько это займёт времени.

▼ Как можно сгруппировать опции внутри тэга `<select>`?

Нельзя вкладывать один optgroup в другой.

label добавляет обобщающий заголовок для сгурпиированных опций.

disabled - находящиеся внутри опции не будут доступны для выбора.

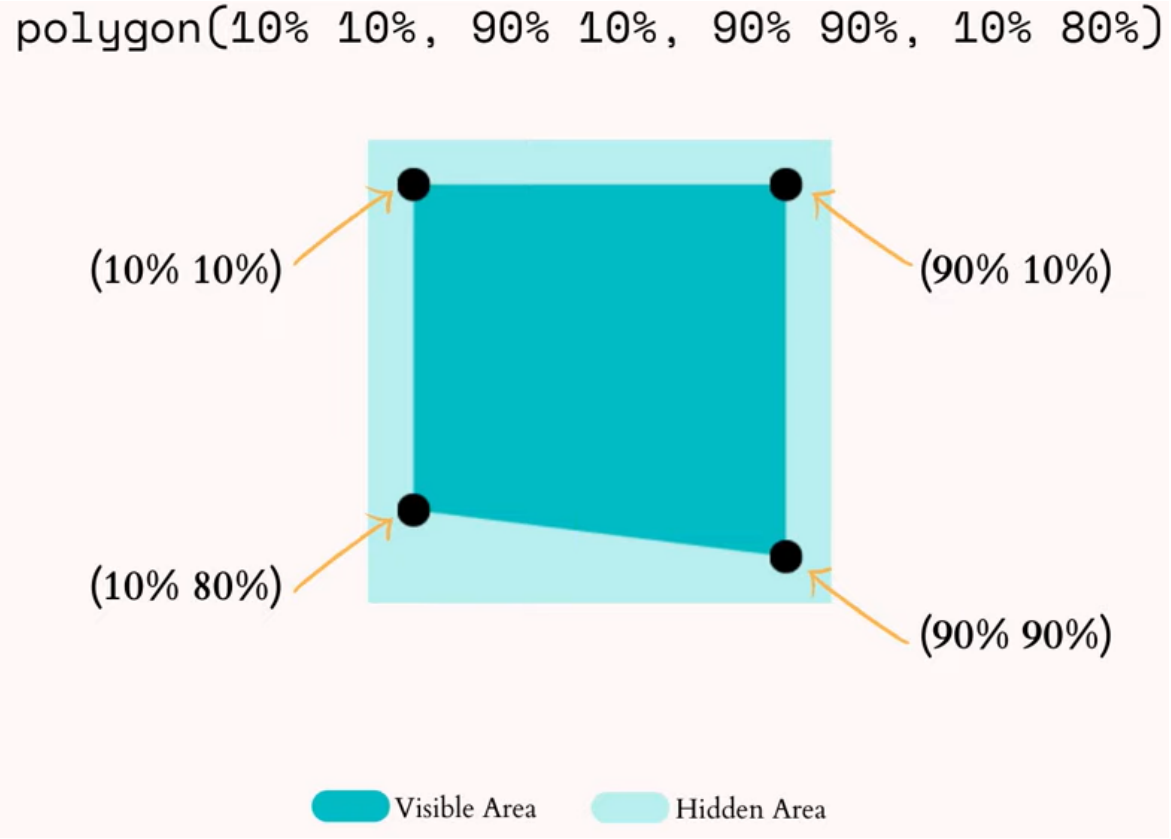



```
<select>
    <optgroup label='HTML'>
        <option>Pug</option>
        <option>Handlebars</option>
</optgroup>
    <optgroup label='CSS'>
        <option>SASS</option>
        <option>PostCSS</option>
```

```
        <option>Stylus</option>
</optgroup>
</select>
```

▼ Как можно изменить форму картинки или HTML элемента?

При помощи css свойства clip-path. Свойство `clip-path` задаёт видимую область изображения. Всё, что выходит за пределы указанной области скрывается.



Задавать область можно базовой формой:

- `inset` — четырёхугольник,
- `circle` — круг,
- `ellipse` — эллипс,
- `polygon` — многоугольник,
- `path` — сложная фигура по правилам заполнения SVG.

Хотя самая частая область применения — это изображения, но свойство `clip-path` может ограничивать и другие HTML-элементы

Областью фигуры может стать блоковая модель:

- `marginbox` — включает отступы,
- `borderbox` — по заданной рамке,
- `paddingbox` — по контенту, включая свойство `padding`,
- `contentbox` — по содержимому,
- `fillbox` — по ограничивающей рамке объекта,

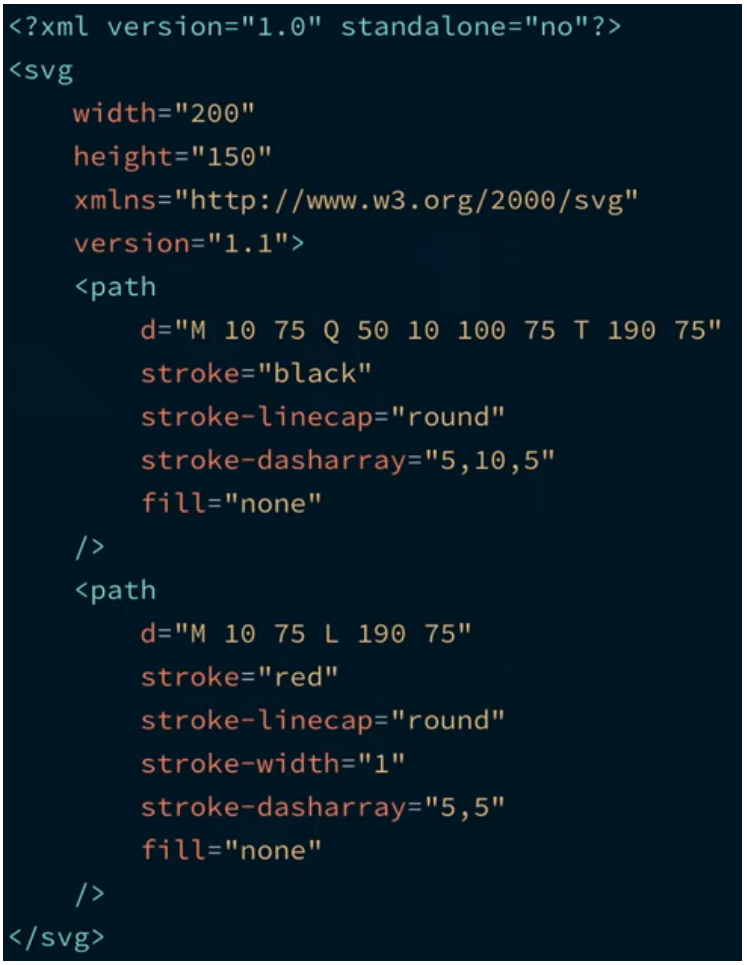
- `strokebox` — по обводке ограничивающей рамки,
- `viewbox` — по окну просмотра SVG.

▼ Чем отличается `<article>` от `<section>`?

- ** ** — контейнер общего назначения, не обязательно смысловой. Дивы используются для разметки мелких блоков, создания сетки и декоративных эффектов.
- **<section>** — более крупный логический контейнер, объединяющий содержание по смыслу. Например, блок «О компании», список товаров, раздел личной информации в профиле и так далее. Объекты располагающиеся внутри него объединены общим сыслом, задает раздел документа. Например блок новостей, контактная информация, список видео. Допускается вкладывать друг в друга.
- **<article>** — самостоятельный, цельный и независимый раздел документа. Этот раздел можно в неизменном виде использовать в различных местах, в том числе и на других сайтах. Примеры: статья, пост в блоге, сообщение на форуме и так далее. Обозначает независимую и самодостаточную часть страницы. Не рекомендуется вкладывать друг в друга.

▼ Расскажите об особенностях стилизации `<svg>`?

Изменение внешнего вида SVG-элементов отличается от стилизации стандартных HTML элементов т.к. основан на XML и стилевые свойтсва отличаются по названию. Например фон назначается свойством fill, а не background-color, граница - stroke вместо border.



```xml
<?xml version="1.0" standalone="no"?>
<svg
    width="200"
    height="150"
    xmlns="http://www.w3.org/2000/svg"
    version="1.1">
    <path
        d="M 10 75 Q 50 10 100 75 T 190 75"
        stroke="black"
        stroke-linecap="round"
        stroke-dasharray="5,10,5"
        fill="none"
    />
    <path
        d="M 10 75 L 190 75"
        stroke="red"
        stroke-linecap="round"
        stroke-width="1"
        stroke-dasharray="5,5"
        fill="none"
    />
</svg>
```

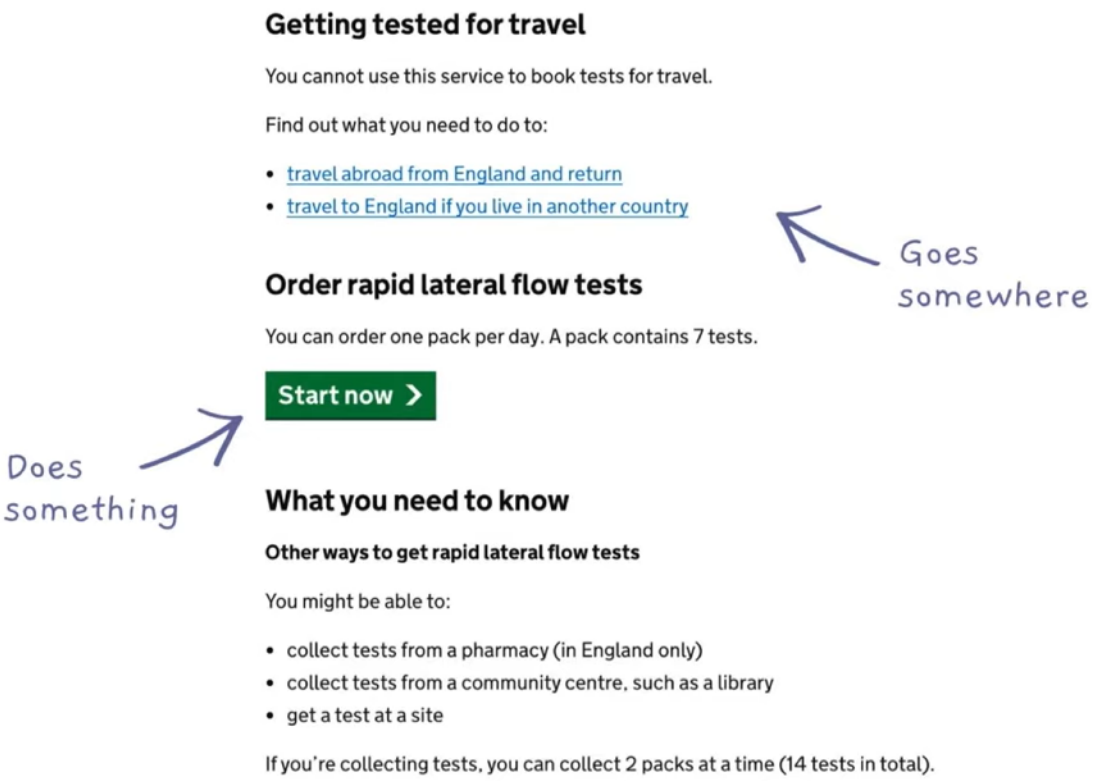
▼ Разница между кнопкой и ссылкой в HTML?

Кнопка (html тег <button>) — это функциональный элемент, то есть элемент, отвечающий за выполнение определенной функции.

- Получает фокус с клавитару по умолчанию
- Отправка и очистка формы
- Блокироваться через dsabled
- Добавляет подсказку скринридеру
- Показывает состояния focus, hover, hover, disabled
- Хорошо подходит для открытия модального окна, всплывающего меню, переключения интерфейса

Ссылка (html тег <a>) — это элемент навигации, то есть элемент, отвечающий за взаимосвязь разделов веб-ресурса (или разных ресурсов) между собой.

- Перевод на новую страницу
- Изменять url адрес
- Вызывать браузерные перериосви и перезагрузки
- Переходить по якорям внутри страницы
- Регистрировать клик по нажатию клавиши enter



▼ Для чего используется атрибут `decoding`?

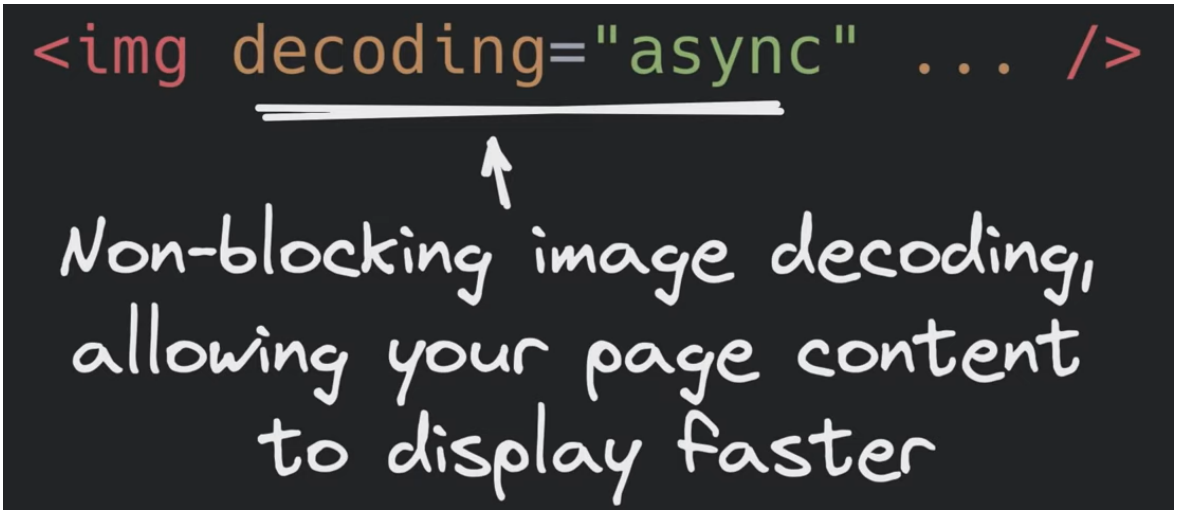
Этот атрибут используется в связке с тегом img, то есть с картинкой, он определяет как браузер должен декодировать изображение. По умолчанию декодирование проходит в основном потоке, что может помешать другим процессам проходящим на странице но тогда браузер рендерит поток более гадко, без скачков.

При async он обрабатывает изображение в параллельном потоке, поэтому рендер на старанице пропускает место в котором должна быть картинка т.к. не знает какого она будет размера, и когда она загрузиться в параллельном потоке будет скачок верстки т.к. она будет вставлена в зарезервированное для нее место без известных заранее размеров, что негативно сказывается на метрике CLS из web vitals .

Чтобы избежать скачка можно задать тегу img размеры прямо в разметке, тогда зарезервированное место будет готово заранее с указанной шириной и высотой.

Он принимает три основных значения:

- **async** вне основного потока.
- **sync** в основном потоке.
- **auto** по логике браузера.

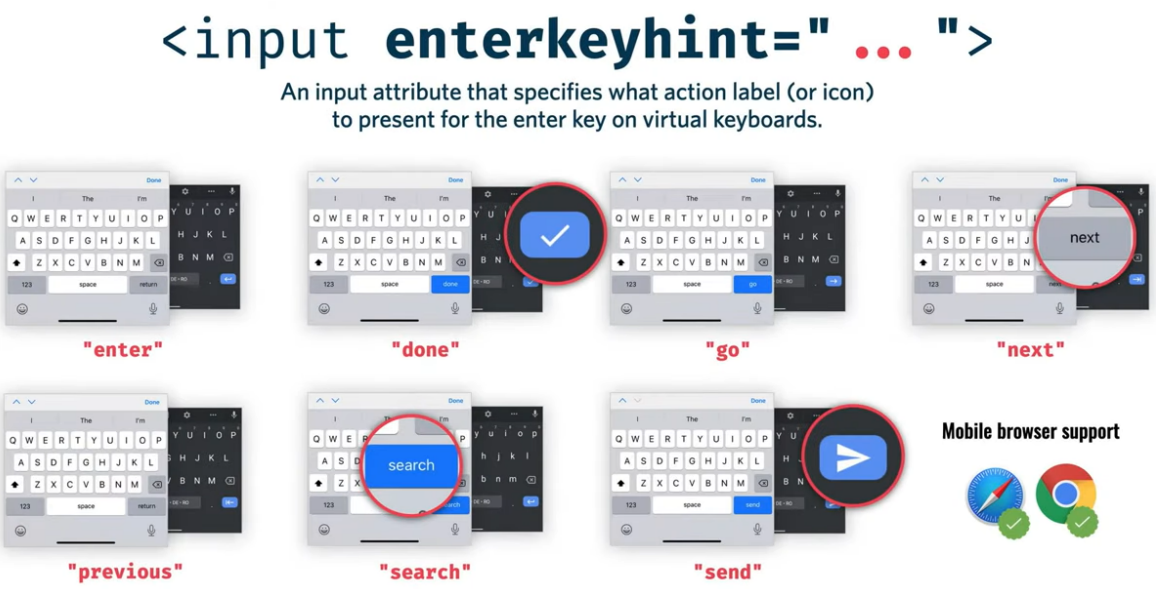


▼ Для чего используется атрибут `enterkeyhint` ?

Это атрибут тега input, он используется в связке с формой, он указывает какую метку действия или значок отобразить на кнопке на виртуальной клавиатуре.

Он может принимать одно из 7 основных значений:

- **enter** - вставка новой строки
- **done** - готово, закрыть поле ввода
- **go** - перевод к цели набранного текста
- **next** - переход к следующему текстовому полю
- **previous** - первод в предыдущее поле
- **search** - переход к результатам поиска введеднного текста
- **send** - отправить введеднный текст



▼ Для чего используют атрибут `novalidate` ?

Атрибут предназначен для отключения нативной валидации формы на стороне браузера, нативные правила будут проигнорированы и блокировка отправки формы при некорректных данных не стработает.

Применяется если форма валидируется с помощью JS и нужно избегать конфликтов с барузерной валидацией, в том числе чтобы убрать нативные подсказки и показывать кастомные.

▼ Для чего используют атрибут `inputmode`?

Это атрибут тегов input и textarea, он говорит браузеру на устройствах с экранной клавиатурой, какой набор символов показать при вводе данных в конкретное поле.

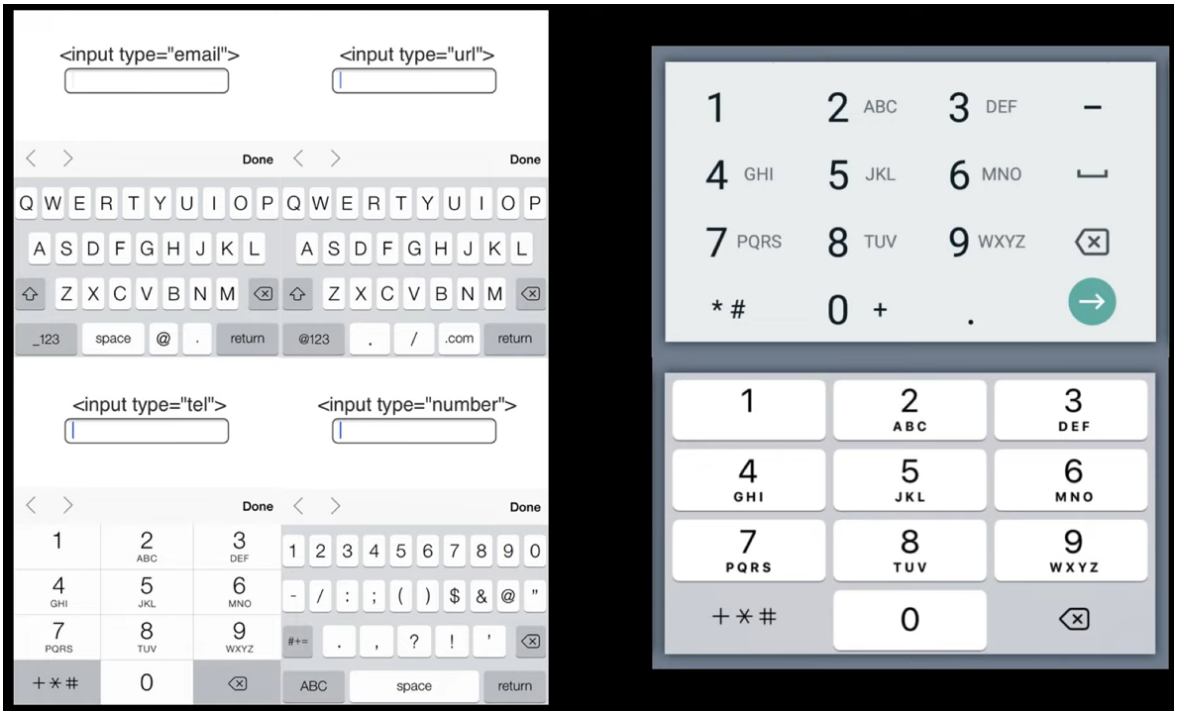
Важно понимать разницу между атрибутом `type` и атрибутом `inputmode`:

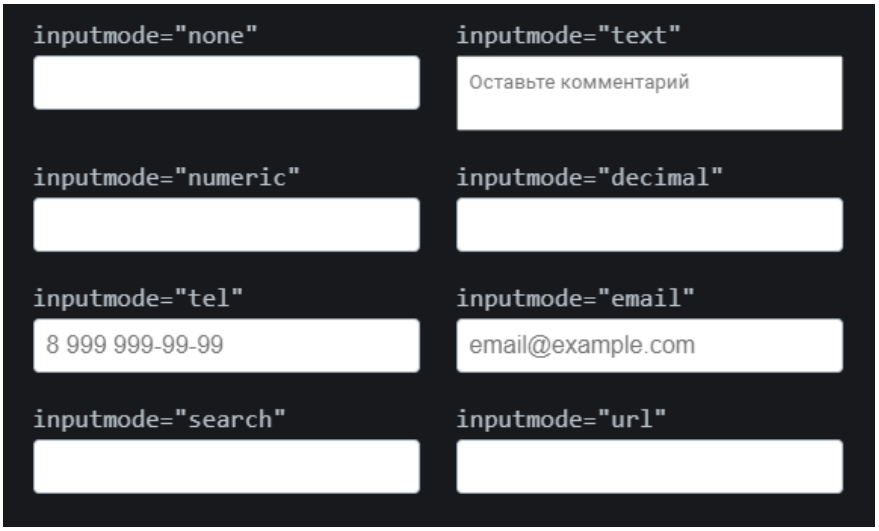
- атрибут `inputmode` только **подсказывает** браузеру, какой набор символов предложить пользователю для взаимодействия с полем ввода;
- атрибут `type` **устанавливает** тип данных, которые пользователь может ввести в поле ввода.

Подробнее тут

Принимает значения:

- email
- url
- tel
- number





▼ Для чего используется атрибут `pattern`?

Атрибут pattern определяет регулярное выражение, которому должно соответствовать значение элементов формы.

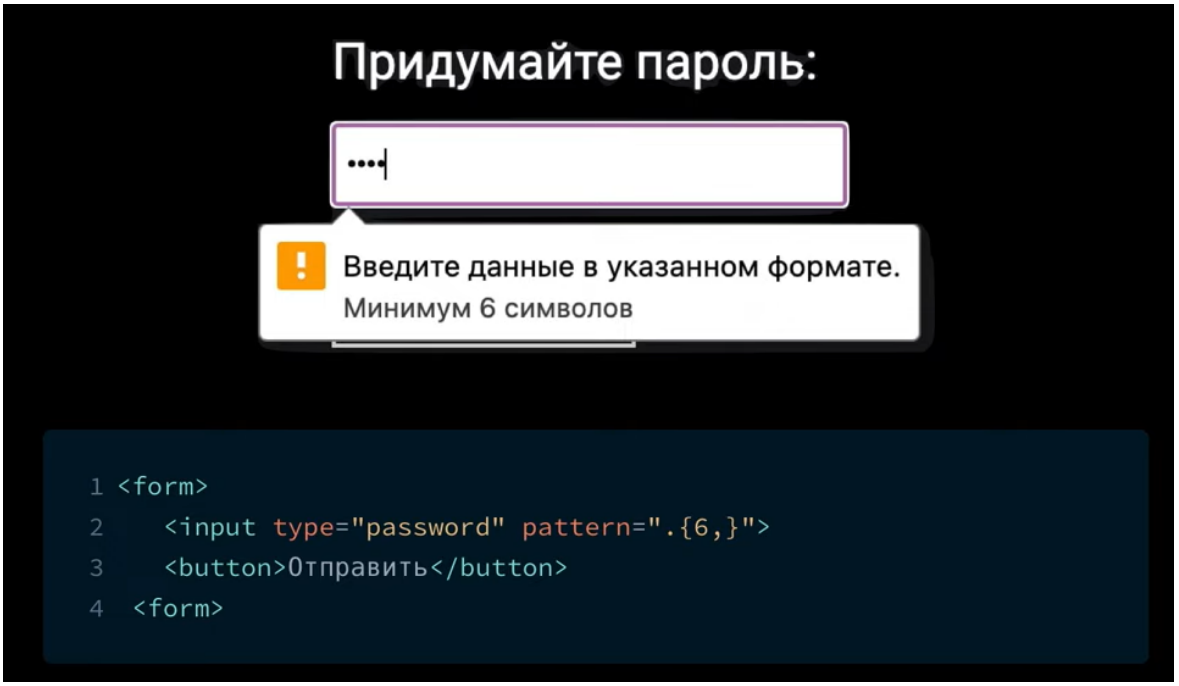
Атрибут `pattern` можно применить только к тегам `<input>` и только со следующими значениями атрибута `type`:

- `text`;
- `tel`;
- `email`;
- `url`;
- `password`;
- `search`.

Значением должно быть регулярное выражение, по которому браузер перед отправкой формы будет проверять то, что введено в поле. Если значение не соответствует регулярному выражению браузер покажет всплывающую подсказку с ошибкой. Механизм с ошибкой сработает только при отправке поля из «настоящей» формы с тегом `<form>`, без формы вам придётся проверять поле самостоятельно.

Текст всплывающей подсказки можно дополнить при помощи атрибута `title`. Большинство браузеров покажут этот текст вместе с ошибкой.

Лучше не писать слишком строгие паттерны для проверки значений. Вы никогда не сможете предугадать все возможные сценарии. Проверяйте только то, что действительно необходимо. К примеру, для валидации имейла достаточно проверить, что в тексте есть символ `@`

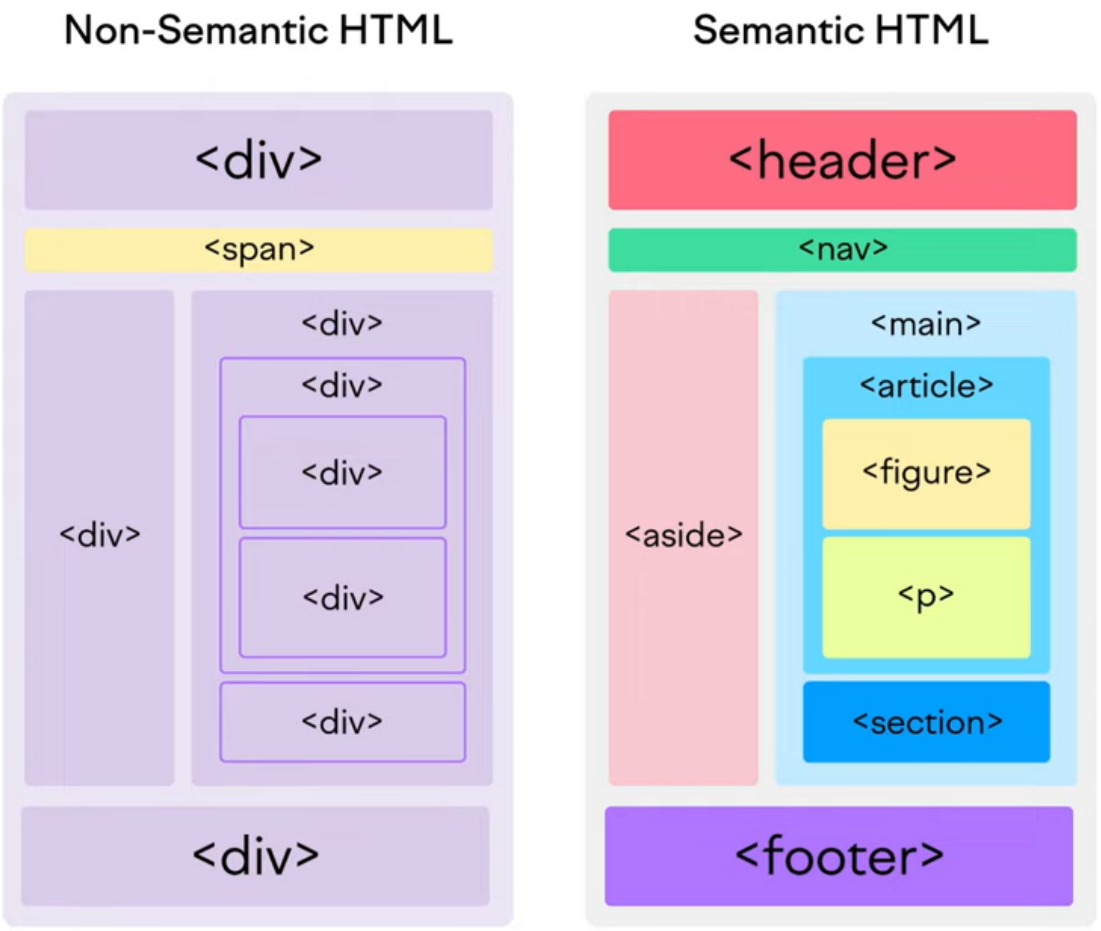


▼ Почему стоит использовать семантические теги в верстке?

Сематническая верстка опирается не на внешний вид сайта, а на смысловое предназначение каждого блока и логическую структуру документа.

Основные плюсы:

- Сделать сайт доступным
- Повысить SEO
- Теги прописаны в стандарте HTML5, для каждого тега указана роль, рекомендуется использовать именно семантические теги вместо div и span
- Удобнее с точки зерния разработки, удобнее читать код



▼ **Для чего используется тэг `<label>`?**

Тег label определяет текстовую метку для элементов

- textarea,
- input,
- select,
- checkbox,
- radio.

При клике на label фокус перемещается на связанный элемент формы.

```html
<!-- using "for" for a "label" -->
<form>
   <label for="male">Male</label>
   <input type="radio" name="sex" id="male">
   <br>
   <label for="female">Female</label>
   <input type="radio" name="sex" id="female">
 </form>

 <!-- Using "label" as a wrapper -->
 <form>
   <label>
     Male
     <input type="radio" name="sex" id="male">
   </label>
   <br>
   <label>
     Female
     <input type="radio" name="sex" id="female">
   </label>
 </form>
```

CSS

▼ Что такое CSS? И для чего он используется?

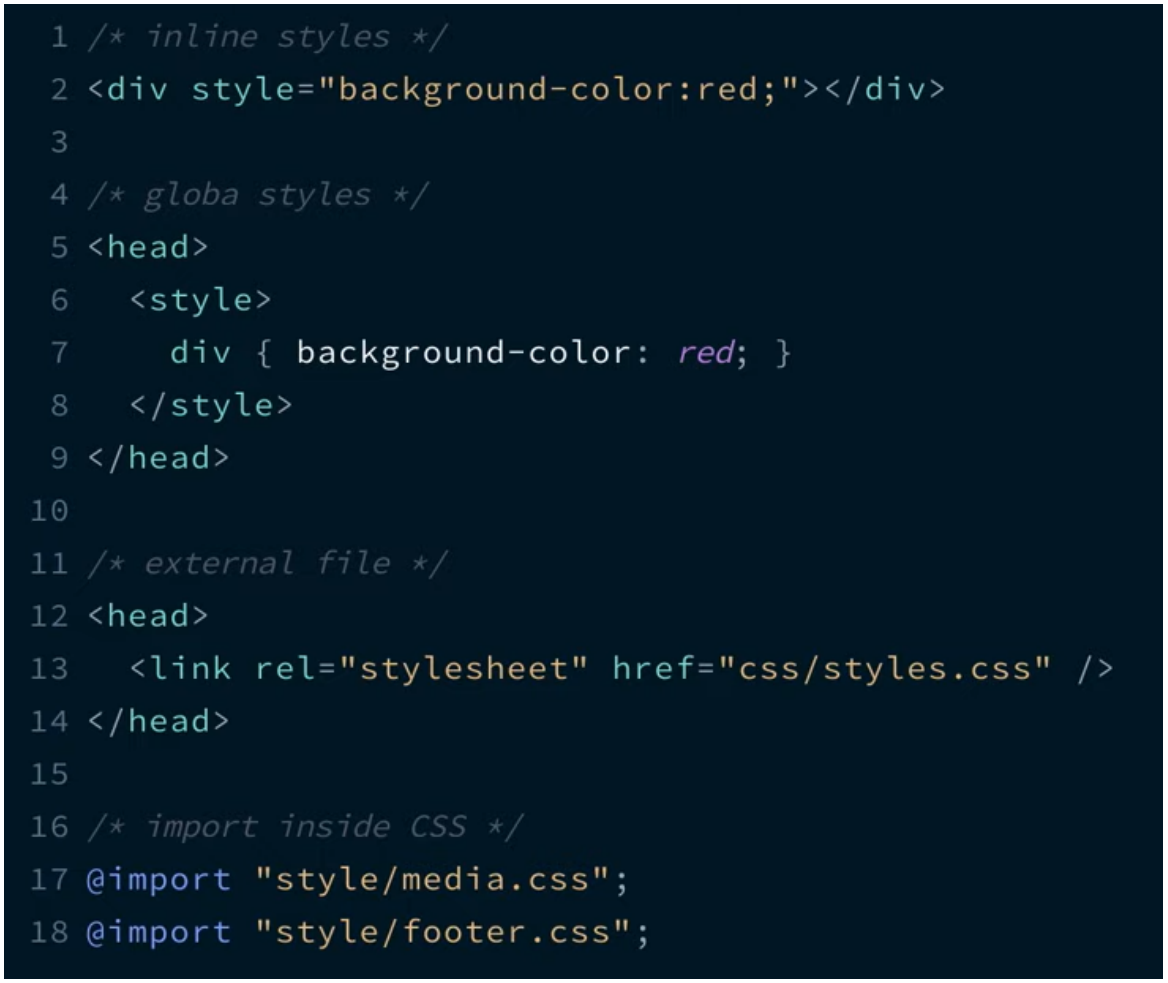
CSS - это каскадная таблица стилей, которая предназначена для добавления css стилей на страницу.

▼ Что такое CSS-правило?

CSS-правило - формируется из двух элементов selector и declaration, который имеет в себе все стили.

▼ Варианты добавление CSS стилей на страницу?

1. Инлайновый стиль, атрибут style="".
2. Использование глобальный стилей через тег <style>.
3. Стилизация во внешнем файле.
4. Импорты в самих файлах стилей.



```
/* inline styles */
<div style="background-color:red;"></div>

/* globa styles */
<head>
  <style>
    div { background-color: red; }
  </style>
</head>

/* external file */
<head>
  <link rel="stylesheet" href="css/styles.css" />
</head>

/* import inside CSS */
@import "style/media.css";
@import "style/footer.css";
```

▼ Типы позиционирования в CSS?

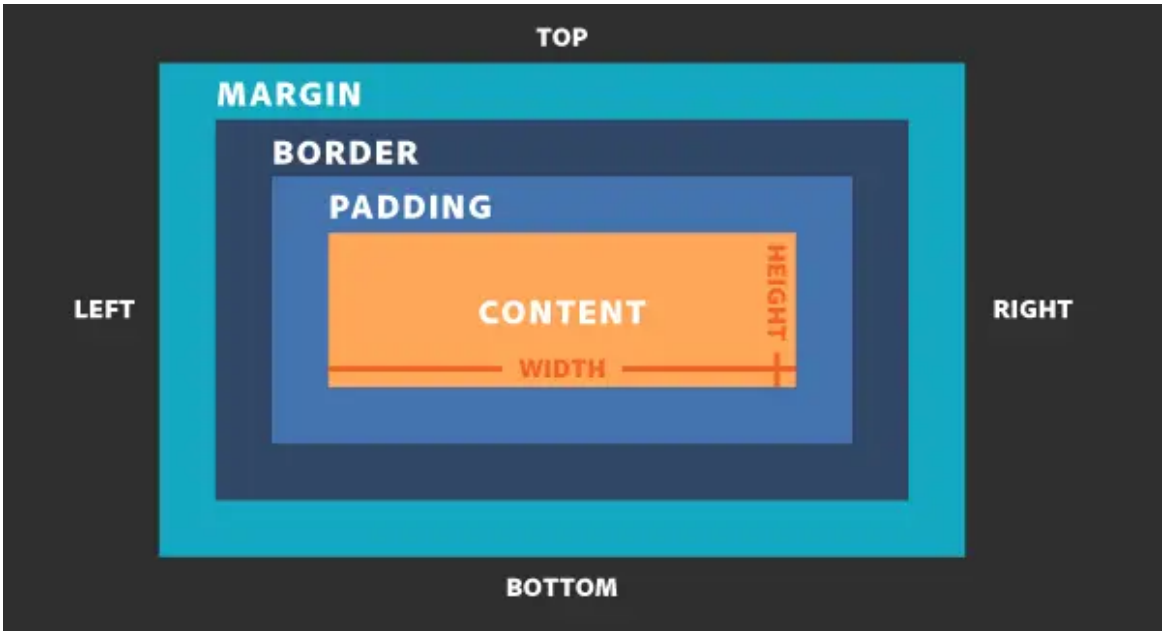
Подробнее тут

- **static** тип позиционирования по умолчанию.
- **relative** относительное позиционирование при котором элемент можно перемещать относительно его начального положения в документе.
- **absolute** абсолютное позиционирование при котором элемент можно перемещать, позиционирование происходит относительно родительского элемента у которого свойство position отличается от дефолтного, если таких нет, то относительно всего браузера.
- **fixed** позиционирование только относительно окна браузера.
- **sticky** комбинирует два вида позиционирование, внутри видимой области браузера ведёт себя, как fixed, при последующей скроле скролится вместе с родителем, что напоминает relative.

▼ Блочная модель CSS?

Блочная модель позволяет рассчитать итоговое пространство элемента, в неё входит контент, padding, border, margin.

Блочная модель, она же box model — это алгоритм расчёта размеров каждого отдельного элемента на странице, которым браузеры пользуются при отрисовке. Чтобы точно понимать, каким в итоге получится блок и сколько места он займёт



Блочная модель состоит из нескольких CSS-свойств, влияющих на размеры элемента:

- `width` — ширина элемента;
- `height` — высота элемента;
- `padding` — внутренние отступы от контента до краёв элемента;
- `border` — рамка, идущая по краю элемента;
- `margin` — внешние отступы вокруг элемента.

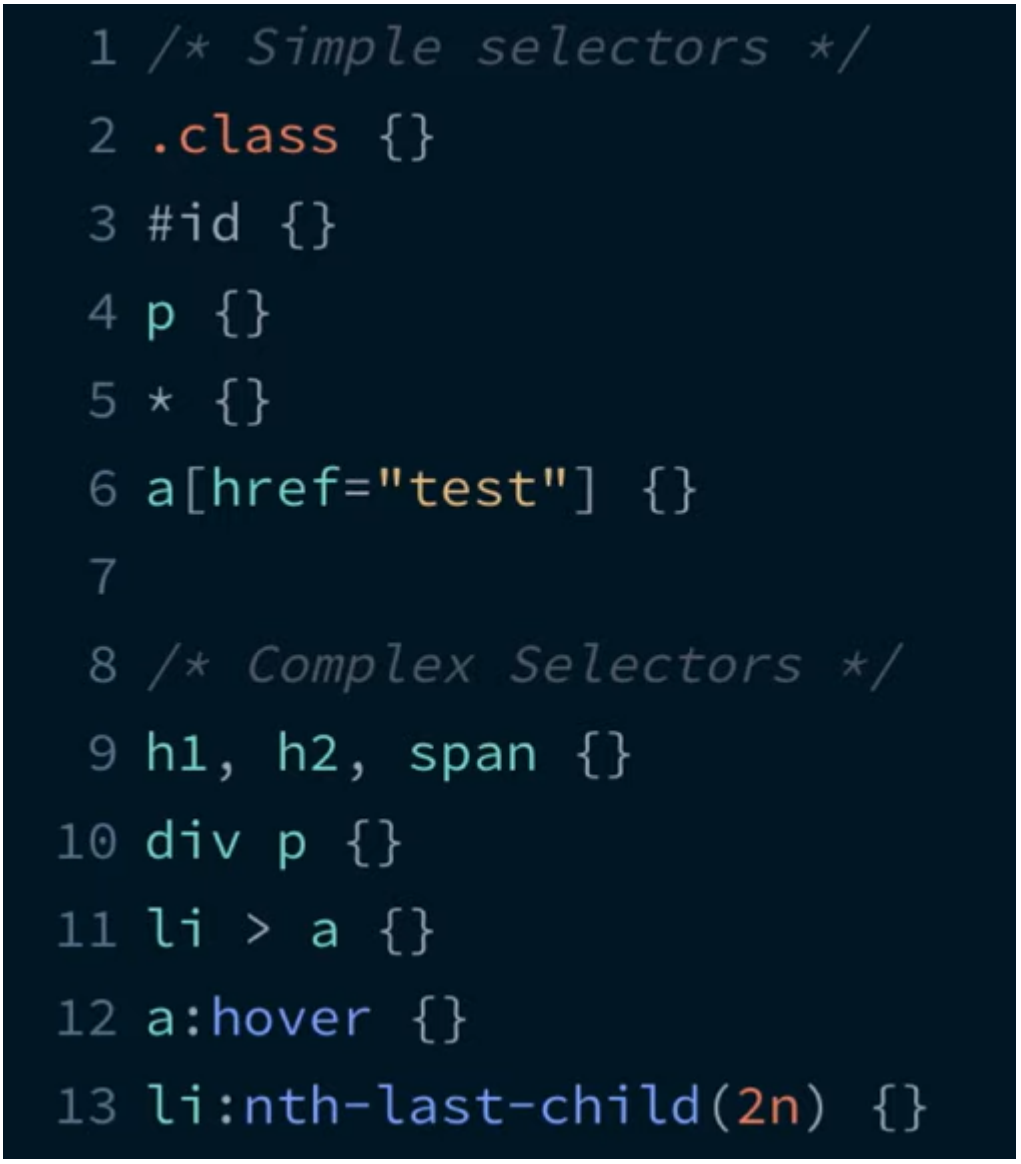
По умолчанию элементы с блочным отображением (`display: block`) занимают всю ширину родителя, если явно не задано другое. А вот высота элемента подстраивается под контент.

Элементы со строчным (`display: inline`) или строчно-блочным (`display: inlineblock`) отображениями по умолчанию подстраивают и ширину, и высоту под вложенный контент. Однако строчно-блочному можно и произвольно задать размеры: ширину (`width`) и высоту (`height`).

▼ Что такое селектор? И какие селекторы существуют?

Подробнее тут

Селектор - это часть css правил, который сообщает браузеру какому элементу или элементам веб страницы будет применен стиль.

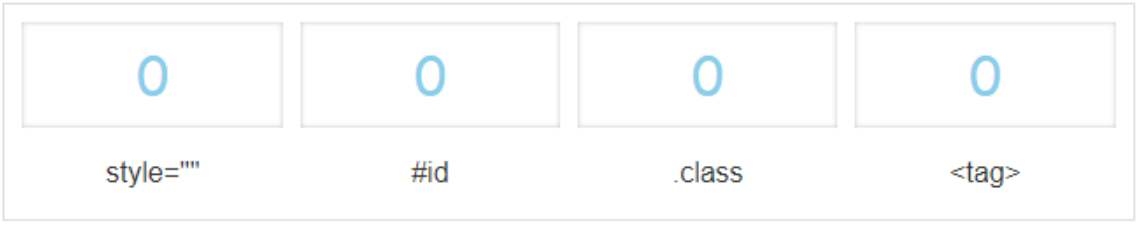


▼ Что такое специфичность селектора? Как считать вес селектора?

Подробнее тут

Специфичность - это способ, с помощью которого браузеры определяют, какие значения свойств CSS наиболее соответствуют элементу и, следовательно, будут применены.

Специфичность селектора рассчитывается по 4-м позициям:



Вес селекторов (по убыванию):

- style="" `1,0,0,0`
- #id `0,1,0,0`
- .class `0,0,1,0`
- [attr=value] `0,0,1,0`
- tag `0,0,0,1`
- * `0,0,0,0`

У стилей, заданных в атрибуте `style`, на первой позиции будет единица — `1,0,0,0`. Это самая высокая специфичность, которая перевешивает свойства, заданные другими способами.

Переопределить стили, заданные в `style`, можно дописав `!important` к значению свойства в таблице стилей.

Обратный вариант — универсальный селектор `*`, он не имеет веса: `0,0,0,0`.

Примеры:

- `LI 0,0,0,1` — селектор по тегу
- `UL LI 0,0,0,2` — селектор с двумя тегами весит больше, чем с одним.
- `.orange 0,0,1,0` — селектор с классом весит больше, чем селектор с тегом.
- `.orange A SPAN 0,0,1,2` — селектор перевесит предыдущий, потому что помимо класса содержит два тега.
- `#page .orange 0,1,1,0` — селектор с ID перевесит всё, кроме inline-стилей.

Теперь сравним селекторы из исходного примера:

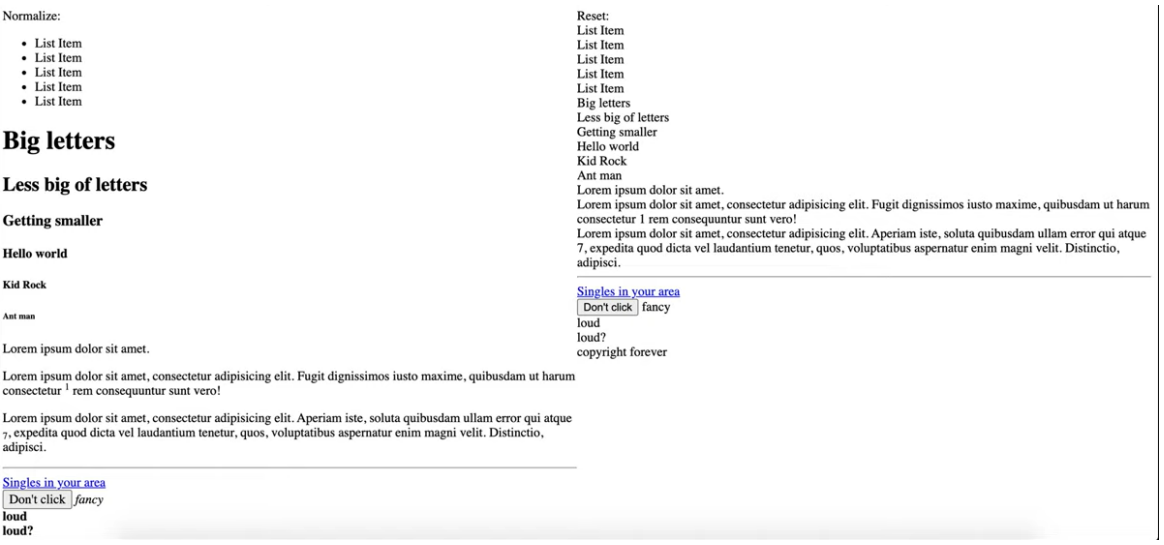
- `#container A 0, 1 ,0, 1`
- `.list A 0,0, 1 , 1`

`0, 1 ,0, 1` **>** `0,0, 1 , 1` — хорошо видно, что селектор с ID весит больше, чем селектор с классом, поэтому все ссылки имеют оранжевый фон, хотя ниже в коде им задан зеленый.

▼ Разница между Reset.css и Normalize.css?

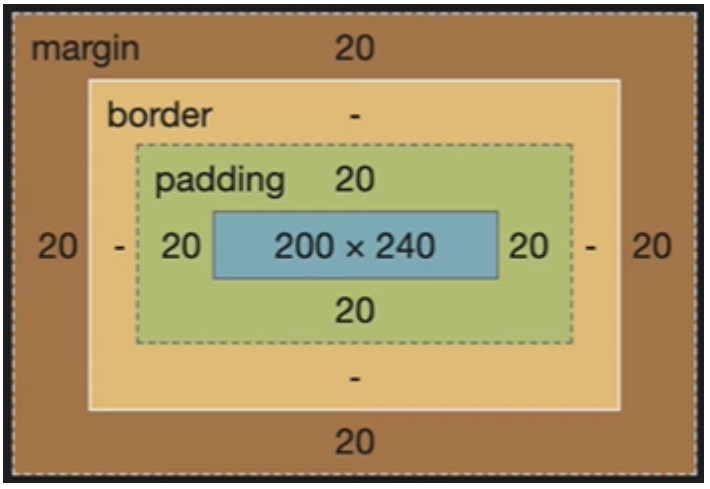
Просто CSS файл подключаемый в начале документа.

- Reset - сброс всех стилей.
- Normalize - приведение всего к единому виду во всех браузерах.



▼ Разница между `margin` и `padding`?

- **margin** это внешний отступ, пространство от границы блока до другого элемента.
- **padding** это внутренний отступ от границы блока до контента.



▼ Разница между `display: none` и `visibility: hidden`?

- **display: none** полностью убирает с HTML страницы и удаляется с потока документов, единственное место где он остается быть доступным это DOM дерево. Не доступен для поисковых роботов.
- **visibility: hidden** - на HTML странице он не виден, но из основного потока он не вырывается и занимает своё место. Дотупен для поисковых роботов.

```
<!-- Styles -->
<style>
  .first { display: none; }
  .second { visibility: hidden; }
</style>

<!-- Markup -->
<body>
  <h2 class="first">Hello world</h2>
  <h2 class="second">It's a webDev channel</h2>
</body>
```

▼ Разница между блочным и строчным (инлайновым) элементами?

Элементы с display: block занимают всю доступную ширину на странице и начинаются с новой строки. Они могут иметь заданные значения ширины и высоты, а также вертикальные и горизонтальные отступы.

Элементы с display: inline занимают только необходимое пространство для отображения содержимого и не начинают новую строку. Они не

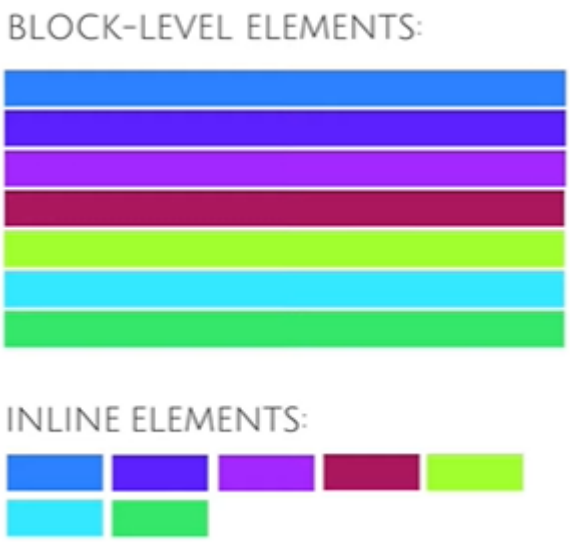
могут иметь заданных значений ширины и высоты, а также вертикальных отступов, но могут иметь горизонтальные отступы.

Элементы с display: inline-block сочетают в себе свойства элементов с display: block и display: inline. Они занимают только необходимое пространство для отображения содержимого, но могут иметь заданные значения ширины и высоты, а также вертикальные и горизонтальные отступы. Они не начинают новую строку.

`none` — полностью скрывает элемент со страницы, не удаляя его при этом из HTML-разметки.

`block` — элемент ведёт себя как блочный.

`inlineblock` — элемент ведёт себя снаружи как строчный, а внутри как блочный.



▼ Разница между классом и идентификатором в CSS?

class - есть негласное правило, для добавления стилей используем class, а для добавления логики id, это связано с определёнными качествами каждого, к примеру id уникально и может встречаться всего 1 раз на странице и id у элемента может быть только одно, class в свою очередь можно указывать сколько угодно.

T.e. class для CSS, a id для JS

▼ Что такое CSS спрайт? И для чего он используется?

CSS спрайты - это техника оптимизации загрузки веб-страниц, которая заключается в объединении нескольких изображений в один файл(чаще для наборов иконок) и использовании CSS свойств для отображения

нужной части изображения на странице. Это позволяет уменьшить количество запросов к серверу и ускорить загрузку страницы. Например предзагрузить иконки котрые показываются при наведении, тогда не будет видно мигания при ожидании загрузкт картинки.

▼ Что такое вендорные префиксы? И для чего они используются?

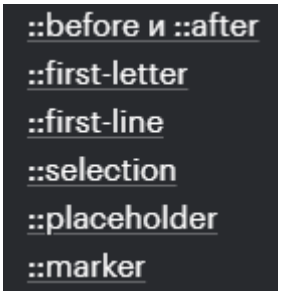
Вендорные префиксы — это приставки перед свойствами, селекторами, функциями или другими сущностями в CSS, позволяющие браузерам внедрять экспериментальные фичи до того, как они полностью стандартизированы и готовы для использования. Когда префикс отбрасывается — это знак, что всё готово.

Самый простой способ проверить поддержку свойства — найти его на сайте Can I use. Там, в том числе, написано, какие префиксы для каких браузеров нужны.

```
/* Vendor Prefixes */
.class {
  -webkit-opacity: 0.5; /* Chrome, Safari */
  -moz-property: 0.5;   /* Firefox */
  -ms-property: 0.5;    /* Internet Explorer & Edge */
  -o-property: 0.5;     /* Opera */
  property: 0.5;
}
```

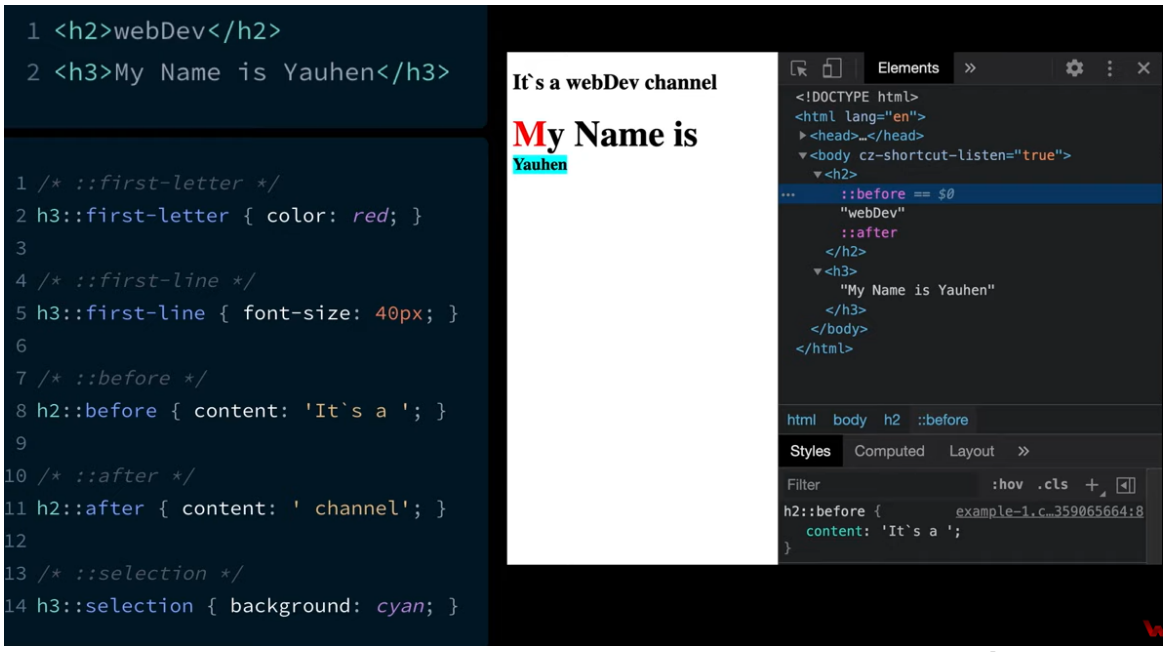
▼ Что такое псевдоэлементы? И для чего они используются?

Псевдоэлементы — это элементы, которых не существует в HTML-разметке. Они создаются и позиционируются исключительно при помощи CSS. Чаще всего используются для создания различных декоративных элементов (которые не несут содержательного смысла).



Также псевдоэлементы приходят на помощь, когда нужно наложить поверх картинки так называемый оверлей (перекрывающий слой). На этом польза от псевдоэлементов не заканчивается.

`content` — обязательное свойство псевдоэлементов `::before` и `::after`



▼ Что такое схлопывание границ (margin collapsing)?

Схлопывание границ - это механизм взаимодействия отступов по вертикали, отступы не суммируются а объединяются между собой. В результате итоговое расстояние получается равным большему из margin

Если блоку задан нижний отступ, а следующему за ним — верхний, то можно ожидать, что итоговый отступ между блоками будет равен сумме этих двух отступов. Но в соответствии со спецификацией соприкасающиеся отступы «схлопываются». То есть как бы проваливаются один в другой. Итоговый отступ будет равен бóльшему отступу из двух.

Происходит только по вертикали.

Выпадение отступов из родителя можно предотвратить несколькими способами:

- Задать родителю вертикальный внутренний отступ `paddingtop` или `paddingbottom` в зависимости от того, с какой стороны мы хотим предотвратить выпадение.
- Задать родителю верхнюю или нижнюю рамку по такой же логике. Рамка может быть прозрачной, главное, чтобы она была
- Задать родителю свойство `overflow` со значением, отличным от `visible`.
- Переопределить родителю свойство `display` на `flex` или `grid`.
- Сделать оба элемента inline-block.

▼ Что такое CSS препроцессор?

CSS-препроцессор - это программа (sass, less, stylus) которая позволяет генерировать css из собственного уникального синтаксиса, то есть на выходе мы всегда будем получать сгенерированный css код который отправляется в браузер, ну а сами css препроцессоры очень активно применяются к разным проектам, потому что у них есть удобный функционал:

- Переменные
- Вложенность, когда один селектор можно вкладывать в другой
- Комбинирование селекторов вложенности когда итоговое имя генерируется на основании вложенности
- Переиспользование кода с помощью миксинов
- Функции и операторы
- Импорты и наследование
- Условные выражения и циклы
- Модульность и повторное использование, опять же благодаря импортам и наседованию

▼ Что такое `z-index` ? Как формируется контекст наложения?

Контекст наложения — это концепция трехмерного расположения элементов по оси z относительно пользователя, смотрящего на экран. Самый базовый контекст наложения, существующий на любой странице формируется корневым элементом <html>. Все элементы внутри этого контекста сортируются и располагаются с оглядкой друг на друга. Но мы можем создавать контексты наложения не только на странице целиком, но и в каждом отдельном блоке. Тогда вложенные в него дочерние блоки будут сортироваться и располагаться уже по правилам этого нового, родительского контекста наложения.

По умолчанию контекст наложения формируется в порядке следования элементов на странице. Если же на элементе явно указано позиционирование то он будет перекрывать собой элементы без явного указания позиционирования.

z-index срабатывает для элементов с позиционированием (свойство position), отличающимся от статичного (значения relative, absolute, fixed, sticky), а на не позиционированных элементах ничего не произойдет

Но мы можем создавать контексты наложения не только на странице целиком, но и в каждом отдельном блоке. Тогда вложенные в него дочерние блоки будут сортироваться и располагаться уже по правилам этого нового, родительского контекста наложения.

Новый контекст наложения формируется если:

- это корневой элемент (`<html>`),
- элемент позиционирован абсолютно (`position: absolute`) или относительно (`position: relative`) со свойством `zindex` , значение которого не `auto` ,
- флекс-элемент со свойством `zindex` , значение которого не `auto` и чей родительский элемент имеет свойство `display: flex` или `display: inlineflex` ,

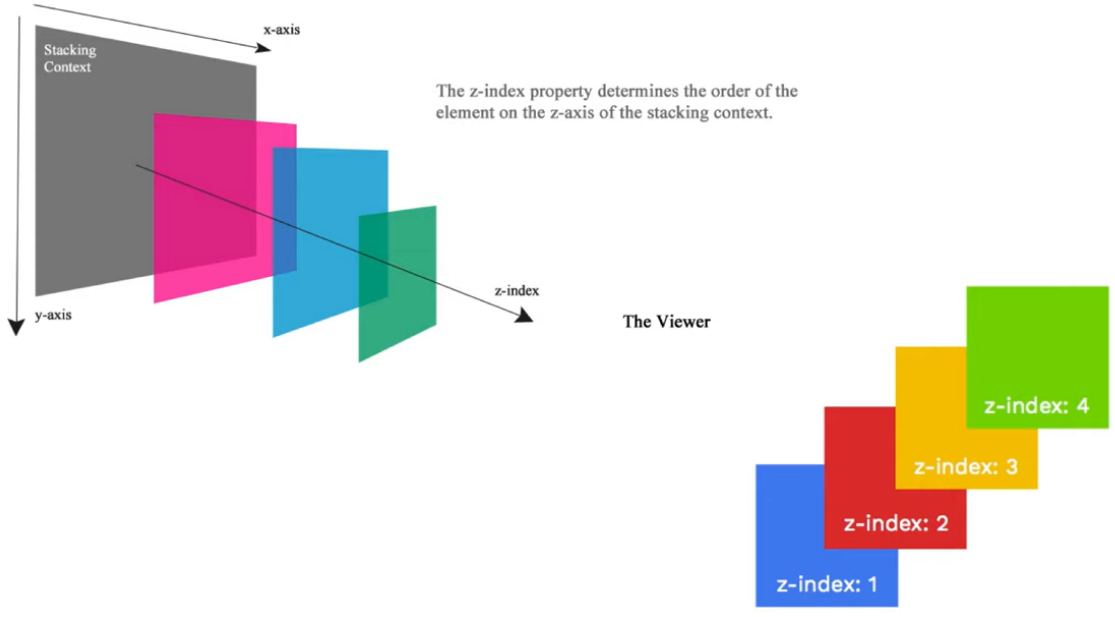
- элементу задано свойство `opacity` со значением меньше `1`,
- элементу задано свойство `transform` со значением не `none`,
- элементу задано свойство `mixblendmode` со значением не `normal`,элементу задано свойство `filter` со значением не `none`,
- элементу задано свойство `isolation` со значением `isolate`,элемент с `position: fixed`,
- элементу задано свойство `willchange` или аналогичный атрибут,
- элементу задано свойство `webkitoverflowscrolling` со значением `touch`.

Остальные элементы, не создающие собственный контекст наложения, используют родительский контекст.

Обычно достаточно запомнить первые три сценария и чуть-чуть помнить про следующие два. Если браузер рисует что-то, чего вы не ожидали, можно всегда вернуться и подсмотреть остальные.

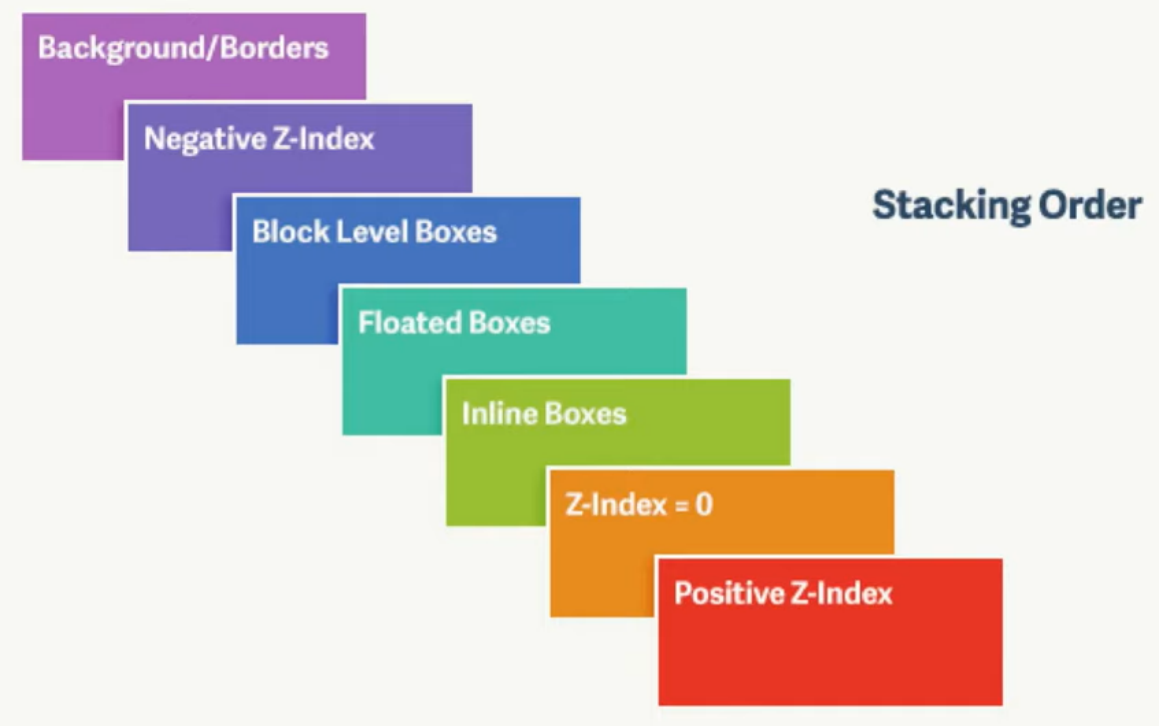
Стоит запомнить одну важную вещь, связанную с контекстом наложения: элементы могут сортироваться находясь на одном уровне внутри контекста наложения. Элементы внутри блока никогда не могут быть выше или ниже своего родителя.

Но мы можем не подчиняться стандартным правилам браузера и устанавливать свой порядок наложения элементов друг на друга с помощью **z-index**.



▼ Порядок наложения элементов в CSS (Stacking Order)?

- Background, Borders.
- z-index < 0.
- Блочные элементы в нормальном потоке(static).
- Float элементы.
- inline элементы.
- z-index = 0
- z-index > 0



▼ Как с помощью CSS определить, поддерживается ли свойство в браузере?

Для проверки используется директива **@supports**, которая работает аналогично media запросам, т.е. при срабатывании выполняется код в фигурных скобках. Часто используется в подходе progressive enhancement

▼ Как поддерживать страницы в браузерах с ограниченными функциями?

Для CSS нельзя писать полифилы.

- Использование Modernizr.
- Использование @support.
- Autoprefixer для вставки браузерных префиксов.
- Использование ресурса caniuse.com.
- Graceful degradation, или Progressive enhancement.

▼ Как исправлять специфичные проблемы со стилями для разных браузеров?

- Использование Autoprefixer.
- Подключение Reset CSS или Normalize CSS.
- Разделение стилей для разных браузеров.
- Использование сторонних библиотек.
- Тестирование приложения в специализированных приложениях типа Browserstack.

▼ Глобальные ключевые слова в CSS?

Для всех свойств помимо стандартных значений можно задать глобальные свойства.

Подробнее тут

Есть 4 глобальных ключевых слова:

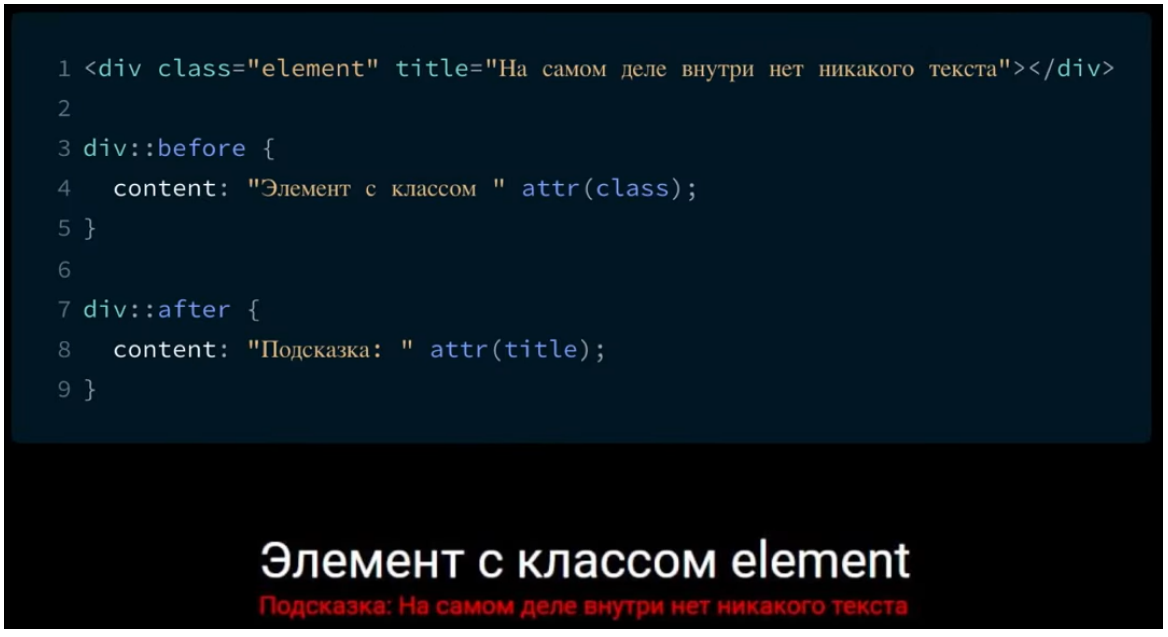
- **initial** сбрасывает все указанные значений свойств до значений по умолчанию.
- **inherit** css свойство делится на наследуемые и не наследуемые, значение наследуемых свойств применяются не только к элементу ну и для всех вложенных дочерних элементов.
- **unset** это ключевое слово ведёт себя по разному с наследуемыми свойствами и не наследуемыми свойствами. С наследуемыми свойствами ведёт себя как inherit, а с не наследуемыми, как initial.
- **revert** данное ключевое слово сбрасывает значение свойство до указанного в стилях браузера.

Также можно сбросить сразу все стили через all, например all: revert

▼ Что такое CSS-атрибут (`attr`)?

Функция attr, которая умеет получать значения любого атрибута элемента, а потом использовать это значение прямо в таблице стилей.

Функцию `attr()` можно использовать в качестве значения любого CSS-свойства, однако полностью поддерживается только свойство `content`. Для остальных свойств поддержка экспериментальная и может различаться от браузера к браузеру. Актуальную информацию о поддержке можно посмотреть на Can I use.



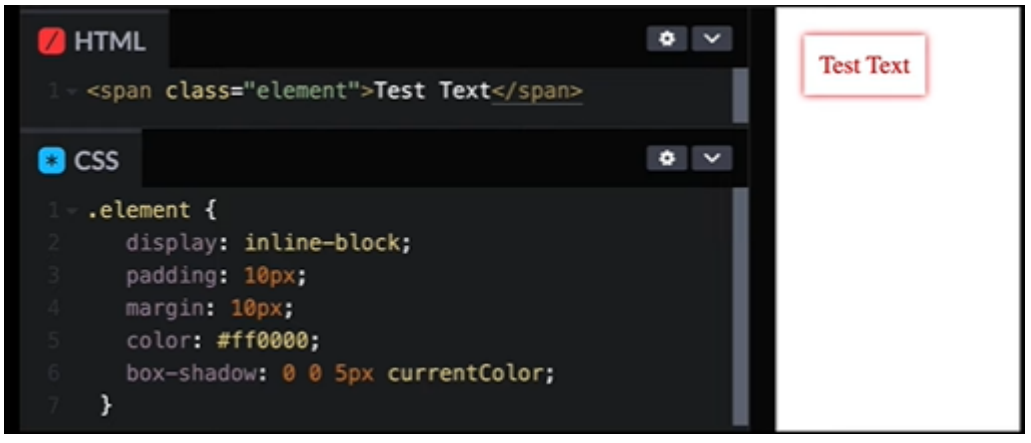
▼ Что такое перечисление селекторов?

При помощи перечисления нескольких селекторов через запятую можно избежать дублирования кода если у элементов есть повторяющиеся стили.

```css
.main-title,
.title {
  position: relative;
  color: #ffffff;
  font-weight: 500;
}

h1, h2, h3, h4, h5, h6 {
  margin: 0;
}
```

▼ Для чего используется ключевое слово `currentColor` в CSS?

Ключевое слово currentColor можно использовать в качестве значения для CSS-свойства, принимающего цвет. Например, color, background-color, box-shadow. Браузер подставит вместо currentColor текущее значение свойства color. Что позволяет при изменении основной палитры автоматически изменить зависящие через currentColor цвета, например цвет тени или фона.



▼ Какие псевдоклассы были добавлены в CSS3?

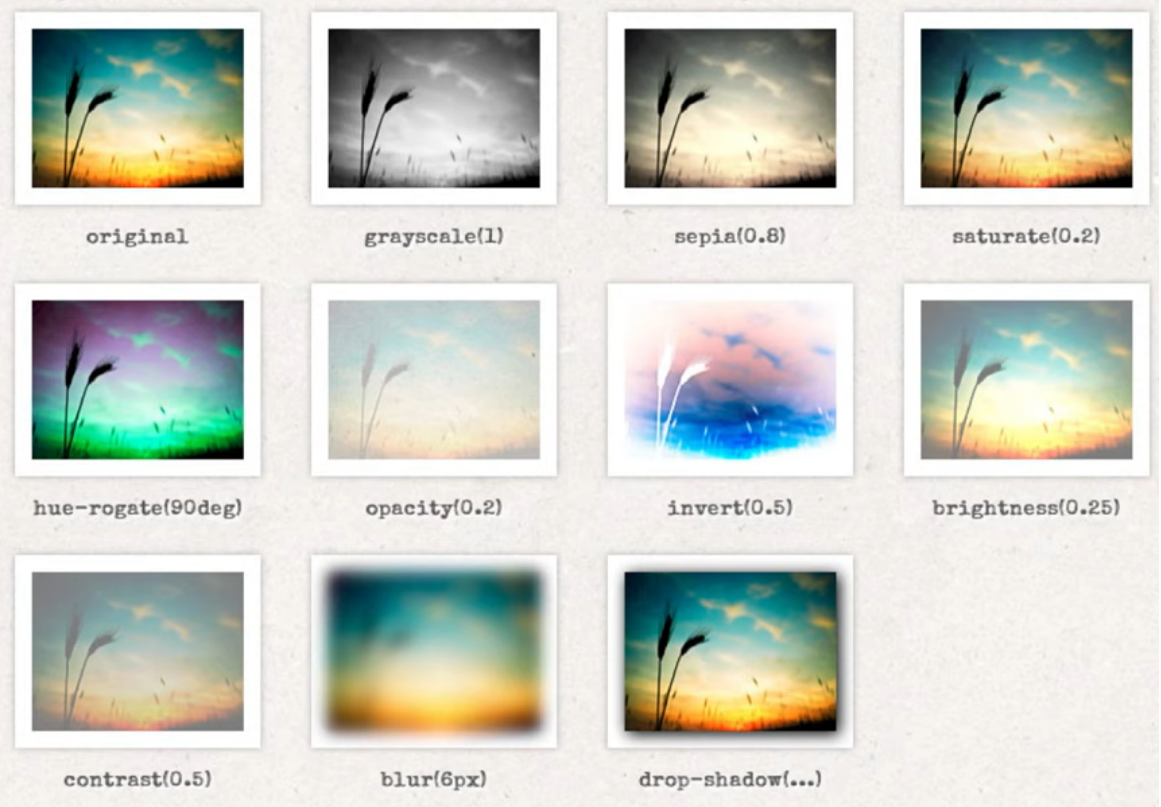
- :nth-child(n)

- :nth-last-child(n)
- :only-child
- :nth-of-type(n)
- :first-of-type
- :last-of-type

▼ Какие фильтры есть в CSS?

CSS позволяет накладывать различные фильтры, это помогает размыть или обесцветить изображения.

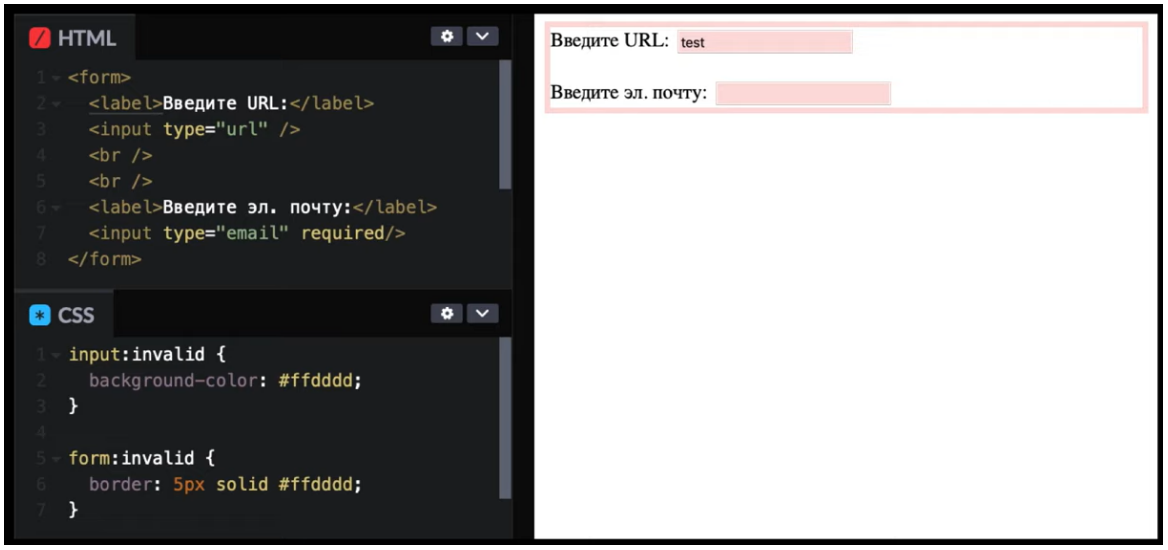
- grayscale
- opacity
- blur
- и т.д.



▼ Для чего используется псевдокласс `:invalid`?

Подробнее тут

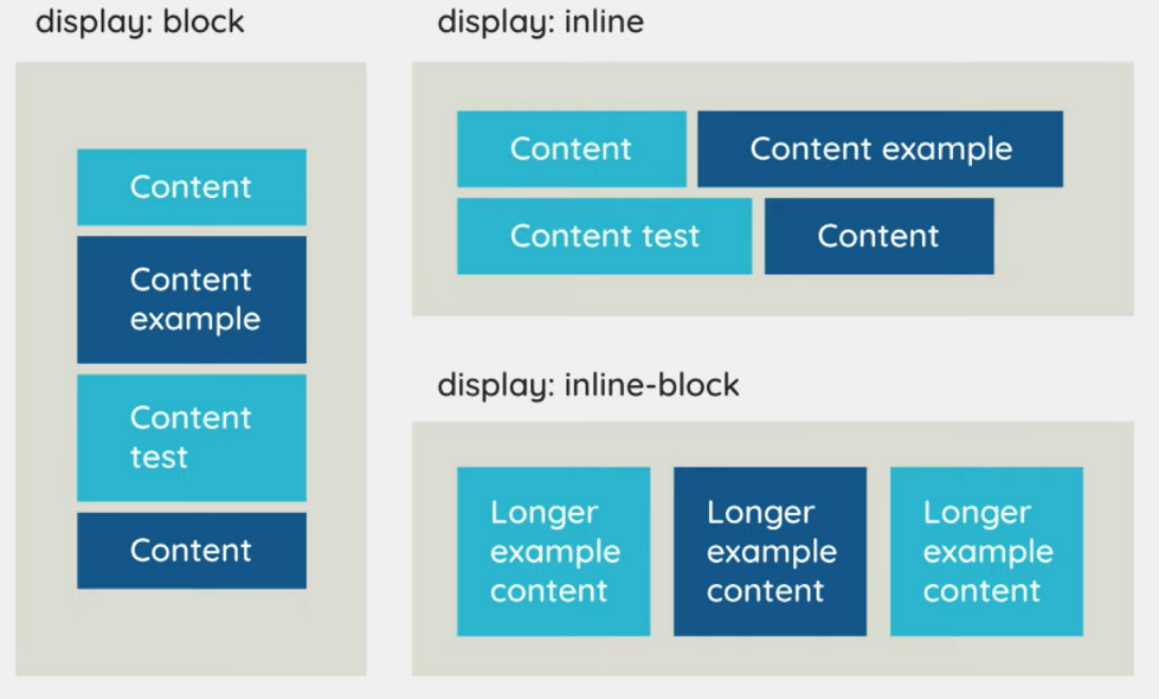
Псевдокласс :invalid CSS работает для тегов <form>, <fieldset>, <input> или другой <form>элемента, содержимое которого не проходит проверку. Это нативный способ подсветить ошибку не добавляя дополнительных классов для ошибок валидации.



▼ Расскажите про свойство `display` в CSS?

По умолчанию все элементы HTML бывают строчными или блочными.

- block
- inline
- inline-block
- flex
- inline-flex
- grid
- table
- none

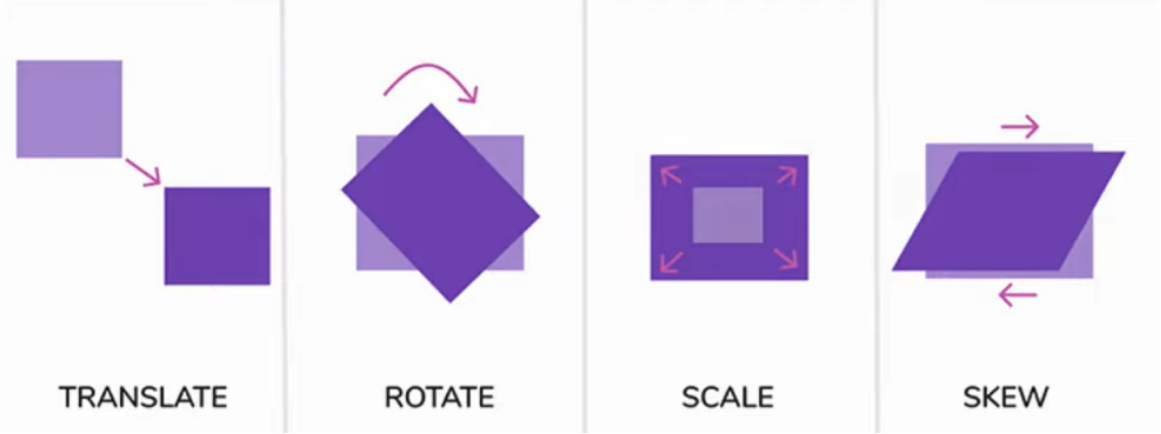


▼ В каком случае лучше использовать `translate()` вместо абсолютного позиционирования?

При анимациях лучше использовать **translate()**, браузер хорошо ее оптимизирует в том числе выносит в отдельный контекст наложения в котром происхдят все изменения и к тому же использует аппаратное ускорение(мощности видеокарты, а не процессора). Абсолютное позиционирование привязывается к позициям пикселей, а translate использует субпиксельную интерполяцию.

Some css properties which are GPU accelerated (not always):

- filter
- opacity
- transform



▼ Что такое плавающие элементы (floats)? Как они работают?

1. Элемент вынимается из потока документа и сдвигается в заданном направлении до того как коснется границы родителя, либо другого float элемента.
2. Если пространства по X не хватает, то смещается вниз пока не начнет помещаться
3. Блочные элементы не видят float элементы т.к. он убран из основного потока.
4. Строчные элементы обтекают float элементы.

▼ Расскажите о свойстве `text-rendering`?

Свойство text-rendering, говорит браузеру какие оптимизации нужно производить с текстом во время рендера. Например объединять несколько символов в лигатуры.

auto - бразер сам решает когда важнаее скорость, читаемость или геометрическая точность

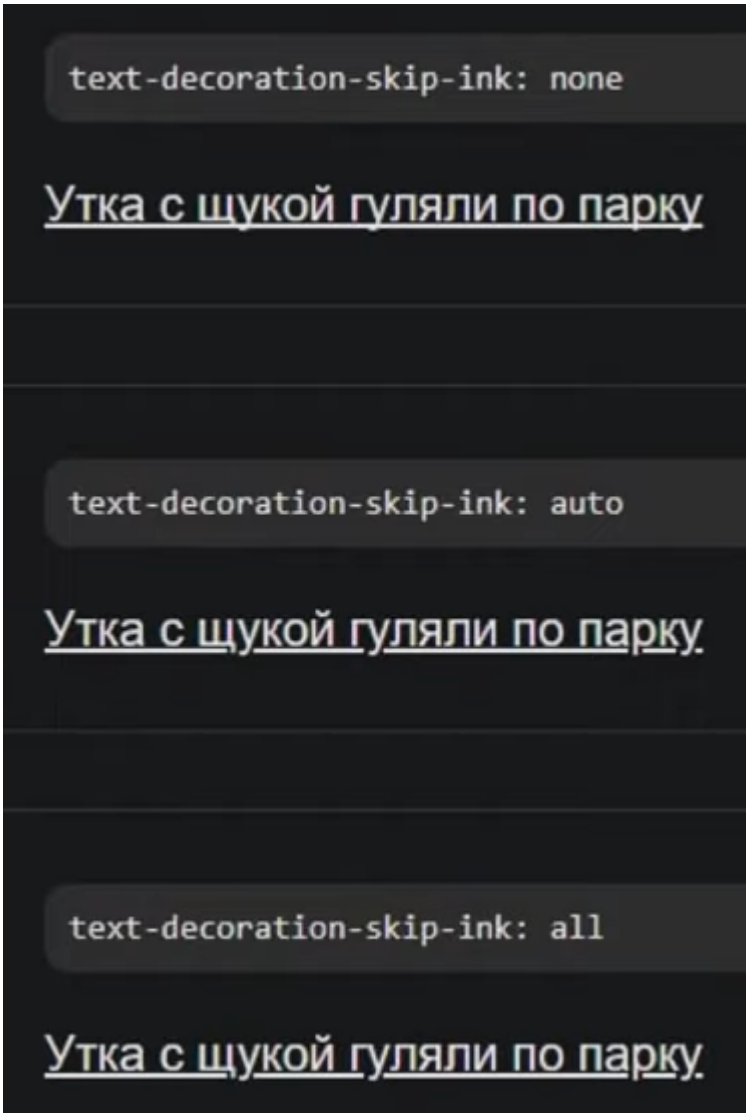
optimizeSpeed - в приоритете скорость отрисовки, лиагтуры и кёрнинг будут отключены

optimizeLegibility - важна не скрость отрисовки, а читаемость текста и визуальная крастоа, активно импользуются лигатуры и используется кёрнинг

geometricPrescision - приоритет геометрическая точность, например когда в некоторых шрифтах кёрнинг непропорционально изменятеся.

▼ Расскажите о свойстве `text-decoration-skip-ink`?

Касается ли подчеркивание букв или обтекает их. Игнорируется если линия перечеркивает текст. По умолчанию auto



▼ Расскажите о свойстве `pointer-events`?

Свойство `pointer-events` управляет тем, как элемент будет реагировать на указатель (pointer): наведение или клик курсора мыши, тап на сенсорном экране, соответствующие события из JavaScript типа click или touch.

`none` — запрещает элементу реагировать на указатель.

`auto` — элемент реагирует на указатель (значение по умолчанию).

Можно запрещать клики по картинкам, запрет интерактивности во время асинхронных запросов, запрет выделение текста, запрещать события hover, click.

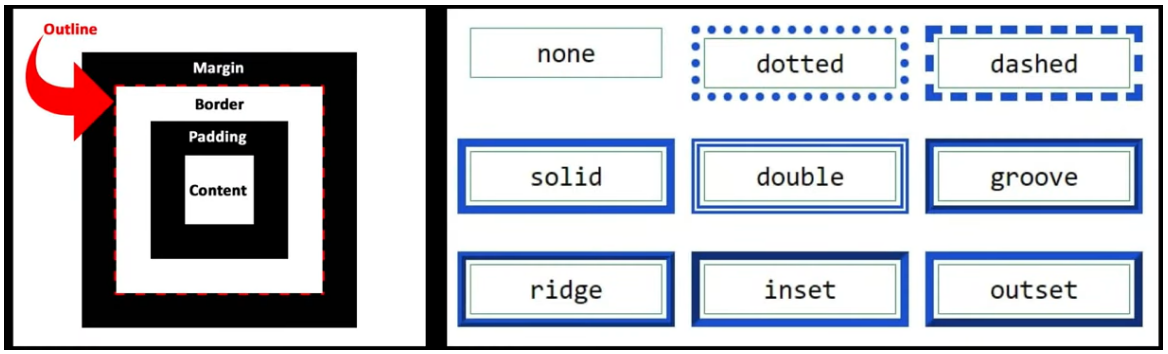
Неоптимальное использование навредит аналитике, которая не сможет собирать информацию о кликах пользователя.

▼ Расскажите о свойстве `outline`?

Что-то вроде `border` на стероидах. Не влияет на размеры элемента, можно отодвинуть от границ на любое расстояние.

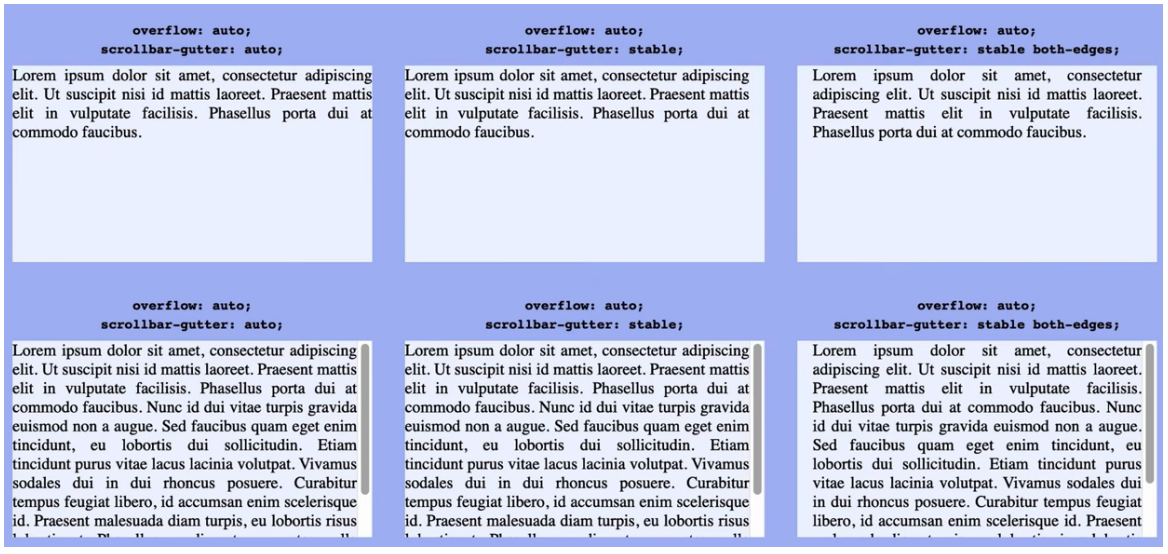
Присутствует в стандартных стилях браузера для акцента на сфокусированных интерактивных элементах. Типа размеры не меняются, вёрстка страницы не ломается, но пользователь наглядно видит, какой элемент в фокусе.

При помощи сочетания `border` и `outline` можно создать множественную рамку вокруг элемента.



▼ Расскажите о свойстве `scrollbar-gutter`?

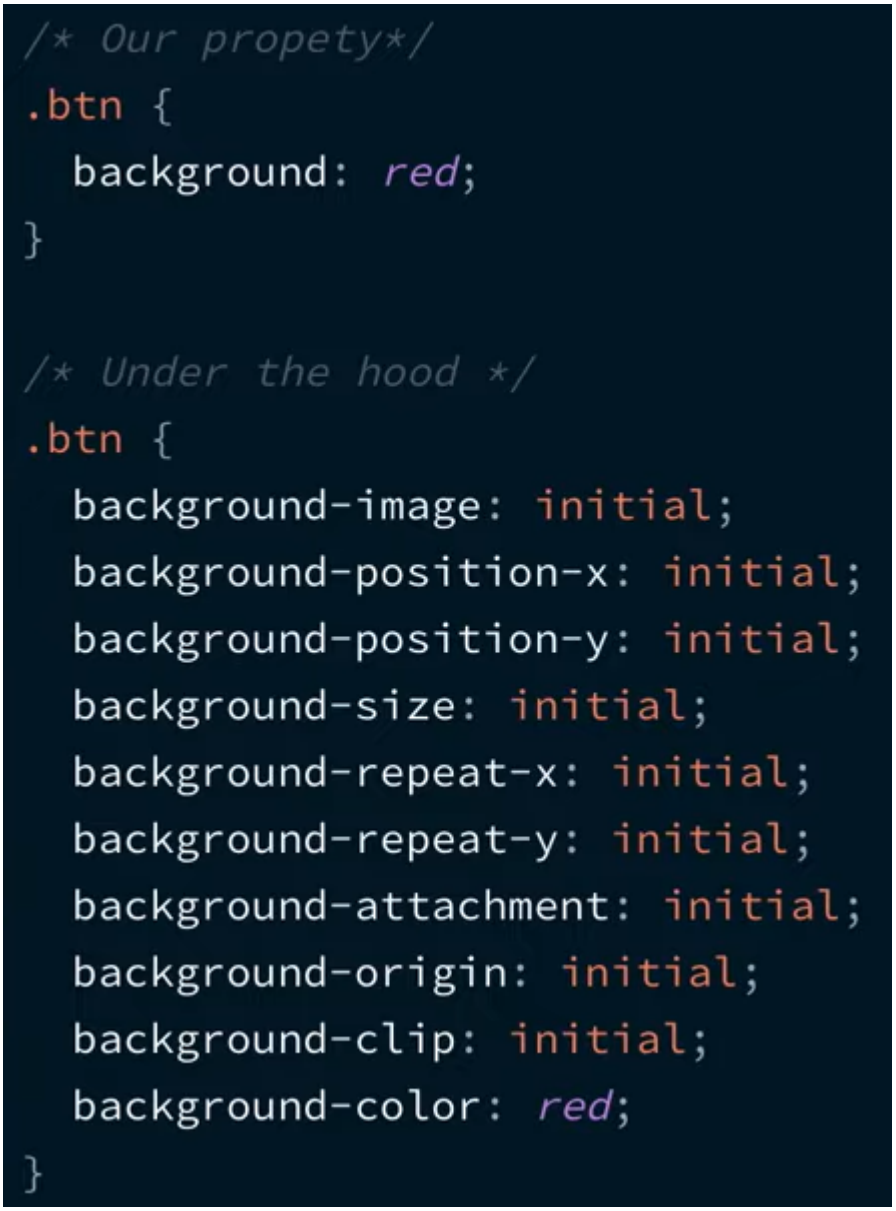
CSS-свойство scrollbar-gutter позволяет зарезервировать место для полосы прокрутки, предотвращая нежелательные изменения макета при появлении полосы прокрутки и избегая скачков содержимого.



▼ Почему не стоит использовать краткую запись свойств CSS?

Использование сокращенных записей CSS свойств под капотом приводит к перезаписи не тронутых нами свойств. Таким образом есть шанс получить переопределение свойств которые не нужно было трогать.

Да и в целом удобнее видеть какие точно свойства меняются, не гадая в каком порядке какое свойство лежит если немного подзабыл этот порядок.



```
/* Our propety*/
.btn {
  background: red;
}

/* Under the hood */
.btn {
  background-image: initial;
  background-position-x: initial;
  background-position-y: initial;
  background-size: initial;
  background-repeat-x: initial;
  background-repeat-y: initial;
  background-attachment: initial;
  background-origin: initial;
  background-clip: initial;
  background-color: red;
}
```

▼ Назовите псевдоэлементы для подсветки текста?

Псевдолементы выделения, это части документа которым присвоен определенный статус.

- **:target-text** - представляет собой текст, на который нацелен фрагмент адреса url, если браузер поддерживает фрагменты текста.
- **:selection** позволяет применить стили к части документа, который был выделен пользователем.

Пока не поддерживаются браузерами:

- **:spelling-error** представляет собой сегмент текста, который агент пользователя пометил как неправильно написанный.
- **:grammar-error** представляет сегмент текста, который user agent пометил как грамматически неверный.

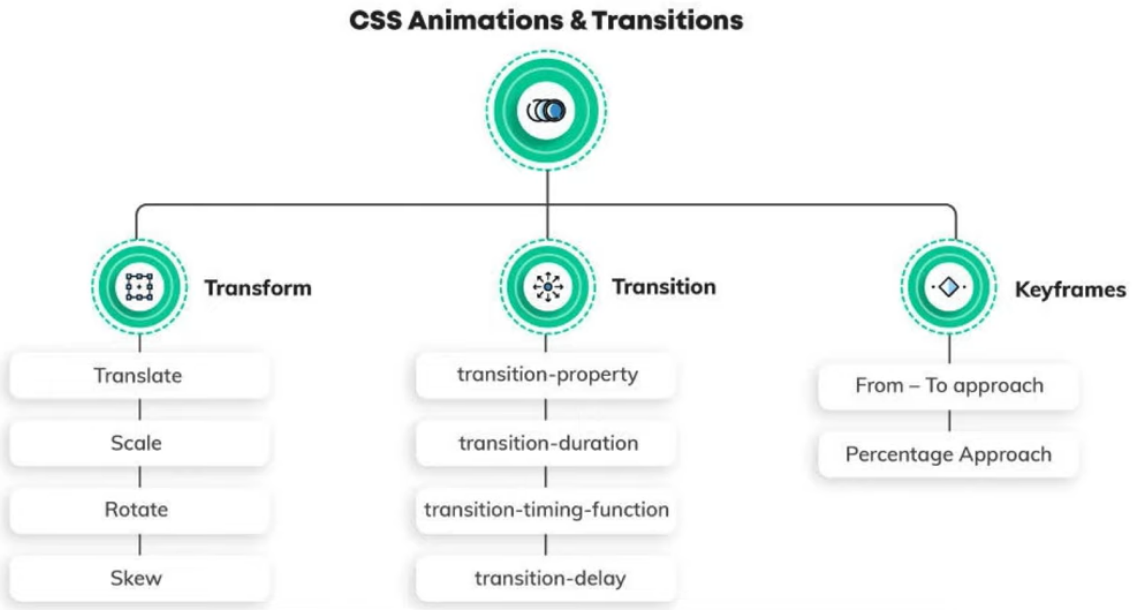
▼ Способы задания цвета в CSS?

HSL(hue, saturate, lightness)

```
color: aqua;

RGB: rgb(255, 0, 0);

RGBA: rgba(255, 255, 255, 0.5);

HSL: hsl(120, 100%, 25%);

HSLA: hsla(30, 100%, 50%, 0.1);

transparent: transparent;

currentColor: currentColor;
```

▼ Какие CSS-свойства используются для создания анимаций и плавных переходов?

Кроме тех что на картинке ещё opacity, filter и scroll-snap - создает плавный скролл или доскролливает до нужного элемента, например заголовка если просто крутнуть колесом мыши.



▼ Принципы и подходы для обеспечения масштабируемости и поддерживаемости CSS-кода?

Семантические классы - более читаемый и понятный код, упрощает поддержку и расширяемость

Модульная архитектура - разбить код на модули которые можно переиспользовать в разных частях сайта, уменьшает дублирование кода

CSS-методологии(БЭМ, OOCSS, SMACSS) - разбить интерфейс на независимые блоки которые можно использовать в разных частях приложения, упрощает поддержку и масштабирование

CSS-препроцессоры - более читаемый и удобный код, дополнительный функционал ускоряюший разработку и облегчающий поддержку и расширяемость.

▼ Плюсы и минусы методологии БЭМ?

Плюсы:

- Улучшенная читаемость кода благодаря именованию
- Поддержка и расширение кода благодаря модульному подходу
- Структура кода, уменьшает вероятность конфликтов и перезаписи стилей
- Работа в комаде благодаря единому подходу

Минусы:

- Увеличенный объем кода, дольше его читать и писать
- Много классов для элементов и модификаторов, что приводит к усложнению
- Ограничение гибкости, т.к. изменения стиля блока затронет все подобные блоки
- Дополниельные усилия для бучения персонала

▼ Какие CSS-препроцессоры вы знаете? Преимущества их использования?

CSS-препроцессор - это программа (sass, less, stylus) которая позволяет генерировать css из собственного уникального синтаксиса, то есть на выходе мы всегда будем получать сгенерированный css код который отправляется в браузер, ну а сами css препроцессоры очень активно применяются к разным проектам, потому что у них есть удобный функционал:

- Переменные
- Вложенность, когда один селектор можно вкладывать в дургой
- Комбинирование селкторов вложенности когда итоговое имя генеируется на основании вложенноси
- Переиспользование кода с помощью миксинов
- Функции и операторы
- Импорты и наследование
- Условные выражения и циклы
- Модульность и повторное использование, опять же благодаря импортам и наседованию

Sass

CSS Pre-processors / Extensions

Less

CSS Pre-processors / Extensions

Stylus

CSS Pre-processors / Extensions

JavaScript

▼ Типы данных в JavaScript?

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 не присвоенных 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.

Специальные типы:

- NaN - number
- Infinity и -Infinity - number

При создании объекта и примитива в JavaScript память выделяется и управляется по-разному.

1. **Примитивные типы данных** обычно хранятся непосредственно в стеке или внутри контекста выполнения функции. Память для

примитивных значений выделяется статически и имеет фиксированный размер, который зависит от типа примитива.

2. **Объекты** хранятся в куче (heap). Память в куче выделяется динамически по мере необходимости. При создании объекта выделяется память для самого объекта и его свойств. В JavaScript объекты могут быть динамически расширены, то есть новые свойства могут быть добавлены в любое время. Каждый объект имеет свой набор свойств, и для каждого свойства выделяется память для хранения его ключа и значения. Кроме того, объекты могут содержать ссылки на другие объекты или функции, что также требует дополнительной памяти.

▼ Разница между `==` и `===` (нестрогое/строгое равенство)?

Оператор == (нестрогое равенство) - приводит оба операнда к одному типу и сравнивает.

Оператор === (строгое равенство) - не приводит к одному типу.

Falsy типы:

- `''`
- `0`
- `undefiend`
- `null`
- `NaN`

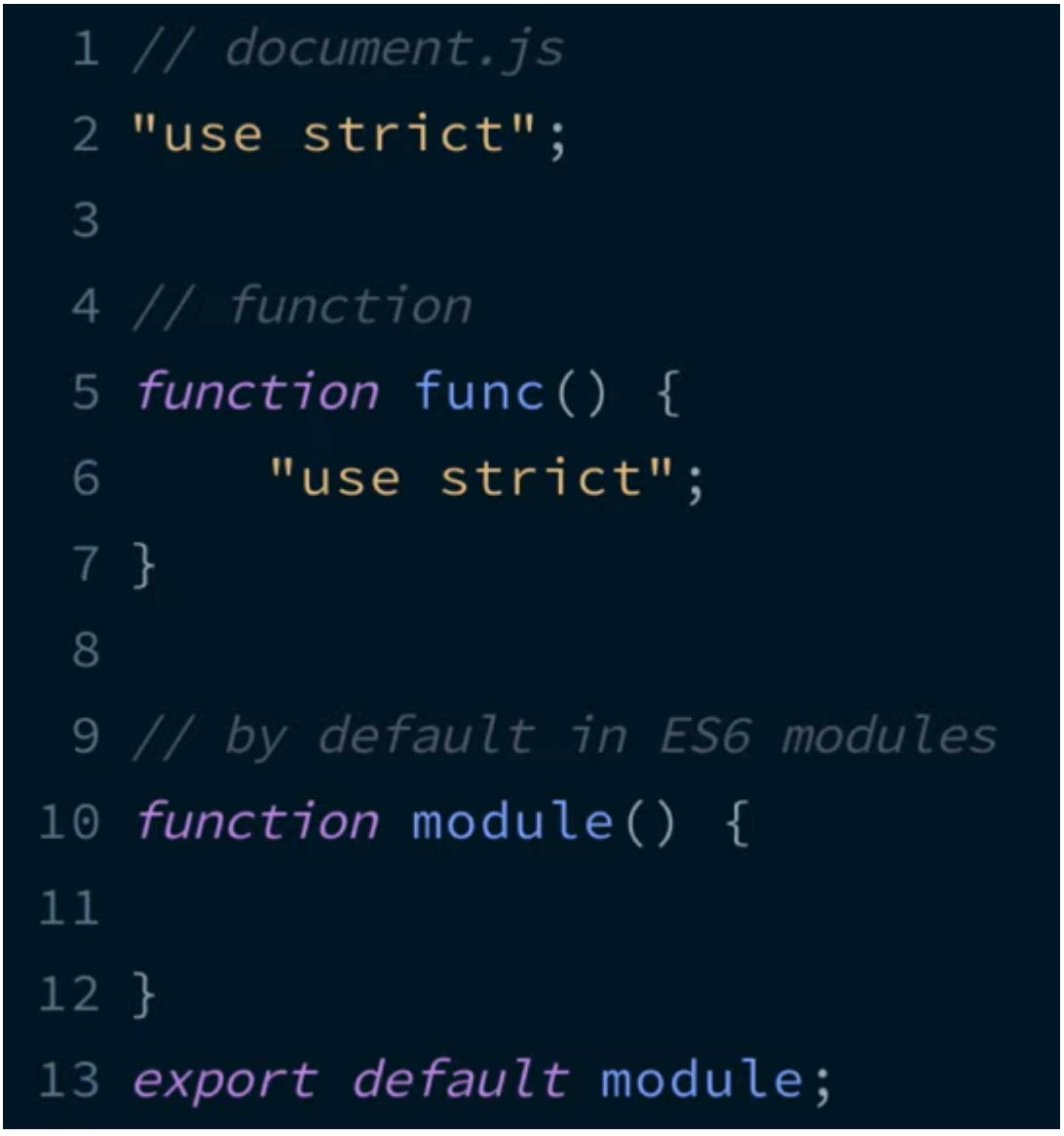
▼ Что такое Strict mode в JavaScript?

Strict mode - это строгий режим работы JS. Он был введен в стандарт ECMAScript 5, чтобы устранить некоторые неоднозначности и проблемы, с которыми могли сталкиваться разработчики, когда писали JavaScript-код.

В режиме Strict:

- Нельзя использовать неопределенные переменные
- Запрещено переопределять свойства объектов только для чтения (readonly)
- В функции this имеет значение undefined если функция вызывается без контекста

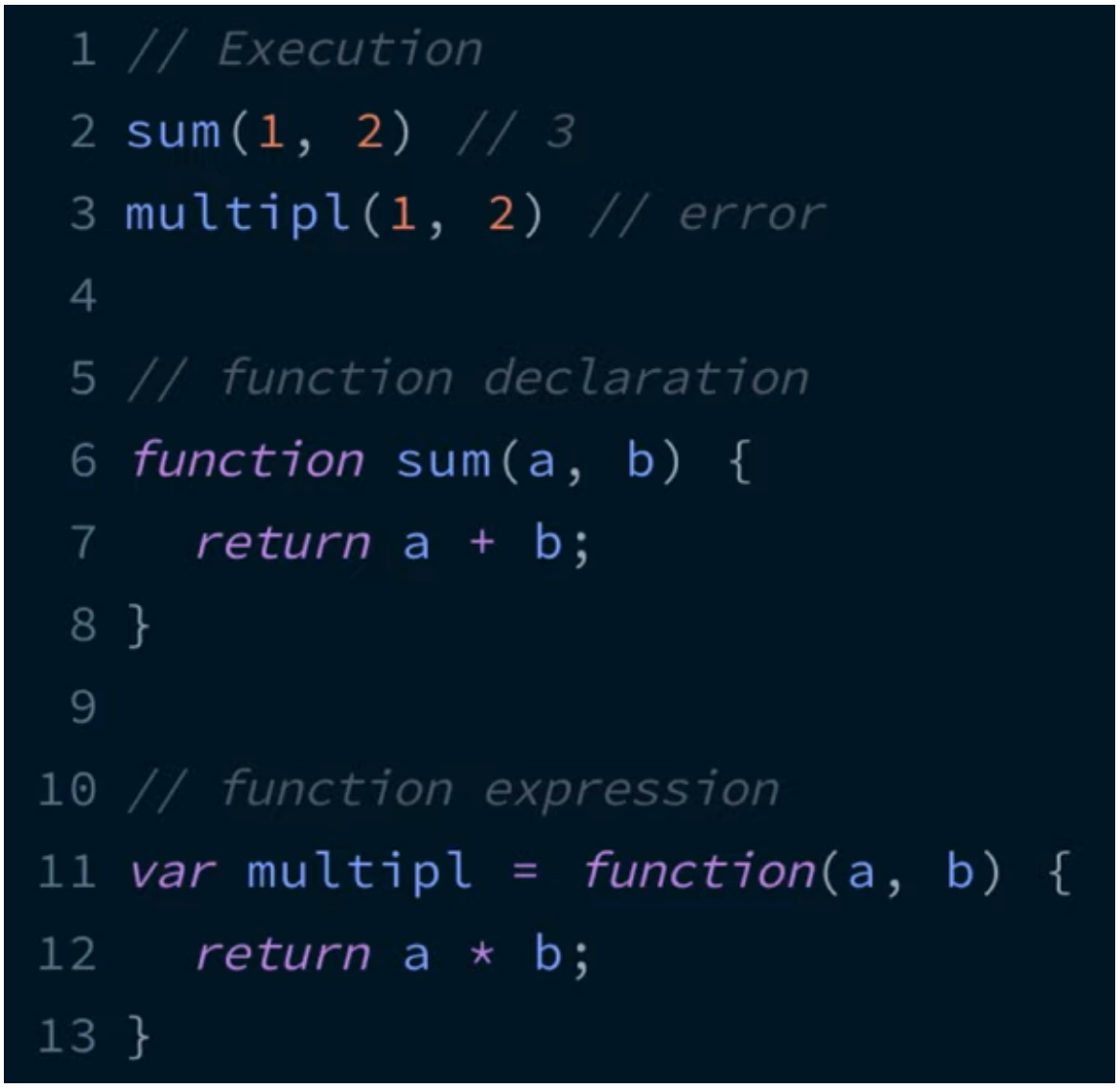
- Запрещено использование ключевых слов **eval** и **arguments** в качестве имен переменных и функций и другие.
- Нельзя использовать переменные без объявления
- Явная ошибка при попытке изменить поле, если значение поля нельзя изменить или удалить
- Параметры функции не могут иметь одинаковые имена
- Другое поведение this, при включённом строгом режиме `this` больше не будет по умолчанию ссылаться на глобальный объект.
- Запрещено использовать новые зарезервированные слова
- Ограничение небезопасных конструкций `with` и `eval`



▼ Разница между function declaration и function expression?

Function Declaration - это объявление функции в основном потоке документа, которое начинается с ключевого слова "function", всплытие есть. Т.е. можно вызвать до объявления.

Function Expression - это определение функции, которое присваивает функцию переменной, у нее нет всплытия.



```
// Execution
sum(1, 2) // 3
multipl(1, 2) // error

// function declaration
function sum(a, b) {
  return a + b;
}

// function expression
var multipl = function(a, b) {
  return a * b;
}
```

▼ Разница между `null` и `undefined`?

Если просто, то **null** мы задали сами, а **undefined** - отсутствие значения.

Основное отличие в том, что **undefined** представляет значение переменной, которая ещё не была инициализирована, а **null** — намеренное отсутствие объекта.

▼ Типы таймеров в JavaScript?

- **setTimeout**
- **setInterval**

Список активных таймеров общий и для одноразовых таймеров, установленных с помощью `setTimeout()` и для регулярных таймеров, установленных с помощью `setInterval()`. Потому в `clearInterval()` можно использовать идентификаторы обоих типов таймеров.

Однако для одноразовых таймеров лучше использовать отдельную функцию `clearTimeout()` для лучшей читаемости.

```
// Base function
const sayHi = () => console.log('Hello');

// setTimeout
setTimeout(sayHi, 1000);
// 'Hello'

// setInterval
const timer = setInterval(sayHi, 1000);
// 'Hello'
// 'Hello'
// 'Hello'
// 'Hello'
// 'Hello'
clearInterval(timer);
```

▼ Что такое поднятие (Hoisting)?

Hoisting (всплытие) - это механизм подъема функции или переменной в глобальную или функциональную область видимости.

Это происходит из-за того, что интерпретатор JS проходит через два шага: поднятие и выполнение.

Срабатывает на переменных **var** и **function declaration**.

```
// variable
console.log(a); // undefined
var a = 'Test string';
console.log(a); // 'Test string'

// let
console.log(b); // Uncaught ReferenceError: b is not defined
let b = 'Super string';
console.log(b); // 'Super string'

// function
sayHi('Jack'); // 'Hello Jack'

function sayHi(name) {
  return `Hello ${name}`;
};
```

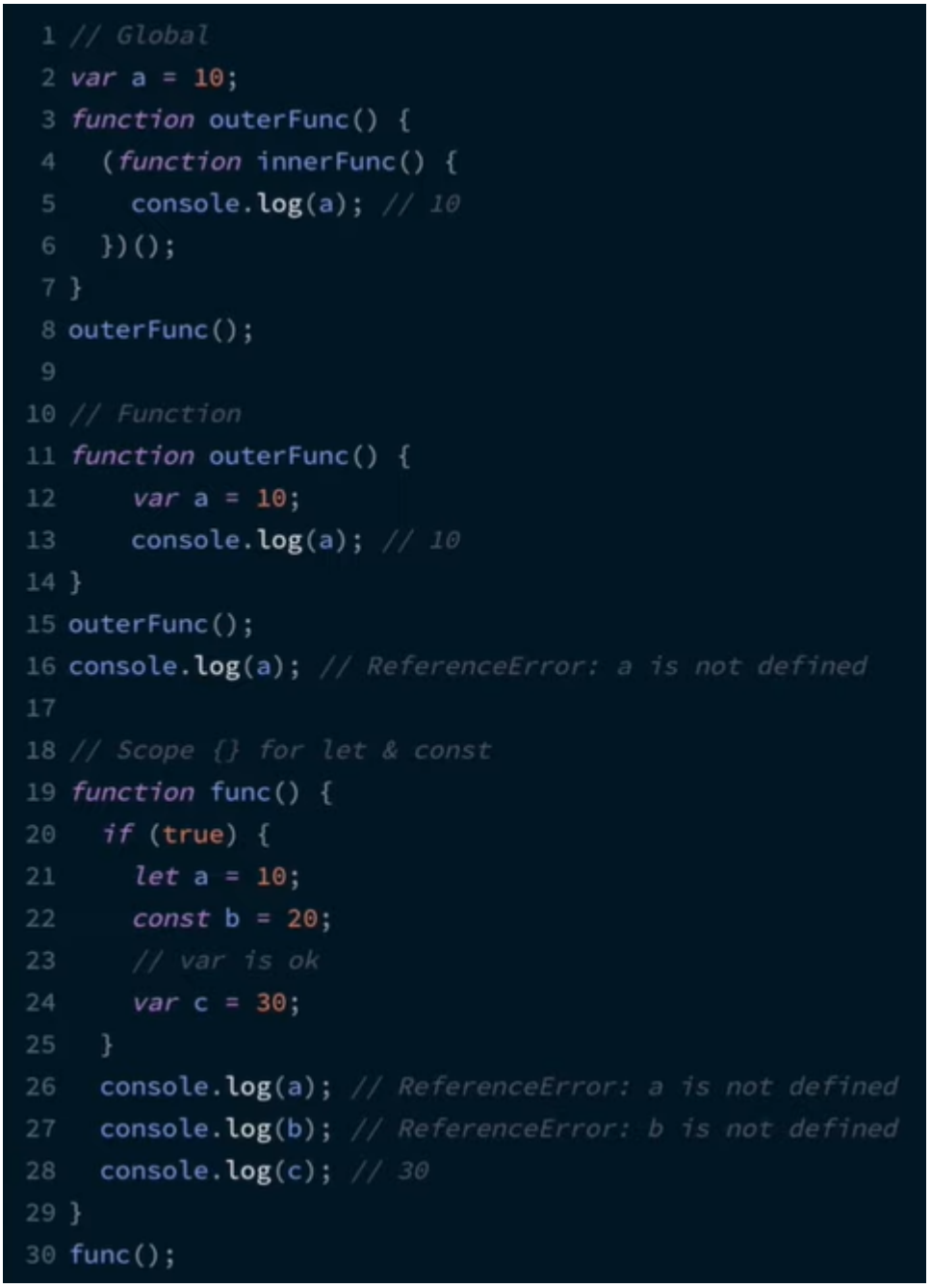
▼ Что такое область видимости (Scope)?

Область видимости - это место откуда мы имеем доступ к переменным или функциям.

В JavaScript есть три типа областей видимости:

- **Глобальная**
- **Функциональная/Локальная** - переменные и функции объявленные внутри функции доступны только этой функции и всем вложенным в нее функциям.
- **Блочная** появилась в ES6 для переменных **let** и **const**, область видимости внутри фигурных скобок, так называемого блока.

Также область видимости это по сути набор правил по которым ищется переменная. Сначала в локальной, затем во внешней и т.д. пока не дойдет до глобальной.



```
// Global
var a = 10;
function outerFunc() {
  (function innerFunc() {
    console.log(a); // 10
  })();
}
outerFunc();

// Function
function outerFunc() {
    var a = 10;
    console.log(a); // 10
}
outerFunc();
console.log(a); // ReferenceError: a is not defined

// Scope {} for let & const
function func() {
  if (true) {
    let a = 10;
    const b = 20;
    // var is ok
    var c = 30;
  }
  console.log(a); // ReferenceError: a is not defined
  console.log(b); // ReferenceError: b is not defined
  console.log(c); // 30
}
func();
```

▼ Разница между `var`, `let` и `const`?

Разные области видимости **let** и **const** - блочная, **var** - функциональная.

const нельзя изменить, но если в **const** лежит обьект или массив, мы можем поменять его элементы.

```
// Hoisting
console.log(a); // Cannot access 'a' before initialization
console.log(b); // Cannot access 'b' before initialization
console.log(c); // undefined
let a = 10;
const b = 20;
var c = 30;

// Scope {} for let & const
if (true) {
  let a = 10;
  const b = 20;
  // var is ok
  var c = 30;
}
console.log(a); // ReferenceError: a is not defined
console.log(b); // ReferenceError: b is not defined
console.log(c); // 30

// let & const
let name = 'Yauhen';
const channel = 'webDev';
name = 'Jack' // it's OK
channel = 'IT' // TypeError: Assignment to constant variable.
```

▼ Что такое замыкание (Closure)?

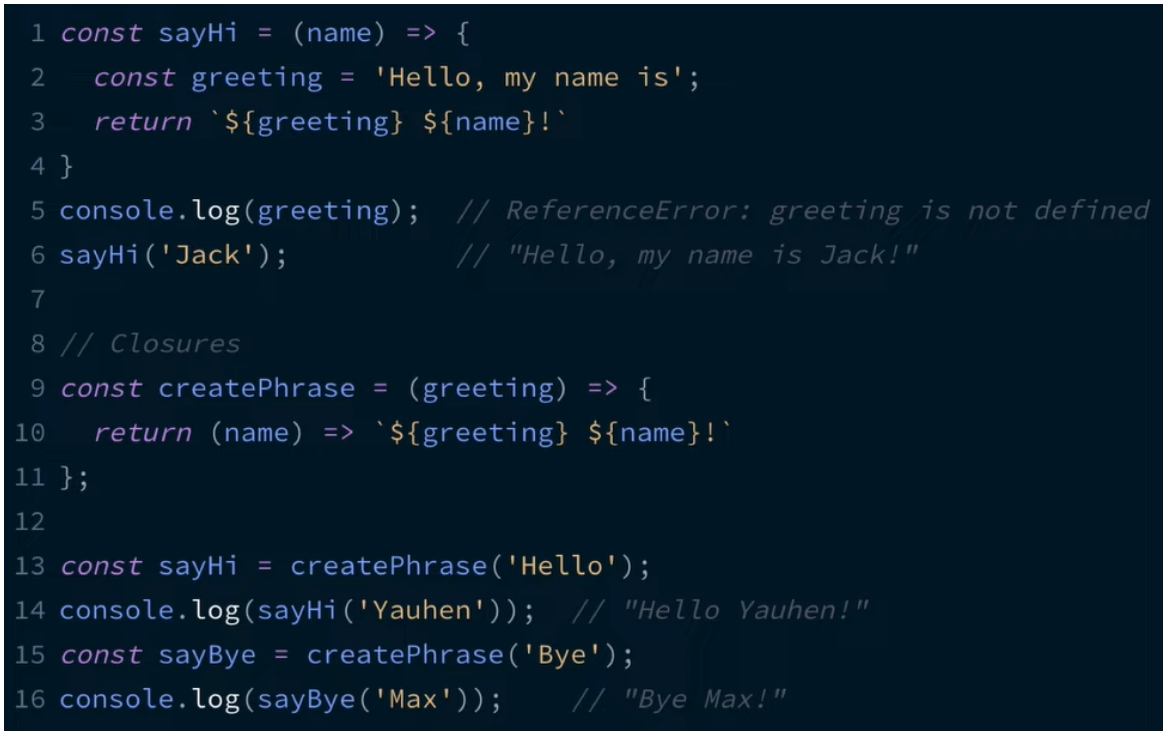
Замыкание (Closure) - функция, которая запоминает своё окружение выполнения (лексическое окружение), даже когда функция вызывается вне этого окружения.

Преимущества замыканий заключаются в том, что они позволяют создавать функции, которые могут сохранять состояние между вызовами и использовать переменные, которые не могут быть переданы в функцию в качестве аргумента. Это делает замыкания очень мощным инструментом для создания более эффективного и гибкого программного кода.

В **JavaScript** у каждой выполняемой функции, блока кода `{...}` и скрипта есть связанный с ними внутренний (скрытый) объект, называемый *лексическим окружением* `LexicalEnvironment`.

Объект лексического окружения состоит из двух частей:

1. *Environment Record* – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение `this`).
2. Ссылка на *внешнее лексическое окружение* – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок).



```
const sayHi = (name) => {
  const greeting = 'Hello, my name is';
  return `${greeting} ${name}!`
}
console.log(greeting);  // ReferenceError: greeting is not defined
sayHi('Jack');          // "Hello, my name is Jack!"

// Closures
const createPhrase = (greeting) => {
  return (name) => `${greeting} ${name}!`
};

const sayHi = createPhrase('Hello');
console.log(sayHi('Yauhen'));  // "Hello Yauhen!"
const sayBye = createPhrase('Bye');
console.log(sayBye('Max'));    // "Bye Max!"
```

▼ Что обозначает `this` в JavaScript?

Подробнее

Подробнее

Ключевое слово `this` в JavaScript используется для ссылки на объект, в котором выполняется код.

Значение `this` зависит от контекста выполнения.

- Без **'use strict': `this`**=== window
- С **'use strict': `this`**=== undefined

Контекст можно определить на основе трех вопросов:

1. Включен строгий режим или нет?
2. Какая функция используется?

3. Как функция вызывается?

В функциях, созданных при помощи ключевого слова `function`:

- **`this`**определяется в момент вызова
- **`this`** можно задать явным образом через методы **`call`, `apply` и `bind` .**

В стрелочных функциях:

- Нет своего **`this`**, они берут его из внешнего окружения в момент создания.
- Нельзя присвоить **`this`** методами **`call`, `apply` и `bind`.**
- НЕ теряют **`this`**.
- В методе объекта, созданном через стрелочную функцию, **`this`** === window, ВСЕГДА! НЕ ВАЖНО КАКОЙ РЕЖИМ!

`this`у методов объектов:

- равен **объекту**, но только в том случае, если мы вызываем метод через объект.

```
'use strict';
const obj = {
    a: 15,
    b: 20,
    sum: function() {
        console.log(this);
    }
}

obj.sum(); // obj
```

- Если присвоить метод объекта в переменную и вызвать переменную как функцию, или передать метод в таймер, то он будет вызываться как обычная функция и **`this`** потеряется

```
'use strict';
const obj = {
    a: 15,
    b: 20,
    sum: function() {
        console.log(this);
    }
}

const a = obj.sum;
console.log(a()); // undefined
setTimeout(obj.a, 1000) // undefined через секунду
```

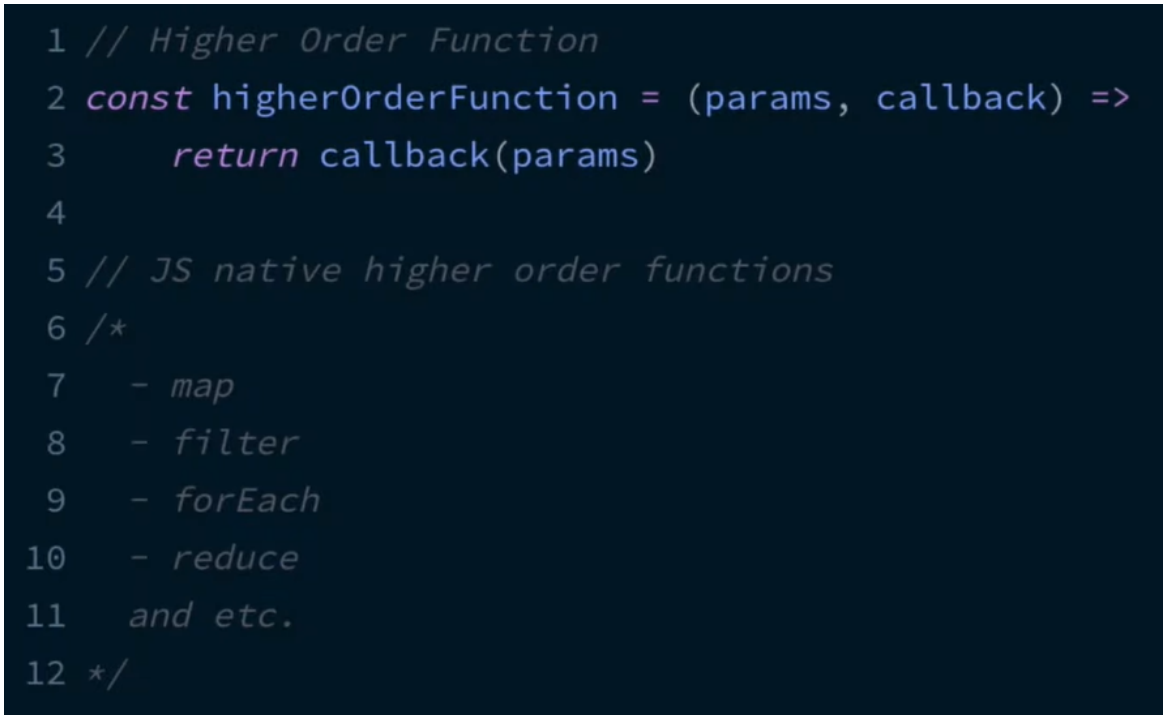
`this` в конструкторах и классах:

- **`this` ===** экземпляру объекта

Таблица

▼ Что такое функции высшего порядка (Higher Order Functions)?

Функции высшего порядка - это функции, которые могут принимать другие функции в качестве аргументов или возвращать функции в качестве результата.



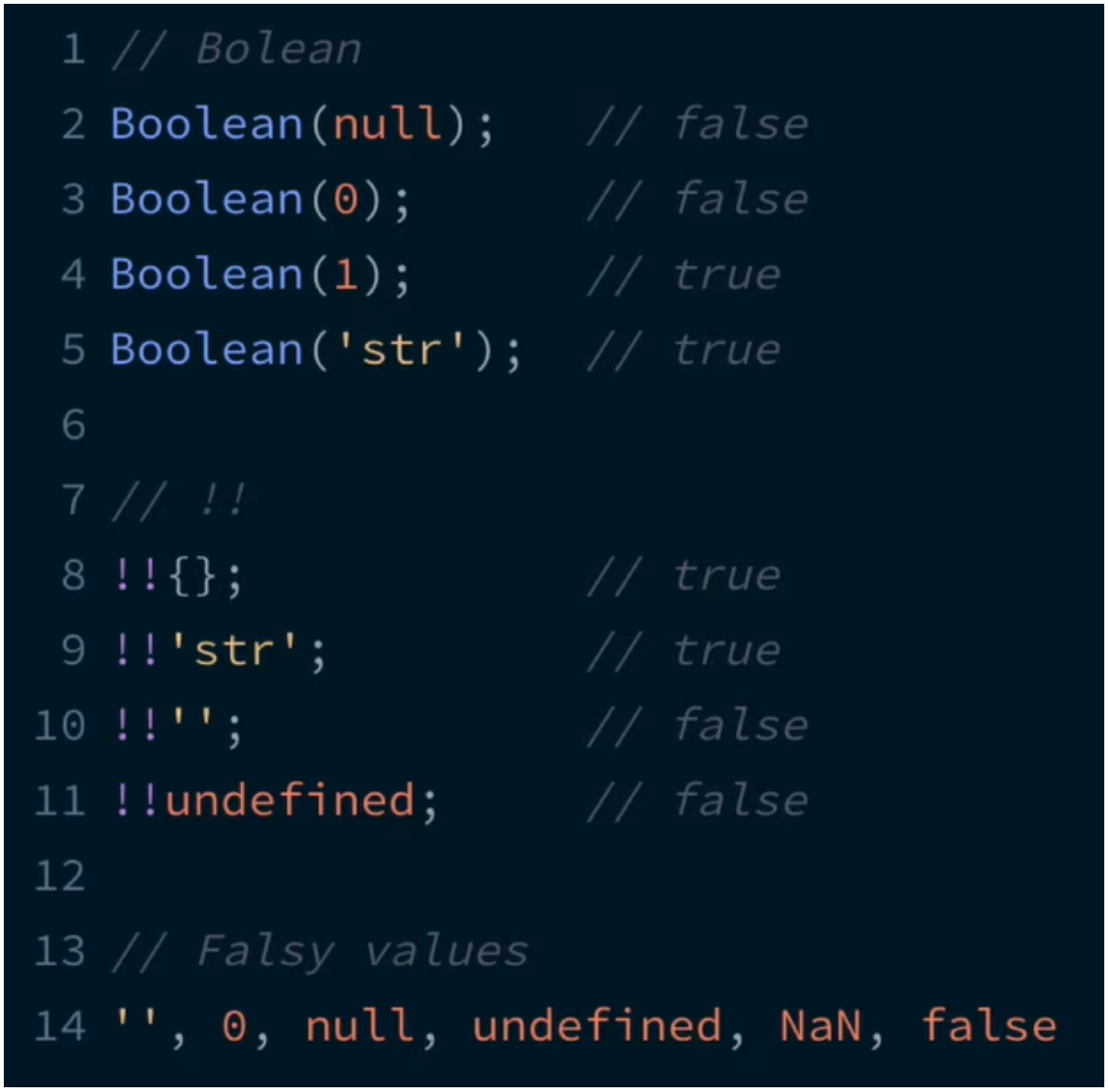
```
// Higher Order Function
const higherOrderFunction = (params, callback) =>
    return callback(params)

// JS native higher order functions
/*
   - map
   - filter
   - forEach
   - reduce
   and etc.
*/
```

▼ Как превратить любой тип данных в булевый? Перечислите ложные значения в JS?

Boolean() и !!

Ложные значения:

- false
- 0 (и -0)
- "" (пустая строка)
- null
- undefined
- NaN (Not a Number)



```
// Bolean
Boolean(null);    // false
Boolean(0);       // false
Boolean(1);       // true
Boolean('str');   // true

// !!
!!{};             // true
!!'str';          // true
!!'';             // false
!!undefined;      // false

// Falsy values
'', 0, null, undefined, NaN, false
```

▼ Методы строк в JavaScript?

1. length - длина строки.
2. charAt() - возвращает символ на указанной позиции в строке.
3. toUpperCase()
4. toLowerCase()
5. indexOf() - возвращает позицию первого вхождения заданного символа или подстроки.
6. lastIndexOf() - возвращает позицию последнего вхождения заданного символа или подстроки.
7. replace() - заменяет заданный символ или подстроку в строке на другую подстроку.

8. split() - разделяет строку на массив подстрок, используя разделитель separator.
9. slice() - извлекает часть строки и возвращает новую строку без изменения оригинальной строки.
10. substring(startIndex, endIndex) - возвращает подстроку строки, начиная с указанного индекса и заканчивая указанным индексом (или до конца строки, если индекс не указан). Разница с методом `slice()` заключается в том, что метод `substring()` не поддерживает отрицательные индексы.

```
const str = 'Hi, my name is Jack!';

// String methods
str.length;                    // 20
str.charAt(1);                 // "i"
str.toUpperCase();             // "HI, MY NAME IS JACK!"
str.toLocaleLowerCase();       // "hi, my name is jack!"
str.indexOf(',');              // 2
str.lastIndexOf('a');          // 16
str.slice(0, 3);               // "Hi,"
str.substr(0, 6);              // "Hi, my"
str.substring(0, 6);           // "Hi, my"
str.replace('Jack', 'Yauhen'); // "Hi, my name is Yauhen!"
'   Hello!     '.trim();       // "Hello!"
```

▼ Методы массивов в JavaScript?

1. **push**() - добавляет элемент в конец массива и возвращает новую длину массива
2. **pop**() - удаляет последний элемент из массива и возвращает его значение
3. **shift**() - удаляет первый элемент из массива и возвращает его значение
4. **unshift**() - добавляет элемент в начало массива и возвращает новую длину массива
5. **concat**() - объединяет два или более массива в один и возвращает новый массив

6. **join**() - объединяет все элементы массива в одну строку, разделенных указанным разделителем
7. **slice**() - возвращает подмассив, начиная с указанного индекса и заканчивая указанным индексом (или до конца массива, если индекс не указан)
8. **splice**() - удаляет, заменяет или добавляет элементы в массиве на указанной позиции
9. **reverse**() - переворачивает порядок элементов в массиве
10. **sort**() - сортирует элементы массива в указанном порядке (по умолчанию - лексикографический)
11. **forEach**
12. **map**
13. **reduce**
14. **every**
15. **some**

```
const arr = ['Tommy', 'Arthur', 'John'];

// Array methods
arr.length;                // 3
arr.push('Finn');          // 4 - ["Tommy", "Arthur", "John", "Finn"]
arr.pop();                 // "Finn" - ["Tommy", "Arthur", "John"]
arr.unshift('Finn');       // 4 - ["Finn", "Tommy", "Arthur", "John"]
arr.shift();               // "Finn" - ["Tommy", "Arthur", "John"]
arr.concat(['Finn']);      // ["Tommy", "Arthur", "John", "Finn"]
arr.splice(1, 1, "Finn"); // ["Tommy", "Finn", "John"]
arr.splice(0, 1);          // ["Finn", "John"]
arr.toString();            // "Tommy,Arthur,John"
arr.join('-');             // "Tommy-Arthur-John"
```

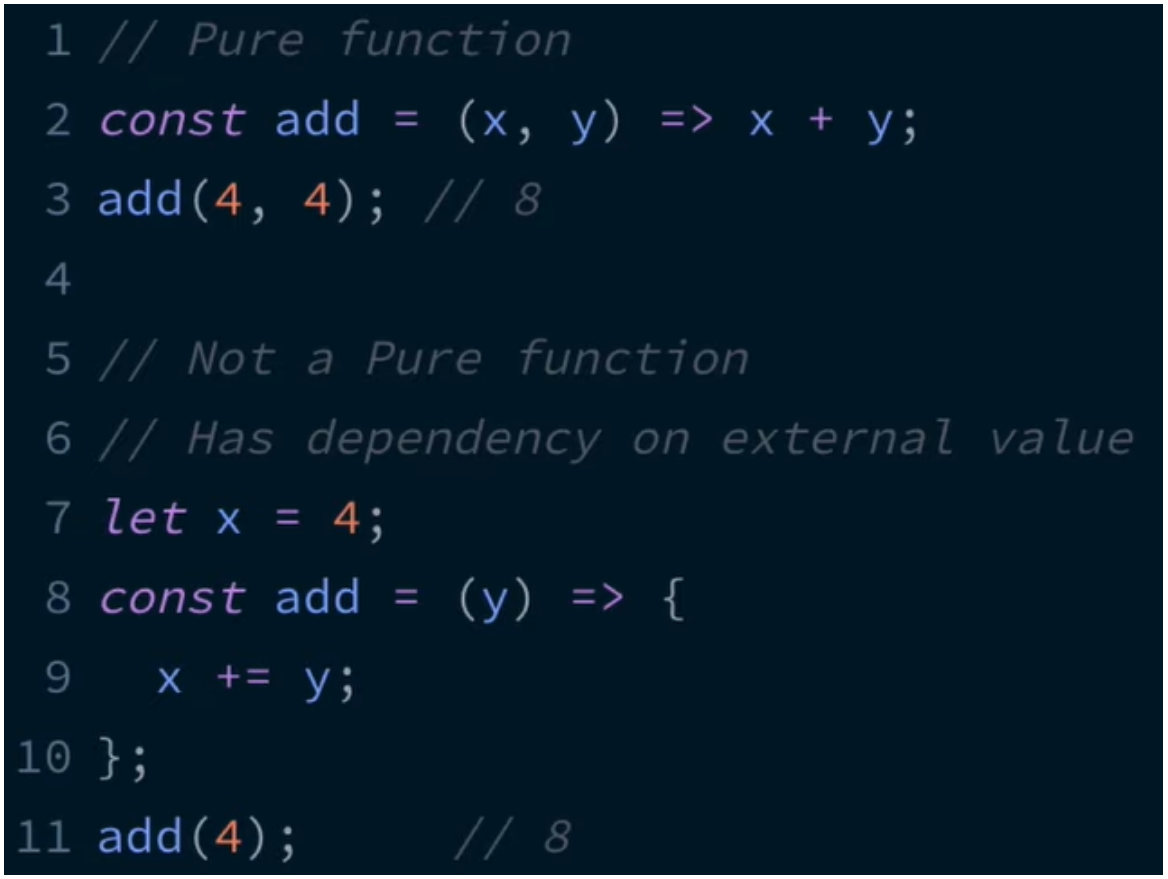
▼ Что такое чистая функция?

Чистая функция - это функция, которая возвращает один и тот же результат при одних и тех же входных данных, а также не имеет побочных эффектов.

К побочным эффектам можно отнести:

- Видоизменение входных параметров.

- HTTP и DOM запросы.
- Изменения в файловой системе.
- Вывод на экран.



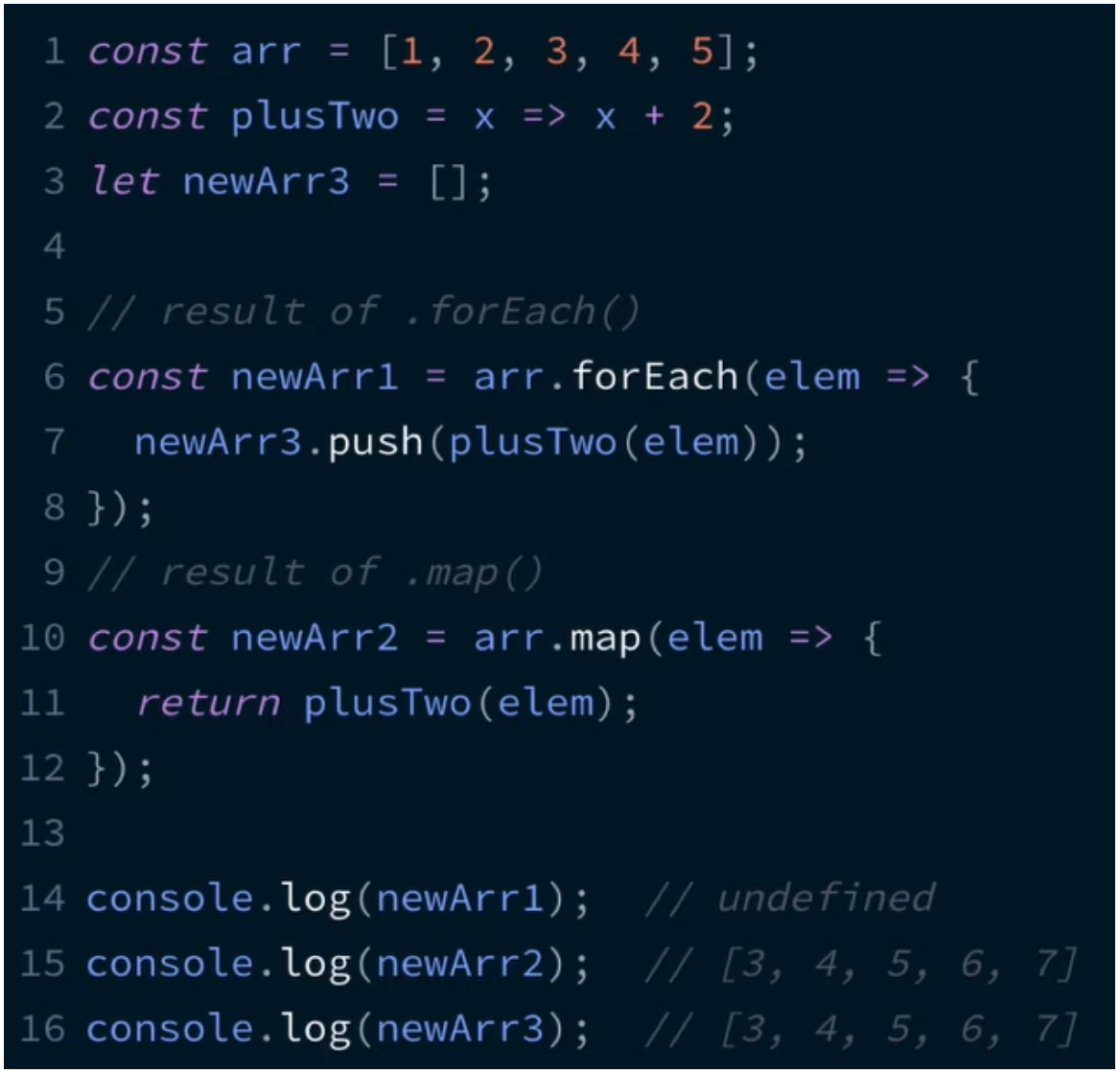
```
// Pure function
const add = (x, y) => x + y;
add(4, 4); // 8

// Not a Pure function
// Has dependency on external value
let x = 4;
const add = (y) => {
  x += y;
};
add(4);     // 8
```

▼ Разница между `.forEach()` и `.map()`?

Не изменяют исходный массив

forEach - перебирает массив и ничего не возвращает.

map - возвращает новый массив.



```
const arr = [1, 2, 3, 4, 5];
const plusTwo = x => x + 2;
let newArr3 = [];

// result of .forEach()
const newArr1 = arr.forEach(elem => {
  newArr3.push(plusTwo(elem));
});
// result of .map()
const newArr2 = arr.map(elem => {
  return plusTwo(elem);
});

console.log(newArr1);  // undefined
console.log(newArr2);  // [3, 4, 5, 6, 7]
console.log(newArr3);  // [3, 4, 5, 6, 7]
```

▼ Разница между `.call()`, `.apply()` и `bind()`?

.call() и .apply() вызывают функцию с определенным контекстом (this) и аргументами.

Разница между ними заключается в том, что .call() передает аргументы в виде списка, а .apply() передает аргументы в виде массива.

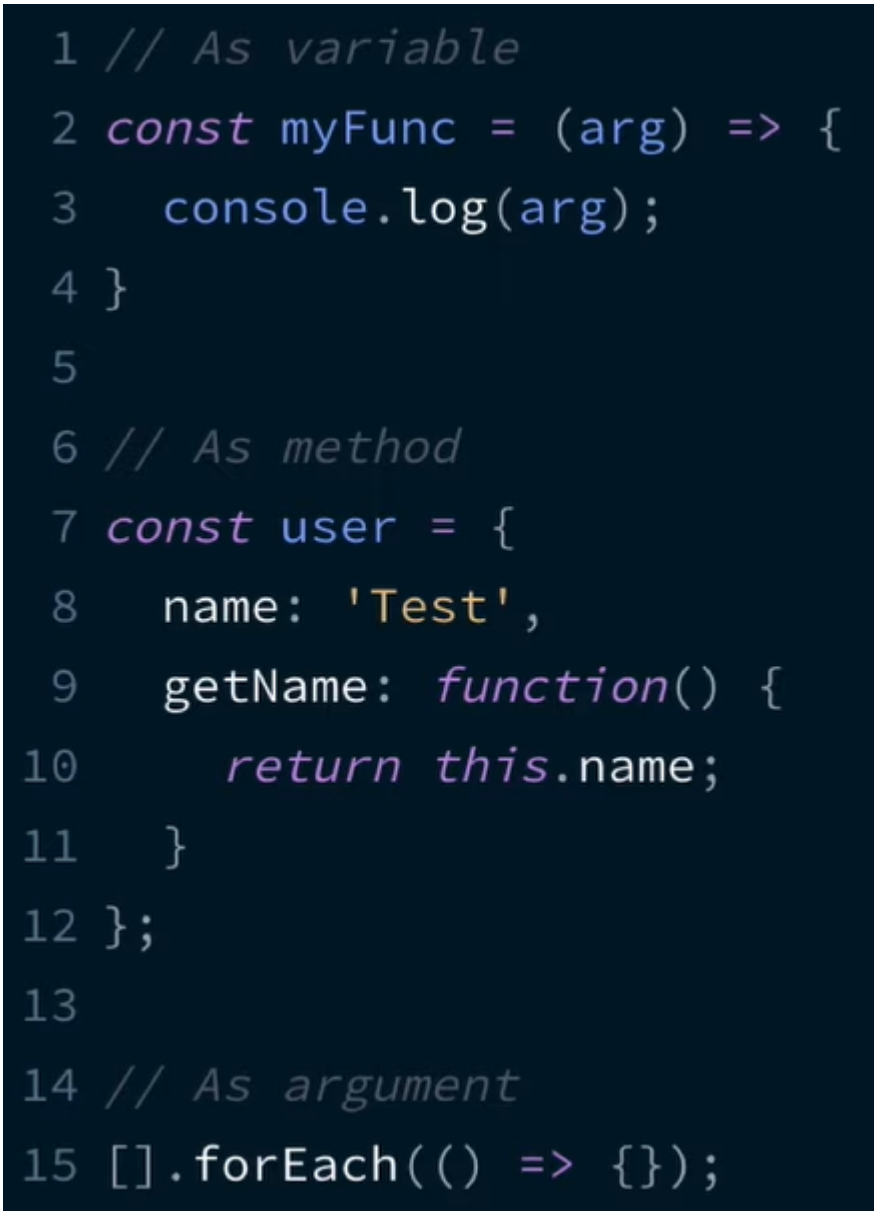
.bind() создает новую функцию с привязанным контекстом (this) и аргументами. Он не вызывает функцию немедленно, а возвращает новую функцию, которую можно вызвать позже.

```js
// Function with a context
function showName(firstPart, lastPart) {
  console.log(`${this[firstPart]} ${this[lastPart]}`);
}

const user = {
  firstName: 'Yauhen',
  lastName: 'Kavalchuk',
}

showName.call(user, 'firstName', 'lastName');        // Yauhen Kavalchuk
showName.apply(user, ['firstName', 'lastName']);     // Yauhen Kavalchuk

const newShowName = showName.bind(user, 'firstName', 'lastName');
newShowName();                                       // Yauhen Kavalchuk
```

▼ Почему в JS функции называют объектами первого класса?

Функции в JavaScript называют объектами первого класса, потому что они являются объектами, которые могут быть присвоены переменным, переданы в качестве аргументов другим функциям, возвращены из функций и использованы как значения свойств объектов.

Это означает, что функции в JavaScript обладают теми же возможностями, что и любые другие объекты, такие как массивы или объекты.



```
// As variable
const myFunc = (arg) => {
  console.log(arg);
}

// As method
const user = {
  name: 'Test',
  getName: function() {
    return this.name;
  }
};

// As argument
[].forEach(() => {});
```

▼ Как определить наличие свойства в объекте?

1. .hasOwnProperty()
2. ‘prop’ in obj
3. obj[‘prop’]

```js
const obj = {
  'prop1': 'foo',
  'prop2': 'bar',
};

// hasOwnProperty:
console.log(obj.hasOwnProperty('prop1'));  // true
console.log(obj.hasOwnProperty('prop2'));  // true
console.log(obj.hasOwnProperty('prop3'));  // false

// in
console.log('prop1' in obj);  // true
console.log('prop2' in obj);  // true
console.log('prop3' in obj);  // false

// index notation
console.log(obj['prop1']);  // foo
console.log(obj['prop2']);  // bar
console.log(obj['prop2']);  // undefined
```

▼ Что такое IIFE?

IIFE (Immediately Invoked Function Expression) - это анонимная самовызывающаяся функция.

Их использовали до появления модулей в ES6, чтобы не засорять глобальную область видимости и изолировать переменные.

```
// IIFE
(function () {
  const greeting = 'Hello world!';
  console.log(greeting);
})();

// 'Hello world!'
```

▼ Что такое псевдомассив `arguments`?

arguments - это коллекция аргументов, которые передаются в функцию.

Это объект подобный массиву, с помощью него можно получить доступ к любому из аргументов, которые были переданы в функцию.

Чтоб превратить его в массив:

1. Array.prototype.slice.call(arguments) - ES5
2. [...arguments] - ES6

Не доступен в стрелочных функциях.

```
// ES5
function createName(firstName, lastName) {
  console.log(arguments.length);        // 2
  console.log(arguments[0]);            // "Yauhen"
  console.log(arguments[1]);            // "Kavalchuk"

  // ES5 (Transformation to array)
  var args1 = Array.prototype.slice.call(arguments);
  // ES6 (Transformation to array)
  const args2 = [...arguments];

  return firstName + ' ' + lastName;
}

createName('Yauhen', 'Kavalchuk'); // "Yauhen Kavalchuk"

// ES6
const createName = (firstName, lastName) => arguments;

createName('Yauhen', 'Kavalchuk') // ReferenceError: arguments is not define
```

▼ Разница между host-объектами и нативными объектами?

host-объекты - это объекты, которые предоставляются средой выполнения, то есть браузером или nodejs.

- window
- document
- location
- history

нативные объекты - объекты, которые являются частью языка JavaScript.

- String
- Object
- RegExp
- Function
- Array

```
/* Host objects
  FE: String, Math, RegExp, Object, Function...
  BE (Node.js): global, console, http, Event...
*/

/* Built-in objects
  window, document, location, history, XMLHttpRequest,
  setTimeout, getElementsByTagName, querySelectorAll...
*/

// User objects
const user = {
  name: 'Yauhen',
};
```

▼ Почему результат сравнения 2х объектов это `false`?

Потому что объекты это ссылочный тип.

```
// Primitives
const a = 10;
const b = 20;
const c = 10;
console.log(a === b); // false
console.log(a === c); // true

// Objects
console.log({ a: 10 } === { a: 10 }); // false
console.log({} === {});               // false

// Equal odjects
const a = { a: 10 };
const b = a;
console.log(a === b); // true
```

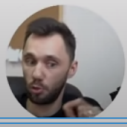
▼ Что такое прототипное наследование? Как создать объект без прототипа?

Прототипное наследование - это особенность JavaScript, которая позволяет объектам наследовать свойства и методы других объектов через использование их прототипов. В JavaScript каждый объект имеет ссылку на прототип, который является другим объектом.

__proto__ - есть у всех объектов! Это ссылка на **.prototype** того класса, с помощью которого создан обьект.

.prototype - есть только у **function** и **class.**

```
let promise = new Promise(() => {}) // new Promise(...)

let man = {} // new Object(...)

let users = [] // new Array(...)

let age = 18  // new Number(...)

let youtube = "it-kamasutra" // new String(...)

function subscribeToChannelRightNowPlease() {} // new Function(...)
let likeThisVideoPlease = function() {} // new Function(...)
let clickTheBellPlease = () => {} // new Function(...)
class YoutubeChannel {} // new Function(...)

let channel1 = new YoutubeChannel() // new YoutubeChannel

let areYouOKAfterThat = true // new Boolean(...)
```



Когда к свойству объекта происходит обращение, и если свойство не находится в текущем объекте, то механизм JavaScript просматривает прототип объекта и ищет это свойство там, затем в прототип прототипа и так пока не найдет нужное свойство или не дойдет до конца цепочки прототипов.

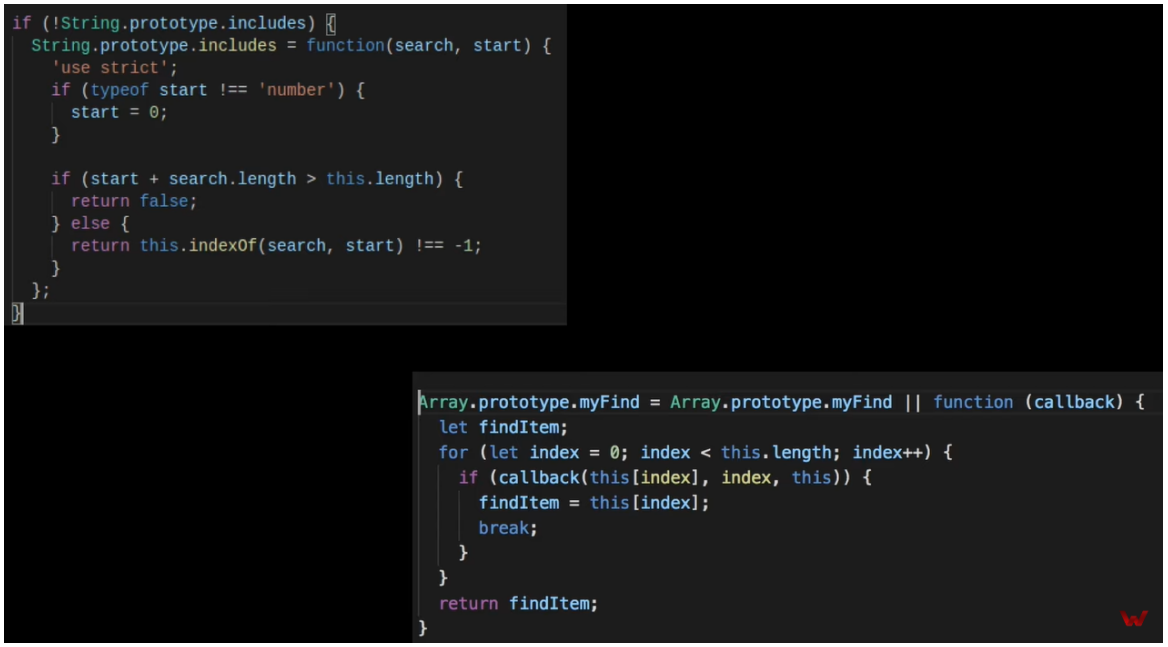
Создать объект без прототипа:

- Object.create(null)

▼ Почему расширение нативных JavaScript-объектов это плохая практика?

Это называется **манкипатчинг**.

1. Это может привести к конфликтам имен и перезаписи существующих методов или свойств.
2. Это может привести к несовместимости с будущими версиями языка или библиотеками.
3. Это может привести к более сложному и трудному для понимания коду.



```
if (!String.prototype.includes) {
  String.prototype.includes = function(search, start) {
    'use strict';
    if (typeof start !== 'number') {
      start = 0;
    }

    if (start + search.length > this.length) {
      return false;
    } else {
      return this.indexOf(search, start) !== -1;
    }
  };
}
```

```
Array.prototype.myFind = Array.prototype.myFind || function (callback) {
  let findItem;
  for (let index = 0; index < this.length; index++) {
    if (callback(this[index], index, this)) {
      findItem = this[index];
      break;
    }
  }
  return findItem;
}
```

▼ Что такое `NaN`? Как определить, что значение равно `NaN`?

NaN (Not a Number) - не число, оно получается когда математическая функция сработала неверно. Не равно ничему, включая самому себе.

Проверка:

- **isNaN()** - это глобальная функция в JavaScript, пытается преобразовать свой аргумент в число перед проверкой. Если аргумент не может быть преобразован в число, функция вернет `true`. Это означает, что строка, символ или объект будут считаться `NaN`, если они не могут быть корректно преобразованы в число.

```
console.log(isNaN(NaN)); // true
console.log(isNaN(123)); // false
console.log(isNaN('Hello')); // true
console.log(isNaN(undefined)); // true
console.log(isNaN({})); // true
```

- **Number.isNaN()** - это статический метод глобального объекта `Number`, введенный в ECMAScript 6 (ES6). В отличие от глобальной функции `isNaN()`, которая преобразует свой аргумент в число перед проверкой, `Number.isNaN()` возвращает `true` только в том случае, если переданный аргумент является строго равным `NaN`. Он не преобразует аргумент в число

```
console.log(Number.isNaN(NaN)); // true
console.log(Number.isNaN(123)); // false
console.log(Number.isNaN('Hello')); // false
console.log(Number.isNaN(undefined)); // false
console.log(Number.isNaN({})); // false
```

▼ Что такое объектная обертка (Wrapper Objects)?

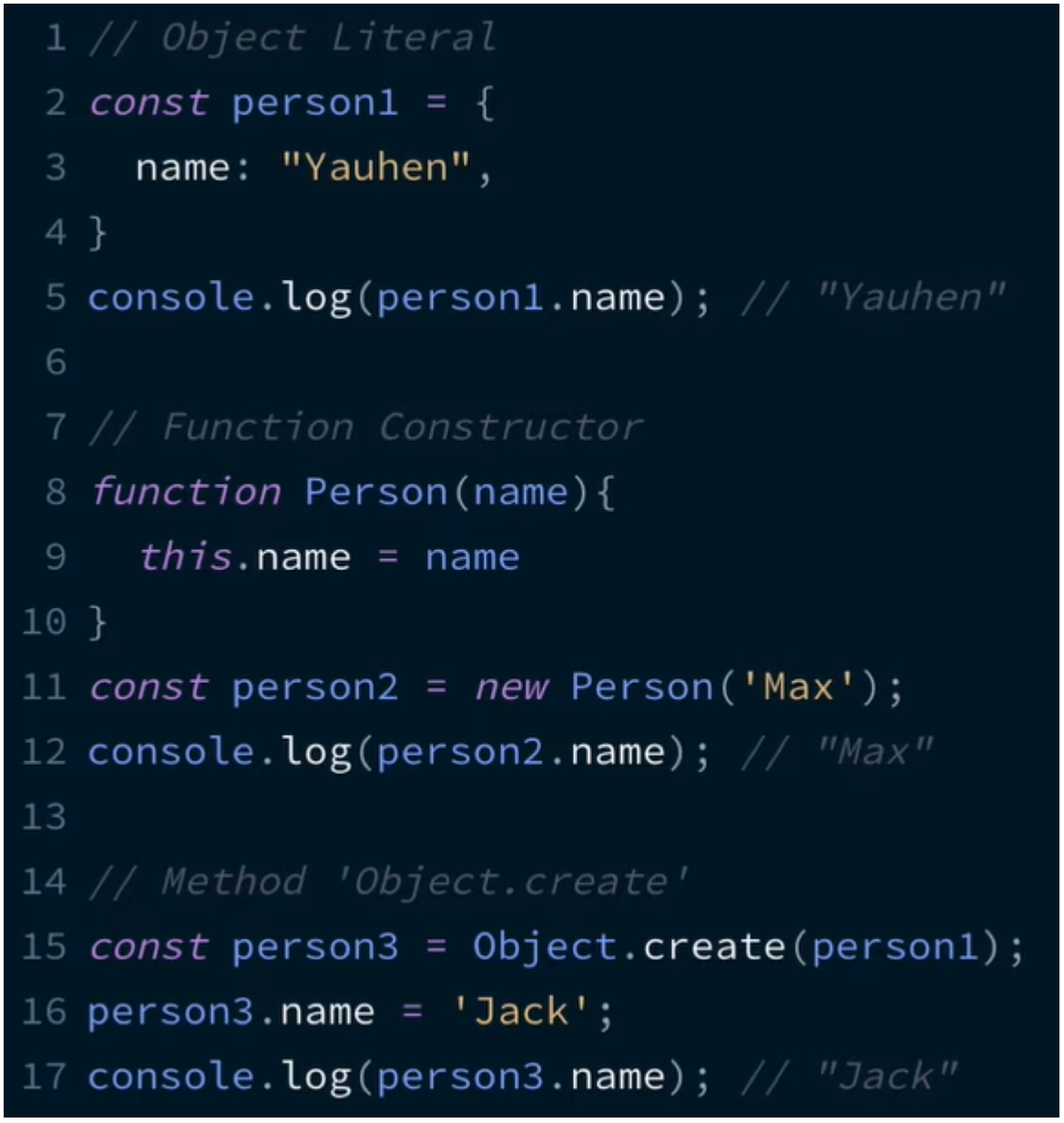
Объектные обертки (Wrapper Objects) - это временные объекты, которые создаются для примитивных типов данных (строки, числа, булевые значения), чтобы предоставить им доступ к методам и свойствам объектов.

У каждого примитива, кроме null и undefined есть объект обертка.

```
// Primitive
const name = "Yauhen";
console.log(typeof name); // string

// No errors
console.log(name.toUpperCase());                // "YAUHEN"
// Wrapper Object 'String'
console.log(new String(name).toUpperCase());  // "YAUHEN"
```

▼ Как в JavaScript создать объект?

1. Объектный литерал - const obj = {}.
2. С помощью функции конструктора.
3. С помощью Object.create().
4. С помощью Класса.



```
// Object Literal
const person1 = {
  name: "Yauhen",
}
console.log(person1.name); // "Yauhen"

// Function Constructor
function Person(name){
  this.name = name
}
const person2 = new Person('Max');
console.log(person2.name); // "Max"

// Method 'Object.create'
const person3 = Object.create(person1);
person3.name = 'Jack';
console.log(person3.name); // "Jack"
```

▼ Для чего используется ключевое слово `new`?

Ключевое слово `new` используется в JavaScript для создания нового объекта на основе функции-конструктора или класса.

Ключевое слово **new** делает 4 вещи:

1. Создает пустой объект.
2. Привязывает к пустому объекту значение this.
3. Устанавливает **prototype**.
4. Возвращает значение this.

```
function Employee(name, position) {
  this.name = name;
  this.position = position;
};

const person = new Employee('Yauhen', 'Front-end dev');
console.log(person.name);       // "Yauhen"
console.log(person.position);   // "Front-end dev"
```

▼ Операторы «И» и «ИЛИ» (`&&` и `||`)?

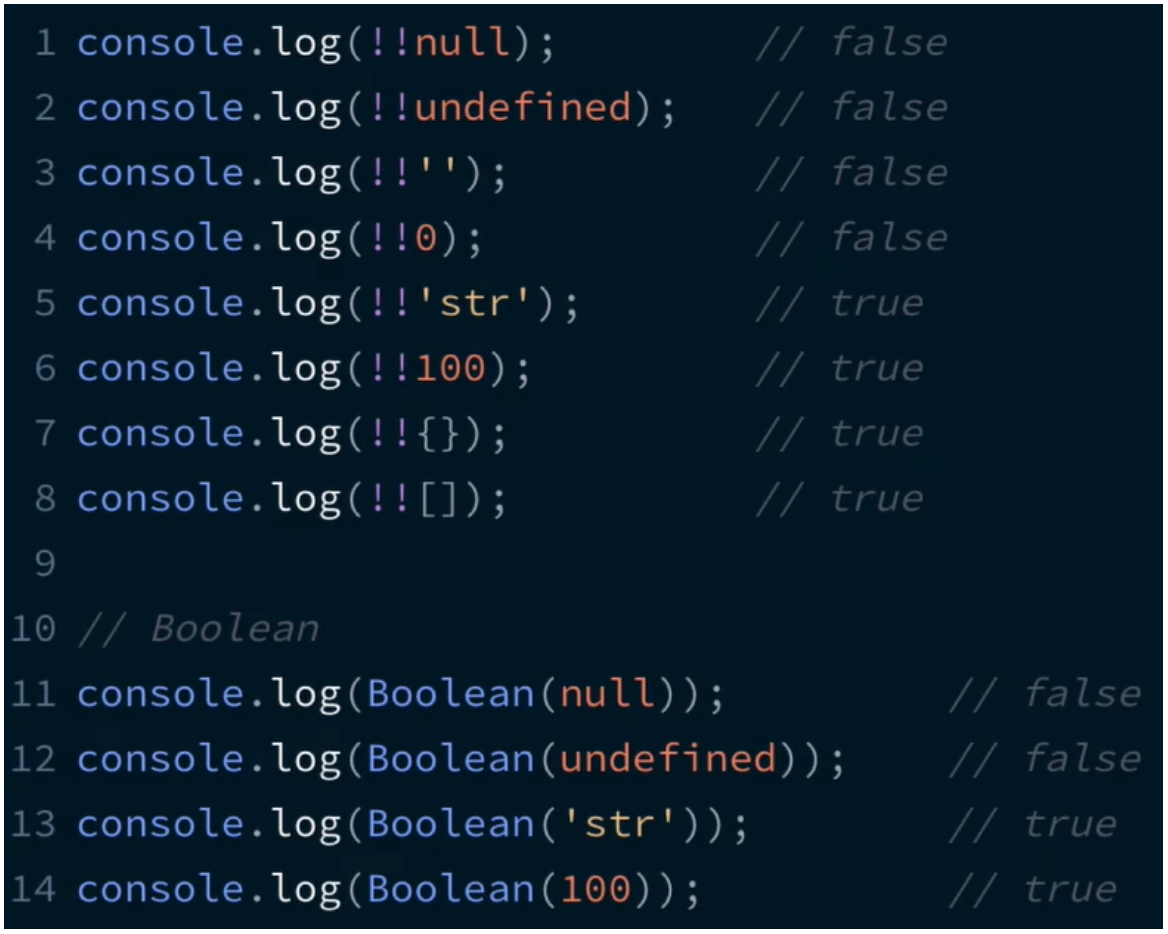
И - спотыкается об ложь.

ИЛИ - спотыкается о правду.

```
// AND - '&&'
console.log(false && 2021 && 'string'); // false
console.log(2021 && {} && 'string'); // 'string'
console.log(true && null && 'string'); // null

// OR - '||'
console.log(false || 2021 || 'string'); // 2021
console.log(2021 || false || 'string'); // 2021
console.log(true || null || 'string'); // null
```

▼ Для чего используется оператор двойного отрицания (`!!`)?

Он приводит значение к boolean типу.

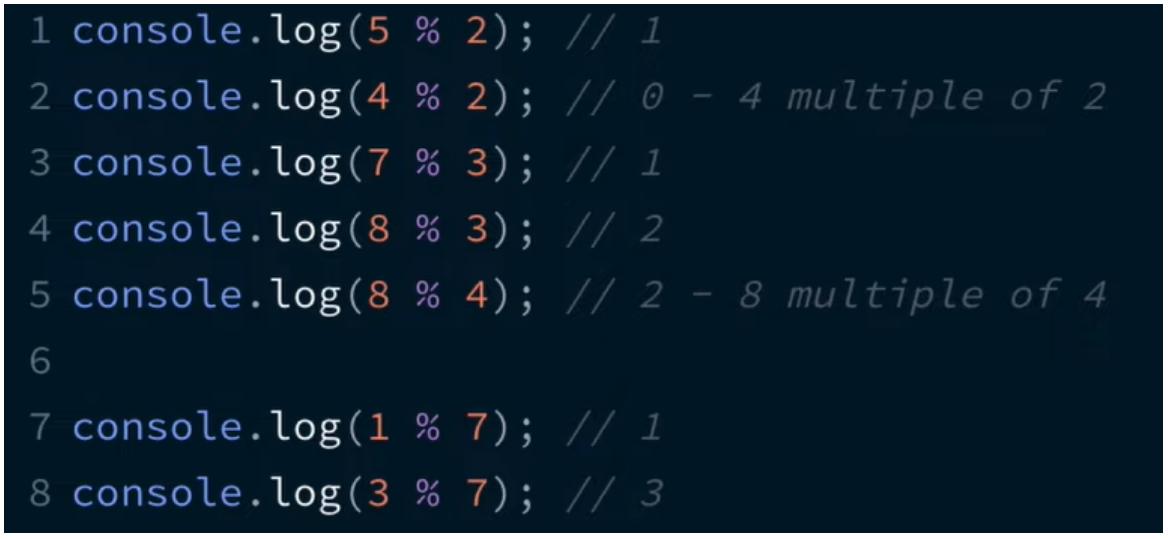


```
console.log(!!null);          // false
console.log(!!undefined);     // false
console.log(!!'');            // false
console.log(!!0);             // false
console.log(!!'str');         // true
console.log(!!100);           // true
console.log(!!{});            // true
console.log(!![]);            // true

// Boolean
console.log(Boolean(null));          // false
console.log(Boolean(undefined));     // false
console.log(Boolean('str'));         // true
console.log(Boolean(100));           // true
```

▼ Для чего используется оператор остатка (`%`)?

Чтоб получить остаток от деления, проверить кратность.

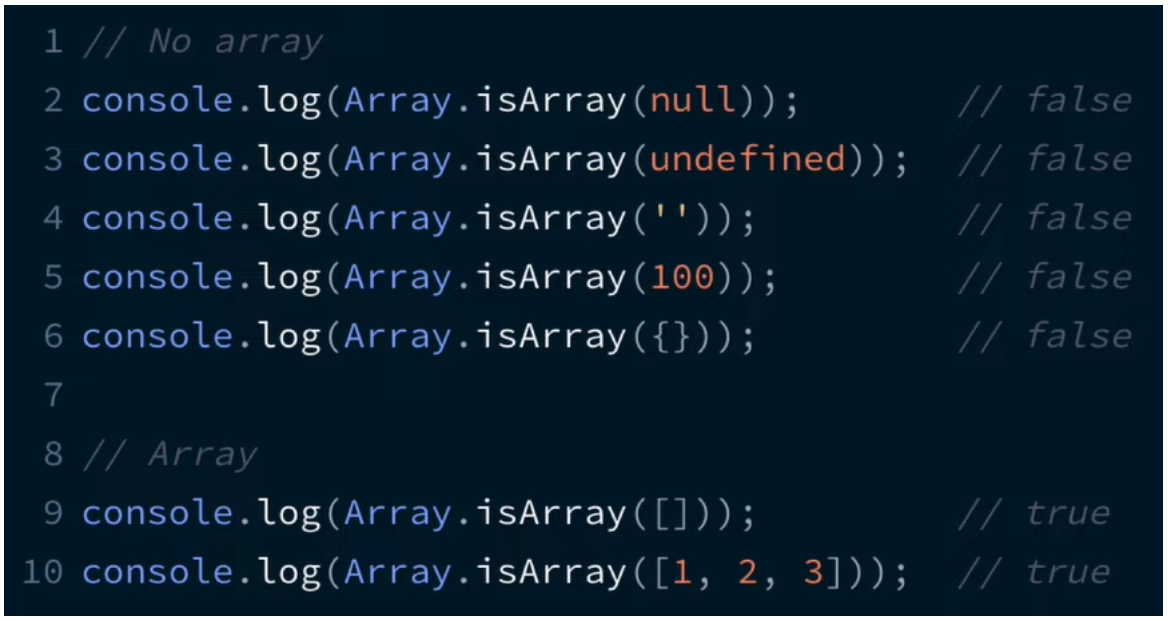
Если делимое число меньше делителя, то результатом будет само число.



```
console.log(5 % 2); // 1
console.log(4 % 2); // 0 - 4 multiple of 2
console.log(7 % 3); // 1
console.log(8 % 3); // 2
console.log(8 % 4); // 2 - 8 multiple of 4

console.log(1 % 7); // 1
console.log(3 % 7); // 3
```

▼ Как проверить, является ли значение массивом?

Array.isArray()



```
// No array
console.log(Array.isArray(null));       // false
console.log(Array.isArray(undefined));  // false
console.log(Array.isArray(''));         // false
console.log(Array.isArray(100));        // false
console.log(Array.isArray({}));         // false

// Array
console.log(Array.isArray([]));         // true
console.log(Array.isArray([1, 2, 3]));  // true
```

▼ Как работает boxing/unboxing в JavaScript?

boxing - когда мы вызываем метод у примитива, то примитив оборачивается в объект обертку, и вызывает нужный нам метод, после исполнения объект уничтожается, а нам возвращается результат работы.

unboxing - обратный процесс, распаковка относится к преобразованию ссылочных типов в примитивные, для данной цели можно использовать методы: valueOf(), toString() или Symbol.toPrimitive()

```
// Boxing
const name = "Yauhen";
console.log(typeof name);          // "string"
console.log(name.toUpperCase());   // string=>String=>String.function=>string

// Explicit boxing
const name = new String("Yauhen");
console.log(typeof name);          // "object"
console.log(name.toUpperCase());   // String.function

// Unboxing
const age = new Number(32);
const name = new String("Yauhen");
console.log(typeof age);       // "object"
console.log(typeof name);      // "object"

console.log(typeof age.valueOf());       // "number" basic number type
console.log(typeof age.toString());      // "string" basic character type
console.log(typeof name.valueOf());      // "string" basic data type
console.log(typeof name.toString());     // "string" basic data type
```

▼ Что такое мемоизация? Реализуйте базовую логику функции для мемоизации?

Мемоизация - это прием создания функции, способной запомнить ранее вычисленное значение.

В результате при повторном вызове функции с одинаковыми аргументами, она не будет выполнена, а результат работы вернется из кэша.

```
function memoize(func) {
    const cache = {}; // Объект для кэширования результатов

    return function(...args) {
        const key = JSON.stringify(args); // Создаем ключ н

        if (!cache.hasOwnProperty(key)) { // Если результат
            cache[key] = func.apply(this, args); // Вызывае
        }

        return cache[key]; // Возвращаем результат из кэша
    };
}
```

```
function expensiveOperation(n) {
    console.log("Performing expensive operation for", n);
    return n * 2;
}

const memoizedExpensiveOperation = memoize(expensiveOperati

console.log(memoizedExpensiveOperation(5)); // Вызовет expe
console.log(memoizedExpensiveOperation(5)); // Вернет резул
console.log(memoizedExpensiveOperation(10)); // Вызовет exp
console.log(memoizedExpensiveOperation(10)); // Вернет резу
```

▼ Разница между оператором `in` и методом `.hasOwnProperty()`?

in и цикл **for in** - проверяют не только объект, но и прототипы

.hasOwnProperty - только объект.

```
1 const test = new Object();
2 test.prop = 'just test';
3
4 console.log('prop' in test);          // true
5 console.log('toString' in test);      // true
6
7 console.log(test.hasOwnProperty('prop'));        // true
8 console.log(test.hasOwnProperty('toString'));    // false
```

▼ Разница между глубокой (deep) и поверхностной (shallow) копиями объекта? Как сделать каждую из них?

При копировании объекта в JavaScript существуют два типа копирования: поверхностное (shallow) и глубокое (deep).

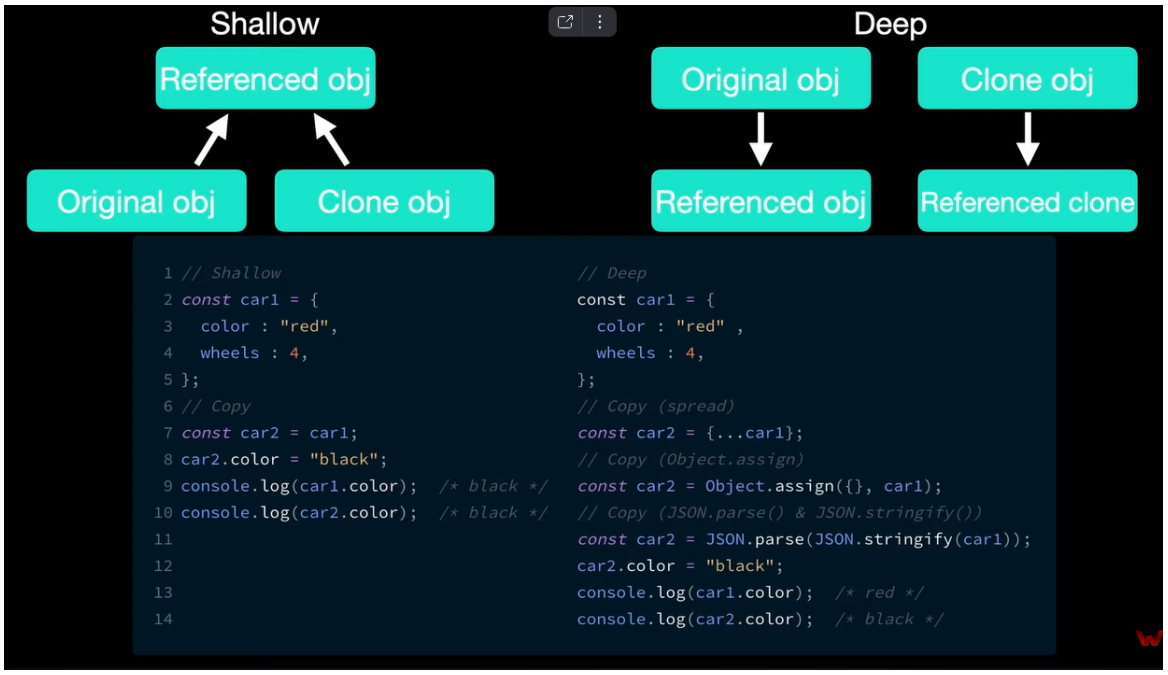
Разница между ними заключается в том, что поверхностная копия создает новый объект, который содержит ссылки на те же самые объекты, что и оригинал, тогда как глубокая копия создает новый объект и копирует все значения свойств в новый объект.

shallow:

- Object.assign
- objA = objB
- {...obj}

deep:

- рекурсивно
- JSON.parse JSON.stringify
- structuredClone



▼ Что такое цепочка вызовов функций (chaining)? Как реализовать такой подход?

Chaining - это подход, при котором методы объекта вызываются один за другим.

В методе возвращать this

```
// Native chaining
'test'.split('').reverse().join('').toUpperCase();

// Promise chaining
myPromise.then(test1).then(test2).catch(handleRejected);

// Custom chaining
const counter = {
  count: 0,
  plusOne() {
    this.count++;
    return this;
  },
  minusOne() {
    this.count--;
    return this;
  },
  showResult() {
    console.log(this.count);
    return this;
  }
}
counter.plusOne().plusOne().minusOne().plusOne().minusOne().showResult();
```

▼ Что такое необъявленная переменная?

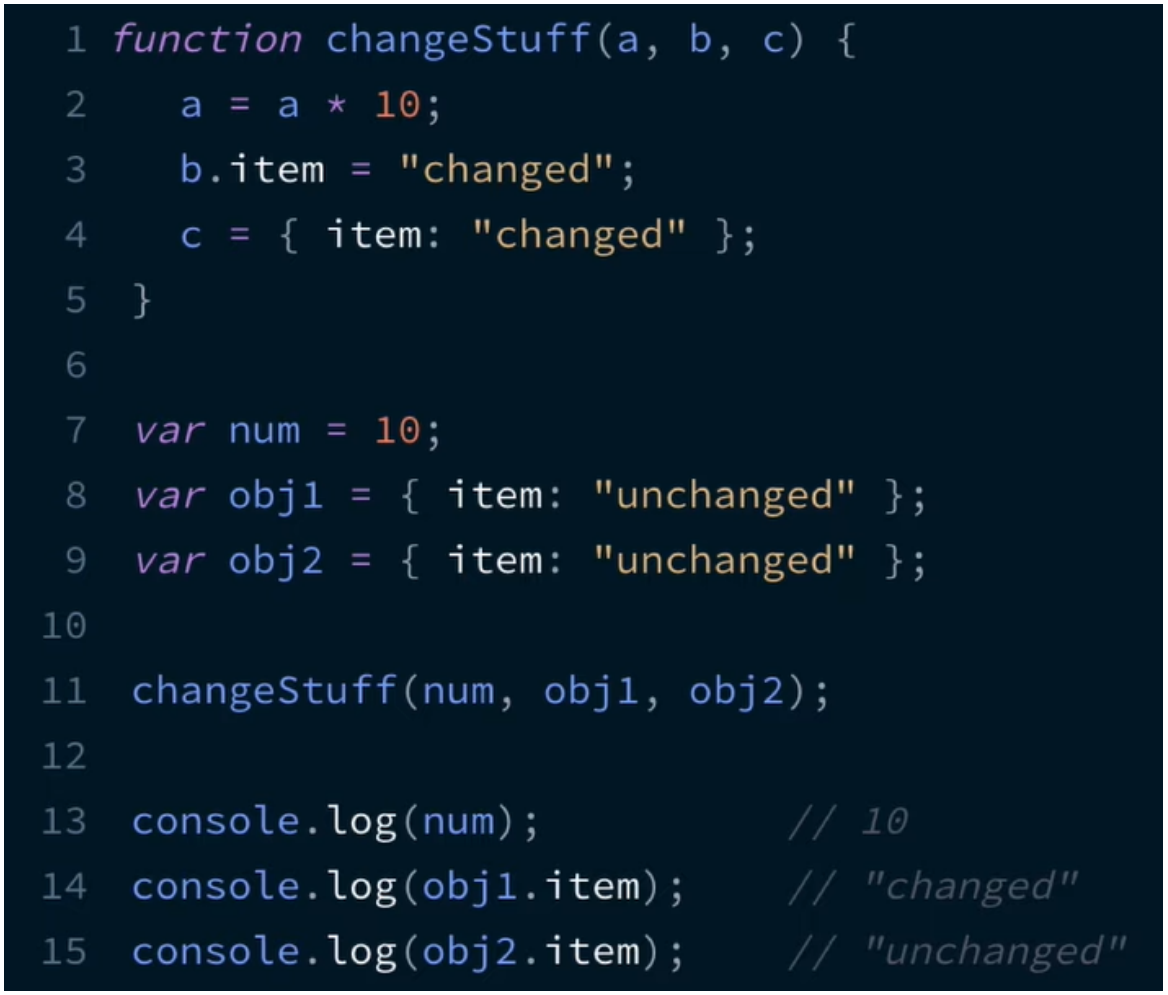
Необъявленная переменная - это переменная, которая используется в программе без объявления ее в начале программы или в функции. Обращение к необъявленной переменной может привести к ошибкам выполнения программы.

```
function func() {
    x = 'test';       // ReferenceError in 'use strict'
}

func();
console.log(x);     // test
```

▼ Как передаются параметры в функцию: по ссылке или по значению?

Примитивы по значению

Ссылочные по ссылке



```
function changeStuff(a, b, c) {
    a = a * 10;
    b.item = "changed";
    c = { item: "changed" };
}

var num = 10;
var obj1 = { item: "unchanged" };
var obj2 = { item: "unchanged" };

changeStuff(num, obj1, obj2);

console.log(num);          // 10
console.log(obj1.item);    // "changed"
console.log(obj2.item);    // "unchanged"
```

▼ Что такое прототип объекта в JavaScript?

Прототип это обьект, который послужил шаблоном для создания дочернего объекта, предоставил ему свои поля и методы.

▼ Как работает метод `Object.create()`?

Метод `Object.create()` используется для создания нового объекта с заданным прототипом.

```
Object.create(proto, propertiesObject)
```

proto: Объект, который станет прототипом нового объекта.

propertiesObject (необязательный): Объект, определяющий дескрипторы свойств нового объекта.

```
const animal = {
    type: 'mammal',
    sound: 'makes noise',
```

```
    makeSound: function() {
        console.log(this.sound);
    }
};

const dog = Object.create(animal);
dog.sound = 'barks';

dog.makeSound(); // Output: barks
```

▼ Разниц между `Object.freeze()` и `Object.seal()`?

Подробнее тут

- **Object.freeze()** - делает объект неизменяемым навсегда. (разфризить нельзя).

  Запрещает добавлять/удалять/изменять свойства. Устанавливает `configurable: false, writable: false` для всех существующих свойств.

  Метод возвращает замороженный объект.

- **Object.seal()** - можно только изменять значения существующих свойств (обратного пути также нет).
  Метод `Object.seal()` запечатывает объект, предотвращая добавление новых свойств и делая все существующие свойства не настраиваемыми. Значения представленных свойств всё ещё могут изменяться, поскольку они остаются записываемыми.

  Запрещает добавлять/удалять свойства. Устанавливает `configurable: false` для всех существующих свойств.

Actions	Object.prevent Extensions	Object.seal	Object.freeze
Can add a new property.	✗	✗	✗
Can modify values of existing properties.	✓	✓	✗
Can delete existing properties.	✓	✗	✗
Can reconfigure existing properties.	✓	✗	✗

▼ Разница между методами `.slice()` и `.splice()`?

Метод .slice() - используется для создания нового массива, содержащего выбранные элементы из исходного массива.

Принимает два необязательных аргумента: начальный индекс (включительно) и конечный индекс (НЕ включительно, не обязательный), которые указывают диапазон элементов, которые нужно скопировать. Если аргументы не указаны, возвращается копия всего исходного массива, если указан только один аргумент, возвращает копию массива начиная с указанного индекса и до конца массива.

Метод .splice() - используется для изменения исходного массива путем удаления, замены или добавления элементов.

Принимает три аргумента: начальный индекс, количество удаляемых элементов и, необязательно, элементы, которые нужно добавить в массив.

Возвращает массив удаленных элементов.

Если количество удаляемых элементов равно 0 и указаны элементы для вставки, элементы будут вставлены в указанную позицию без удаления.

```
const array = [1, 2, 3, 4, 5];

const slicedArray = array.slice(1, 4);
console.log(slicedArray); // Output: [2, 3, 4]
console.log(array); // Output: [1, 2, 3, 4, 5]

const splicedArray = array.splice(1, 3, 'a', 'b', 'c');
console.log(splicedArray); // Output: [2, 3, 4]
console.log(array); // Output: [1, 'a', 'b', 'c', 5]
```

▼ Как работают методы `.find()`, `.findIndex()` и `.indexOf()`?

- Метод **.find(() ⇒ {})** используется для поиска первого элемента в массиве, который удовлетворяет заданному условию в функции, возвращает этот элемент.
- Метод **.findIndex(() ⇒ {})** работает аналогично методу .find(), но возвращает индекс найденного элемента в массиве вместо его значения.

- Метод **.findLastIndex(() ⇒ {})** работает аналогично методу **.findIndex()**, но ищет с конца массива.
- Метод **indexOf(item, from)** ищет `item` начиная с индекса `from` (не обязательный) и возвращает индекс, на котором был найден искомый элемент, в противном случае `-1`.

```
// find
const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const found = arr.find(el => el > 5);
console.log(found);          // 6

// findIndex
const arr = ["John", "Arthur", "Tommy", "Finn"];
const foundIndex = arr.findIndex(el => el === "Arthur");
console.log(foundIndex);     // 1

// indexOf
const arr = ["John", "Arthur", "Tommy", "Finn"];
const foundIndex = arr.indexOf("Arthur");
console.log(foundIndex);     // 1
```

▼ Плюсы и минусы использования `use strict` ?

Плюсы использования use strict:

- Помогает избежать ошибок и неопределенного поведения в коде, так как требует более строгого синтаксиса и правил использования переменных.
- Позволяет более точно определить, что происходит в коде, так как запрещает использование устаревших функций и методов.
- Улучшает производительность кода, так как некоторые оптимизации могут быть применены только при использовании use strict.

Минусы использования use strict:

- Некоторые старые скрипты могут не работать с use strict, так как требуется более строгий синтаксис.

- Некоторые разработчики могут забыть добавить use strict в свой код и продолжить использовать устаревшие методы и функции.

Плюсы:

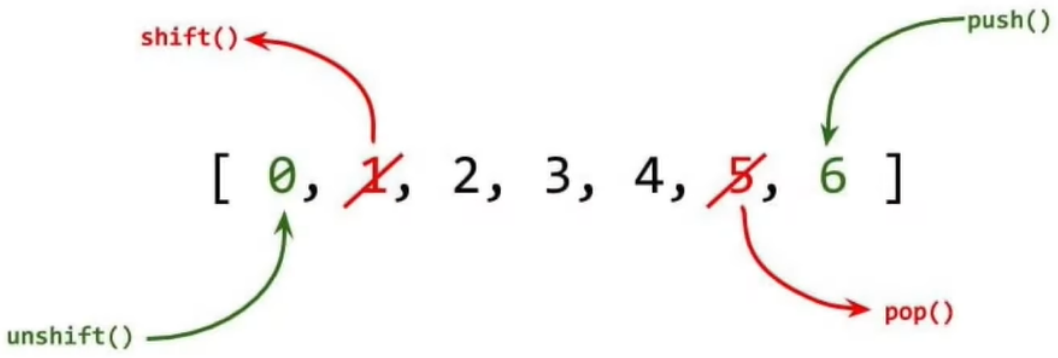
- Не позволяет случайно создавать глобальные переменные
- Любое присваивание, которое в обычном режиме завершается неудачей, в строгом режиме выдаст исключение
- При попытке удалить неудаляемые свойства, выдаст исключение
- Требует, чтобы имена параметров функции были уникальными
- this в глобальной области видимости равно undefined
- Перехватывает распространенные ошибки, выдавая исключения
- Исключает неочевидные особенности языка

Минусы:

- Нельзя использовать некоторые особенности языка
- Нет доступа к function.caller и function.arguments
- Объединение скриптов, в строгом режиме может вызвать проблемы

▼ Разница между методами `.push()`, `.pop()`, `.shift()` и `.unshift()`?

- .push() добавляет один или несколько элементов в конец массива.
- .pop() удаляет последний элемент массива и возвращает его.
- .shift() удаляет первый элемент массива и возвращает его, сдвигая все остальные элементы влево.
- .unshift() добавляет один или несколько элементов в начало массива, сдвигая все остальные элементы вправо.



▼ Плюсы и минусы иммутабельности? Как достичь иммутабельности в JS?

Плюсы иммутабельности:

- Безопасность: неизменяемые объекты невозможно изменить случайно или специально, что обеспечивает безопасность данных.
- Предсказуемость: неизменяемые объекты всегда имеют одинаковое состояние, что обеспечивает предсказуемость поведения программы.
- Переиспользование: неизменяемые объекты могут быть переиспользованы в разных частях программы, что уменьшает количество дублирующего кода.

Минусы иммутабельности:

- Неэффективность: создание новых объектов вместо изменения существующих может привести к увеличению потребления памяти и процессорного времени.
- Сложность: использование неизменяемых объектов может быть сложнее, чем изменяемых, особенно в случае больших и сложных структур данных.

Достичь иммутабельности в JS можно с помощью следующих методов:

- Использование const для объявления переменных, которые не должны изменяться.
- Использование Object.freeze() для замораживания объектов, чтобы они не могли быть изменены.
- Использование библиотек, таких как Immutable.js, которые предоставляют неизменяемые версии стандартных объектов JavaScript.
- Использование функционального программирования, которое поощряет использование неизменяемых данных и функций без побочных эффектов.

▼ Типы всплывающих окон в JavaScript?

Их нельзя стилизировать, имеют системные стили

В JavaScript существует несколько типов всплывающих окон:

1. **Alert** это простое всплывающее окно, которое выводит сообщение и кнопку "OK". Используется для информирования пользователя о

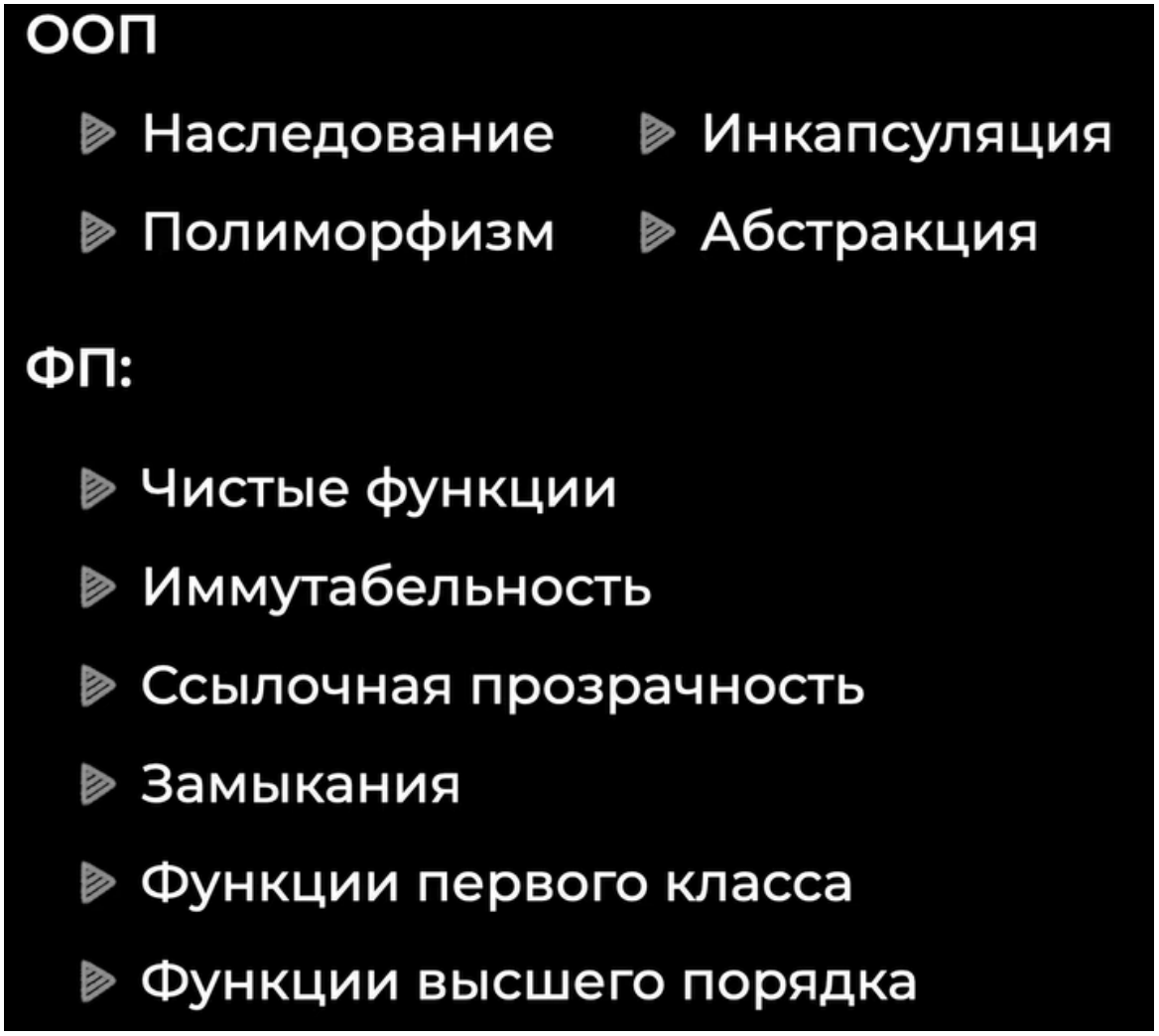
каком-либо событии.

2. **Confirm** это всплывающее окно, которое выводит сообщение и две кнопки "OK" и "Cancel". Используется для получения подтверждения от пользователя на выполнение какого-либо действия.
3. **Prompt** это всплывающее окно, которое выводит сообщение, поле для ввода и две кнопки "OK" и "Cancel". Используется для получения данных от пользователя.
4. **Custom** это всплывающее окно, которое создается с помощью HTML, CSS и JavaScript. Может быть настроено на любой вид в зависимости от потребностей приложения.

▼ Типы объектов JavaScript?

- Array
- String
- Date
- Number
- Boolean
- Function
- Math
- RegExp
- Object

▼ Парадигмы программирования в JavaScript?



▼ ООП:

Идея в том, что любую структуру в приложении можно представить в виде объекта или класса. Эти свойства и методы можно расширять, изменять, наследовать, передавать и т.д.

первая концепция - это объект.

Объект это элементарная единица ООП, которая представляет из себя набор свойств.

Каждое свойство состоит из имени и значения ассоциированного с этим именем.

Это удобная единица, но не настолько вариативная как хотелось бы.

Поэтому дополнительной надстройкой над объектом можно назвать класс.

В ООП Класс - это расширяемый шаблон кода для создания объектов, который устанавливает у них начальное значение и реализацию поведения.

Конструктор - особый метод который вызывается в момент инициализации класса.

Значения из конструктора присваиваются экземпляру.

- Наследование создание новых классов на основе существующих
- Полиморфизм - способность вызывать один и тот же метод для разных объектов и каждый объект будет реагировать по разному

Compile-time (статический) полиморфизм:

- **Перегрузка методов (Method Overloading):** Это когда в классе объявлены несколько методов с одним и тем же именем, но с разными параметрами (типами, количеством аргументов).

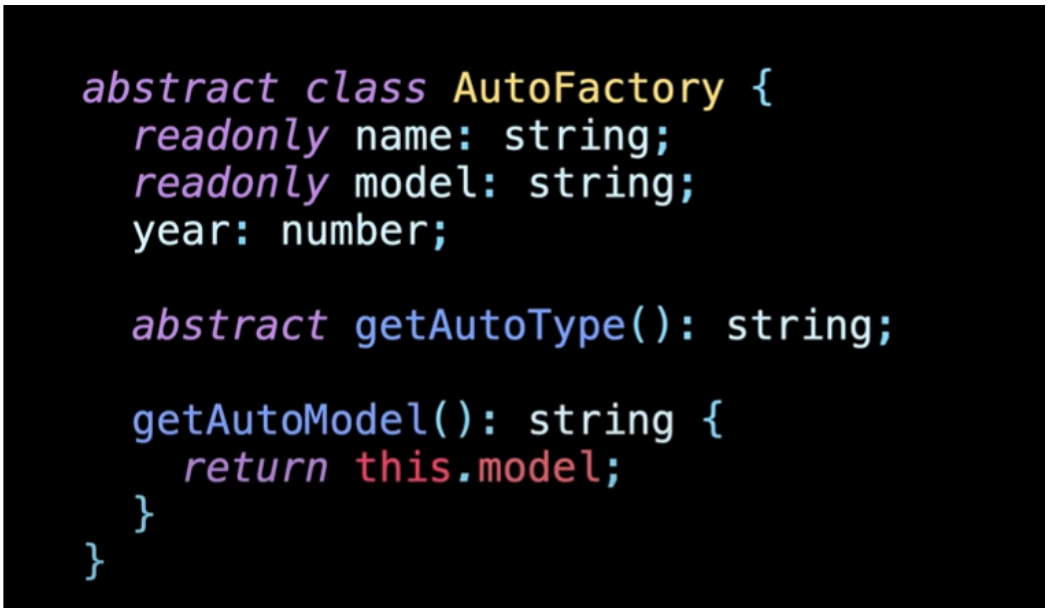
```
class Calculator:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c
```

Run-time (динамический) полиморфизм:

- **Перегрузка операторов (Operator Overloading):** Это позволяет объектам разных классов использовать общие операторы, но с различными реализациями.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
```

- Инкапсуляция - сокрытие состояния объекта от прямого доступа из вне. Общедоступные свойства мы можем пометить флагом public, а скрытых - с помощью модификатора private. Обратиться напрямую к private свойству мы не можем, но можем установить геттер. Тем самым прочитать можно, изменить - нельзя
- Абстракция - способ создания простой модели, которая содержит только важные свойства. На основании абстрактного класса можно создать обычный класс с extends



```
abstract class AutoFactory {
  readonly name: string;
  readonly model: string;
  year: number;

  abstract getAutoType(): string;

  getAutoModel(): string {
    return this.model;
  }
}
```

- Интерфейс - определяет свойства и методы которые объект или класс может реализовать. Внутри себя он не содержит реализаций. Пример - оглавление книги. Структура есть, описания нет. Readonly только для чтения. Можно наследоваться с помощью ключевого слова implements
- Композиция - подход при котором одни классы или объекты содержат в себе другие. Мы создаем классы(деталей) и затем в общем классе(машине) инициализируем эти детали. Реализовываем делегирование.

  Без общего класса Car все сущности внутри него перестанут существовать.

```
class Car {
  constructor() {
    // Composition
    this.engine = new Engine();
    this.wiring = new Wiring();
    this.fuelPump = new FuelPump();
  }
```

- Агрегация - когда мы другой класс не инициализируем внутри, а передаем как параметр. Он инициализируется снаружи и может существовать вне класса Car

```
class Car {
  constructor(autopilot) {
    this.engine = new Engine();
    this.wiring = new Wiring();
    this.fuelPump = new FuelPump();
    this.autopilot = autopilot;
  }

  start() {
    this.engine.start();
    this.wiring.start();
    this.fuelPump.start();
  }
}
```

▼ ФП:

сводится к написанию функций.

приложения написанные на ОП проходят через 3 основных этапа:

1)Получение аргументов для работы

2) выполнение функций

3)Возвращение результата

Но такой подход имеет ряд недостатков

1) большое количество кода

2) трудности в чтении и анализе логики

3) сложность в организации отдельных модулей и структур

4) сложность декомпозиции и разделения логики

- Чистые функции
- Иммутабельность
- Ссылочная прозрачность
- Замыкания
- Функции первого порядка
- Функции высшего порядка

▼ Типы ошибок в JavaScript?

- **SyntaxError** ошибки в написании кода, которые приводят к невозможности его выполнения.
- **TypeError** ошибки, которые возникают при попытке выполнить операцию над неправильным типом данных.
- **RangeError** число не попадает в диапазон.
- **ReferenceError** ошибка ссылки, когда логика ссылается на несуществующую переменную
- **URIError** - при encodieURI и decodeURI
- **ThrowingError** - возникает при использовании ключевого слова throw
- **CustomError** - созданный пользователем класс ошибок
- Логические ошибки - ошибки, которые не приводят к сбою программы, но приводят к неправильным результатам.
- Ошибки времени выполнения - ошибки, которые возникают во время выполнения программы, например, попытка обращения к несуществующей переменной.
- Ошибки времени загрузки - ошибки, которые возникают при загрузке веб-страницы.

- Ошибки обработки событий - ошибки, которые возникают при обработке событий, например, если обработчик не был назначен или неправильно настроен.

▼ Разница между `typeof` и `instanceof`?

Оператор typeof используется для определения типа данных переменной или выражения, например, "string", "number", "boolean", "undefined", "function" и "object". Он возвращает строку, указывающую на тип данных.

Оператор instanceof используется для проверки, является ли объект экземпляром определенного класса или конструктора. Он возвращает логическое значение true или false.

▼ JavaScript статически, или динамически типизированный язык?

Динамически

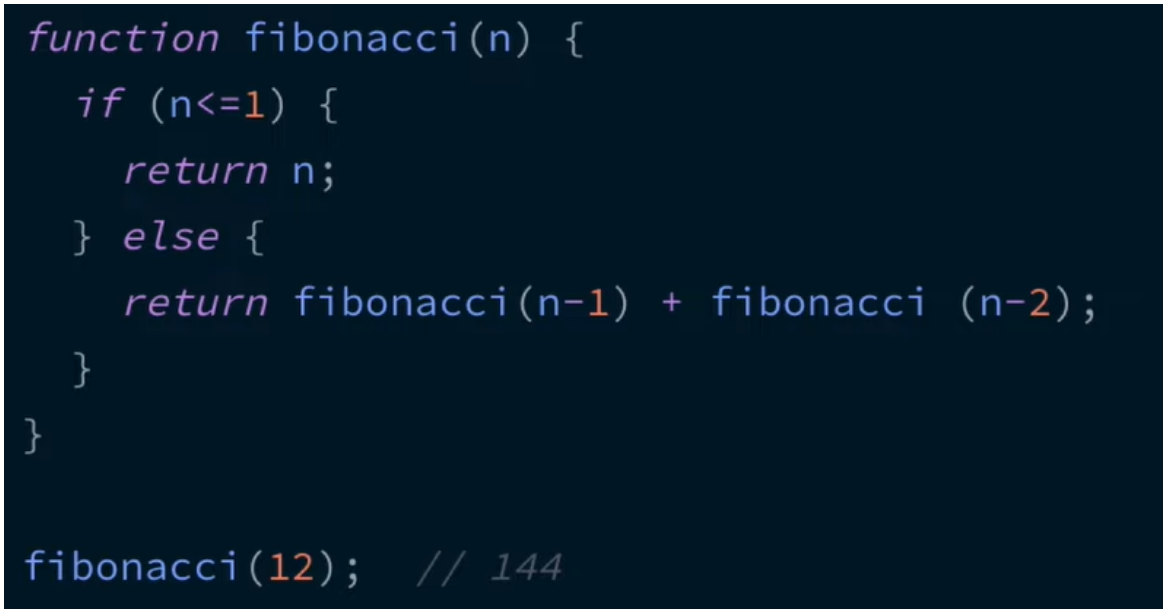
▼ Что такое регулярное выражение (Regular Expression)?

Регулярное выражение (Regular Expression) - это шаблон, состоящий из символов и метасимволов, который используется для поиска и обработки текстовых данных.

▼ Что такое рекурсия?

Рекурсивная функция(Рекурсия) - это когда функция вызывает саму себя.

Рекурсивный вызов - вызов рекурсивной функции



▼ Что такое прототип (Prototype) объекта?

Прототип - это объект, который используется в качестве шаблона для создания других объектов. Каждый объект в JavaScript имеет ссылку на свой прототип, который определяет его свойства и методы. Если свойство или метод не найден в самом объекте, JavaScript будет искать его в прототипе этого объекта, затем в прототипе прототипа и так далее, пока не будет найдено или не достигнут конец цепочки прототипов (Object.prototype). Прототипы позволяют создавать иерархию объектов и упрощать наследование свойств и методов.

▼ Какие методы используются в регулярных выражениях?

- `test()` - проверяет, существует ли совпадение между регулярным выражением и указанной строкой. Возвращает `true` или `false`.

```
const text = 'The quick brown fox jumps over the lazy dog';
const pattern = /fox/;
console.log(pattern.test(text)); // Output: true
```

- `exec()` - ищет совпадения между регулярным выражением и указанной строкой. Если совпадение найдено, он возвращает массив, содержащий информацию о совпадении. Если совпадение не найдено, возвращается `null`.

```
const text = 'The quick brown fox jumps over the lazy dog';
const pattern = /fox/;
```

```
console.log(pattern.exec(text)); // Output: ["fox", index:
```

- `match()` - возвращает массив всех совпадений между регулярным выражением и строкой.

```
const text = 'The quick brown fox jumps over the lazy dog';
const pattern = /o/g;
console.log(text.match(pattern)); // Output: ["o", "o", "o"
```

- `replace()` - заменяет совпадения с регулярным выражением в строке на указанную подстроку или результат вызова функции.

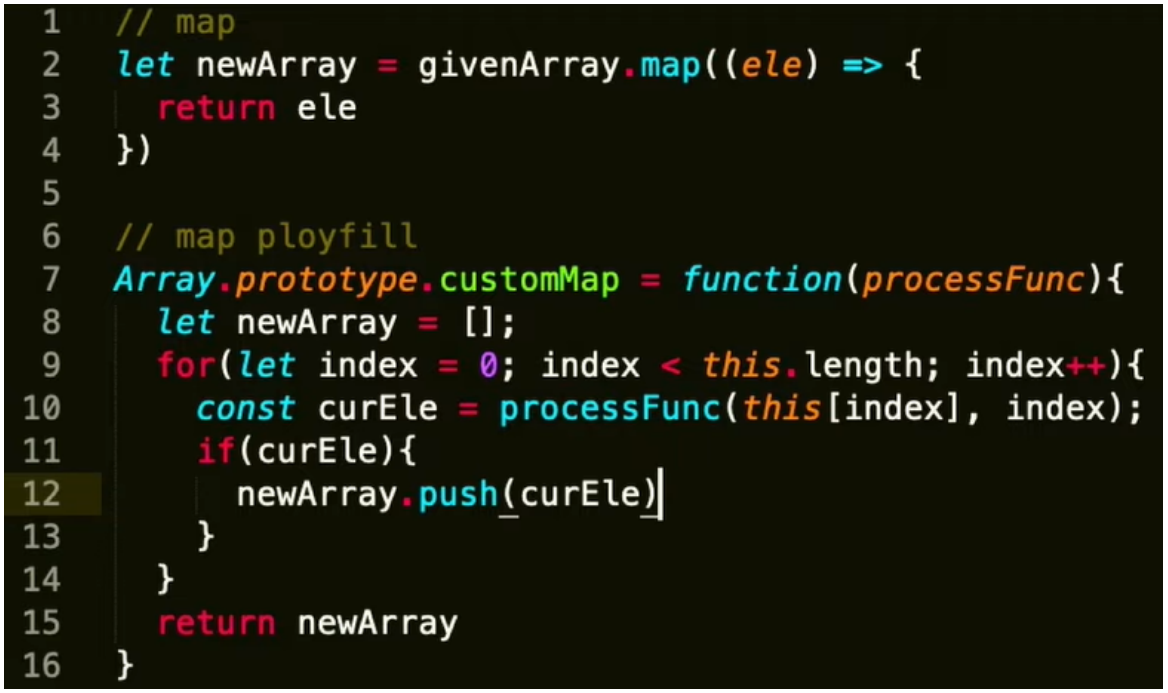
```
const text = 'The quick brown fox jumps over the lazy dog';
const pattern = /fox/;
console.log(text.replace(pattern, 'cat')); // Output: "The
```

- `search()` - возвращает индекс первого совпадения между регулярным выражением и строкой. Если совпадение не найдено, возвращается `-1`.

```
const text = 'The quick brown fox jumps over the lazy dog';
const pattern = /fox/;
console.log(text.search(pattern)); // Output: 16
```

▼ Что такое полифил (polyfill)?

Полифил - это код, который эмулирует функциональность, которая может отсутствовать в старых версиях браузеров или в браузерах, которые не поддерживают новые стандарты языка.



```
// map
let newArray = givenArray.map((ele) => {
  return ele
})

// map ployfill
Array.prototype.customMap = function(processFunc){
  let newArray = [];
  for(let index = 0; index < this.length; index++){
    const curEle = processFunc(this[index], index);
    if(curEle){
      newArray.push(curEle)
    }
  }
  return newArray
}
```

▼ Что такое `switch/case`? Правила использования `switch/case`?

switch/case - является альтернативой if/else, и представляет собой более наглядный способ выполнения кода в зависимости от переданного условия.

В конце указывается не обязательный блок default.

- Условие число или строка
- Не допускается дублирование значений
- Блок default является опциональным
- break опционален

```
const score = 20;

switch(score){
  case 10:
    console.log("Score value is 10");
    break;
  case 20:
      console.log("Score value is 20");
      break;
  default:
      console.log("Score value is neither 10 or 20");
}
```

▼ Типы функций по способности принимать другие функции?

- **Функция первого порядка** - не принимает и не возвращает функцию, но может быть передана как аргумент или возвращена как результат работы функции.
- **Функция высшего порядка** - принимает функцию в качестве аргумента или возвращает функцию.
- **Унарная функция** функция, которая принимает только 1 аргумент, который не является функцией.

```
// First Order Function
const firstOrder = () => console.log('Hello');

// Higher Order Function
const higherOrder = firstOrderReturn => firstOrderReturn();
higherOrder(firstOrder);

// Unary Function
const unaryFunction = (a) => console.log(`${a} + world!`);
```

▼ Что такое выражения (expression) и инструкции (statement) в JavaScript?

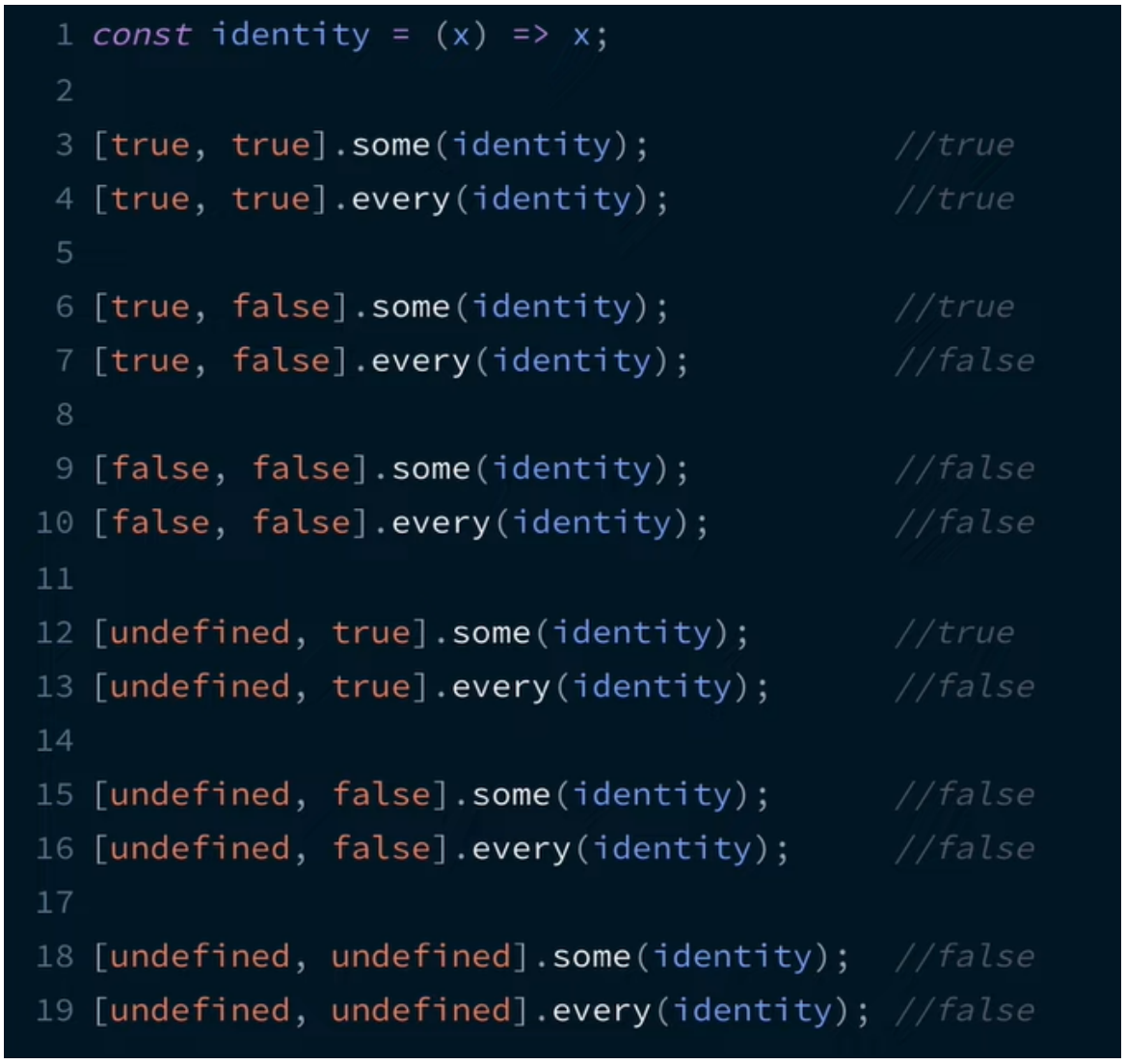
- **Выражение** это фрагмент кода, комбинация значений, переменных и функций, которые в ходе своего выполнения приводят к вычислению и возврату некоторого значения.
- **Инструкция** это фрагмент кода, который выполняет какое-то определенное действие. Примеры инструкций: if, while, for, else, switch

```
// expressions
const x = 5;
const y = getAnswer();
const z = 2 + 5;
Math.random();

// statements
if (a > 5) {
  9 + 9;
}

if (true) { return 2 }

switch (expr) {
  case 'Oranges':
    console.log('Oranges are $0.59 a pound.');
    break;
  case 'Papayas':
    console.log('Mangoes and papayas are $2.79 a pound.');
    break;
  default:
    console.log(`Sorry, we are out of ${expr}.`);
}
```

▼ Разница между `.some()` и `.every()`?

- **.some()** - соответствует ли хотя бы один элемент какому-то условию.

- **.every()** - соответствуют ли все элементы какому-то условию.



```
const identity = (x) => x;

[true, true].some(identity);            //true
[true, true].every(identity);           //true

[true, false].some(identity);           //true
[true, false].every(identity);          //false

[false, false].some(identity);          //false
[false, false].every(identity);         //false

[undefined, true].some(identity);       //true
[undefined, true].every(identity);      //false

[undefined, false].some(identity);      //false
[undefined, false].every(identity);     //false

[undefined, undefined].some(identity);  //false
[undefined, undefined].every(identity); //false
```

▼ Как сгенерировать случайное число в JavaScript?

Math.random()

Лучше использовать специализированные библиотеки, т.к. Math.random генерирует псевдослучайное число, от 0 до 1 не включая 1. работает благодаря зашитому в него алгоритму PRNG - генератор псевдослучайных чисел

```js
// Getting a random number between 0 and 1
function getRandom() {
  return Math.random();
}

// Getting a random integer between two values, inclusive
function getRandomIntInclusive(min, max) {
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min + 1) + min);
}
```

▼ Типы операторов в JavaScript?

- Операторы присваивания = += %=
- Операторы сравнения === == !== !=
- Арифметические операторы * + - /
- Битовые операторы & | ~
- Логические операторы && || !
- Строковый оператор +
- Тернарный оператор ? :
- Оператор запятая ,
- Унарный оператор delete typeof
- Операторы отношения in, instance
- Оператор нулевого слияния ??

Результат выражения `a ?? b` возвращает первый аргумент, если он не `null/undefined`, иначе второй.

Операторы присваивания	x = y, x += y, x %= y, ...
Операторы сравнения	x === y, x == y, x !== y, x != y
Арифметические операторы	x * y, x + y, x - y, ...
Битовые операторы	x & y, x | y, ~ x, ...
Логические операторы	x && y, x || y, !x
Строковый оператор	"test1" + "test2"
Тернарный оператор	condition ? x : y
Оператор запятая	let x = 0, y = 1
Унарный оператор	delete x, typeof y, ...
Операторы отношения	x in y, x instance y, ...
Оператор нулевого слияния	x ?? y

▼ Разница между параметром и аргументом функции?

Аргументы - при вызове.

Параметры - при создании функции.

▼ Правила задания имён для переменных и функций в JavaScript?

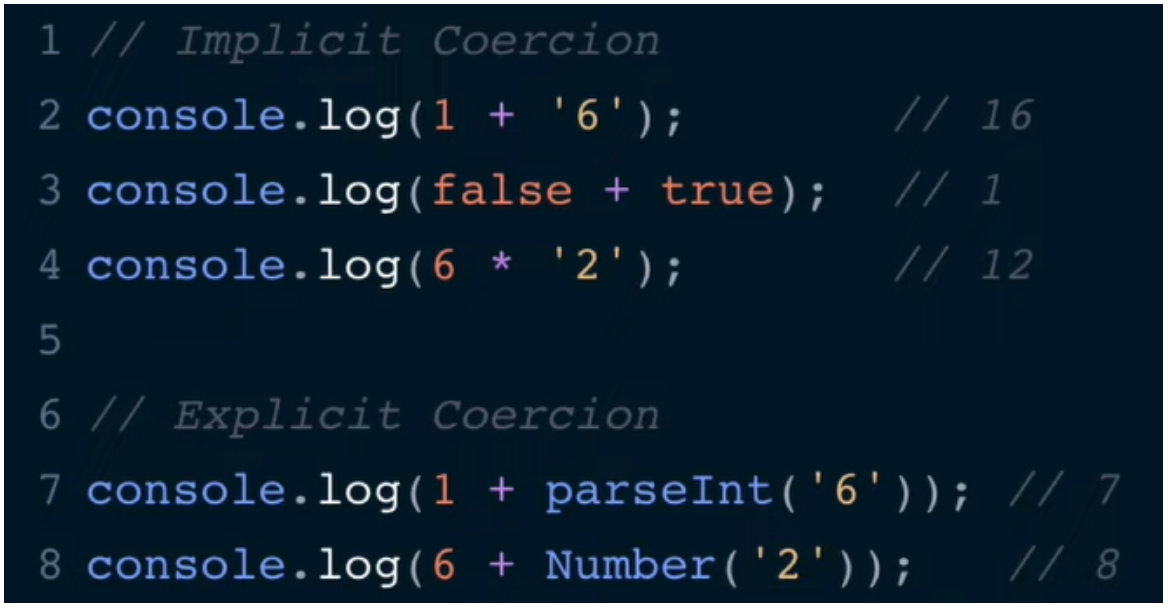
- Имя переменной должно содержать только буквы, цифры или символы доллара и нижнего подчеркивания.
- Первый символ не должен быть цифрой.
- Имя функции должно понятно отражать, что она делает и что возвращает.

```
// Correct
let width = 0;
let name = 'Jack';
let _nickName = 'Max';
let $ = 1;
const checkAge = () => {};
const checkName = () => {};
const sortList = () => {};

// Incorrect
let 1name = 'Jack';
let 2nickName = 'Max';
let my-name = 'Stan';
const admin = () => {};
const listOfPeople = () => {};
```

▼ Разница между явным и неявным преобразованием (Implicit and Explicit Coercion)?

Неявное преобразование - это способ приведения значения к другому типу без участия разработчика. Данный тип преобразования происходит автоматически.

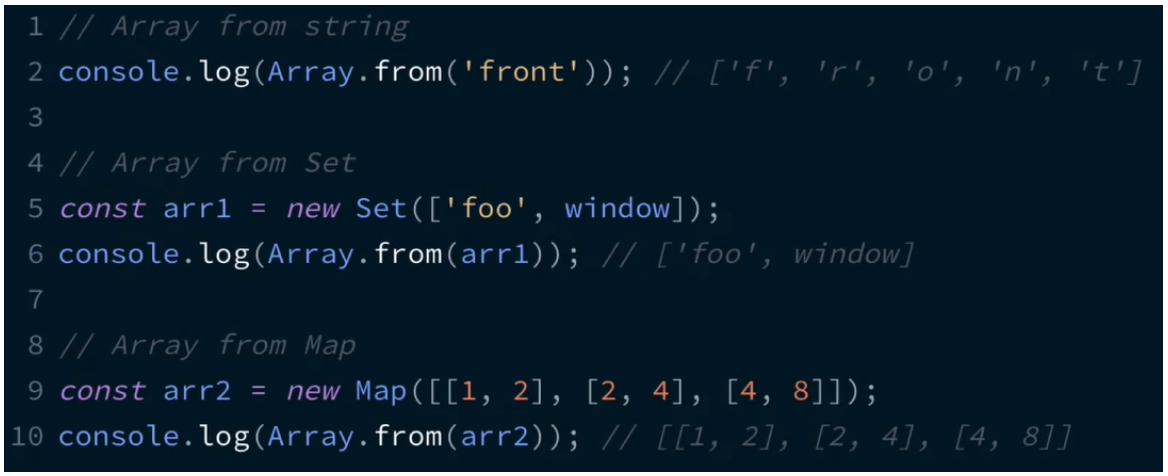
Явное преобразование - предполагает участие разработчика в приведении значения к другому типу. Пример: parseInt(), Number(), String().



```
// Implicit Coercion
console.log(1 + '6');        // 16
console.log(false + true);   // 1
console.log(6 * '2');        // 12

// Explicit Coercion
console.log(1 + parseInt('6')); // 7
console.log(6 + Number('2'));   // 8
```

▼ Для чего применяется метод `Array.from()`?

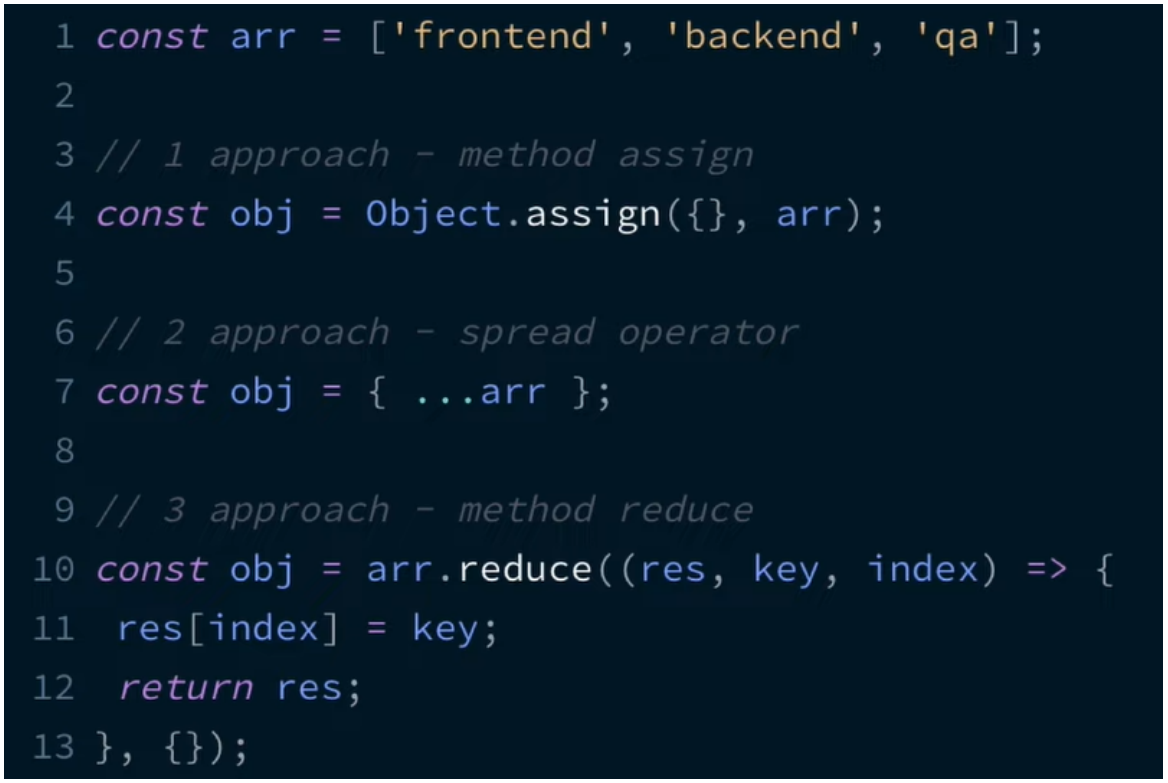
Метод Array.from(obj, cb, this) - создает новый массив на основе переданного объекта, объект должен быть либо массивоподобным, как строка или объект **arguments**, либо итерируемым как коллекции **Set** и **Map**. cb для преобразования элемента перед его добавлением в массив.



```
// Array from string
console.log(Array.from('front')); // ['f', 'r', 'o', 'n', 't']

// Array from Set
const arr1 = new Set(['foo', window]);
console.log(Array.from(arr1)); // ['foo', window]

// Array from Map
const arr2 = new Map([[1, 2], [2, 4], [4, 8]]);
console.log(Array.from(arr2)); // [[1, 2], [2, 4], [4, 8]]
```

▼ Назовите способы преобразования массива в объект?

1. Object.assign({}, arr)
2. { ...arr }
3. .reduce

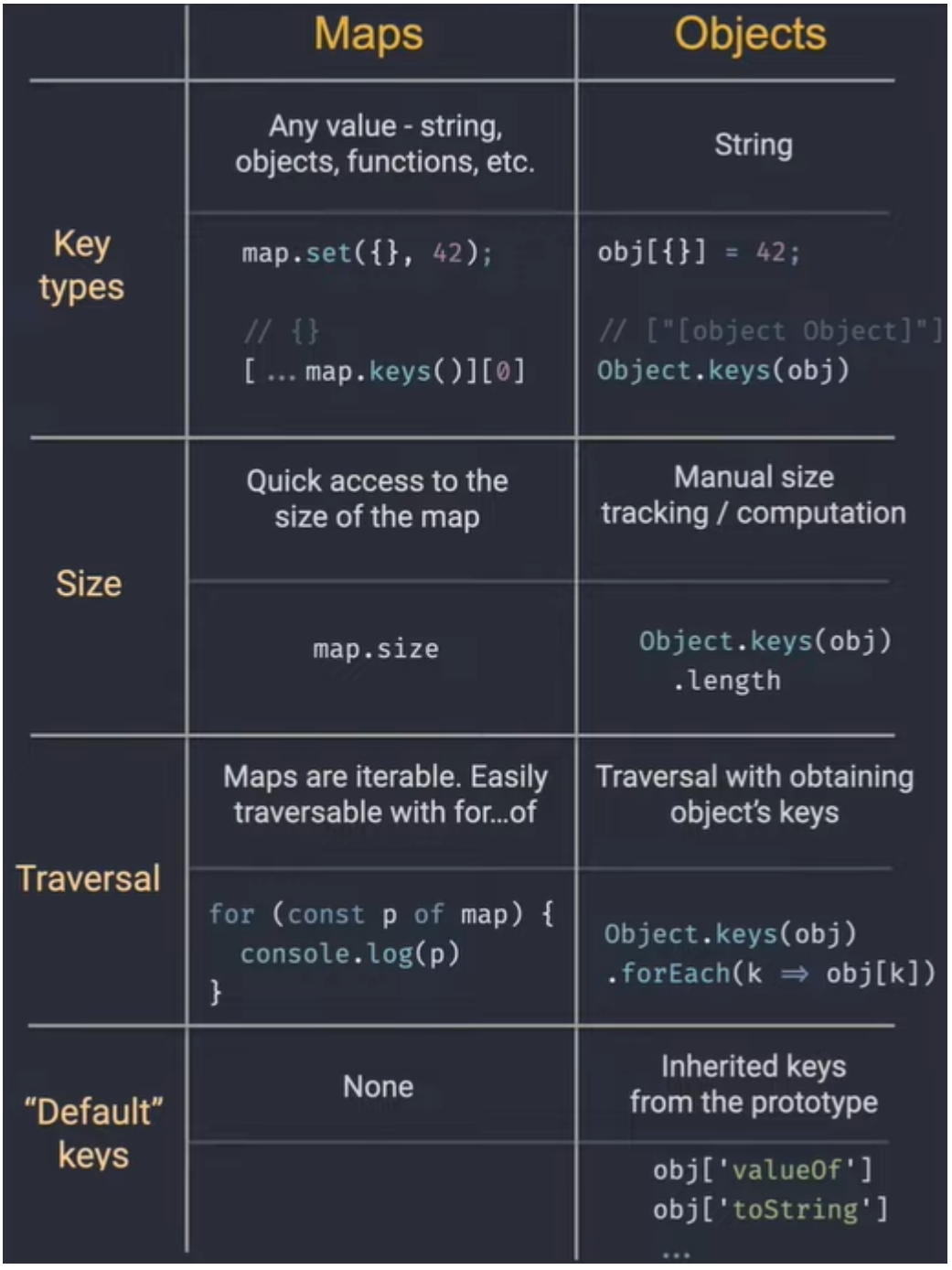


```
const arr = ['frontend', 'backend', 'qa'];

// 1 approach - method assign
const obj = Object.assign({}, arr);

// 2 approach - spread operator
const obj = { ...arr };

// 3 approach - method reduce
const obj = arr.reduce((res, key, index) => {
 res[index] = key;
 return res;
}, {});
```

▼ Разница между `Object` и `Map`?

Основная разница между **Object** и **Map** заключается в том, что Object использует строки, числа и символы в качестве ключей, а **Map** может использовать любой тип данных в качестве ключей.

Ключи в Map хранятся в порядке добавления



	Maps	Objects
Key types	Any value - string, objects, functions, etc.	String

	`map.set({}, 42);`
	
	`// {}`
	`[ ... map.keys()][0]`
		`obj[{}] = 42;`
		
		`// ["[object Object]"]`
		`Object.keys(obj)`

Size	Quick access to the size of the map	Manual size tracking / computation
	`map.size`	`Object.keys(obj)` `.length`
Traversal	Maps are iterable. Easily traversable with for...of	Traversal with obtaining object's keys

	`for (const p of map) {`
	`  console.log(p)`
	`}`
		`Object.keys(obj)`
		`.forEach(k ⇒ obj[k])`

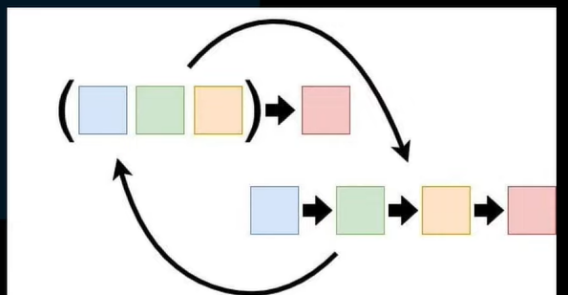
"Default" keys	None	Inherited keys from the prototype

		`obj['valueOf']`
		`obj['toString']`
		`...`

▼ Что такое каррирование?

Каррирование - это процесс преобразования функции с несколькими аргументами в последовательность функций, каждая из которых имеет только 1 аргумент. Работает за счет замыканий.

```
// example
function curry(f) {
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// using
function sum(a, b) {
  return a + b;
}
let carriedSum = curry(sum);
alert( carriedSum(1)(2) ); // 3
```



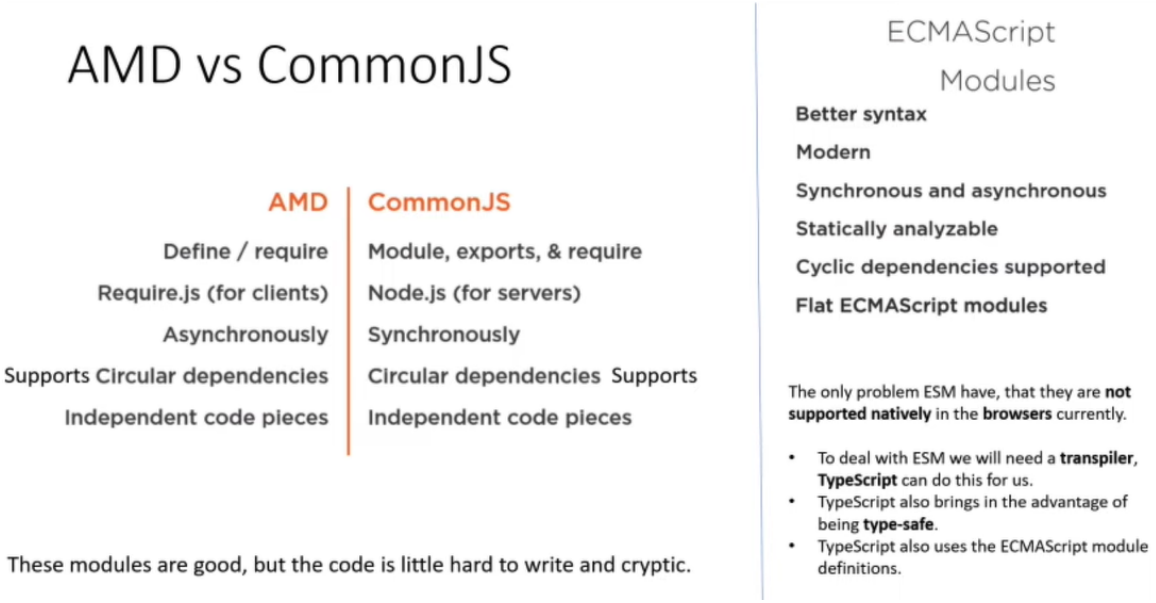
▼ Для чего используется свойство `.dataset`?

Свойство **dataset** позволяет считывать или устанавливать любые data-атрибуты на HTML элемент и хранить в них данные.

```
<ul>
  <li data-id="1541" data-episode="1">Дарт Мол</li>
  <li data-id="9434" data-episode="4">Дарт Вейдер</li>
  <li data-id="5549" data-episode="4">Дарт Сидиус</li>
</ul>

<script>
  const items = document.querySelectorAll('li');
  const firstItem = items[0];
  console.log(firstItem.dataset); // { id: '1541', episode: '1' }
</script>
```

▼ Каким образом можно обмениваться кодом между файлами?

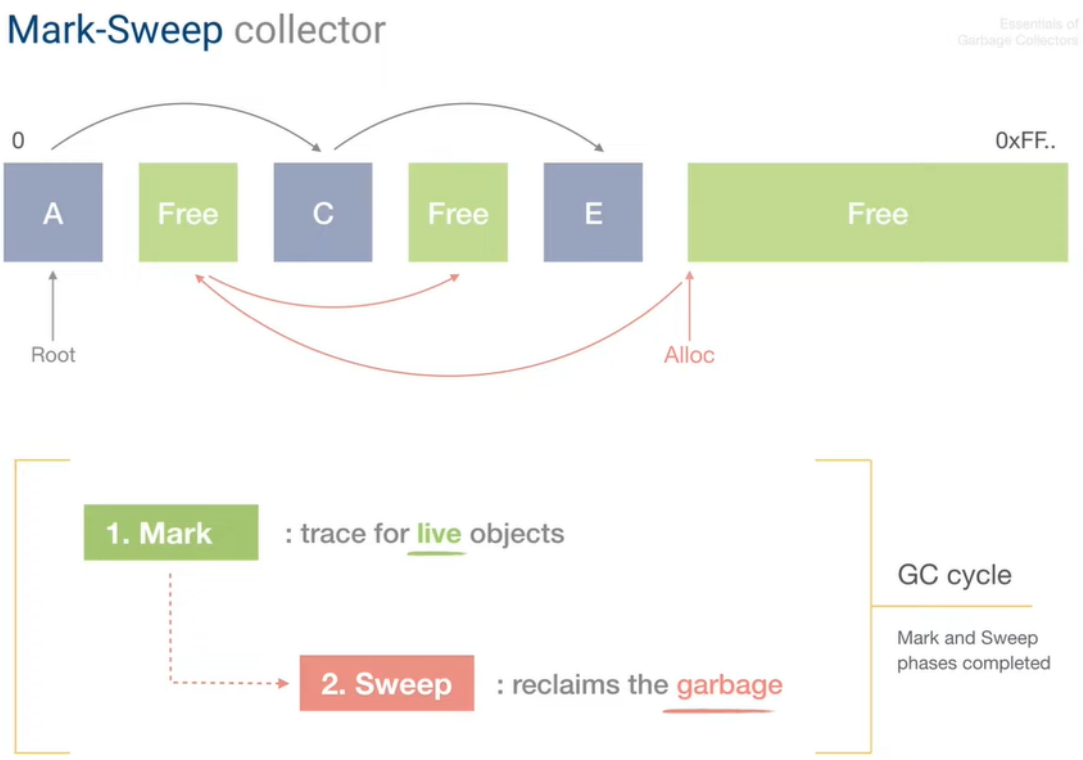
- export, require
- import, export



▼ Как работает «сборщик мусора» в JavaScript?

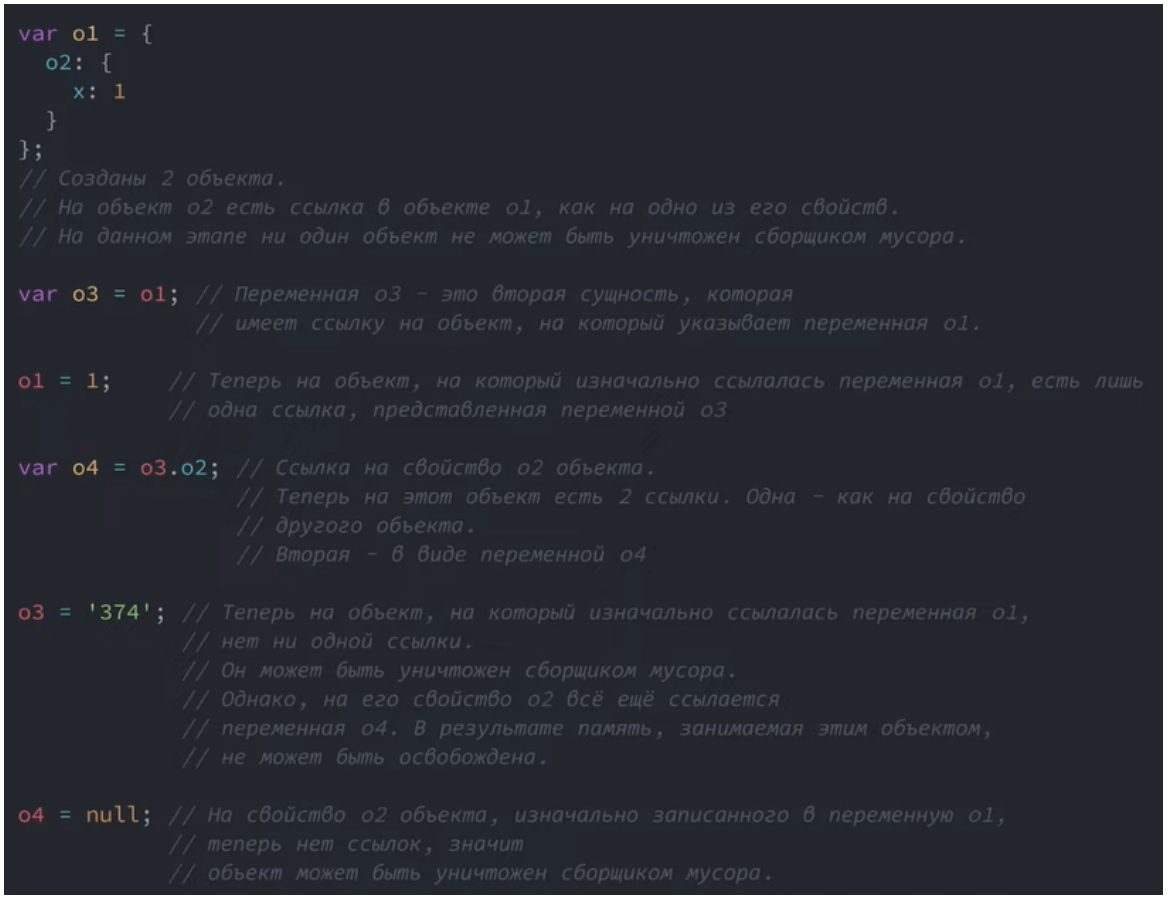
Сборщик мусора в большинстве браузеров использует алгоритм Mark-and-Sweep.

1. **Отслеживание ссылок**: Сборщик мусора отслеживает все ссылки на объекты в программе. Объект считается доступным для сборки мусора, если на него нет ссылок из глобальной области видимости или из других объектов.
2. **Маркировка и очистка**: Сборщик мусора использует алгоритм маркировки и очистки для определения, какие объекты доступны для сборки мусора. В этом процессе сначала помечаются все объекты, которые достижимы из корневого набора объектов (например, глобальных переменных и активных стеков вызова). Затем все объекты, которые не были помечены, считаются недоступными и могут быть безопасно удалены из памяти.
3. **Освобождение памяти**: После того как недоступные объекты были идентифицированы и помечены, сборщик мусора освобождает память, занимаемую этими объектами, чтобы она могла быть повторно использована другими частями программы.
4. **Циклические ссылки**: Один из особых случаев для сборщика мусора - это обработка циклических ссылок, когда два или более объекта имеют ссылки друг на друга. Для обнаружения и сборки таких циклических ссылок сборщик мусора использует алгоритмы, основанные на подсчете ссылок или алгоритмы обхода графа.



▼ Что такое утечки памяти?

Утечки памяти - это ситуация, когда выделенная для программы память не освобождается после того, как она больше не нужна.



```
var o1 = {
  o2: {
    x: 1
  }
};
// Созданы 2 объекта.
// На объект o2 есть ссылка в объекте o1, как на одно из его свойств.
// На данном этапе ни один объект не может быть уничтожен сборщиком мусора.

var o3 = o1; // Переменная o3 - это вторая сущность, которая
             // имеет ссылку на объект, на который указывает переменная o1.

o1 = 1;      // Теперь на объект, на который изначально ссылалась переменная o1, есть лишь
             // одна ссылка, представленная переменной o3

var o4 = o3.o2; // Ссылка на свойство o2 объекта.
                // Теперь на этот объект есть 2 ссылки. Одна - как на свойство
                // другого обьекта.
                // Вторая - в виде переменной o4

o3 = '374'; // Теперь на обьект, на который изначально ссылалась переменная o1,
            // нет ни одной ссылки.
            // Он может быть уничтожен сборщиком мусора.
            // Однако, на его свойство o2 всё ещё ссылается
            // переменная o4. В результате память, занимаемая этим объектом,
            // не может быть освобождена.

o4 = null; // На свойство o2 объекта, изначально записанного в переменную o1,
           // теперь нет ссылок, значит
           // объект может быть уничтожен сборщиком мусора.
```

▼ Назовите основные типы утечек памяти в JavaScript?

- Глобальные переменные.
- Таймеры или забытые коллбэки.
- Замыкания.
- Ссылки на объекты DOM за пределами DOM дерева.

▼ Как работает контекст выполнения (execution context) в JavaScript?

Контекст выполнения (execution context) в JavaScript - это абстрактное понятие, которое описывает окружение, в котором выполняется код JavaScript. Каждый раз, когда выполняется какой-либо код в JavaScript, создается контекст выполнения, который включает в себя следующие компоненты:

- Глобальный контекст выполнения. Это базовый, используемый по умолчанию контекст выполнения. Если некий код находится не внутри какой-нибудь функции, значит этот код принадлежит глобальному контексту. Глобальный контекст характеризуется наличием глобального объекта, которым, в случае с браузером,

является объект `window`, и тем, что ключевое слово `this` указывает на этот объект. В программе может быть лишь один глобальный контекст.

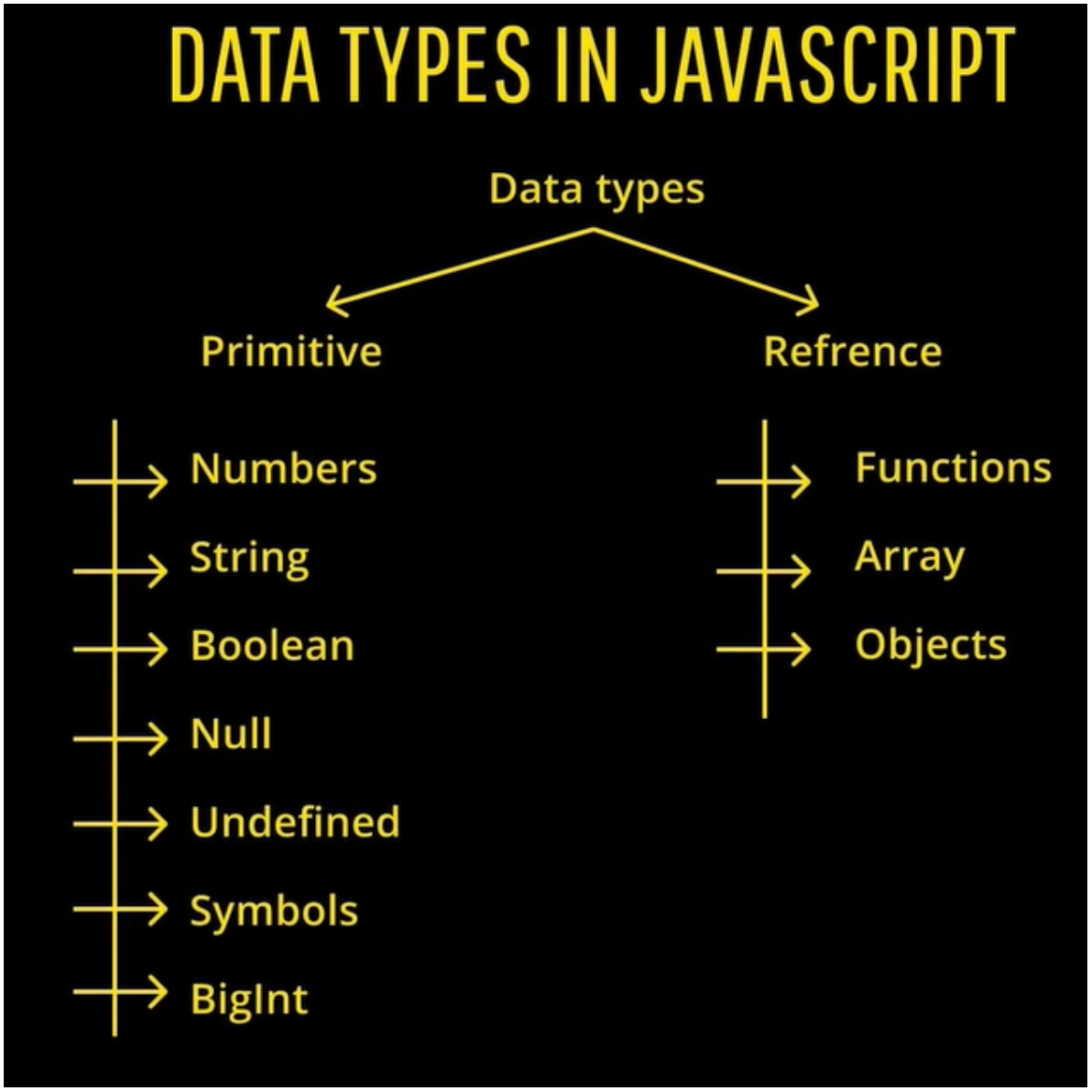
- Контекст выполнения функции. Каждый раз, когда вызывается функция, для неё создаётся новый контекст. Каждая функция имеет собственный контекст выполнения. В программе может одновременно присутствовать множество контекстов выполнения функций. В функциональном контексте выполнения сохраняются все локальные переменные функции, параметры, ссылка на объект `this`, а также информация о блоке кода, который выполняется в данный момент.
- Контекст выполнения функции `eval`. Код, выполняемый внутри функции `eval`, также имеет собственный контекст выполнения. Однако функцией `eval` пользуются очень редко.

Когда функция завершает выполнение, её функциональный контекст выполнения удаляется из стека вызовов, и управление возвращается в контекст, из которого была вызвана функция. Этот процесс повторяется при выполнении кода в JavaScript, создавая и удаляя контексты выполнения в зависимости от потребностей приложения.

▼ Разница между примитивом и объектом?

Примитивные значение нельзя изменить напрямую, т.е. при изменении будет создан новый примитив, а старый удален. Имеют фиксированный размер и передаются по значению.

Объекты - сложные типы данных для хранения свойств и методов, передаются по ссылке.



▼ Что такое Object.is()?

Object.is() - это статический метод в JavaScript, который используется для сравнения двух значений на строгое равенство. Этот метод отличается от операторов сравнения `===` и `==` в том, что он считает `NaN` равен `NaN`, а также `-0` и `+0` как не равные.

▼ Как можно использовать try catch?

try catch finally

try catch

try finally

▼ Минусы JSON parse/stringify?

Происходит сериализация, то есть превращение объекта в строковую форму.

- **Не копирует методы** - если объект содержит функции в качестве свойств, они будут пропущены при сериализации в JSON с помощью `JSON.stringify()`. При парсинге JSON, содержащего функции, `JSON.parse()` также пропустит их.
- **Преобразование ограничено поддерживаемыми типами данных -** `JSON.stringify()` может преобразовывать только примитивные типы данных (строки, числа, булевы значения, null), массивы и объекты. Другие типы данных, такие как объекты с прототипами, не могут быть корректно преобразованы.
- **Сериализация/десериализация Map и Set -** Несмотря на то что JSON поддерживает массивы и объекты, `Map` и `Set` не могут быть напрямую сериализованы в JSON, и при использовании `JSON.stringify()` они будут преобразованы в пустые объекты. При десериализации JSON строки, содержащей `Map` или `Set`, результатом будет просто пустой объект.

▼ Proxy

`new Proxy()` позволяет создавать обертки над объектами и определять специфичное поведение для операций с этими объектами, например, для управления доступом к их свойствам или логгирования операций.

Аргументы

`new Proxy(target, handler)`

- `target`: Целевой объект, над которым будет создан прокси.
- `handler`: Объект-обработчик, который определяет пользовательскую логику для перехватываемых операций с целевым объектом.

Аргументы функций внутри прокси

Функции внутри объекта-обработчика (handler) принимают следующие аргументы:

- `target`: Целевой объект, над которым создан прокси.
- `property`: Имя свойства или метода, к которому происходит доступ.
- `receiver`: Объект, к которому был применен вызов. (не обязательный)

```
// Целевой объект
const targetObject = {
  name: "Alice",
  age: 30
};

// Обработчик, который меняет функциональность целевого объ
const handler = {
  get: function(target, property) {
    console.log(`Чтение свойства ${property}`); // Добавили
    return target[property];
  },
  set: function(target, property, value) {
    console.log(`Запись свойства ${property} со значением $
    target[property] = value;
    return true;
  }
};

// Создаем объект прокси
const proxyObject = new Proxy(targetObject, handler);

console.log(proxyObject.name); // Чтение свойства name / Al
proxyObject.age = 31; // Запись свойства age со значением 3
```

Кейсы использования

1. **Логирование операций**: Можно создать прокси, который логирует доступ к свойствам и методам целевого объекта.
2. **Валидация данных**: Прокси может проверять и валидировать данные, прежде чем они будут установлены в свойства объекта.
3. **Обеспечение безопасности**: Прокси позволяет контролировать доступ к членам объекта, предотвращая нежелательные изменения.
4. **Интерцептирование вызовов методов**: Можно использовать прокси для изменения поведения методов объекта, например, добавляя логику до или после вызова метода.

OOP & FP

▼ Основные принципы ООП?

- **Абстракция -** представляет собой концепцию, которая позволяет скрыть детали реализации объекта и предоставить только необходимый набор данных и операций для работы с ним. Это как использование автомобиля без необходимости знать, как он работает внутри. Вам не нужно знать, как каждая деталь двигателя взаимодействует друг с другом; вам просто нужно знать, как управлять машиной с помощью руля и педалей. Абстракция в программировании позволяет скрыть сложные детали реализации, чтобы облегчить использование объекта или системы.
- **Наследование -** способность объекта/класса наследоваться от другого объекта/класса. Наследование позволяет создавать новые классы на основе существующих (родительских) классов. Подкласс (или дочерний класс) наследует свойства и методы родительского класса, а также может определять свои собственные свойства и методы. Это позволяет избегать повторного кодирования и обеспечивает повторное использование кода.
- **Инкапсуляция** - объединение данных и методов, которые работают с этими данными, в единый объект/класс. Объект скрывает свою внутреннюю реализацию от внешнего мира, предоставляя только определенные методы и свойства для взаимодействия, делается это с помощью геттеров и сеттеров, публичных и приватных полей в объекте/классе. Это позволяет обеспечить безопасность данных и упрощает их использование.
- **Полиморфизм -** позволяет объектам разных классов реагировать на вызовы методов с одинаковыми именами разным для каждого класса способом. Это делает код более гибким и легким для поддержки и расширения, поскольку различные объекты могут вести себя по-разному, но при этом использовать общий интерфейс для взаимодействия с ними.

```
class Human {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    // Метод для приветствия
    greet() {
        console.log(`Привет, меня зовут ${this.name}`);
    }
}

class Employee extends Human {
    constructor(name, age, position) {
        super(name, age);
        this.position = position;
    }

    // Метод для отображения информации о должности
    greet() {
        console.log(`Привет,  я работаю на позиции ${this.p
    }
}

class Developer extends Employee {
    constructor(name, age, position, language) {
        super(name, age, position);
        this.language = language;
    }

    // Метод для отображения информации о языке программиро
    greet() {
        console.log(`Привет, я пишу на ${this.language}`);
    }
}

const man = new Human('Alex', 25);
const rabotnik = new Employee ('Peter', 20, 'manager');
```

```
const frontend = new Developer ('David', 25, 'frontend', 'J

const persons = [man, rabotnik, frontend]

function massGreeting(persons) {
    for(let i = 0; i < persons.length; i++) {
        const person = persons[i];
        person.greet();
    }
}
massGreeting(persons)
// Привет, меня зовут Alex
// Привет, я работаю на позиции manager
// Привет, я пишу на JS.
```

Это и есть полиморфизм, каждый обьект поприветствовал нужным образом, несмотря на то, что классы **Developer** или **Emploee** не задействованы в функции, но отработали соответствующие методы. Т.е. функция работает с разными объектами и вызывает метод, логика в котором может быть разная.

▼ Разница между классовым и прототипным наследованием?

1. **Классовое наследование**:
    - В классовом наследовании используется концепция классов и объектов, подобная той, что применяется в языках программирования, таких как **Java** или **C++**.
    - Наследование происходит путем создания подкласса от родительского класса с использованием ключевого слова `extends`.
    - Все методы и свойства родительского класса копируются в подкласс, который может быть изменен или дополнен.
2. **Прототипное наследование**:
    - В прототипном наследовании используется концепция прототипов, которая уникальна для **JavaScript**.

- Все объекты в JavaScript имеют ссылку на прототип, который может быть использован для наследования свойств и методов.
- Наследование происходит путем создания нового объекта с использованием существующего объекта в качестве прототипа с помощью метода `Object.create()` или изменения прототипа объекта напрямую с помощью свойства `__proto__`.
- Подклассы могут наследовать свойства и методы от своих прототипов и расширять их, добавляя свои собственные методы и свойства.

Хорошо упомянуть:

- Классы: тесные связи, иерархия.
- Прототипы: конкатенативное наследование, делегирование, функциональное наследование, композиция.

Следует избегать:

- Не указать на преимущества прототипного наследования перед классовым.

▼ Однонаправленный поток данных и двусторонняя связь данных? В чем между ними разница?

При однонаправленной связи каждый процесс, участвующий в общении, может использовать средство связи либо только для приема информации, либо только для ее передачи.

При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных.

▼ Что такое функциональное программирование?

Функциональное программирование - это парадигма программирования, которая стремится использовать функции как основной строительный блок программы и обращает особое внимание на избегание изменяемого состояния и побочных эффектов. Ключевые концепции функционального программирования:

1. **Чистые функции (Pure Functions)**: Чистые функции - это функции, которые при одних и тех же входных данных всегда возвращают одинаковый результат и не имеют побочных эффектов, то есть они не изменяют состояние программы или внешние данные.

2. **Неизменяемость (Immutability)**: Данные в функциональном программировании обычно считаются неизменяемыми, что означает, что после создания структура данных не может быть изменена. Вместо этого операции над данными создают новые структуры данных.
3. **Функции высшего порядка (Higher-Order Functions)**: Функции высшего порядка - это функции, которые могут принимать другие функции в качестве аргументов или возвращать их в качестве результатов. Они позволяют создавать более абстрактные и гибкие функции.
4. **Рекурсия (Recursion)**: Рекурсия широко используется в функциональном программировании вместо циклов для выполнения итераций.
5. **Композиция функций (Function Composition)**: Композиция функций - это процесс комбинирования нескольких функций в одну, чтобы создать новую функцию.
6. **Преобразование данных (Data Transformation)**: Функциональное программирование поддерживает мощные средства для преобразования данных с помощью функций отображения, фильтрации и свертки.
7. **Строгая типизация (Strong Typing)**: Некоторые языки функционального программирования имеют строгую типизацию, которая обеспечивает большую надежность программ и упрощает рефакторинг.

▼ Недостатки паттерна MVW?

MVW (Model View Whatever) - этим паттерном легко управлять в простом приложении содержащим несколько моделей и контроллеров, но при росте приложения можно столкнуться со следующими проблемами:

1. Модели и контроллеры взаимодействуют, эти модули меняют состояние друг друга.
2. Асинхронные сетевые запросы добавляют элемент неожиданности в то, когда модель будет изменена или модифицирована.
3. Изменение состояние модели добавляет еще один уровень сложности - мутацию, требуется определить каким образом

изменяется состояние или модель и какие инструменты необходимы для распознания мутации.

4. Код совместно используемого приложения, такого как например Гугл докс, где множество данных меняются в режиме реального времени будет просто огромным.
5. Нет возможности отменять действия, возвращаться назад во времени без добавления большого кол-ва дополнительного кода.

▼ Разница между функцией и методом?

- Функция - это независимый фрагмент кода, который выполняет определенную задачу и может принимать аргументы и возвращать результат. Функция может быть объявлена глобально или внутри другой функции, а также использоваться в качестве аргументов для других функций (функции высшего порядка) или присваиваться переменным.
- Метод - это функция, связанная с определенным объектом или классом, и выполняющая определенное действие с этим объектом. Методы вызываются на объекте, к которому они принадлежат, и имеют доступ к свойствам и методам этого объекта через ключевое слово `this`.

▼ Плюсы и минусы ФП и ООП?

Плюсы ООП

- Удобство моделирования систем. Когда каждый компонент системы представлен в виде объекта, отношения между этими объектами проще регламентировать и зафиксировать.

Минусы ООП

- Проблема конкурентных вычислений. Но это минус не только ООП, а вообще любой парадигмы, в которой есть общая память или общие данные.
- Простое наследование не всегда полностью отражает отношения компонентов.

Плюсы ФП

- Мощь функционального программирования проявляется в параллельных вычислениях.

- Кроме этого чистые функции отлично тестируются, всегда видно какие нужны аргументы.

Минусы ФП

- Потребление памяти
- Концепция чистых функций

▼ Какие принципы можно использовать вместе с наследованием?

- **Делегация** перепоручение задачи от одного объекта другому.
- **Композиция -** это принцип предполагает создание объекта из других объектов, называемых компонентами. Компоненты связаны между собой и составляют более сложный объект. Например, если у вас есть объект "Автомобиль", он может состоять из компонентов, таких как "Двигатель", "Колеса", "Кузов" и т.д.
- **Агрегация -** это отношение между объектами, когда один объект является частью другого объекта, но может существовать независимо от него. Например, мы можем дополнить обьект “Автомобиль” объектом “Елочка” (ароматизатор такой), но при этом “Елочка” это отдельный объект, который может существовать независимо от “Автомобиля”

▼ Какие ещё принципы кроме SOLID вы знаете?

- **DRY/DIE** - Don't Repeat Yourself / Duplication Is Evil.
- **KISS** Keep It Simple Stupid - призывает к тому, чтобы решения были максимально простыми. Он подразумевает, что чем проще и понятнее код, тем проще его поддерживать и изменять.
- **YAGNI** You Ain't Gonna Need It - не следует включать в программу функциональность, которая не требуется в данный момент. Он предостерегает от излишней сложности и избыточности кода.
- **APO** Avoid Premature Optimization - не следует оптимизировать код заранее, до того как станет ясно, что он действительно требует оптимизации. Ранняя оптимизация может привести к излишней сложности и утрате читаемости кода.

▼ Что такое дескрипторы свойств объектов?

Дескриптор свойства – это JavaScript-объект, описывающий атрибуты и значение свойства.

```
let user = {
  name: "John"
};

let descriptor = Object.getOwnPropertyDescriptor(user, 'nam

console.log( JSON.stringify(descriptor, null, 2 ) );
/* дескриптор свойства:
{
  "value": "John",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
*/
```

Метод `Object.getOwnPropertyDescriptor` позволяет получить *полную* информацию о свойстве.
Его синтаксис:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, prope
```

Чтобы изменить флаги, мы можем использовать метод `Object.defineProperty`.

Его синтаксис:

```
Object.defineProperty(obj, propertyName, descriptor)
```

`obj`, `propertyName` - Объект и его свойство, для которого нужно применить дескриптор.

`descriptor` - Применяемый дескриптор.

Если свойство существует, `defineProperty` обновит его флаги. В противном случае метод создаёт новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему

присваивается значение `false`.

Дескрипторы свойств, присутствующие в объектах, бывают двух основных типов:

Дескриптор данных - это свойство, имеющее значение, которое может быть записываемым.

Дескриптор доступа - это свойство, описываемое парой функций - геттером и сеттером.

▼ В чем заключаются особенности геттеров и сеттеров?

- Используются для определения вычисляемых свойств.
- Поскольку геттеры и сеттеры могут вызываться так же, как и обычные свойства объекта, они создают единый интерфейс доступа к данным. Например, если у вас есть класс `Person` с геттером и сеттером для свойства `name`, то вы можете использовать их так же, как и обычные свойства объекта, что создает однородный интерфейс доступа к данным
- Использование геттеров и сеттеров позволяет контролировать доступ к данным объекта и применять проверки и фильтры к этим данным. Например, вы можете добавить проверку в сеттер свойства `age` объекта `Person`, чтобы убедиться, что возраст всегда остается положительным числом.
- Внешний код может взаимодействовать с объектом через геттеры и сеттеры, не зная, каким образом эти операции реализованы внутри объекта.

▼ Что такое статический метод класса (`static`)? Как осуществляется его вызов?

Статические методы принадлежат только классам, а не экземплярам класса и работают только с классом, в котором были созданы.

```
class Users{
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
  static compareAge(user1, user2) {
```

```
    return user1.age === user2.age;
  }
}

const u1 = new Users("Alex", 19);
const u2 = new Users("Peter", 25);
console.log(Users.compareAge(u1, u2))
```

▼ Разница между композицией и наследованием?

Главное отличие между композицией и наследованием заключается в том, что композиция дает возможность переиспользовать код без расширения существующего класса, как это происходит в случае с наследованием. Не менее важно и то, что композиция позволяет нам выполнять повторное использование кода даже из final-класса, в то время как наследоваться от него мы не сможем.

▼ Что такое композиция в контексте JavaScript?

Композиция - создание сложной функциональности за счет объединения более простых функций. В некотором смысле, композиция - это вложение функций, каждая из которых передает свой результат в качестве входных данных для другой функции.

▼ Что такое паттерн, или шаблон проектирования?

Шаблон или паттерн проектирования — повторно используемые решения для однотипных проблем, возникающих при проектировании программного обеспечения. Паттерн представляет собой не какой-то конкретный код, а общую концепцию или пример решения той или иной проблемы.

▼ Типы паттернов? Типы GOF паттернов?

- **Порождающие -** помогают создавать объекты и структуры объектов в вашем коде. Они предоставляют решения для типичных проблем, связанных с созданием объектов, делая процесс более гибким и управляемым.
- **Структурные -** касаются композиции объектов и классов. Они помогают определить отношения между различными объектами и классами для создания более крупных структур.

- **Поведенческие** - касаются взаимодействия между объектами и ответственны за задание способов взаимодействия объектов и их распределение обязанностей.

▼ Что такое GRASP паттерны?

GRASP — набор принципов проектирования программного обеспечения, который помогает разработчикам определять, как распределить ответственности между классами и объектами в системе.

Некоторые из основных принципов GRASP включают:

1. **Высокая связанность (High Cohesion)**: Классы должны иметь высокую степень связанности с заданными задачами, что означает, что функции и данные класса должны быть тесно связаны и взаимосвязаны.
2. **Низкая связанность (Low Coupling)**: Классы должны иметь низкую степень связанности между собой, что означает, что изменения в одном классе должны иметь минимальное воздействие на другие классы.
3. **Ответственность (Responsibility Assignment)**: Классы должны быть ответственны за выполнение своих задач. Принципы GRASP помогают определить, какие классы должны быть ответственны за какие задачи.

▼ Типы полиморфизма?

Полиморфизм - это принцип ООП, который позволяет объектам использовать методы с одинаковыми именами, но с различной реализацией.

В общем, полиморфизм включает в себя три основных типа:

Перегрузка методов (Method Overloading):

- В JavaScript нет явной поддержки перегрузки методов, как в некоторых других языках, но вы можете эмулировать это поведение, проверяя количество переданных аргументов или их типы.

```
class Calculator {
    add(x, y) {
```

```
        if (typeof x === 'number' && typeof y === 'number')
            return x + y;
        } else if (typeof x === 'string' && typeof y === 's
            return x.concat(y);
        }
    }
}

const calc = new Calculator();
console.log(calc.add(2, 3));        // Output: 5
console.log(calc.add('hello', 'world')); // Output: hellowo
```

Переопределение методов (Method Overriding):

- Этот тип полиморфизма широко используется в JavaScript при работе с наследованием.

```
class Animal {
    makeSound() {
        console.log("Some generic sound");
    }
}

class Dog extends Animal {
    makeSound() {
        console.log("Woof");
    }
}

const dog = new Dog();
dog.makeSound(); // Output: Woof
```

Параметрический полиморфизм (Parametric Polymorphism:

- В JavaScript параметрический полиморфизм может быть реализован с использованием обобщенных функций или функций высшего порядка.

```
function add(x, y) {
    return x + y;
}

console.log(add(2, 3));        // Output: 5
console.log(add("hello", "world")); // Output: helloworld
```

▼ Можно ли в JavaScript реализовать абстрактный класс и как это сделать?

В JavaScript нет встроенной поддержки абстрактных классов, как в некоторых других объектно-ориентированных языках программирования, таких как Java или C#.

▼ Основные принципы функционального программирования?

1. **Чистые функции (Pure Functions)**:
    - Функции должны возвращать одинаковый результат для одинаковых входных данных и не иметь побочных эффектов, таких как изменение глобальных переменных или ввод-вывод. Чистые функции обеспечивают предсказуемость и упрощают отладку и тестирование кода.
2. **Неизменяемость данных (Immutable Data)**:
    - Данные должны быть неизменяемыми, то есть после создания их нельзя изменять. Вместо этого создаются новые данные при необходимости изменений. Это помогает избежать неожиданных побочных эффектов и упрощает управление состоянием приложения.
3. **Функции высшего порядка (Higher-Order Functions)**:
    - Функции могут принимать другие функции в качестве аргументов или возвращать функции как результат. Это позволяет абстрагировать общие операции и создавать более гибкий и переиспользуемый код.
4. **Рекурсия (Recursion)**:
    - Рекурсия является важным инструментом в функциональном программировании, позволяющим решать задачи путем вызова

функций из самих себя. Рекурсивные алгоритмы могут быть более лаконичными и элегантными, чем их итеративные аналоги.

5. **Функциональные компоненты (Functional Composition)**:
    - Функциональное программирование обычно строится на композиции функций, где результат одной функции передается в качестве аргумента для другой функции. Это позволяет создавать более сложные операции из простых функциональных блоков.
6. **Ленивые вычисления (Lazy Evaluation)**:
    - В функциональном программировании используется подход ленивого вычисления, при котором выражения вычисляются только при необходимости. Это позволяет оптимизировать производительность и эффективность программы.
7. **Работа с данными как с потоками (Data Flow)**:
    - В функциональном программировании данные рассматриваются как поток информации, который проходит через функциональные преобразования. Это облегчает обработку данных и упрощает понимание программы.

▼ Плюсы функционального программирования?

1. **Чистота и предсказуемость**: Благодаря чистым функциям и неизменяемым данным программы становятся более предсказуемыми и проще в понимании.
2. **Устойчивость к ошибкам**: Функциональное программирование сокращает количество побочных эффектов, что уменьшает возможность ошибок и облегчает отладку кода.
3. **Масштабируемость и параллелизм**: Функциональные программы обладают хорошей масштабируемостью и могут быть легко адаптированы для параллельного выполнения, что улучшает производительность приложений.
4. **Краткость и выразительность кода**: Функциональное программирование позволяет писать более короткий и выразительный код благодаря использованию функций высшего порядка и композиции функций.

5. **Легкость тестирования**: Чистые функции облегчают тестирование программы, поскольку они не зависят от состояния окружения и дают одинаковый результат при одинаковых входных данных.

▼ Разница между императивным и декларативным подходами программирования?

Императивное программирование:

- В императивном программировании программа описывает последовательность шагов, которые компьютер должен выполнить для достижения конкретной цели. Основное внимание уделяется тому, *как* должны быть выполнены операции. Здесь программист указывает, какие действия должны быть выполнены и в каком порядке.
- Примеры императивных языков программирования включают C, C++, Java, Python (в некоторых случаях), и т. д.
- Пример императивного кода на JavaScript:

```
let sum = 0;
for (let i = 1; i <= 10; i++) {
    sum += i;
}
console.log(sum); // Output: 55
```

Декларативное программирование:

- В декларативном программировании программа описывает, *что* должно быть сделано, но не указывает, *как* это должно быть сделано. Вместо того, чтобы описывать шаги для достижения результата, программист описывает желаемый результат или свойства этого результата.
- Декларативный код более абстрактный и выражает цели и требования, а не конкретные действия.
- Примеры декларативных подходов включают SQL, HTML, CSS, и функциональное программирование в JavaScript (например, использование функций высшего порядка, например `map`, `filter`, `reduce`).

- Пример декларативного кода на JavaScript (используя функцию `reduce` для нахождения суммы чисел от 1 до 10):

```
const sum = Array.from({ length: 10 }, (_, i) => i + 1).
reduce((acc, curr) => acc + curr, 0);
console.log(sum); // Output: 55
```

▼ Что такое реактивное программирование?
▼ Плюсы и минусы реактивного программирования?
▼ Что такое Inversion of control?
▼ Что такое Dependency injection?
▼ Разница между агрегацией и композицией?
▼ Разница между процедурным и функциональным программированием?
▼ JS PATTERN(FABRIC ETC)

Паттерн проектирования — это часто встречаемое решение определенной проблемы при проектировании архитектуры программ. В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию или пример решения той или иной проблемы, которое нужно будет подстроить под нужды вашей программы

Декоратор - оборачивать объект в класс декоратора тем самым добавляя новые свойства и методы.

Есть некий базовый класс. Декоратор принимает в себя базовый класс и добавляет (изменяет) новые методы.

Потом мы инстансу класса присваиваем декоратор куда передаем инстанс.

Помогает избежать создание множества подклассов.

```
// Version with Autopilot & Parktronic
let tesla = new Tesla();
tesla = new Autopilot(tesla);
tesla = new Parktronic(tesla);

console.log(tesla.getPrice(), tesla.getDescription());
// 33000 "Tesla with autopilot with parktronic"
```

Singleton - объект, который есть в системе в одном экземпляре. К нему есть глобальная точка доступа.

Наблюдатель - subscribe, subscribe, notify

Fabric - создание класса, который создает объекты на основании входных данных.

Использовать когда создавать надо множество объектов с одинаковой структурой, но разными данными.

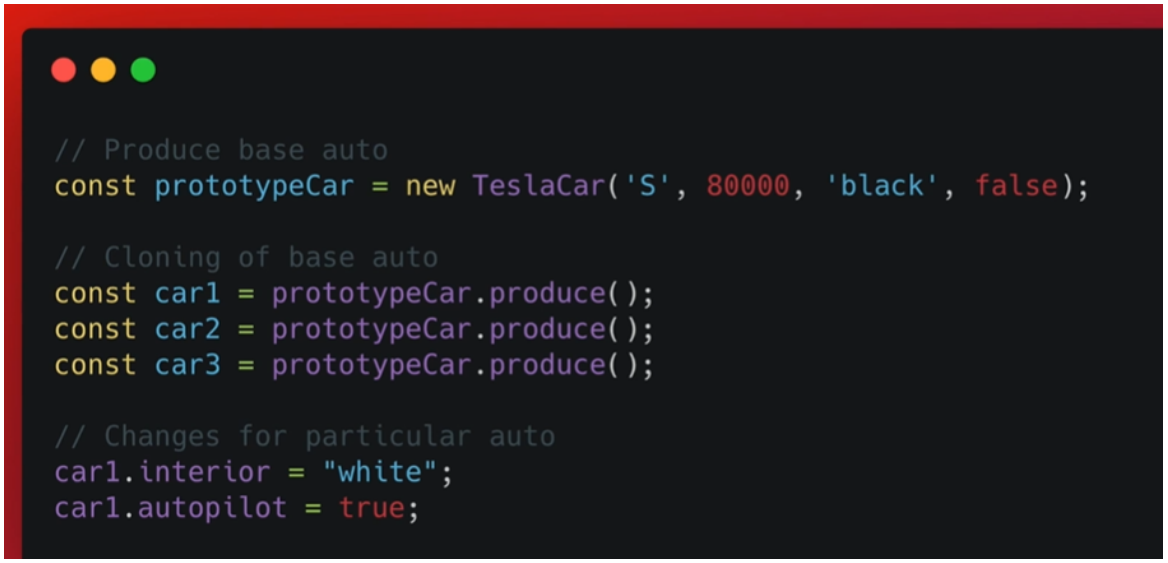
Состоит из класса конструктора, который генерирует объект и класса который вызывает это создание с определенными параметрами.

Создаем экземпляр фабрики, вызываем метод create и передаем нужные данные.

Прототип

Клонируем авто с помощью метода produce который возвращает копию объект класса Car.

Меняем конфигурацию в одном объекте точечно при необходимости.



```
// Produce base auto
const prototypeCar = new TeslaCar('S', 80000, 'black', false);

// Cloning of base auto
const car1 = prototypeCar.produce();
const car2 = prototypeCar.produce();
const car3 = prototypeCar.produce();

// Changes for particular auto
car1.interior = "white";
car1.autopilot = true;
```

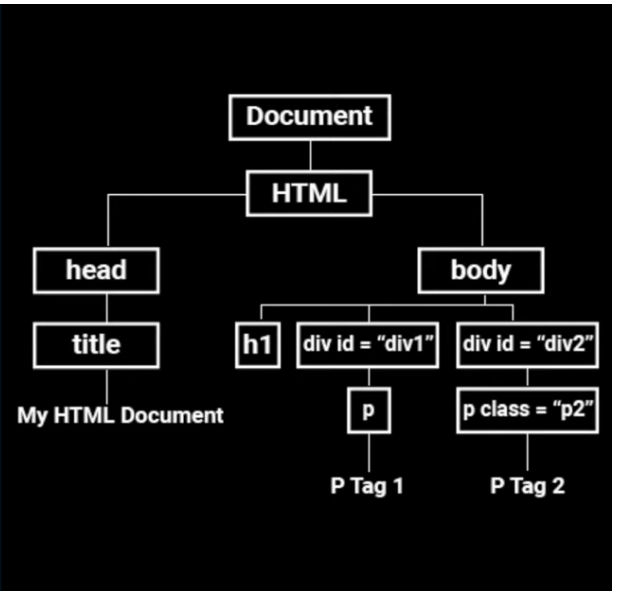
JS in Browser

▼ Что такое DOM?

DOM (Document Object Model) - это объектная модель документа, которую браузер создает в памяти компьютера на основании HTML кода полученного им от сервера. С помощью DOM, JavaScript может взаимодействовать с элементами страницы.

Интрефейс с помощью которого JS взаимодействует с элементами страницы.

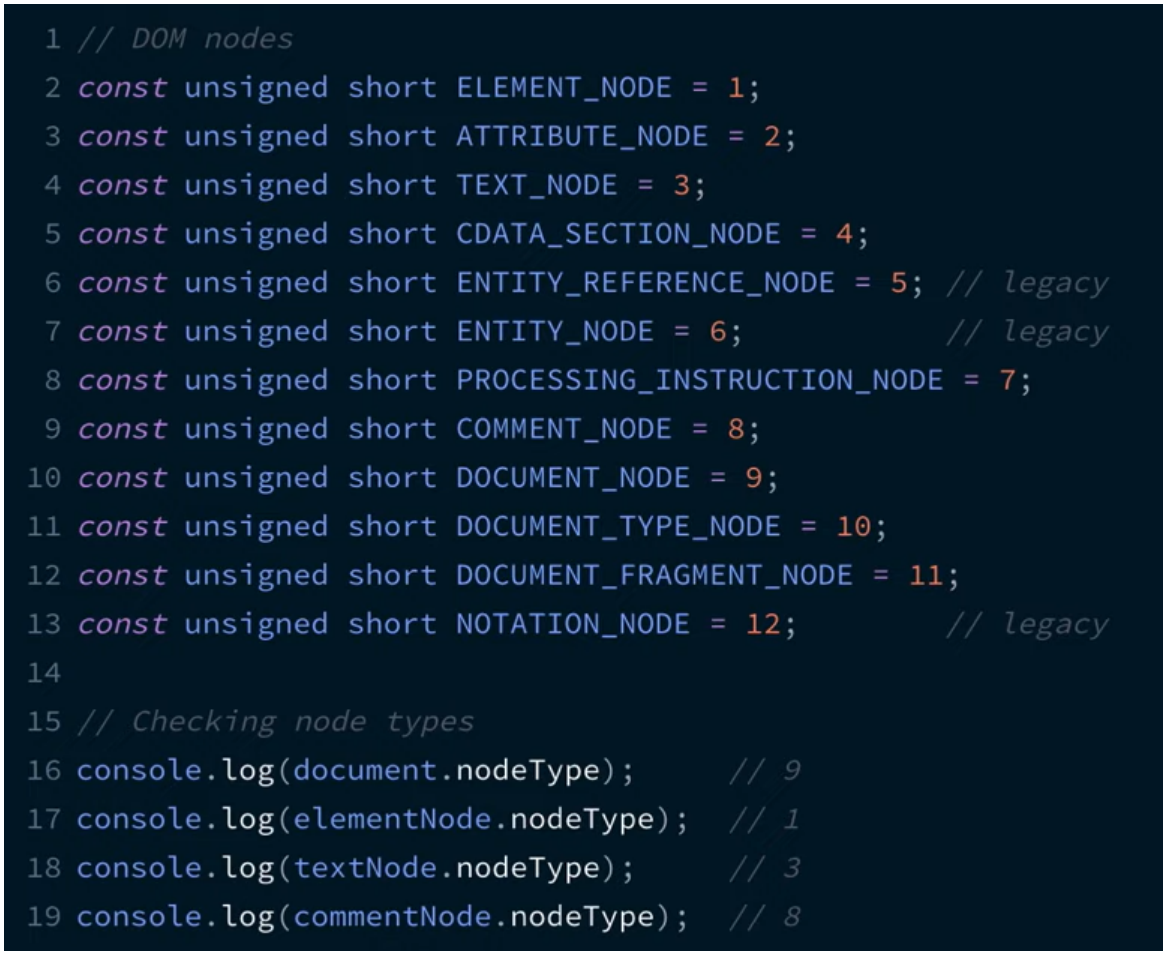
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My HTML Document</title>
</head>
<body>
  <h1>Heading</h1>
  <div id="div1">
    <p>P Tag 1</p>
  </div>
  <div id="div2">
    <p class="p2">P Tag 2</p>
  </div>
</body>
</html>
```



▼ Типы узлов DOM-дерева?

Популярные 4:

- document.nodeType
- element.nodeType
- textNode.nodeType
- commentNode.nodeType



```
// DOM nodes
const unsigned short ELEMENT_NODE = 1;
const unsigned short ATTRIBUTE_NODE = 2;
const unsigned short TEXT_NODE = 3;
const unsigned short CDATA_SECTION_NODE = 4;
const unsigned short ENTITY_REFERENCE_NODE = 5; // legacy
const unsigned short ENTITY_NODE = 6;           // legacy
const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
const unsigned short COMMENT_NODE = 8;
const unsigned short DOCUMENT_NODE = 9;
const unsigned short DOCUMENT_TYPE_NODE = 10;
const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
const unsigned short NOTATION_NODE = 12;        // legacy

// Checking node types
console.log(document.nodeType);     // 9
console.log(elementNode.nodeType);  // 1
console.log(textNode.nodeType);     // 3
console.log(commentNode.nodeType);  // 8
```

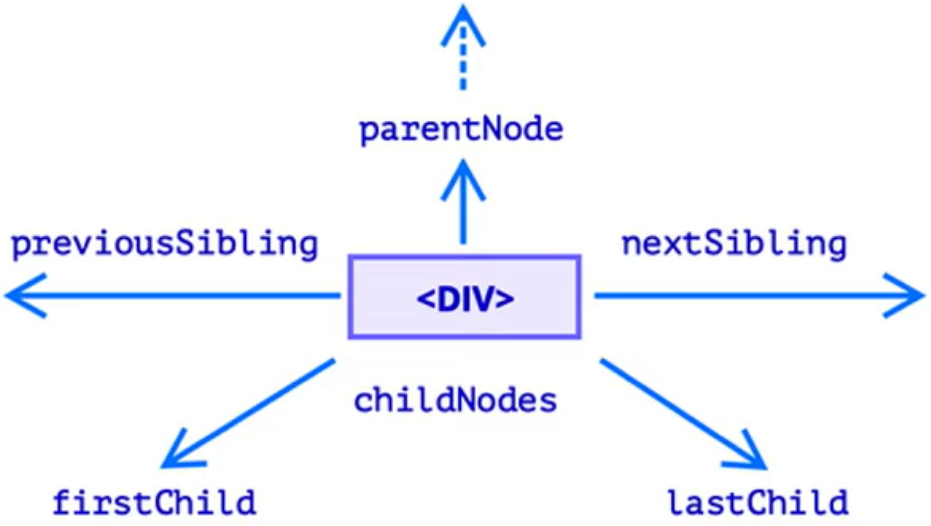
▼ Методы поиска элементов в DOM?

- getElementById()
- getElementsByName()
- getElementsByTagName()
- getElementsByClassName()
- querySelector()
- querySelectorAll()

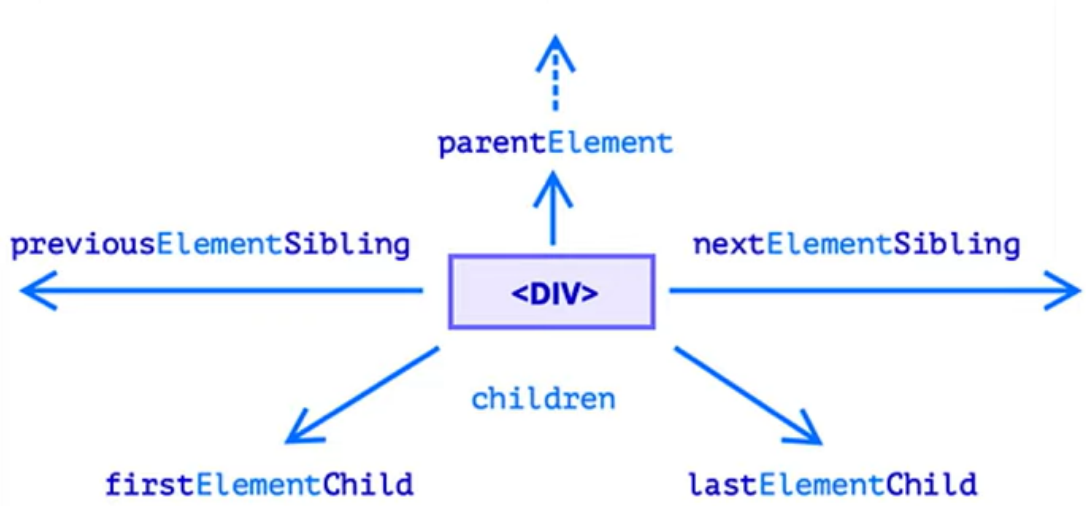
```
// Searching elements methods
document.getElementById('id');
document.getElementsByName('name');
document.getElementsByTagName('div');
document.getElementsByClassName('class');
document.querySelector('nav + p');
document.querySelectorAll('nav > ul > li');
```

▼ Свойства для перемещения по DOM-дереву?

1. **parentNode** возвращает родительский элемент текущего элемента.
2. **childNodes** возвращает коллекцию дочерних элементов текущего элемента.
3. **firstChild** возвращает первый дочерний элемент текущего элемента.
4. **lastChild** возвращает последний дочерний элемент текущего элемента.
5. **nextSibling** возвращает следующий соседний элемент текущего элемента.
6. **previousSibling** возвращает предыдущий соседний элемент текущего элемента.



7. **parentElement** возвращает родительский элемент текущего элемента (только для элементов, не для узлов текста).
8. **children** возвращает коллекцию дочерних элементов текущего элемента (только для элементов, не для узлов текста).
9. **firstElementChild** возвращает первый дочерний элемент текущего элемента (только для элементов, не для узлов текста).
10. **lastElementChild** возвращает последний дочерний элемент текущего элемента (только для элементов, не для узлов текста).
11. **nextElementSibling** возвращает следующий соседний элемент текущего элемента (только для элементов, не для узлов текста).
12. **previousElementSibling** возвращает предыдущий соседний элемент текущего элемента (только для элементов, не для узлов текста).



▼ Разница между attribute и property у DOM-элементов?

attribute - статичное значение определенного DOM элемента, которое неизменяемо и в большинстве своем может быть добавлено в HTML разметке.

property - вычисляемое значение, может быть изменено. Пример: value у input

- Атрибуты – это то, что написано в HTML.
- Свойства – это то, что находится в DOM-объектах.

Небольшое сравнение:

	Свойства	Атрибуты

Тип	Любое значение, стандартные свойства имеют типы, описанные в спецификации	Строка
Имя	Имя регистрозависимо	Имя регистронезависимо

Методы для работы с атрибутами:

- `elem.hasAttribute(name)` – проверить на наличие.
- `elem.getAttribute(name)` – получить значение.
- `elem.setAttribute(name, value)` – установить значение.
- `elem.removeAttribute(name)` – удалить атрибут.
- `elem.attributes` – это коллекция всех атрибутов.

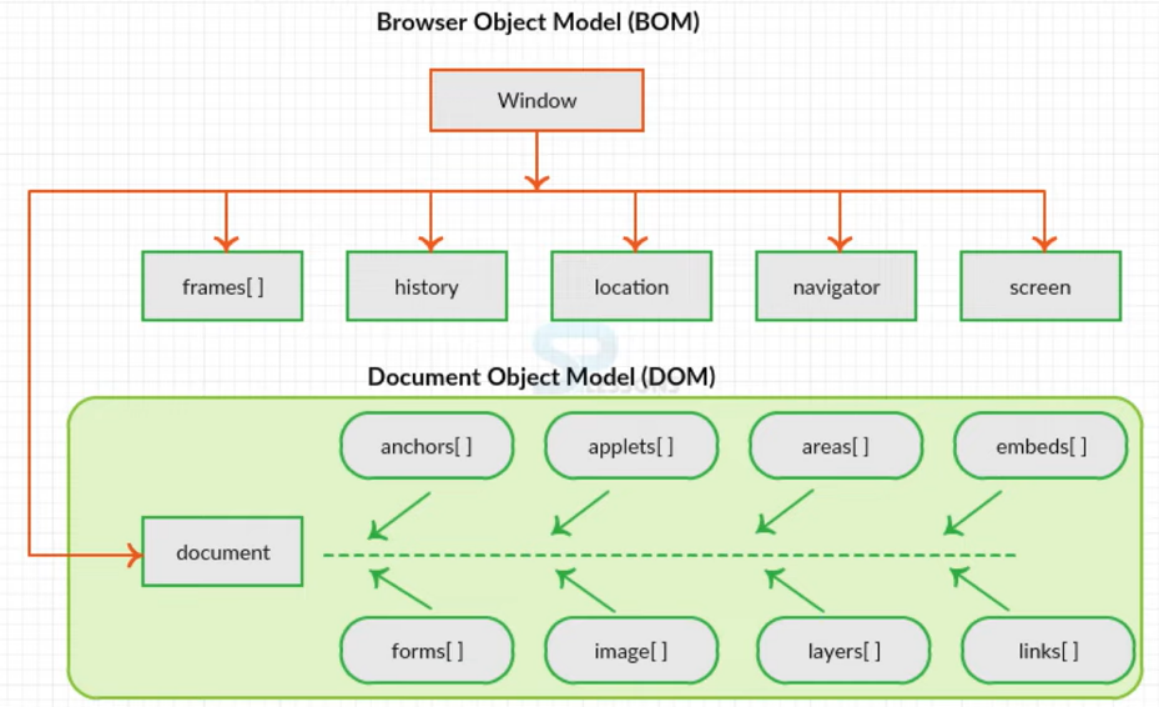
В большинстве ситуаций предпочтительнее использовать DOM-свойства. Нужно использовать атрибуты только тогда, когда DOM-свойства не подходят, когда нужны именно атрибуты, например:

- Нужен нестандартный атрибут. Но если он начинается с `data-`, тогда нужно использовать `dataset`.
- Мы хотим получить именно то значение, которое написано в HTML. Значение DOM-свойства может быть другим, например, свойство `href` – всегда полный URL, а нам может понадобиться получить «оригинальное» значение.

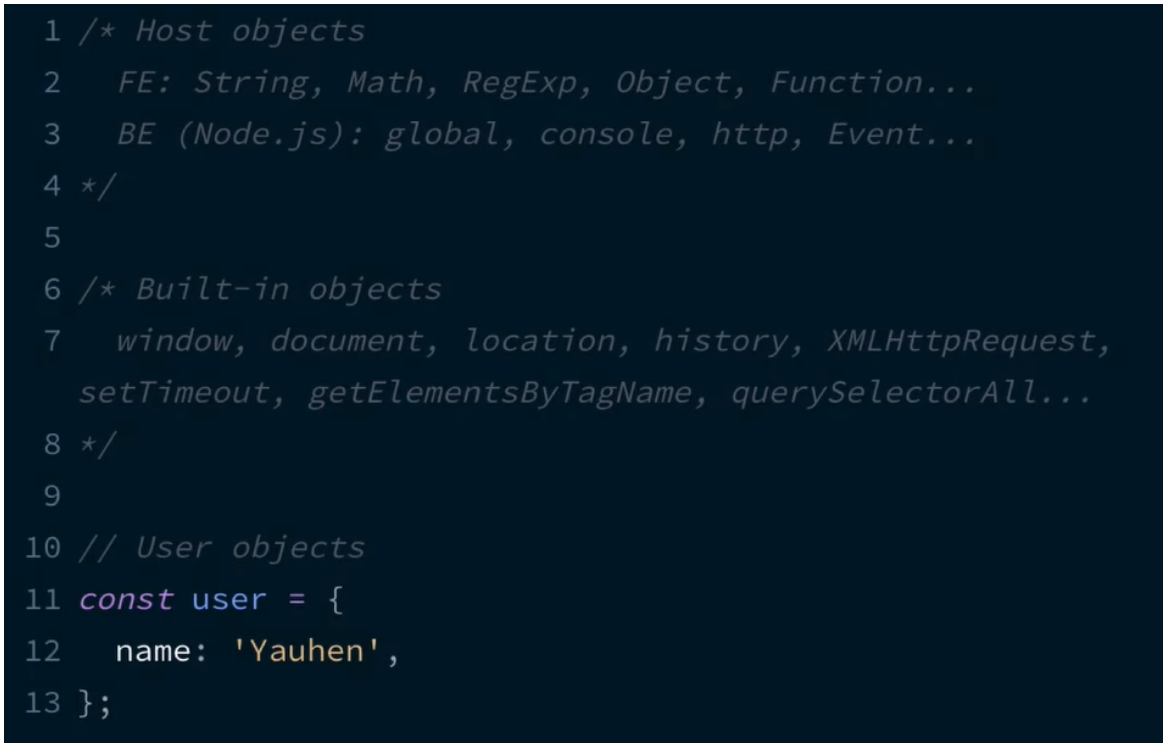
```html
<!-- After page is rendered -->
<input type="text" value="Hello world">
<script>
  input.getAttribute(value); // "Hello world"
  input.value;               // "Hello world"
</script>

<!-- After user intered any value into input field -->
<input type="text" value="New Text">
<script>
  input.getAttribute(value); // "Hello world"
  input.value;               // "New Text"
</script>
```

▼ Что такое BOM?

Не стандартизировна и реализация может отличаться отбразера к браузеру.



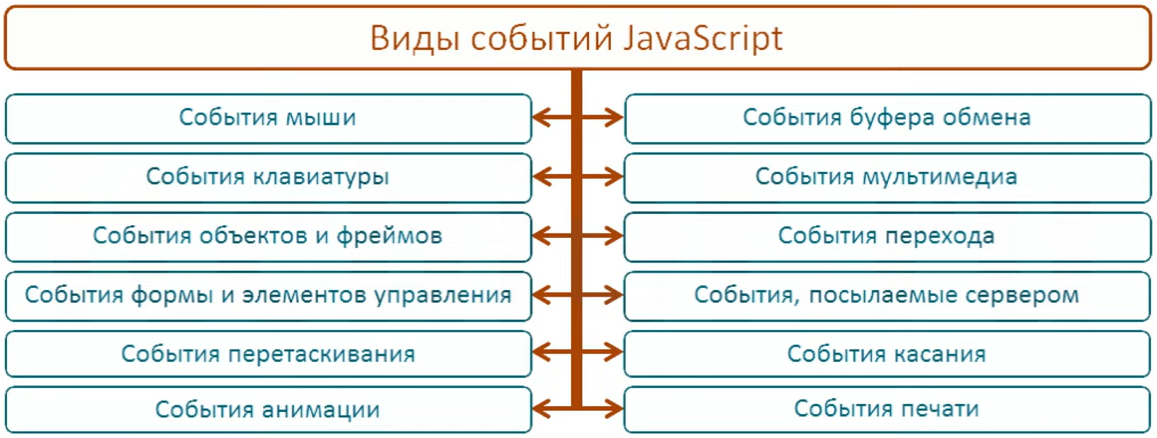
Содержит API для работы JS с браузером. Все браузерные методы типа fetch, xmlhttprequest, localStorage и прочие built-in objects.



```
/* Host objects
   FE: String, Math, RegExp, Object, Function...
   BE (Node.js): global, console, http, Event...
*/

/* Built-in objects
   window, document, location, history, XMLHttpRequest,
  setTimeout, getElementsByTagName, querySelectorAll...
*/

// User objects
const user = {
   name: 'Yauhen',
};
```

▼ Виды событий в JavaScript?

1. События мыши: click, dblclick, mouseover, mouseout, mousedown, mouseup, mousemove.
2. События клавиатуры: keydown, keyup, keypress.
3. События формы: submit, reset, focus, blur, change.
4. События документа: load, unload, ready, scroll, resize.
5. События таймера: setTimeout, setInterval.
6. События перетаскивания: dragstart, dragend, dragenter, dragleave, dragover, drop.
7. События анимации: animationstart, animationend, animationiteration.
8. События сети: online, offline.
9. События истории браузера: popstate.
10. События мультимедиа: play, pause, ended.
11. События геолокации: geolocation.
12. События сенсоров: deviceorientation, devicemotion.



▼ Как добавить обработчик события на DOM-элемент?

```
1  const testFunc = () => console.log('Hello world!');
2
3  // Inline event handler
4  <button onclick="testFunc()">Press me</button>
5
6  // Event handler properties
7  const btn = document.querySelector('button');
8  btn.onclick = testFunc();
9
10 // Function 'addEventListener'
11 const btn = document.querySelector('button');
12 btn.addEventListener('click', testFunc);
```

```
// не нужно искать, но может быть только один обработчик
<button onclick='func()'></button>

// нужно искать элемент, и только один обработчик
const btn = document.querySelector('button')
btn.onclick = func()

// нужно искать элемент, но можно повесить много обработчик
const btn = document.querySelector('button')
btn.addEventListener('click', () => {})
```

▼ Как удалить обработчик события с DOM-элемента?

removeEventListener, очистка поля onClick и подобных, с помощью JS очистить обработчик заданный в HTML.

```
const testFunc = () => console.log('Hello world!');

const btn = document.querySelector('button');

btn.addEventListener('click', testFunc);    // Add event handler
btn.removeEventListener('click', testFunc); // Remove event handler
```

▼ Что такое распространение события (Event Propagation)?

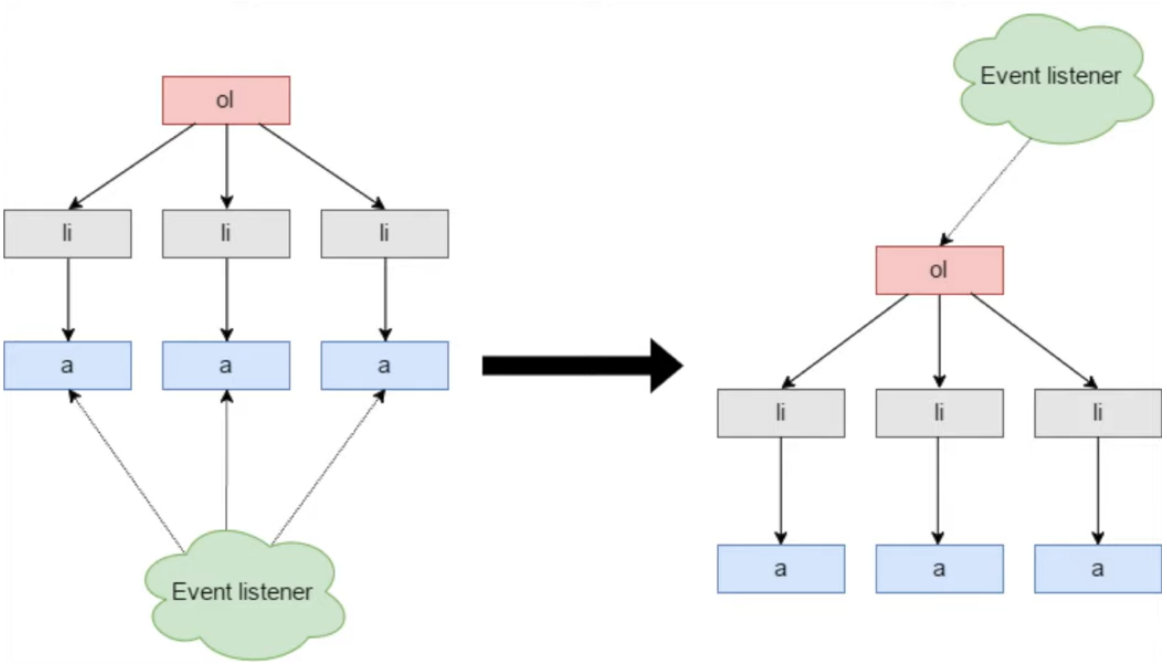
Подробнее тут

Event Propagation - это механизм, который срабатывает, когда какой-либо событие происходит в документе. Событие распространяется от объекта window до вызывающего его элемента, при этом событие последовательно затрагивает всех предков целевого элемента. Все это делится на 3 фазы:

1. **Capturing** фаза захвата, событие начинается от корня документа и проходит по дереву до целевого элемента.
2. **Target** событие достигает целевого элемента.
3. **Bubbling** фаза всплытия, событие возвращается обратно до корня, параллельно вызывая все события на родительских элементах.

▼ Что такое делегирование событий (Event Delegation)?

Event Delegation - это прием разработки, когда вместо того чтобы вешать кучу однотипных обработчиков на все элементы, можно добавить только один на общего предка.



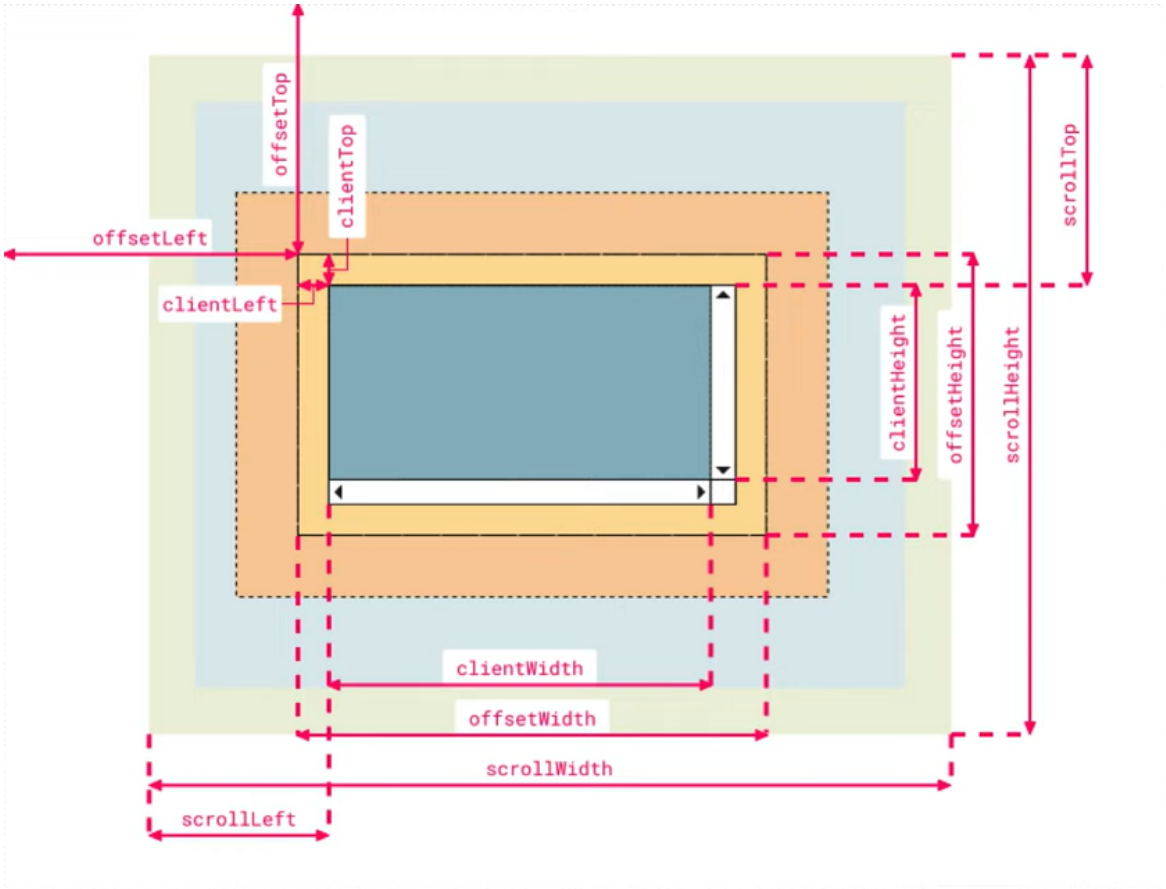
▼ Как использовать media выражения в JavaScript?

Media выражения можно использовать в JavaScript с помощью объекта window.matchMedia(). Этот метод возвращает объект MediaQueryList, который содержит информацию о том, соответствует ли текущее состояние окна браузера заданному медиа-выражению.

```
1 const mediaQuery = window.matchMedia('(min-width: 768px)');
2
3 console.log(mediaQuery);
4 // {media: '(min-width: 768px)', matches: false, onchange: null}
5
6 if (mediaQuery.matches) {
7   console.log('Media Query Matched!');
8 }
```

▼ Расскажите про координаты в браузере?

Подробнее тут



▼ Разница между HTMLCollection и NodeList?

HTMLCollection - это динамическая коллекция, которая представляет собой массивоподобный итерируемый объект дочерних элементов.

NodeList - статический список нод или узлов, в который входят все найденные в документе элементы

getElementsByClassName - HTMLCollection

querySelectorAll - NodeList

Различие между методами в том, что если в разметку динамически добавить элемент, то обновленные данные будут присутствовать только в HTMLCollection.

```html
<ul id="list">
  <li class="item">First Item</li>
  <li class="item">Second Item</li>
</ul>
<script>
  document.getElementsByClassName('item'); // HTMLCollection(2)
  document.querySelectorAll('.item');      // NodeList(2)
</script>

<!-- Dynamicaly added one LI-element -->
<ul id="list">
  <li class="item">First Item</li>
  <li class="item">Second Item</li>
  <li class="item">Third Item</li>
</ul>
<script>
  document.getElementsByClassName('item'); // HTMLCollection(3)
  document.querySelectorAll('.item');      // NodeList(2)
</script>
```

▼ Как динамически добавить элемент на HTML-страницу?

```js
// Create element
const newP = document.createElement("p");
// Add class to element
newP.classList.add("pStyle");
// Create text-node
const textNode = document.createTextNode("Hello world");
// Add text-node inside created element
newP.appendChild(textNode);
// Add created element inside DOM
document.getElementById("test").appendChild(newP);
```

```js
const p = document.createElement('p')
p.classList.add('pStyle')
const textNode = document.createTextNode('Hello World')
p.appendChild(textNode)
document.body.appendChild(p)
```

▼ Разница между feature detection, feature inference и анализом строки user-agent?

feature detection - поддерживает ли браузер какие либо методы, и если нет то будет выполнен аналог или полифил. Рекомендуется использовать именно этот подход. Так работает например библиотека modernizr.

```
// Feature detection
if ('geolocation' in navigator) {
  // use navigator.geolocation
} else {
  // another code
}

// Feature inference
if (document.getElementsByTagName) {
  element = document.getElementById(id);
}

// User Agent
console.log(navigator.userAgent);
// "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.106
  Safari/537.36"
```

▼ Разница между `e.preventDefault()` и `e.stopPropagation()`?

Метод preventDefault() - отключает поведение элемента по умолчанию.

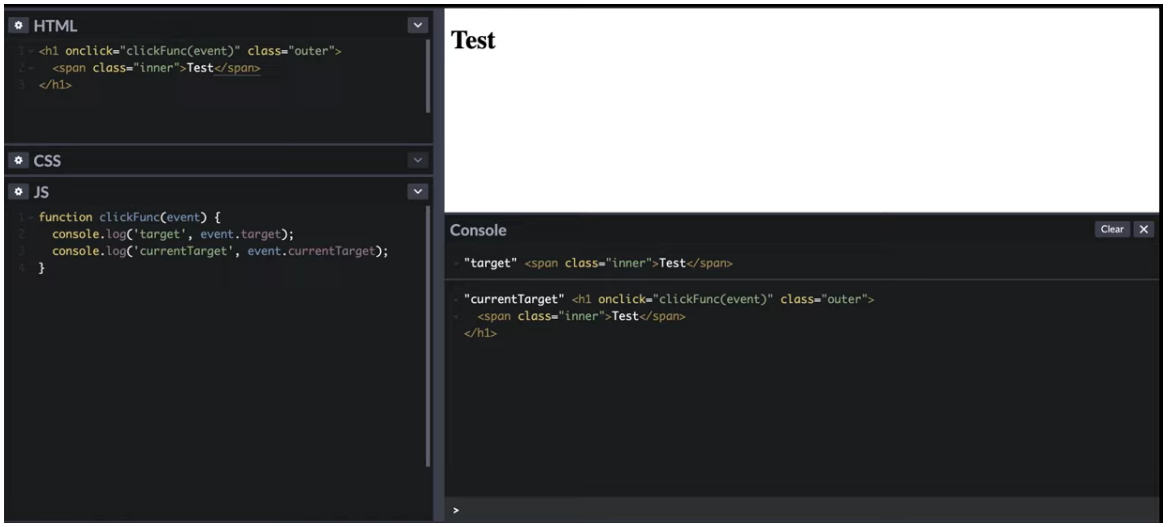
Метод stopPropagation() - отключает распространение события всплытия/погружения.

```
// Stop link event
const link = document.getElementById('link')
link.addEventListener('mousedown', event => {
  event.preventDefault();
});

// Stop form sending
const form = document.getElementById('form')
link.addEventListener('submit', event => {
  event.preventDefault();
});

// Stop propagation
const link = document.getElementById('link')
link.addEventListener('mousedown', event => {
  event.stopPropagation();
});
```

▼ **Разница между `event.target` и `event.currentTarget` ?**

target - элемент в котором происходит событие или элемент вызвавший событие.

currentTarget - элемент к которому прикреплен прослушиватель события.



▼ Разница между `.stopPropagation()` и `.stopImmediatePropagation()`?

.stopPropagation() - остановит всплытие, но все прикрепленные к элементу обработчики будут выполнены.

.stopImmediatePropagation() - остановит всплытие + остановит обработку оставшихся событий на текущем элементе.

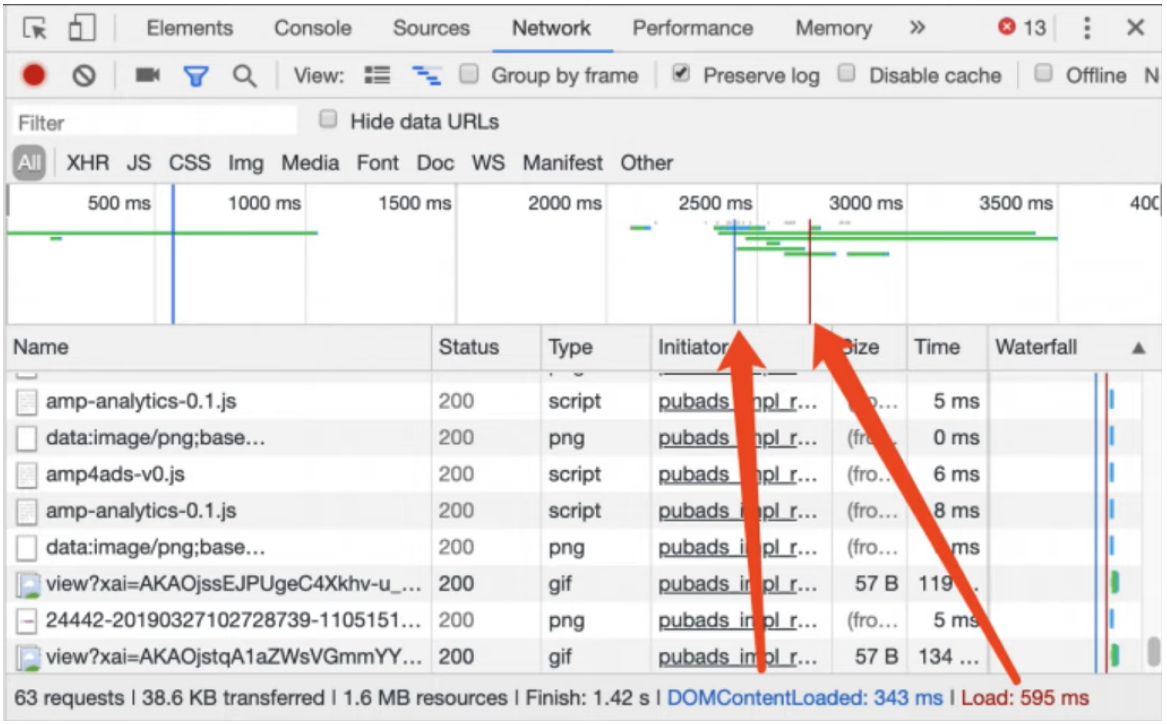
▼ Разница между событиями `load` и `DOMContentLoaded`?

DOMContentLoaded - полностью загружен HTML и построено DOM дерево, но внешние ресурсы еще не загружены.

load - загружены все внешние ресурсы.

beforeunload - запускается, когда окно, документ и его ресурсы вот-вот будут выгружены. Документ все еще виден, и событие в этот момент может быть отменено.

unload возникает когда window выгружает свое содержимое и ресурсы. Удаление ресурсов происходит после возникновения события unload.



▼ Сколько аргументов принимает `addEventListener`?

3 аргумента:

- название события
- колбэк
- options / useCapture

Для Options доступны следующие варианты:

- capture: Boolean перехват на стадии погружения
- once: Boolean указывает, что обработчик должен быть вызван не более одного раза после добавления. Если true, обработчик автоматически удаляется при вызове.
- passive: Boolean указывает, что обработчик никогда не вызовет preventDefault(). Если всё же вызов будет произведён, браузер должен игнорировать его и генерировать консольное

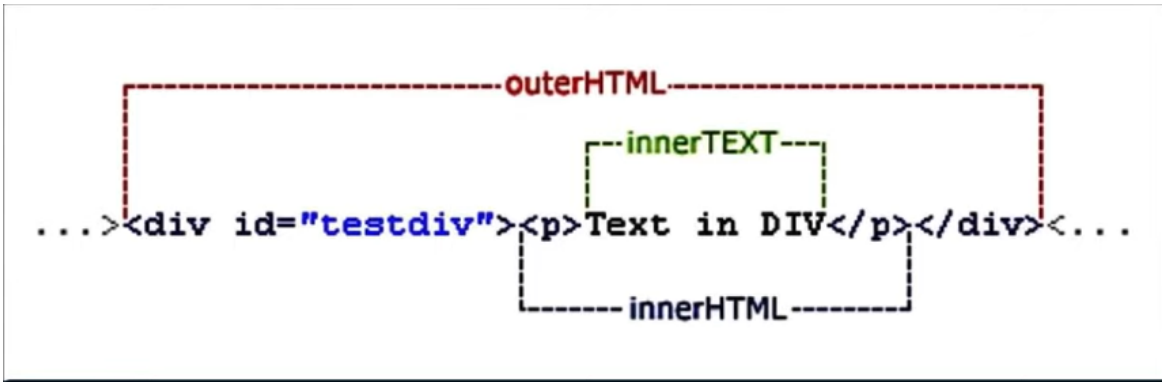
предупреждение. Пример Улучшение производительности прокрутки с помощью passive true

```
const testFunc = () => console.log('Hello world!');

const btn = document.querySelector('button');

// Default
btn.addEventListener('click', testFunc);
// once
btn.addEventListener('click', testFunc, { once: true });
// capture
btn.addEventListener('click', testFunc, { capture: true });
btn.addEventListener('click', testFunc, true);
// passive
btn.addEventListener('click', testFunc, { passive: true });

// Combination
btn.addEventListener('click', testFunc, {
  once: true,
  capture: true,
  passive: true,
});
```

▼ Разница между `innerHTML` и `outerHTML`?

Свойство innerHTML - содержит код внутри найденного элемента.

Свойство outerHTML - возвращает полный код найденного элемента, а не его часть.




```
// DOM
<div class='outer'>
  <p class='inner'></p>
</div>

// innerHTML:
<p class='inner'></p>

// outerHTML:
<div class='outer'><p class='inner'></p></div>
```

▼ Разница между `JSON` и `XML` ?

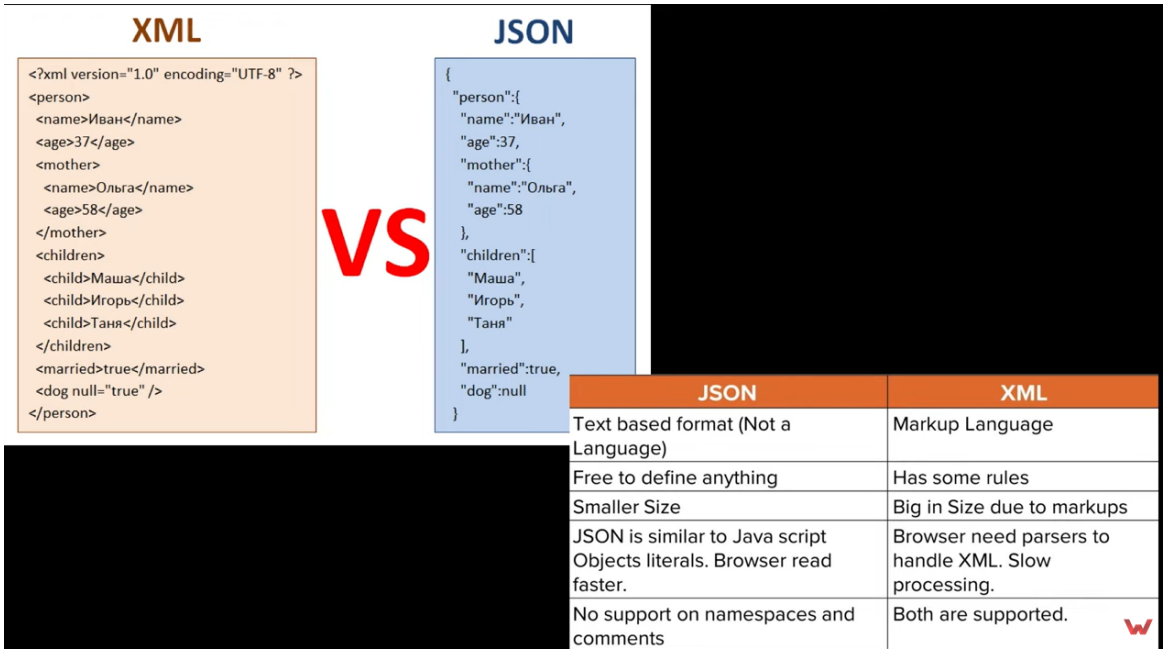
JSON - формат обмена данными.

XML - язык разметки, в котором можно задавать синтаксис, структуру, типы данных и их модель.

JSON - позволяет определять данные любого формата, в то время как в XML есть свои правила и ограничения.

JSON более компактный, т.к. представляет из себя формат ключ: значение

XML объемный за счет того, что данные оборачиваются в разметку.



▼ Как узнать об использовании метода `event.preventDefault()`?

```
const anchor = document.getElementById('myAnchor');

anchor.addEventListener('click', (event) => {
  event.defaultPrevented; // false
  event.preventDefault();
  event.defaultPrevented; // true
});
```

```
event.defaultPrevented
```

▼ Для чего используется свойство `window.navigator`?

Свойство window.navigator используется для получения информации о браузере и операционной системе, на которых запущен текущий скрипт. Можно узнать язык, браузер, включены ли куки, ОС.

```
navigator.userAgent
 // "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0
 Safari/537.36"

navigator.language
 // "ru"

 navigator.languages
 // ["ru", "en-US", "es-ES"]

navigator.cookieEnabled
// true
```

▼ Для чего используется метод `.focus()` ?

Вызов метода .focus() на дом элементе, устанавливает фокус на этот элемент, перехватывая события клавиатуры. Нельзя установить на заблокированный элемент. Прокручивает страницу до элемента с фокусом по умолчанию, но можно отменить передав в метод объект со свойством preventScroll: true

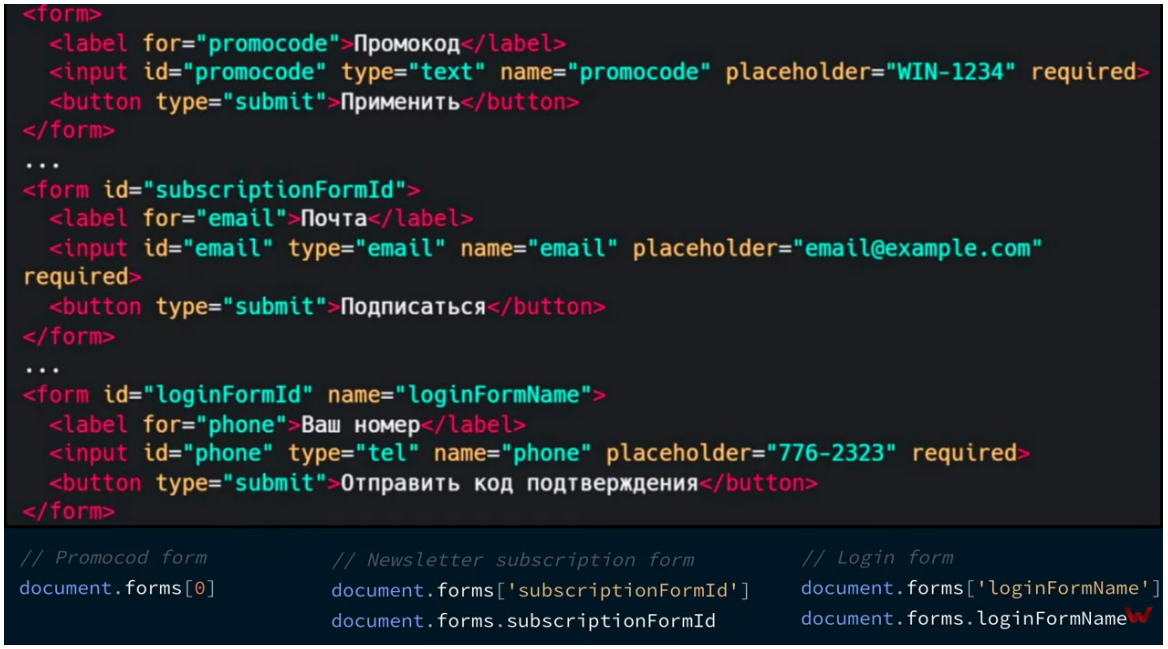
```
<input value="text" id="input">
<input type="button" value="focus" id="focus">
```

```
let input = document.querySelector('#input');
let focus = document.querySelector('#focus');

// По клику на кнопку focus установим фокус инпуту:
focus.addEventListener('click', function() {
  input.focus();
});
```

Результат выполнения кода:

▼ Для чего используется свойство `.forms` ?

.forms - это поле или свойство объекта document, оно хранит коллекцию всех элементов форм, которые есть на текущей открытой странице, данная коллекция доступна только для чтения. Получаем живую коллекцию.



```
<form>
  <label for="promocode">Промокод</label>
  <input id="promocode" type="text" name="promocode" placeholder="WIN-1234" required>
  <button type="submit">Применить</button>
</form>
...
<form id="subscriptionFormId">
  <label for="email">Почта</label>
  <input id="email" type="email" name="email" placeholder="email@example.com"
required>
  <button type="submit">Подписаться</button>
</form>
...
<form id="loginFormId" name="loginFormName">
  <label for="phone">Ваш номер</label>
  <input id="phone" type="tel" name="phone" placeholder="776-2323" required>
  <button type="submit">Отправить код подтверждения</button>
</form>
```

```
// Promocod form
document.forms[0]

// Newsletter subscription form
document.forms['subscriptionFormId']
document.forms.subscriptionFormId

// Login form
document.forms['loginFormName']
document.forms.loginFormName
```

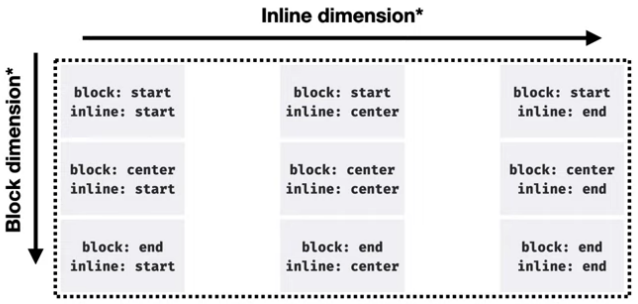
▼ Для чего используется метод `.scrollIntoView()`?

Метод .scrollIntoView() - используется для прокрутки страницы так, чтобы определенный элемент стал видимым в области просмотра браузера. Есть аргумент булевого типа, true - верхняя граница элемента совпадает с верхней границей окна. false - нижняя граница элемента совпадает с нижней границей окна. Также можно указать объект настроек который определит механизм прокрутки, behavior - задает анимацию прорутки, auto или smooth(плавная). Вертикальное и горизонтальное выравнивание через block и inline - start, center, end, nearest.

element.scrollIntoView

Scroll elements into the visible area

```
element.scrollIntoView({
  block: '...',
  inline: '...'
});
```



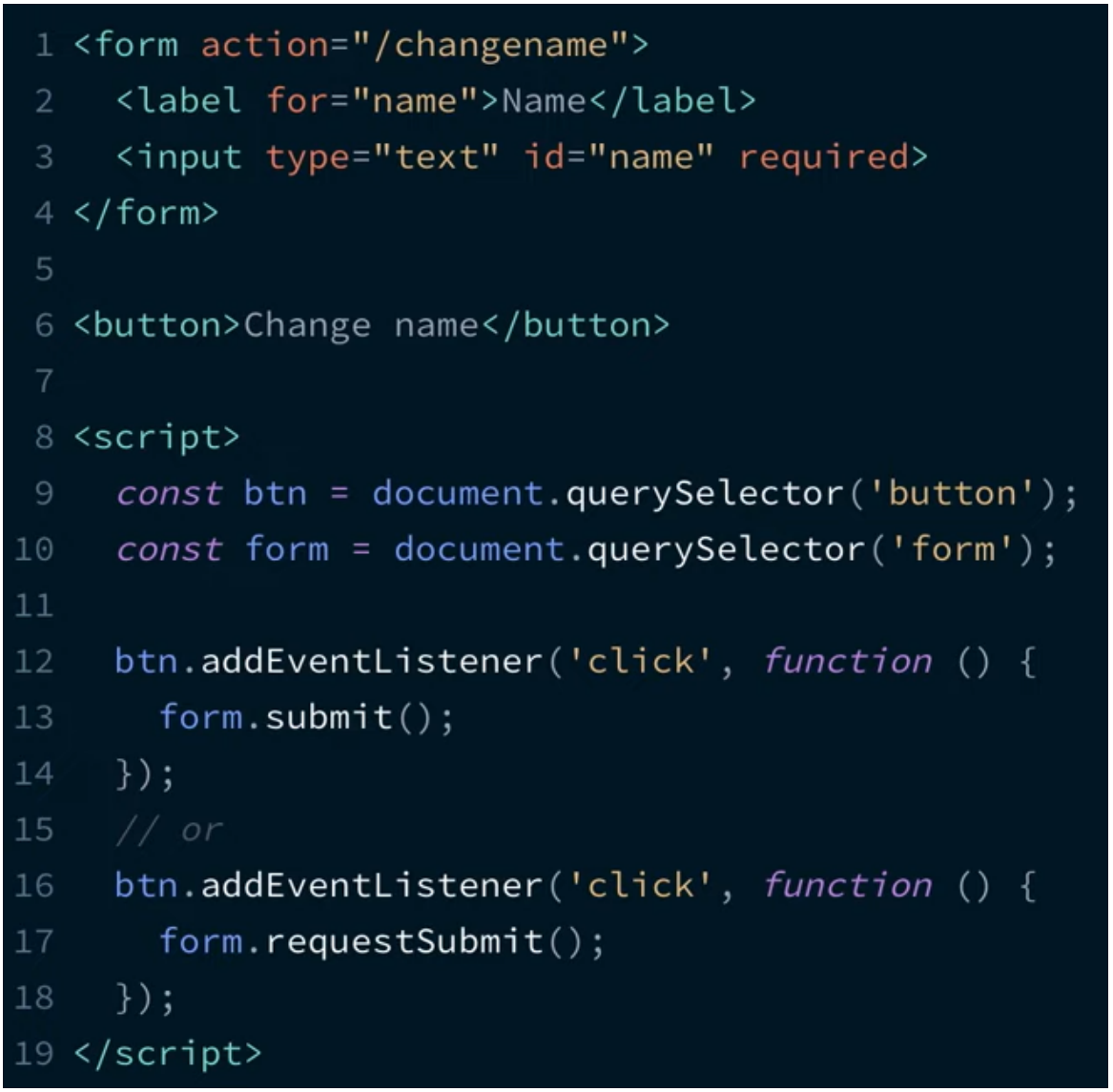
* block and inline dimension direction depend on the element writing mode
(Example shows top-to-bottom and left-to-right)

▼ Разница между методами `.submit()` и `.requestSubmit()`?

Это события формы, а не тип кнопки внутри формы, поэтому работает немного иначе чем кнопка с типом submit.

.**submit**() - отправит невалидную форму. Событие отправки не вызывается. В частности, обработчик события onsubmit формы не запускается. Проверка ограничения не запускается.

.**requestSubmit**() - отправит только валидную форму. Идентичен активации кнопки с типом <submit> формы. Содержимое формы проверяется, и форма отправляется только в том случае, если проверка прошла успешно. Как только форма была отправлена, событие отправки отправляется обратно объекту формы.

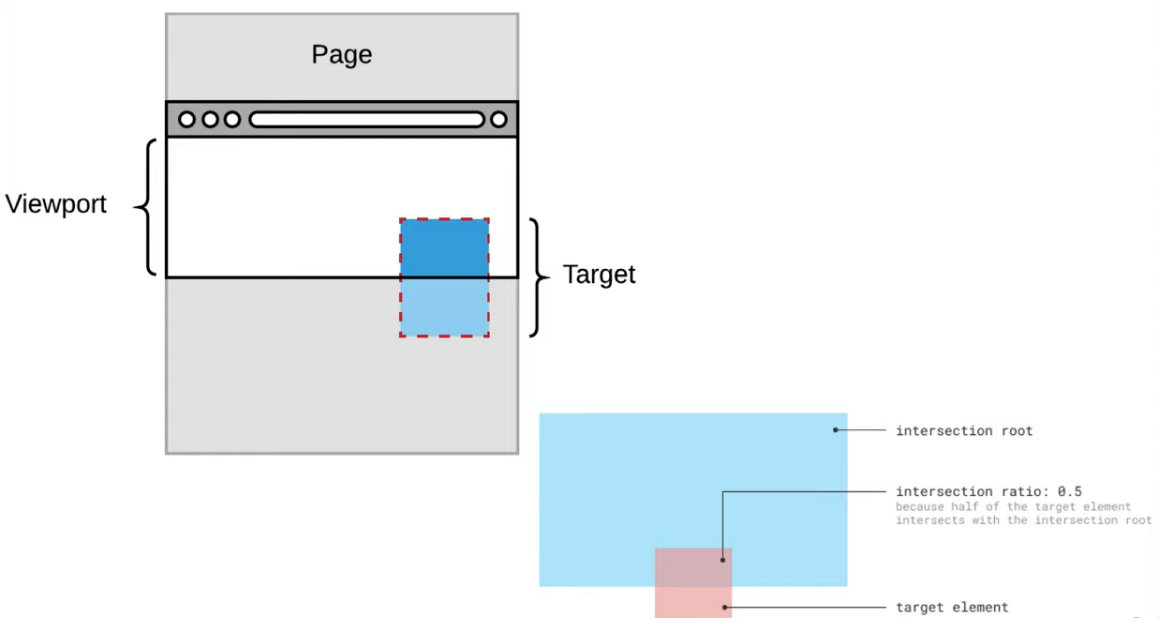


```html
<form action="/changename">
  <label for="name">Name</label>
  <input type="text" id="name" required>
</form>

<button>Change name</button>

<script>
  const btn = document.querySelector('button');
  const form = document.querySelector('form');

  btn.addEventListener('click', function () {
    form.submit();
  });
  // or
  btn.addEventListener('click', function () {
    form.requestSubmit();
  });
</script>
```

▼ Расскажите о `IntersectionObserver`?

Подробнее тут

Intersection Observer — браузерный API, который позволяет асинхронно отслеживать пересечение элемента с его родителем или областью видимости документа (viewport). В момент пересечения можно запустить какое-либо действие, например, подгрузить дополнительные посты в ленте новостей («бесконечный скролл») или сделать «ленивую» загрузку контента.



▼ Расскажите о `URLSearchParams`?

Подробне тут

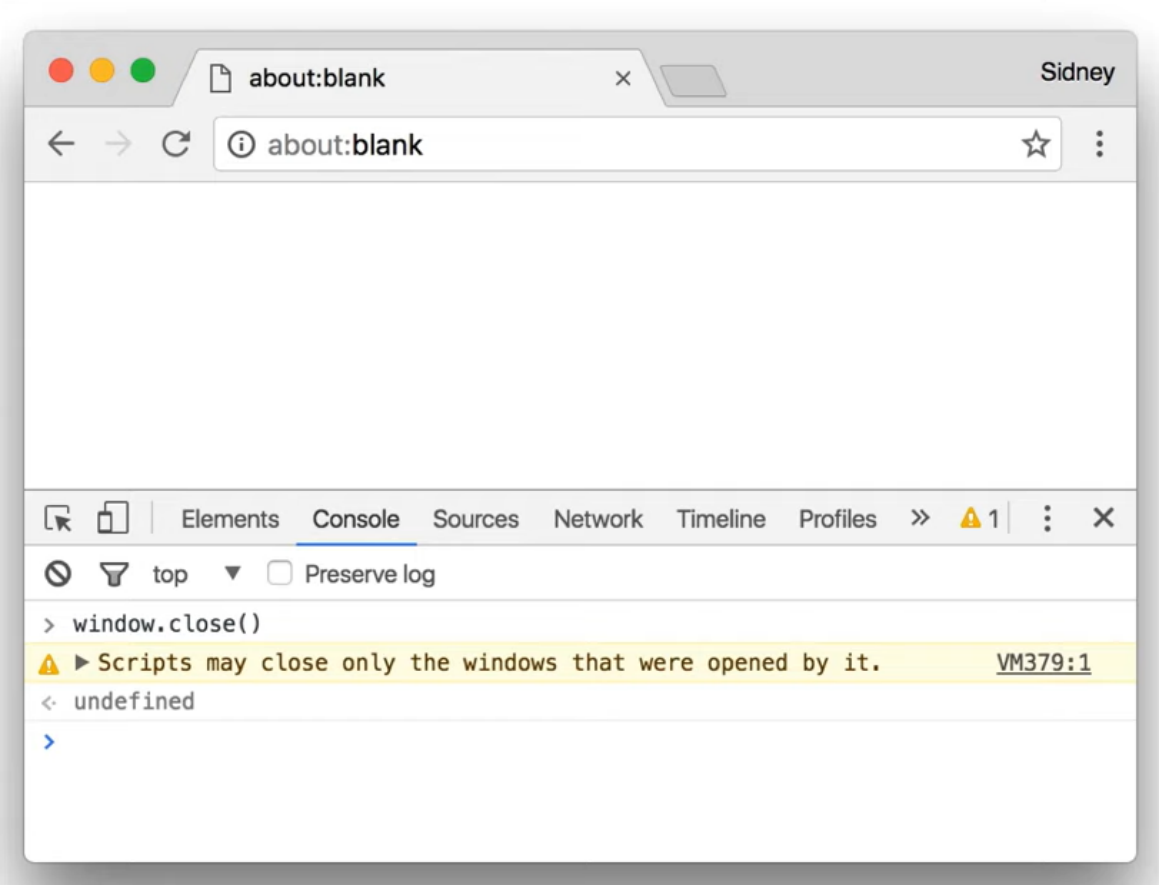
`URLSearchParams` — это класс, предоставляющий удобное API для формирования строки поисковых параметров, которую потом можно использовать для формирования полного адреса. Все параметры в строке будут закодированы для безопасной вставки в адрес. Также этот класс можно встретить как часть класса `URL`.

```
const params = new URLSearchParams();
params.append('foo', '1');
params.append('bar', '2');

console.log(params.toString()); // 'foo=1&bar=2'

params.set('foo', '3');

console.log(params.get('foo')); // '3'

params.delete('bar');

console.log(params.has('bar')); // false
```

▼ Какие есть ограничения у `window.close()`?

Этот метод разрешено использовать только для окон, которые были открыты с помощью функции `window.open()` (en-US), или для окон верхнего уровня(активного окна), имеющих единственную запись в истории. Если окно не соответствует этим требованиям, в консоли появится ошибка, похожая на эту: `Scripts may not close windows that were not opened by script.`

Работает только в пределах открытой вкладки в которой выполняется этот скрипт, соседние вкладки таким образом закрыть нельзя.



▼ Что такое Стек и Куча?

Подробнее тут

Стек (англ. stack) – это структура данных, которая используется для хранения статических данных, т.е. тех, чей размер известен во время компиляции. В JavaScript сюда включаются примитивные значения (string, number, boolean, undefined и null) и ссылки на функции и объекты.

Движок «понимает», что размер данных не меняется, поэтому выделяет фиксированный объем памяти для каждого из значений. Процесс выделения памяти до исполнения называется статическим выделением памяти (static memory allocation). Так как браузер выделяет память заранее для каждого типа данных есть ограничение на размер данных, которые можно туда положить.

Куча (англ. memory heap) используется для хранения таких данных, как объекты и функции. Но в отличие от стека движок не может «знать», какой объем памяти необходим для того либо иного объекта, поэтому память выделяется по мере необходимости. И этот способ выделения памяти называется «динамическим» (dynamic memory allocation).

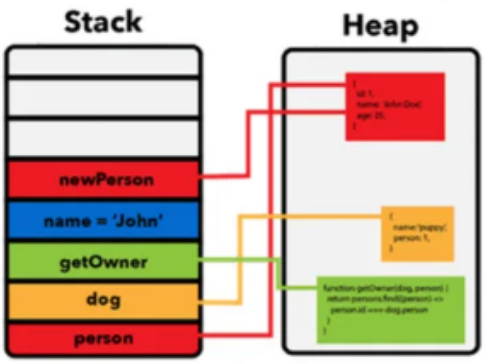
Стек	Куча
Примитивные значения и ссылки	Объекты и функции
Размер известен во время компиляции	Размер известен во время выполнения
Выделяется фиксированный объем памяти	Объем ничем не ограничен

Все переменные в первую очередь указывают на стек. В случае, если значение не является примитивным, в стеке содержится ссылка на объект из кучи.

В куче нет какого-либо определенного порядка, в связи с чем ссылка на нужную нам область памяти должна храниться в стеке: в этом смысле объект в куче похож на дом, а ссылка – на его адрес.

Примечание JavaScript хранит **объекты и функции** в куче, в то время как **примитивные значения и ссылки** находятся в стеке.

```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
    person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```



Async JS

▼ Разница между синхронными и асинхронными функциями?

Синхронные функции являются блокирующими, в то время как асинхронные нет.

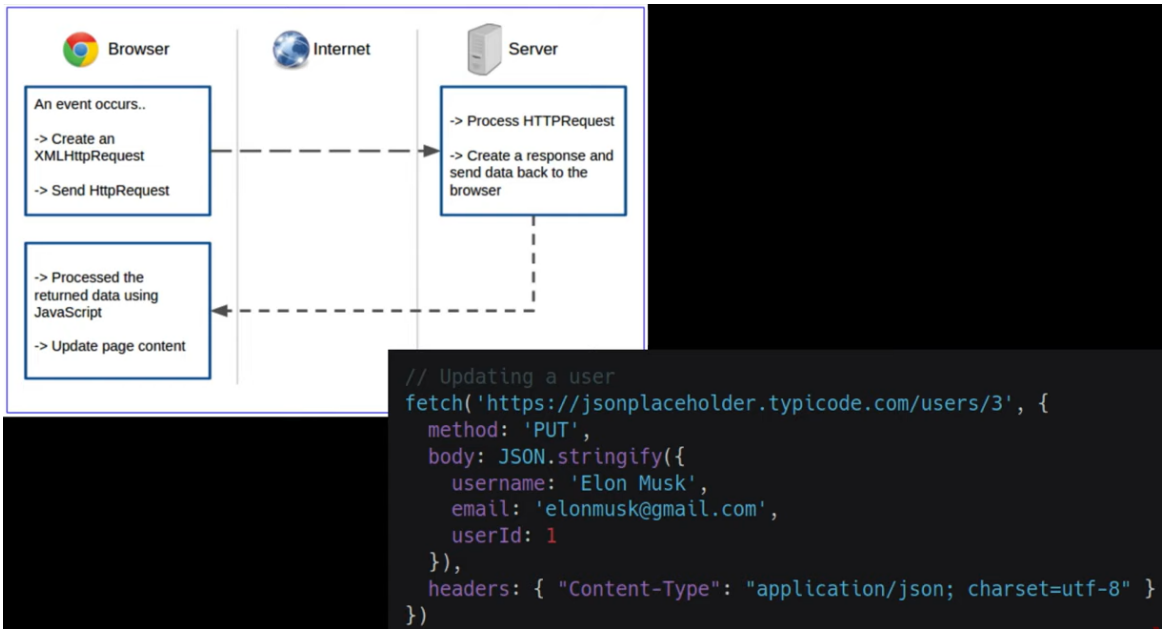
Когда интерпретатор JavaScript натыкается на синхронную функцию он блокирует дальнейшее выполнение операций, пока она не выполнится.

Что относится к асинхронному коду:

- таймеры (setInterval/setTimeout)
- все коллбеки
- Promises
- Async/await
- FetchAPI/XMLHttpRequest
- Слушатели событий (addEventListener и т.д.)
- Взаимодействие с файловой системой (чтение или запись файлов)

▼ Что такое AJAX?

AJAX (Asynchronous Javascript and XML) - это технология, которая позволяет отправлять и получать данные с сервера без перезагрузки страницы. AJAX использует комбинацию JavaScript и XML (или других форматов данных, таких как JSON), чтобы обмениваться данными с сервером асинхронно, то есть без блокирования пользовательского интерфейса. AJAX используется для загрузки дополнительных данных на страницу, отправки форм, обновления содержимого страницы и многих других задач.



▼ Что такое same-origin policy в контексте JavaScript?

Same-Origin Policy (SOP) - это концепция веб-безопасности, которая применяется в браузерах для ограничения взаимодействия между документами или скриптами, загруженными из разных источников.

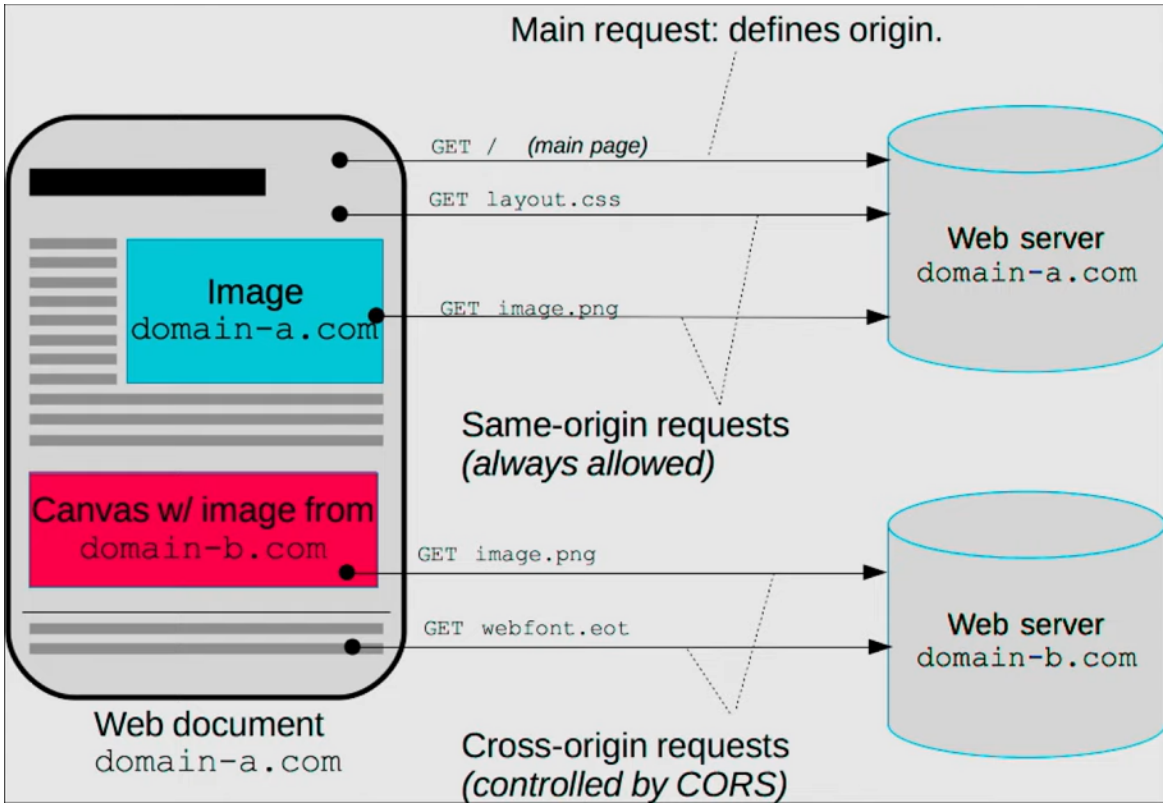
Основная идея заключается в том, что документ или скрипт, загруженные из одного источника («оригина»), должны иметь доступ только к данным из того же самого источника, а не к данным из других источников.

Конкретно в контексте JavaScript это означает, что скрипты, загруженные с одного домена, протокола и порта (оригина), не могут получить доступ к содержимому документа или ресурсам, загруженным из другого домена, протокола или порта. Это ограничение распространяется на такие операции, как чтение свойств документа, отправка XMLHttpRequest запросов к другому домену, доступ к cookies, localStorage и т. д.

Эта политика помогает предотвратить атаки, такие как Cross-Site Scripting (XSS) и Cross-Site Request Forgery (CSRF), защищая данные пользователя от несанкционированного доступа и модификации.

Однако, веб-приложения могут управлять CORS (Cross-Origin Resource Sharing) заголовками для переопределения политики same-origin и разрешения или запрета доступа к ресурсам из других источников.

Обойти их можно с помощью CORS.



▼ Что такое цикл событий (event loop) и как он работает?

Event loop - бесконечный цикл, в котором движок **JavaScript** ожидает задачи, исполняет их и снова ожидает появления новых.

Как работает Event Loop?

1. Выполняет все синхронные задачи
2. Выполняет все микротаски.
3. Выполняет одну макротаску.

Более подробный алгоритм событийного цикла (хоть и упрощённый в сравнении со спецификацией):

1. Выбрать и исполнить старейшую задачу из очереди макрозадач.
2. Исполнить все микрозадачи
3. Отрисовать изменения страницы, если они есть.
4. Если очередь макрозадач пуста – подождать, пока появится макрозадача.
5. Перейти к шагу 1.

▼ Разница между микро и макрозадачами в event loop?

Разница между микрозадачами (microtasks) и макрозадачами (macrotasks) заключается в их приоритете выполнения и времени запуска.

Микрозадачи (Microtasks):

- Микрозадачи имеют более высокий приоритет выполнения по сравнению с макрозадачами.
- Они выполняются после завершения текущего цикла событий и до начала следующего цикла событий.

Макрозадачи (Macrotasks):

- Макрозадачи имеют более низкий приоритет выполнения по сравнению с микрозадачами.
- Они обычно включают в себя операции, которые требуют более длительного времени выполнения, такие как обработка событий ввода/вывода (например, пользовательский ввод, загрузка ресурсов), таймеры (`setTimeout`, `setInterval`) и выполнение скриптов.
- Макрозадачи добавляются в конец текущего цикла событий, выполняются после выполнения всех микрозадач текущего цикла и до начала следующего цикла событий.

Микротаски включают в себя:

- **Промисы (Promises)**
- **Обратные вызовы промисов:** обратные вызовы, переданные методам **then, catch, finally** становятся микрозадачами
- **Мутации DOM:** добавление, удаление элементов, изменение атрибутов или текстового содержимого, обработчик событий мутаций DOM (MutationObserver)
- **Микротаск-очередь:** Специальная очередь, в которую можно поместить пользовательские микрозадачи с помощью функции **queueMicrotask().**Функции, переданные в **queueMicrotask()**, становятся микрозадачами и выполняются после выполнения текущей макрозадачи и перед началом следующей макрозадачи.
- **Обратные вызовы событий (Event Callbacks)**: Обратные вызовы, переданные методам, таким как `addEventListener()`, также могут быть

обработаны как микрозадачи, хотя это зависит от специфики браузера.

- **Методы Promise.resolve() и Promise.reject()**: Вызов методов `resolve()` и `reject()` объекта Promise также создает микрозадачи для обработки промисов.

Макротаски это:

1. Таймеры
2. События
3. Загрузка ресурсов
4. Браузерные нюансы (рендер, I/O и т.д.)

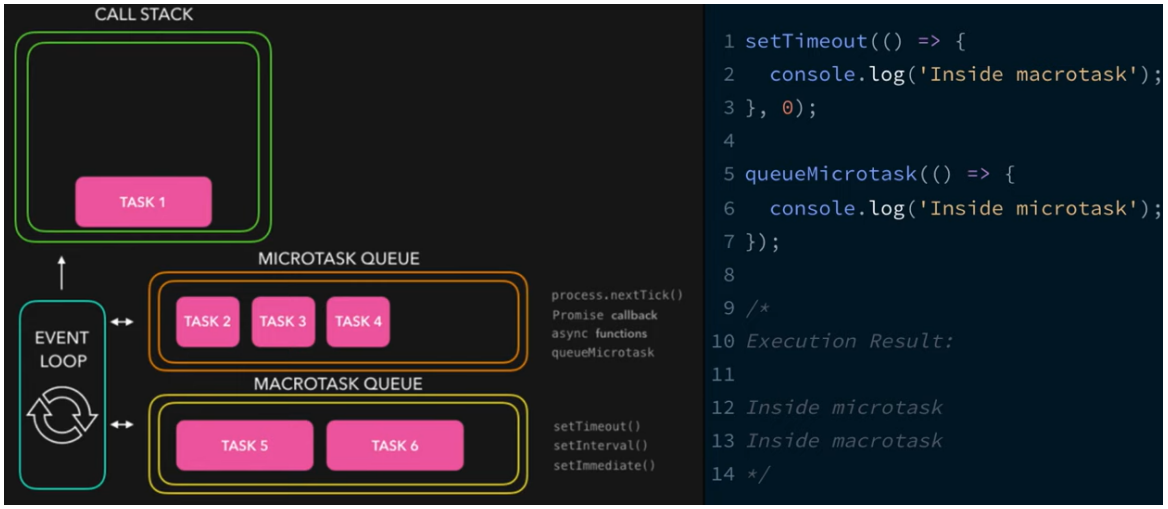
```
setTimeout(() => alert("timeout"));

Promise.resolve()
  .then(() => alert("promise"));

alert("code");
```

Какой здесь будет порядок?

1. **code** появляется первым, т.к. это обычный синхронный вызов.
2. **promise** появляется вторым, потому что **.then** проходит через очередь микрозадач и выполняется после текущего синхронного кода.
3. **timeout** появляется последним, потому что это макрозадача.

▼ Расскажите о `queueMicrotask`?

Добавляет функцию в очередь микрозадач



▼ Что такое промисы (Promises)?

Подробнее тут

Промис (Promise) — это объект, который используется для выполнения асинхронных операций и управления их результатами. Он представляет собой обещание выполнить асинхронную операцию и вернуть ее результат в будущем, либо отклонить ее в случае возникновения ошибки.

Асинхронные функции возвращают объект `Promise` в качестве значения. Внутри промиса хранится результат вычисления, которое может быть уже выполнено или выполнится в будущем.

Состояния промиса:

- ***pending*** — исходное состояние, когда промис еще не был выполнен или отклонен.
- ***fulfilled*** — операция завершилась успешно и результат доступен.
- ***rejected*** — операция завершилась с ошибкой и результат недоступен.

Поменять состояние можно только один раз: перейти из `pending` либо в `fulfilled`, либо в `rejected`

Для обработки результатов выполнения **Promise** используются методы `then()` и `catch()`:

- **then()**: Выполняется при успешном выполнении Promise и принимает функцию обратного вызова, которая обрабатывает результат.

- **catch()**: Выполняется при возникновении ошибки и принимает функцию обратного вызова для обработки ошибки.

Пример:

```
const myPromise = new Promise((resolve, reject) => {
  // Асинхронная операция
  if (/* операция выполнена успешно */) {
    resolve("Результат успешного выполнения");
  } else {
    reject("Ошибка");
  }
})
.then(result => {
  console.log("Результат:", result);
})
.catch(error => {
  console.error("Ошибка:", error);
});
```

Методы Promise:

- **Promise.all(iterable)**: Принимает массив (или итерируемый объект) промисов и возвращает новый промис, который выполняется, когда все промисы в массиве выполнятся, или отклонятся, если хотя бы один промис отклоняется.
- **Promise.race(iterable)**: Принимает массив (или итерируемый объект) промисов и возвращает новый промис, который выполняется или отклоняется в зависимости от того, какой Promise в массиве выполнится или отклонится первым.

▼ Плюсы и минусы использовании Ajax?

Плюсы:

1. Повышение интерактивности т.к. новые данные могут загружаться и обновляться частично без полной перезагрузки страницы.
2. Сокращение количества обращений к серверу, скрипты и таблицы запрашиваются один раз.
3. Хранение сосояния страницы, имеется в виду что переменные в JS и состояние в DOM не перезагружается и сохраняет свои значения.

4. Меньшая нагрузка на сервер - данная технология позволяет не использовать такое количество запросов к базе данных;
5. Быстрая работа веб-страниц - реакция на действия пользователя более динамична;

Минусы:

1. Проблемная реализация добавления динамической веб страницы в закладки(приходится сохранять query параметры для сохранения состояния страницы)
2. Не работает без JS
3. Безопасность не гарантируется - весь код программного скрипта доступен для просмотра в браузерах и может быть использован злоумышленниками;
4. Не работает история просмотров, что плохо для отслеживания статистики и анализа поведения пользователей;
5. Проблемы с индексацией - роботы поисковых систем иногда испытывают трудности с просмотром страниц, использующих AJAX. В некоторых случаях такие страницы вообще не индексируются.

▼ Подходы при работе с асинхронным кодом?

Нужны для последовательной обработки асинхронного кода

1. callback
2. promise
3. async/await

```
// Callback
fs.readFile("./test.txt", 'utf-8', (error, data) => {
  fs.mkdir("./files", () => {
    fs.writeFile("./files/test2.txt", `${data}New text!`, (error) =>
{});
  });
});

// Promice
const fetchUser = () => {
  fetch('https://randomuser.me/api/')
    .then(response => response.json())
    .then(data => { setUser(data.results[0]); })
    .catch(error) { console.log(error); }
}

// Async/Await
const fetchUser = async () => {
  const response = await fetch('https://randomuser.me/api/');
  const data = await response.json();
  setUser(data.results[0]);
}
```

▼ Преимущества использования промисов вместо колбэков?

- Помогает избежать callback hell
- Упрощает написание последовательного читаемого кода с помощью then, а также обработку ошибок с помощью catch.
- Упрощает написание параллельного асинхронного кода с помощью Promise.All()
- С использованием промисов, можно избежать следующих проблем с колбэками: колбэк функция была вызвана слишком рано или слишком поздно, пропуск ошибок или исключений

▼ Что такое callback-функция? Что такое Callback Hell?

Callback - это функция, которая передается в другую функцию в качестве параметра/аргумента и вызывается после выполнения определенной операции.

Callback Hell - это ситуация, когда вложенные коллбэки становятся слишком глубокими и сложными для понимания и отладки.

```
// HOC
[1, 2, 3, 4].forEach(element => console.log(element));

// Timer
setTimeout(() => alert('Hello world'), 1000);

// Even callback
btnAdd.addEventListener('click', function clickCallback(e) {
    // do something
});

// Callback Hell
fs.readFile('somefile.txt', function (e, data) {
  // do something
  fs.readdir('directory', function (e, files) {
    // do something
    fs.mkdir('directory', function (e) {
      // do something
    });
  });
});
```

▼ Что такое `async/await`?

Это синтаксический сахар над промисами.

Это способ написания асинхронного кода в JS делающий его более читаемым. Async/Await заставляет код, который выполняется асинхронно, выглядеть как синхронный код. Для отлова ошибок необходимо оборачивать в **try catch.**

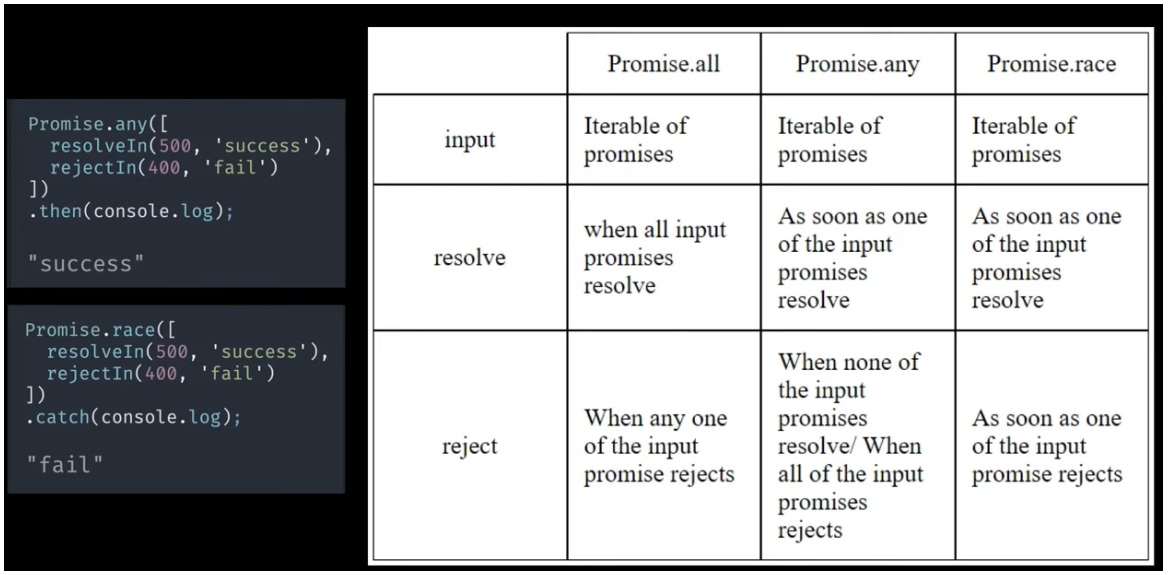
Ключевое слово **async** пишется перед функцией и гарантирует нам, что функция вернет промис.

Ключевое слово **await** заставит интерпретатор ждать до тех пор, пока промис справа от него не выполнится, после чего вернет его результат и выполнение кода продолжится.

await можно использовать только внутри async, но есть top level await

▼ Разница между `Promise.all()`, `Promise.any()` и `Promise.race()`?

- **Promise.all()** - принимает массив промисов и возвращает новый промис, который будет выполнен только тогда, когда все промисы в массиве будут выполнены. Если хотя бы один промис из массива отклонится, то Promise.all() тоже отклонится.
- **Promise.any()** - также принимает массив промисов и возвращает новый промис, который будет выполнен, когда хотя бы один из промисов в массиве будет выполнен успешно. Если все промисы отклонятся, то Promise.any() тоже отклонится.
- **Promise.race()** также принимает массив промисов и возвращает новый промис, который будет выполнен, когда любой из промисов в массиве будет выполнен, независимо от того, успешно или нет. Результат выполнения первого завершившегося промиса будет использован для выполнения Promise.race().



```
Promise.any([
  resolveIn(500, 'success'),
  rejectIn(400, 'fail')
])
.then(console.log);

"success"
```

```
Promise.race([
  resolveIn(500, 'success'),
  rejectIn(400, 'fail')
])
.catch(console.log);

"fail"
```

	Promise.all	Promise.any	Promise.race
input	Iterable of promises	Iterable of promises	Iterable of promises
resolve	when all input promises resolve	As soon as one of the input promises resolve	As soon as one of the input promises resolve
reject	When any one of the input promise rejects	When none of the input promises resolve/ When all of the input promises rejects	As soon as one of the input promise rejects

▼ Расскажите про статический метод `.allSettled()`?

Подробнее тут

1. `Promise.allSettled(promises)` (добавлен недавно) – ждёт, пока все промисы завершатся и возвращает их результаты в виде массива с объектами, у каждого объекта два свойства:
    - `status`: состояние промиса, может быть одним из значений: "fulfilled" (разрешен), "rejected" (отклонен).
    - `value`: значение, если промис разрешен.
    - `reason`: причина отклонения, если промис был отклонен.

2. `Promise.resolve(value)` – возвращает успешно выполнившийся промис с результатом `value`.
3. `Promise.reject(error)` – возвращает промис с ошибкой `error`.

Пример кода:

```
const promises = [
  Promise.resolve('Успех'),
  Promise.reject('Ошибка'),
  Promise.resolve('Еще один успех')
];

Promise.allSettled(promises)
  .then(results => {
    results.forEach(result => {
      if (result.status === 'fulfilled') {
        console.log(`Промис разрешен с результатом: ${resul
      } else {
        console.log(`Промис отклонен с причиной: ${result.r
      }
    });
  });
```

▼ Плюсы и минусы асинхронного программирования в JavaScript?

Плюсы асинхронного программирования

- Все скрипты загружаются по одному. Это означает скорость, отзывчивость и лучший пользовательский интерфейс.
- Это устраняет задержки загрузки страницы. Таким образом, нет необходимости в последующем обновлении страницы при выполнении новых запросов.
- Вы можете использовать несколько функций одновременно, даже если другие запросы все еще выполняются.
- Асинхронные приложения хорошо масштабируются и требуют немного ресурсов для работы.

- Даже если один запрос откликается медленно, это не влияет на время отклика других.
- Сбой одного потока не останавливает рендеринг других.
- Встроенные обратные вызовы позволяют настраивать сообщения об ошибках.

Минусы асинхронного программирования

- Это требует множества обратных вызовов и рекурсивных функций, которые могут быть громоздкими во время разработки.
- Если обратные вызовы используются неэффективно, пользователь не может узнать, не выполняется ли запрос, особенно при выполнении запросов POST.
- Задержка при отображении начальной страницы может повлиять на ваше восприятие.
- Веб-приложения, использующие асинхронную загрузку, могут быть трудно сканировать поисковыми системами, такими как Google и Bing.
- В некоторых языках программирования может быть сложно реализовать асинхронные сценарии.
- Код может стать беспорядочным и трудным для отладки.

Плюсы синхронного программирования

- Он требует меньше знаний в области кодирования и поддерживается всеми языками программирования.
- Даже если нет настраиваемых обратных вызовов для сбоев запросов, это сразу очевидно для вас, поскольку клиент (браузер) обрабатывает такие ошибки по умолчанию.
- Лучше для выполнения задач ЦП.
- Поисковые системы находят, что синхронные веб-страницы легче сканировать.
- Идеально подходит для простых запросов.

Минусы синхронного программирования

- Время загрузки может быть медленным.
- Нет встроенных методов обратного вызова.
- Когда поток заблокирован, блокируются и другие.
- Невозможность одновременного выполнения нескольких операций может ухудшить взаимодействие с пользователем.
- После сбоя запроса вся программа также перестает отвечать.
- Для обработки большего количества потоков может потребоваться огромное количество ресурсов, если запросы становятся чрезмерными.

▼ Проблемы при использовании callback-функций?

- Callback hell
- Ошибки колбеков которые трудно отлавливать
- Нет контроля над потоком выполнения, зависим от времени ответа от сервера блокируя поток и приводит к зависанию интерфейса

```
getUserData(function (userData) {
  getPosts(userData.id, function (posts) {
    getComments(posts[0].id, function (comments) {
      getRate(rates[0].id, function (rate) {
        renderPosts(posts, comments, rate, function (html) {
          showOnPage(html);
        });
      });
    });
  });
});
```

▼ Как выполнить несколько асинхронных операций последовательно?

- Цепочка then
- Async/await

```
// using "".then":
fetchData()
  .then(result1 => {
    return secondAsyncOperation(result1);
  })
  .then(result2 => {
    return thirdAsyncOperation(result2);
  })
  .catch(error => {
    console.error('Some error', error);
  });

// using "async/await":
async function fetchData() {
  try {
    const response = await fetch('https://example.com/data');
    const data = await response.json();
  } catch (error) {
    console.error('Some error', error);
  }
}
```

▼ Какие проблемы может вызвать неправильное использование асинхронности в JavaScript?

- Потеря контекста при передаче или вызове функции, решается привязыванием контекста или использованием стрелочных функций
- Callback hell, решается использованием промисов или async/await
- Race conditions - когда множество операций зависят от одних и тех же ресурсов и могут работать с ними одновременно. Решается использованием блокировки или использованием атомарных операций
- Утечки памяти - когда появляется множество неуправляемых ссылок на объекты, которые не очищаются сборщиком мусора, решается своевременным удалением ссылок и очисткой ресурсов
- Недостаточная производительность, когда использование асинхронности приводит к замедлению работы приложения. Тогда нужно применять асинхронность только в необходимых местах.

▼ Что такое mutationObserver?

MutationObserver - это встроенный объект, наблюдающий за DOM-элементом и запускающий колбэк в случае изменений.

Синтаксис:

```
let observer = new MutationObserver(callback);
```

Потом прикрепляем его к DOM-узлу:

```
observer.observe(node, config);
```

`config` – это объект с булевыми параметрами «на какие изменения реагировать»:

- `childList` – наблюдать за добавлением или удалением дочерних элементов (Включая текстовые узлы (text nodes)),
- `subtree` – наблюдать за потомками целевого элемента.
- `attributes` – наблюдать за изменениями атрибутов целевого элемента.,
- `attributeFilter` – массив имён атрибутов, чтобы наблюдать только за выбранными.
- `characterData` – наблюдать за изменениями значения текстового содержимого целевого узла (текстовых узлов дочернего элемента).
- `characterDataOldValue` – если `true`, будет передавать и старое, и новое значение `node.data` в колбэк, иначе только новое (также требуется опция `characterData`),
- `attributeOldValue` – если `true`, будет передавать и старое, и новое значение атрибута в колбэк, иначе только новое (также требуется опция `attributes`).

Затем, после изменений, выполняется `callback`, в который изменения передаются первым аргументом как список объектов MutationRecord, а сам наблюдатель идёт вторым аргументом.

Метод, останавливающий наблюдение за узлом:

- `observer.disconnect()` – останавливает наблюдение.

Вместе с ним используют метод:

- `mutationRecords = observer.takeRecords()` – получает список необработанных записей изменений, которые произошли, но колбэк для них ещё не выполнился.

Пример кода:

```
const box = document.querySelector('.box');
// создаем обсервер
const observer = new MutationObserver(mutationRecords => {
    console.log(mutationRecords);
});
// запускаем слежение
observer.observe(box, { childList: true });
```

2 ВАЖНЫХ НЮАНСА:

1 - Обсервер срабатывает **после(!!!)** изменений, т.е. мы не можем что-то сделать **во время** или **до** изменений. Мы работаем с результатом.

2 - **mutationObserver** это асинхронная операция!!! Т.е. выполнение каких-то действий может немного задержаться.

Сборка мусора

Объекты `MutationObserver` используют внутри себя так называемые «слабые ссылки» на узлы, за которыми смотрят. Так что если узел удалён из DOM и больше не достижим, то он будет удалён из памяти вне зависимости от наличия наблюдателя.

Когда это может быть нужно?

Представим ситуацию, когда вы подключаете сторонний скрипт, который добавляет какую-то полезную функциональность на страницу, но при этом делает что-то лишнее, например, показывает рекламу `<div class="ads">Ненужная реклама</div>`.

Разумеется, сторонний скрипт не даёт каких-то механизмов её убрать.

Используя `MutationObserver`, мы можем отследить, когда в нашем DOM появится такой элемент и удалить его. А полезную функциональность оставить.

Есть и другие ситуации, когда сторонний скрипт добавляет что-то в наш документ, и мы хотели бы отследить, когда это происходит, чтобы

адаптировать нашу страницу, динамически поменять какие-то размеры и т.п.

ES6

ECMAScript

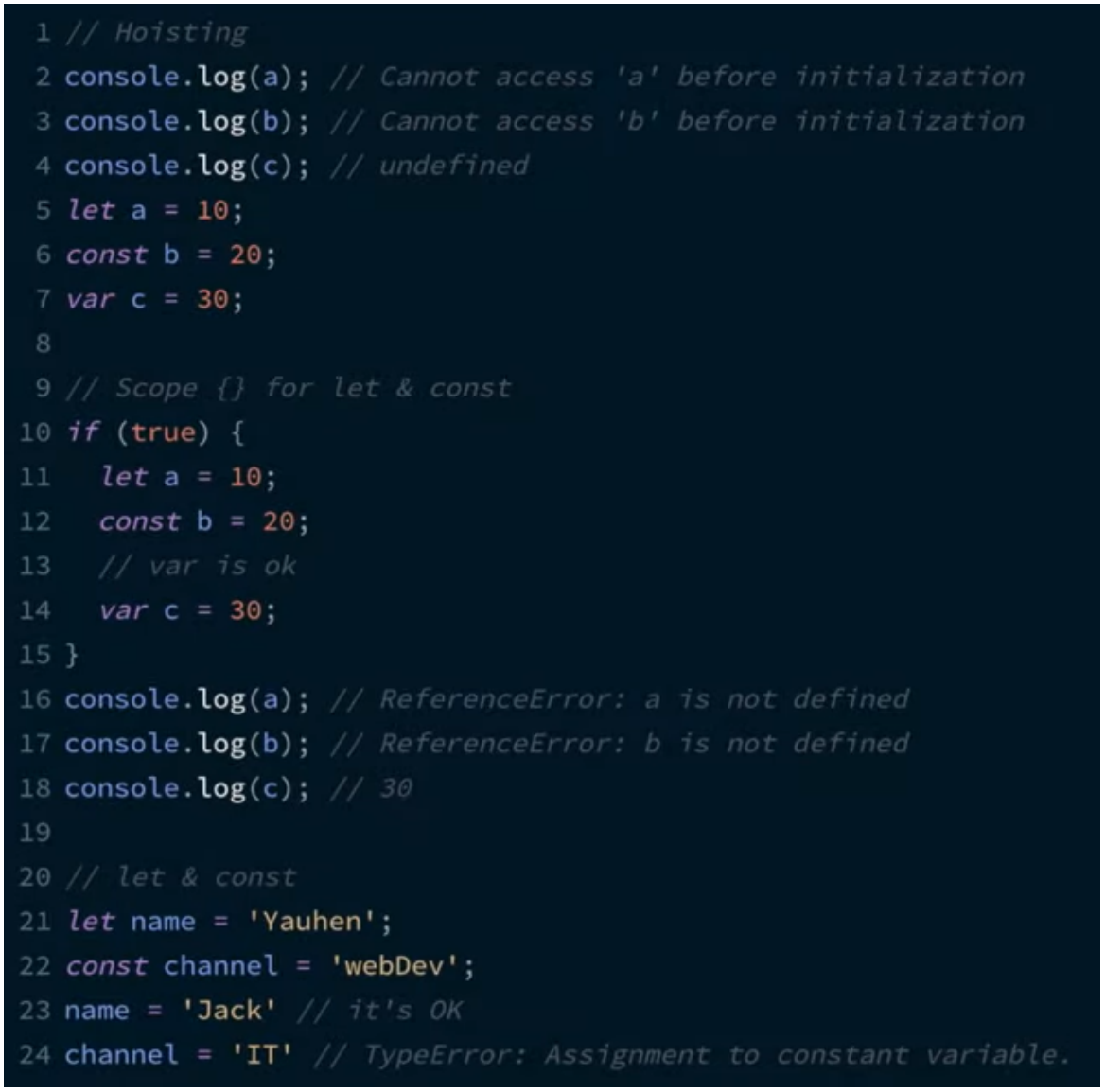
▼ Что такое ECMAScript? В чём отличие от JavaScript?

ECMAScript - это спецификация, стандарт, определяющий язык программирования JavaScript. Он описывает синтаксис, типы данных, операторы, объекты и функции языка JavaScript. ECMAScript был создан для обеспечения единого стандарта для разработчиков языка, и JavaScript это одна из реализация стандарта. На данный момент последней версией стандарта является ECMAScript 2023.

▼ Разница между `let`, `const` и `var`?

Подробнее тут

- Переменные, объявленные при помощи var, имеют глобальную или локальную (в рамках функции) область видимости. Если вы объявляете переменные с использованием let или const, их область видимости будет блочной.
- Переменные, объявленные при помощи var, могут обновляться и объявляться заново. Использование let позволяет обновлять переменную, но не объявлять заново, а использование const не предполагает ни того, ни другого.
- Переменные всех видов поднимаются в верх своей области видимости. Но переменные, объявленные при помощи var, инициализируются как undefined, а объявленные с использованием let или const — не инициализируются.
- При помощи var или let можно объявлять переменные без их инициализации, но если вы объявляете переменную при помощи const, она должна инициализироваться при объявлении.



```
// Hoisting
console.log(a); // Cannot access 'a' before initialization
console.log(b); // Cannot access 'b' before initialization
console.log(c); // undefined
let a = 10;
const b = 20;
var c = 30;

// Scope {} for let & const
if (true) {
  let a = 10;
  const b = 20;
  // var is ok
  var c = 30;
}
console.log(a); // ReferenceError: a is not defined
console.log(b); // ReferenceError: b is not defined
console.log(c); // 30

// let & const
let name = 'Yauhen';
const channel = 'webDev';
name = 'Jack' // it's OK
channel = 'IT' // TypeError: Assignment to constant variable.
```

▼ Можно ли изменить значение определённое через `const` ?

Свойства и методы объекта объявленные через const могут быть изменены. Т.к. хранится ссылка, а не сами данные, сохраняя ссылку данные можно поменять.

```
// Object
const a = { a: 10 };
// Reassign Error
a = { a: 20 }; // Uncaught TypeError: Assignment to constant variable.
// Adding Property
a.b = 20
console.log(a); // { a: 10, b: 20 }

// Array
const c = [];
// Reassign Error
c = {}; // Uncaught TypeError: Assignment to constant variable.
// Adding Values
c.push(5);
c.push(10);
console.log(c); // [5, 10]
```

▼ Что такое временная мёртвая зона (temporal dead zone)?

Дело в том, что let и const всплывают также, как и var, но для того, чтобы легче отлавливать ошибки, которые в ES5 были вызваны hoisting'ом, в ES6 для переменных и констант создали мертвую зону, а это значит, что переменные будут созданы, когда интерпретатор зайдет в область видимости, но они не будут доступны пока выполнение кода не дойдет до места их фактического объявления.

```
// START - temporal dead zone for 'b' and 'c'
console.log(a); // undefined
console.log(b); // ReferenceError: b is not defined
console.log(c); // ReferenceError: c is not defined

var a = 10;
// END - temporal dead zone for 'b'
let b = 20;
// END - temporal dead zone for 'c'
const c = 30;
```

▼ Разница между Rest и Spread операторами?

Rest оператор, обозначенный тремя точками (...), используется для объединения аргументов в массив.

Spread оператор, также обозначенный тремя точками (...), используется для распаковки значений из массива или объекта. Spread syntax (кроме случаев spread properties) может быть применён только к итерируемым объектам (iterable objects)

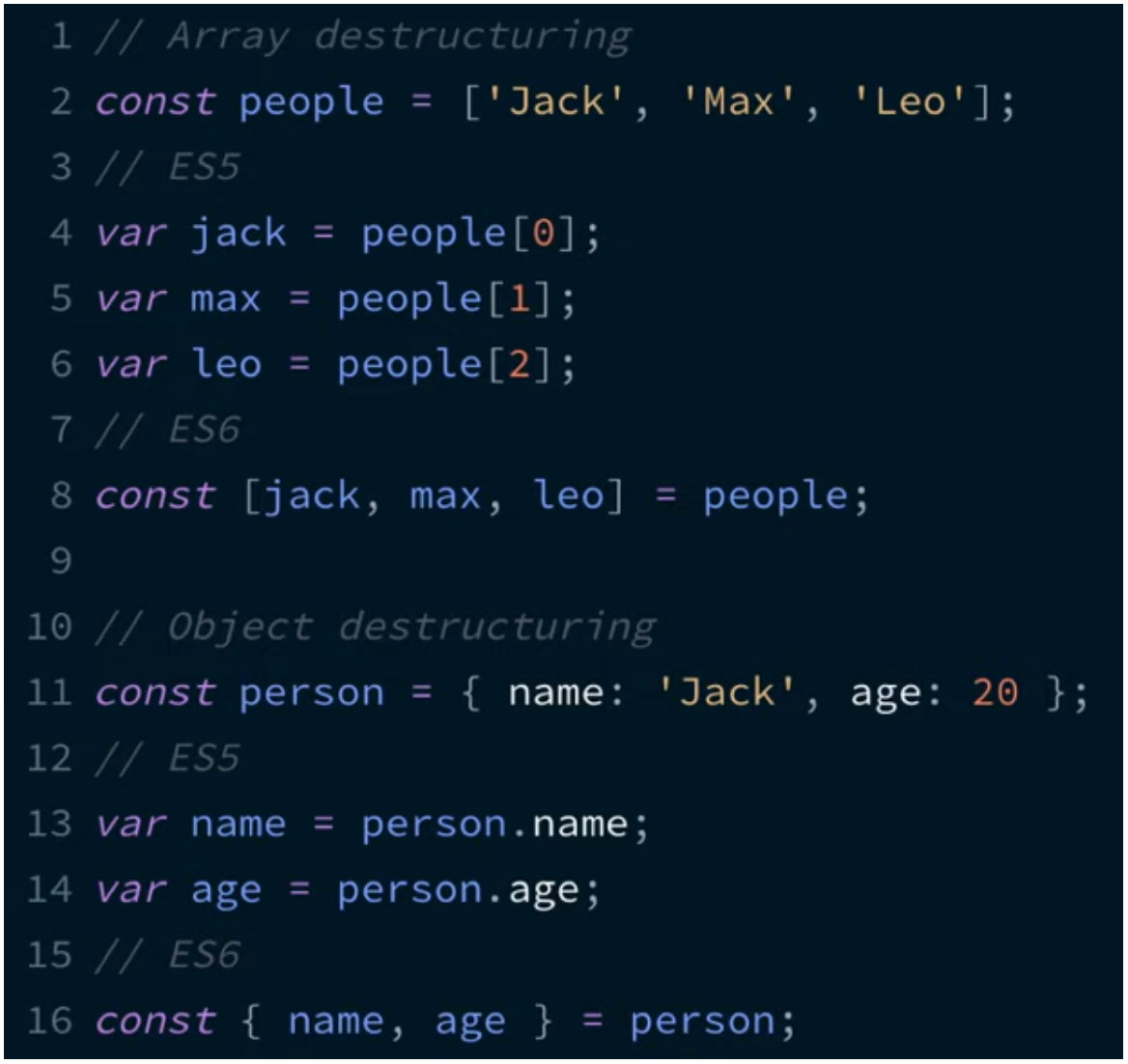
```
// Spread
// with array
const names1 = ['Jack', 'Max'];
const names2 = ['Leo', 'Tommy', ...names1];
console.log(names2); // ['Leo', 'Tommy', 'Jack', 'Max']
// with object
const obj1 = { name: 'Jack' };
const obj2 = {
  name: 'Max',
  age: 30,
  ...obj1
};
console.log(obj2); // { name: 'Jack', age: 30 }

// Rest
const log = (a, b, ...rest) => {
  console.log(a, b, rest);
};
log('Basic', 'rest', 'operator', 'usage'); // Basic rest ['operator', 'usage']
```

▼ Что такое деструктуризация?

Деструктуризация - это выражение доступное с версии стандарта ES6.

Деструктуризация - деструктурирующее присваивание, это способ извлечения значений из массивов или объектов и присваивания их переменным. Она позволяет более удобно работать с данными, избегая необходимости обращаться к каждому элементу или свойству по отдельности.

Из объектов деструктурируется по имени свойства, из массивов в порядке следования.



```
// Array destructuring
const people = ['Jack', 'Max', 'Leo'];
// ES5
var jack = people[0];
var max = people[1];
var leo = people[2];
// ES6
const [jack, max, leo] = people;

// Object destructuring
const person = { name: 'Jack', age: 20 };
// ES5
var name = person.name;
var age = person.age;
// ES6
const { name, age } = person;
```

▼ Для чего используется цикл `for…of`?

for...of появился в ES6

`for...of` используется для перебора элементов итерируемых объектов, таких как массивы, строки, Map, Set, NodeList и другие объекты, поддерживающие итерацию. Он перебирает значения элементов, а не их индексы.

`for...in` используется для перебора свойств объекта. Он перебирает имена (ключи) всех перечисляемых свойств объекта, включая свойства, унаследованные от его прототипа. Это означает, что если вы используете `for...in` для перебора массива, он также перечислит не только элементы массива, но и его свойства, включая индексы элементов и свойства, добавленные в прототип `Array`.

```
const names = ['Jack', 'Max', 'Leo'];

// for...in
for(let index in names) {
  console.log(index);           // 0, 1, 2
  console.log(names[index]);    // 'Jack', 'Max', 'Leo'
}

// for...of
for(let name of names) {
  console.log(name);            // 'Jack', 'Max', 'Leo'
}
```

▼ Что такое шаблонные литералы (Template Literals)?

Шаблонные литералы (Template Literals) - шаблонные строки, это специальный синтаксис для создания строк в JavaScript. Они позволяют создавать многострочные литералы без использования символов перевода строки, вставлять значения переменных внутрь строки без необходимости использования конкатенации или сложных операций со строками типа экранирования кавычек.

```
const name = "Yauhen";
// ES5
var greet = 'Hi I\'m ' + name  + '!';   // "Hi I'm Yauhen!"
var job = '\n'
    + ' I \n'
    + ' am \n'
    + 'Front-end developer \n';

// ES6
const greet = `Hi I'm ${name}!`;    // "Hi I'm Yauhen!"
const job = `
    I
    am
    Front-end developer
`;
```

▼ Что такое `Set`, `Map`, `WeakMap` и `WeakSet`?

Set - это коллекция уникальных значений любого типа данных. Он не допускает дубликатов и позволяет быстро проверять наличие элемента в коллекции.

WeakSet - это коллекция уникальных объектов. Он используется для хранения "слабых" ссылок на объекты, что позволяет им быть автоматически удаленными из коллекции при сборке мусора, если это единственная ссылка на элемент.

Map - это коллекция ключ-значение, где ключ может быть любого типа данных. Он позволяет быстро получать значение по ключу и выполнять другие операции с коллекцией.

WeakMap - это коллекция ключ-значение, где ключ должен быть объектом. Он используется для хранения "слабых" ссылок на объекты, что позволяет им быть автоматически удаленными из коллекции при сборке мусора, если это единственная ссылка на элемент.

```
// Map
let map = new Map();
// Adding elements in Map
map
    .set('str', 'string')       // key is string
    .set(1, 'number')           // key is number
    .set(true, 'boolean');      // key is boolean

// Set
let users = new Set();
// Data
const jack = { name: "Jack" };
const max = { name: "Max" };
const leo = { name: "Leo" };
users
    .add(jack)
    .add(max)
    .add(leo)
    .add(jack)              // Duplicate
    .add(max);              // Duplicate
console.log(users.size);    // 3 (no dubplicated values)
```

▼ Разница между обычными функциями и стрелочными?

У обычных функций this динамическое и зависит от контекста исполнения.

- Не создает свой контекст исполнения, а использует внешний, запоминая его в момент объявления
- Нельзя изменить контекст через привязку контекста
- Стрелочную функцию нельзя использовать как функцию конструктор.
- У стрелочной функции нет доступа к псевдомассиву arguments.
- В стрелочных функциях можно обойтись без return.
- Можно передать как коллбек, например в таймер, и она не потеряет контекст

- Стрелочную функцию можно использовать как метод внутри классов, если такой метод передать как коллбек, например в таймер, то функция не потеряет контекст

```
// this
const myObject = {
  myMethod(items) {
    console.log(this);      // myObject
    const callback = () => {
      console.log(this);    // myObject
    };
  }
};

// constructor
const User = (name) => {
  this.name = name;
}
const admin = new User('Yauhen');   // => TypeError: User is not a constructor

// arguments
const myFunc = () => {
  console.log(arguments);
}
myFunc('a', 'b');   // => ReferenceError: arguments is not defined
```

```
// return
const double = (num) => num * 2;
console.log(double(2)); // 4

// context
class Hero {
  constructor(heroName) {
    this.heroName = heroName;
  }

  logName = () => {
    console.log(this.heroName);
  }
}
const batman = new Hero('Batman');
setTimeout(batman.logName, 1000);    // Batman
```

▼ Разница между методом `Object.freeze()` и `const` ?

Разные вещи, и const позволяет менять свойства в объекте, а freeze нет.

```
// const
const person = {
  name: "Yauhen",
};
person = 'test';    // Uncaught TypeError: Assignment to constant variable

// Object.freeze()
const person = {
  name: "Yauhen",
};
// freezing
Object.freeze(person);

person.name = "Jack";    // Uncaught TypeError: Cannot assign to read only property
console.log(person);     // { name: "Yauhen" }
```

▼ Что такое итераторы?

Подробнее тут

Итератор — это объект, который умеет обращаться к элементам коллекции по одному за раз, при этом отслеживая своё текущее положение внутри этой последовательности.

Иными словами итератор — это такой механизм, который позволяет перемещаться (итерироваться) по элементам коллекции в определённом порядке и делает их доступными.

Объекты, которые можно использовать в цикле `for..of`, называются *итерируемыми*.

- Технически итерируемые объекты должны иметь метод `Symbol.iterator`.
  - Результат вызова `obj[Symbol.iterator]` называется *итератором*. Он управляет процессом итерации.
  - Итератор должен иметь метод `next()`, который возвращает объект `{done: Boolean, value: any}`, где `done:true` сигнализирует об окончании процесса итерации, в противном случае `value` – следующее значение.
- Метод `Symbol.iterator` автоматически вызывается циклом `for..of`, но можно вызвать его и напрямую.
- Встроенные итерируемые объекты, такие как строки или массивы, также реализуют метод `Symbol.iterator`.
- Строковый итератор знает про суррогатные пары.

```
function makeIterator(array) {
  var nextIndex = 0;
  console.log("nextIndex =>", nextIndex);

  return {
    next: function() {
      return nextIndex < array.length
        ? { value: array[nextIndex++], done: false }
        : { done: true };
    }
  };
}

var it = makeIterator(["simple", "iterator"]);

console.log(it.next()); // {value: 'simple, done: false}
console.log(it.next()); // {value: 'iterator, done: false}
console.log(it.next()); // {done: true}
```

▼ Что такое генераторы? Когда стоит использовать генераторы?

Подробнее тут

Выполняясь они могут остановиться и выдать промежуточное значение, а затем продолжить выполнение. Чтобы превратить функцию в генератор необходимо добавить ей `*`.

Гнереатор отвечает только за возварт, управление им осуществляется извне.

Основным методом генератора является `next()`. При вызове он запускает выполнение кода до ближайшей инструкции `yield <значение>` (значение может отсутствовать, в этом случае оно предполагается равным `undefined`). По достижении `yield` выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода `next()` всегда является объект с двумя свойствами:

- `value`: значение из `yield`.
- `done`: `true`, если выполнение функции завершено, иначе `false`.

- Генераторы создаются при помощи функций-генераторов `function* f(…) {…}`.
- Внутри генераторов и только внутри них существует оператор `yield`.
- Внешний код и генератор обмениваются промежуточными результатами посредством вызовов `next/yield`.

В современном JavaScript генераторы используются редко. Но иногда они оказываются полезными, потому что способность функции обмениваться данными с вызывающим кодом во время выполнения

```
function* makeRangeIterator(start = 0, end = 4, step = 1) {
   let iterationCount = 0;
   for (let i = start; i < end; i += step) {
     iterationCount++;
     yield i;
   }
   return iterationCount;
 }

const iterator = makeRangeIterator();

iterator.next();     // { value: 0, done: false }
iterator.next();     // { value: 1, done: false }
iterator.next();     // { value: 2, done: false }
iterator.next();     // { value: 3, done: false }
iterator.next();     // { value: 4, done: true }
```

▼ Что такое ES6 модули?

Модули позволяют разделить код на несколько файлов, для удобства поддержки, облегчения читаемости, сохранения читсоты глобального пространства имен, и переиспользуемости кода.

```
// Export primitive
export const one = 1;
// Export function
export const isNull = (val) => val === null;
// Default export
class Helpers {
  static isUndefined(val) {
    return val === undefined;
  }
}
export default Helpers;
// Multiple export
let two = 2; let three = 3;
export { two, three };

// Import primitive
import { one } from './file.js';
// Import primitive with renaming
import { one as num1 } from './file.js';
// Import function
import { isNull } from './file.js';
// Default import
import Helpers from './file.js'
```

▼ Что такое символ (Symbol) в ES6?

Символ (Symbol) - это новый тип данных, представляющий уникальный и неизменяемый идентификатор. Каждый символ имеет уникальное значение и может быть использован в качестве ключа для свойств объектов. Символы создаются с помощью глобальной функции Symbol(), которая возвращает новый уникальный символ при каждом вызове. Символы могут использоваться для определения специальных свойств объектов, таких как итераторы, генераторы и т.д. Они также могут быть использованы для создания приватных свойств объектов, которые не могут быть доступны извне.

```
// Symbol creation
const symbol = Symbol();
console.log(typeof symbol); // "symbol"

// Creation symbol with name
const symbol1 = Symbol('mySymbol');
const symbol2 = Symbol('mySymbol');
console.log(symbol1 === symbol2); // false
```

▼ Для чего используется метод `.includes()`?

.includes(element, с какой позиции начинать) - есть у массивов и строк, с помощью него, можно проверить включает ли в себя строка или массив, какое-то значение.

```
1  const dead = ["Joffrey", "Ned Stark", "Night king"];
2
3  const isJonDead = dead.includes("Jon Snow");
4  console.log(isJonDead);      // false
5
6  const isJoffreyDead = dead.includes("Joffrey");
7  console.log(isJoffreyDead); // true
8
9  const text = "Test Lorem ipsum"
10 console.log(text.includes("Lorem"));      // true
11 console.log(text.includes("string"));     // false
12 console.log(text.includes("lorem"));     // false (register)
```

▼ Для чего используется метод `.getOwnPropertyDescriptors()`?

Возвращает все сведения для всех свойств объекта.

value, writable, enumerable configurable, также позволяет клонировать объекты клонируя в том числе геттеры и сеттеры

```
const person = {
  name: 'Max',
  age: 30,
  set personName(name) {
    this.name = name;
  },
  get password() {
    return `${this.name}${this.age}`;
  }
};

console.log(Object.getOwnPropertyDescriptors(person));
/*
age: {value: 30, writable: true, enumerable: true, configurable: true}
name: {value: "Max", writable: true, enumerable: true, configurable: true}
password: {get: ƒ, set: undefined, enumerable: true, configurable: true}
personName: {get: undefined, set: ƒ, enumerable: true, configurable: true}
*/

const admin = Object.defineProperties({}, Object.getOwnPropertyDescriptors(person));

console.log(Object.getOwnPropertyDescriptors(admin));
/*
age: {value: 30, writable: true, enumerable: true, configurable: true}
name: {value: "Max", writable: true, enumerable: true, configurable: true}
password: {get: ƒ, set: undefined, enumerable: true, configurable: true}
personName: {get: undefined, set: ƒ, enumerable: true, configurable: true}
*/
```

▼ Расскажите о методах `.keys()`, `.values()`, `.entries()`?

- Метод .**keys**() возвращает массив, содержащий все ключи объекта, за исключением символьных ключей.
- Метод .**values**() возвращает массив, содержащий все значения объекта, за исключением символьных ключей.
- Метод .**entries**() возвращает массив собственных перечисляемых свойств в формате [ключ, значение], содержащий все перечисляемые записи объекта, не включая символьные ключи и свойства из прототипа.

```js
const user = {
  firstName: "Yauhen",
  lastName: "Kavalchuk",
};
Object.keys(user); // ['firstName', 'lastName']
Object.values(user); // ['Yauhen', 'Kavalchuk']
Object.entries(user); // [['firstName', 'Yauhen'], ['lastName', 'Kavalchuk']]

const name = ['M', 'a', 'x'];
Object.entries(name); // [ [ '0', 'M' ], [ '1', 'a' ], [ '2', 'x' ] ];

const admin = {
  [Symbol('password')]: '123pass',
  name: 'Yauhen',
};
Object.entries(admin); // [ [ 'name', 'Yauhen' ] ];
```

▼ Для чего используется метод `.fromEntries()`?

Object.fromEntries() - может создать объект из массива массивов с парами [ключ, значение]

```js
const user = {
  firstName: "Yauhen",
  lastName: "Kavalchuk",
};
Object.entries(user); // [['firstName', 'Yauhen'], ['lastName', 'Kavalchuk']]

conat arr1 = [['firstName', 'Yauhen'], ['lastName', 'Kavalchuk']];
Object.fromEntries(arr1); // { firstName: "Yauhen", lastName: "Kavalchuk" }

const arr2 = [['one', 1], ['two', 2], ['three', 3]];
Object.fromEntries(arr2); // { one: 1, two: 2, three: 3 }
```

▼ Для чего используются методы `.flat()` и `.flatMap()`?

Метод `flat()` возвращает новый массив и уменьшает вложенность массива на заданное количество уровней.

Метод принимает необязательный аргумент `depth` — количество уровней, на которые нужно уменьшить вложенность. Значение по умолчанию — 1.

Если вложенность неизвестна, но нужно получить из массива с вложенными элементами плоский массив, то передайте аргумент `Infinity`. Тогда метод рекурсивно обойдёт массив и сделает на его основе новый плоский.

Результатом вызова метода `flat()` будет новый массив меньшей вложенности.

Метод `flatMap()` позволяет сформировать массив, применяя функцию к каждому элементу, затем уменьшает вложенность, делая этот массив плоским, и возвращает его.

Метод идентичен последовательному вызову `map().flat()` с параметром `depth = 1`, соответственно, применяется для трансформации исходных данных, с уменьшением вложенности.

Данный метод более эффективный, чем вызов этих функций по отдельности, поскольку обход массива совершается только один раз. Но глубина уменьшается всего на один уровень.

Метод flat() делает входной массив плоским, глубина указывается в аргументах.

Метод flatMap() применяет функцию, трансформируя массив и делает массив плоским. Два в одном.

```
const arr1 = [1, 2, [3, 4]];
arr1.flat(); // [1, 2, 3, 4]

const arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat(); // [1, 2, 3, 4, [5, 6]]

const arr3 = [1, 2, [3, 4, [5, 6]]];
arr3.flat(2); // [1, 2, 3, 4, 5, 6]

const arr4 = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]];
arr4.flat(Infinity); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

const arr5 = [1, 2, , 4, 5];
arr5.flat(); // [1, 2, 4, 5]

const arr6 = [[1], [2], [3], [4]];
arr7.flatMap(x => [x * 2]); // [2, 4, 6, 8]
```

▼ Для чего используются методы `.padStart()` и `.padEnd()`?

.padStart(length, чем заполнять) - дополняет строку с начала до нужного размера.

.padEnd(length, чем заполнять) - дополняет строку с конца до нужного размера.

По умолчанию заполняет пробелами.

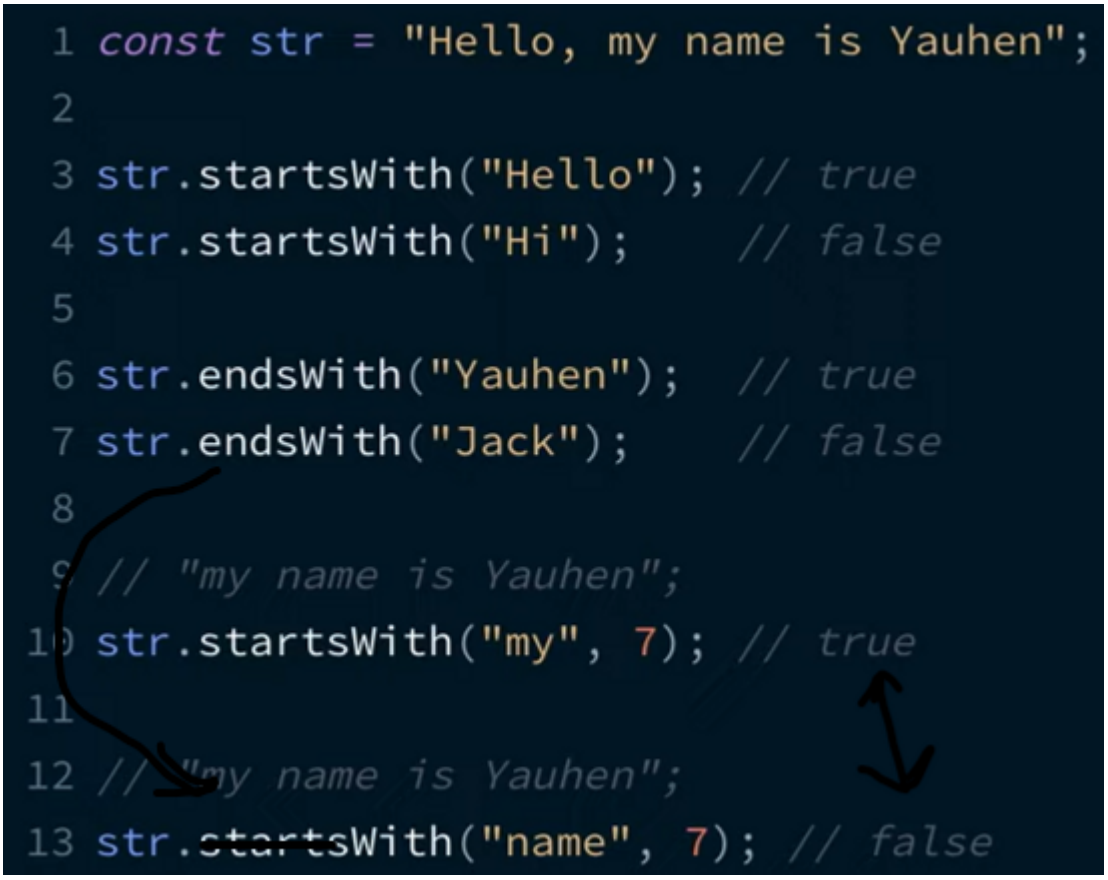
Если исходная строка длиннее указанного числа, то строка останется неизменной.

```
const str = "test";

str.padStart(10, '~'); // '~~~~~~test'
str.padEnd(10, '~');   // 'test~~~~~~'

str.padStart(10);      // '      test'
str.padEnd(10);        // 'test      '
```

▼ Для чего используются методы `.startsWith()` и `.endsWith()`?

Это методы строк, которые как называются, за то и отвечают.

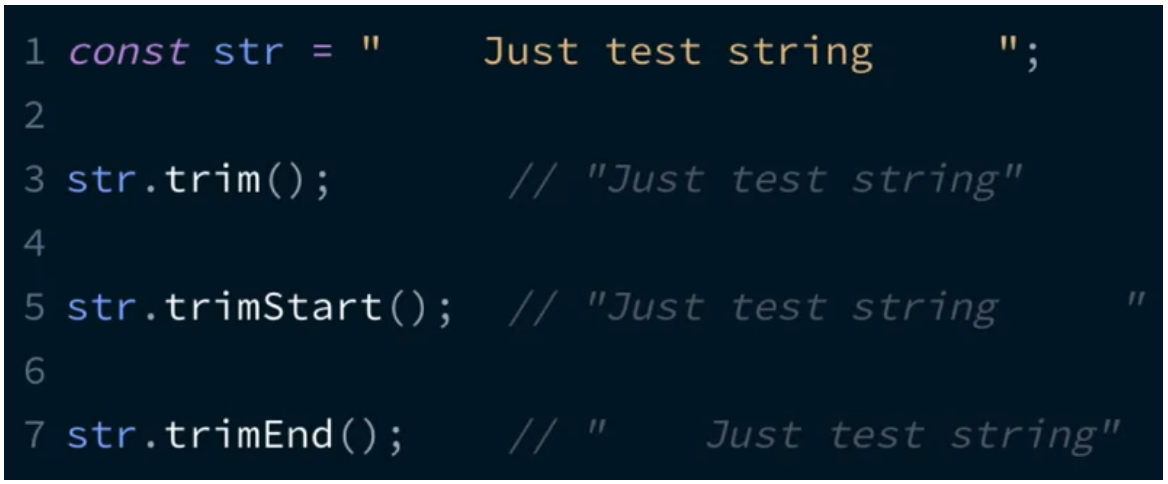
Необязательный параметр. Позиция в строке, с которой начинать поиск `searchString`; по умолчанию 0. Либо начинает поиск с заданного индекса, либо обрезает строку до заданного индекса



```
const str = "Hello, my name is Yauhen";

str.startsWith("Hello"); // true
str.startsWith("Hi");    // false

str.endsWith("Yauhen");  // true
str.endsWith("Jack");    // false

// "my name is Yauhen";
str.startsWith("my", 7); // true

// "my name is Yauhen";
str.startsWith("name", 7); // false
```

▼ Как в JavaScript удалять пробельные символы в начале и в конце строки?

- .trim()
- .trimStart()
- .trimEnd()

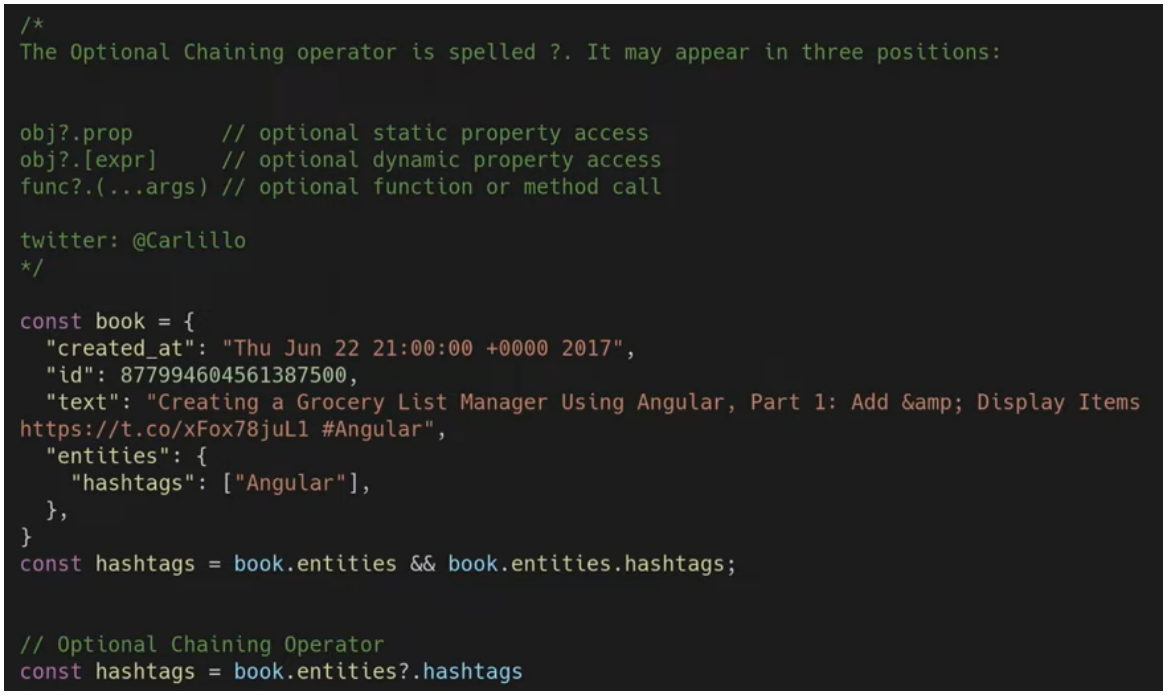


```
const str = "    Just test string     ";

str.trim();      // "Just test string"

str.trimStart(); // "Just test string     "

str.trimEnd();   // "    Just test string"
```

▼ Расскажите об операторе Optional Chaining (`?.`)?

При обращении к свойству объекта, которое лежит очень глубоко, зачастую приходится добавлять проверки, на существование промежуточных узлов.

Оператор optional chaining позволяет получить значение свойства находящегося на любом уровне вложенности, без необходимости проверять каждое из промежуточных свойств на существование. Не выбрасывает исключение, а просто возвращает undefined.



```
/*
The Optional Chaining operator is spelled ?. It may appear in three positions:


obj?.prop       // optional static property access
obj?.[expr]     // optional dynamic property access
func?.(...args) // optional function or method call

twitter: @Carlillo
*/

const book = {
  "created_at": "Thu Jun 22 21:00:00 +0000 2017",
  "id": 877994604561387500,
  "text": "Creating a Grocery List Manager Using Angular, Part 1: Add &amp;amp; Display Items
https://t.co/xFox78juL1 #Angular",
  "entities": {
    "hashtags": ["Angular"],
  },
}
const hashtags = book.entities && book.entities.hashtags;


// Optional Chaining Operator
const hashtags = book.entities?.hashtags
```

▼ Для чего используется метод `.replaceAll()`?

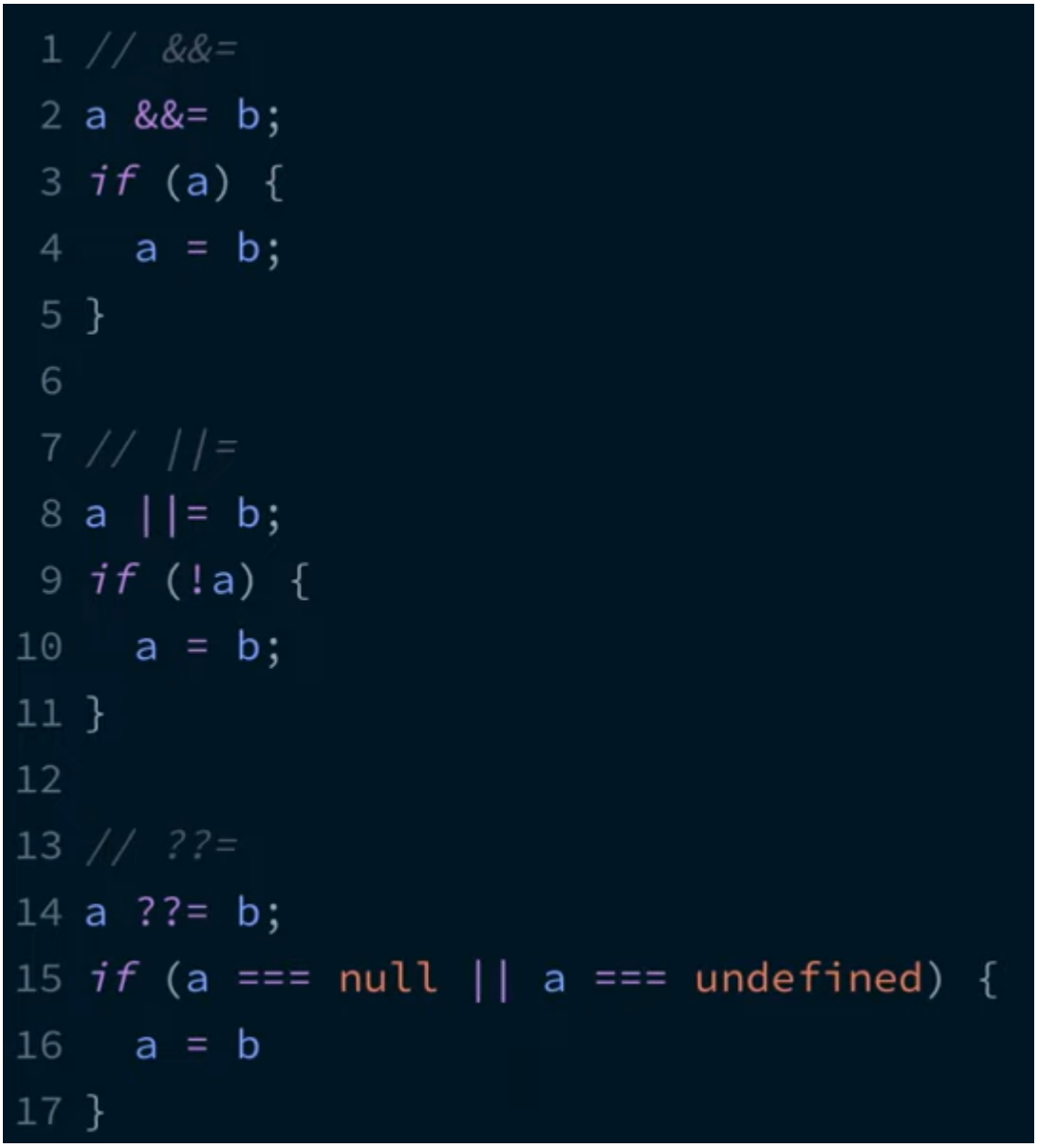
.replaceAll() - заменяет **все** совпадения строки на другие строки в отличии от простого .replace который меняет только первое вхождение.

Также может принимать регулярное выражение.

```js
const str = "Backbencher sits at the Back";

// .replace()
const newStr1 = str.replace("Back", "Front");
console.log(newStr1); // "Frontbencher sits at the Back"

// .replace() with RexExp
const newStr2 = str.replace(/Back/g, "Front");
console.log(newStr2); // "Frontbencher sits at the Front"

// .replaceAll()
const newStr3 = str.replaceAll("Back", "Front");
console.log(newStr3); // "Frontbencher sits at the Front"
```

▼ Что такое оператор логического присваивания?

Оператор логического присваивания объединяет логические операции && || и оператор нулевого слияния с присваиванием.



```
// &&=
a &&= b;
if (a) {
   a = b;
}

// ||=
a ||= b;
if (!a) {
   a = b;
}

// ??=
a ??= b;
if (a === null || a === undefined) {
   a = b
}
```

```
//a &&= b
if (a) {
    a = b
}

//a ||= b
if (!a) {
    a = b
}

//a ??= b
if (a === null || a === undefined) {
```

```
    a = b
}
```

▼ Как увеличить читаемость больших чисел?

Появилось в EcmaScript 2021, можно делить числа по разрядам нижним подчеркиванием.

let number = 1_000_000_000 - будет числом 1000000000

```
let number = 1_000_000_000;
console.log(number); // 1000000000
```

▼ Что такое приватные аксессоры?

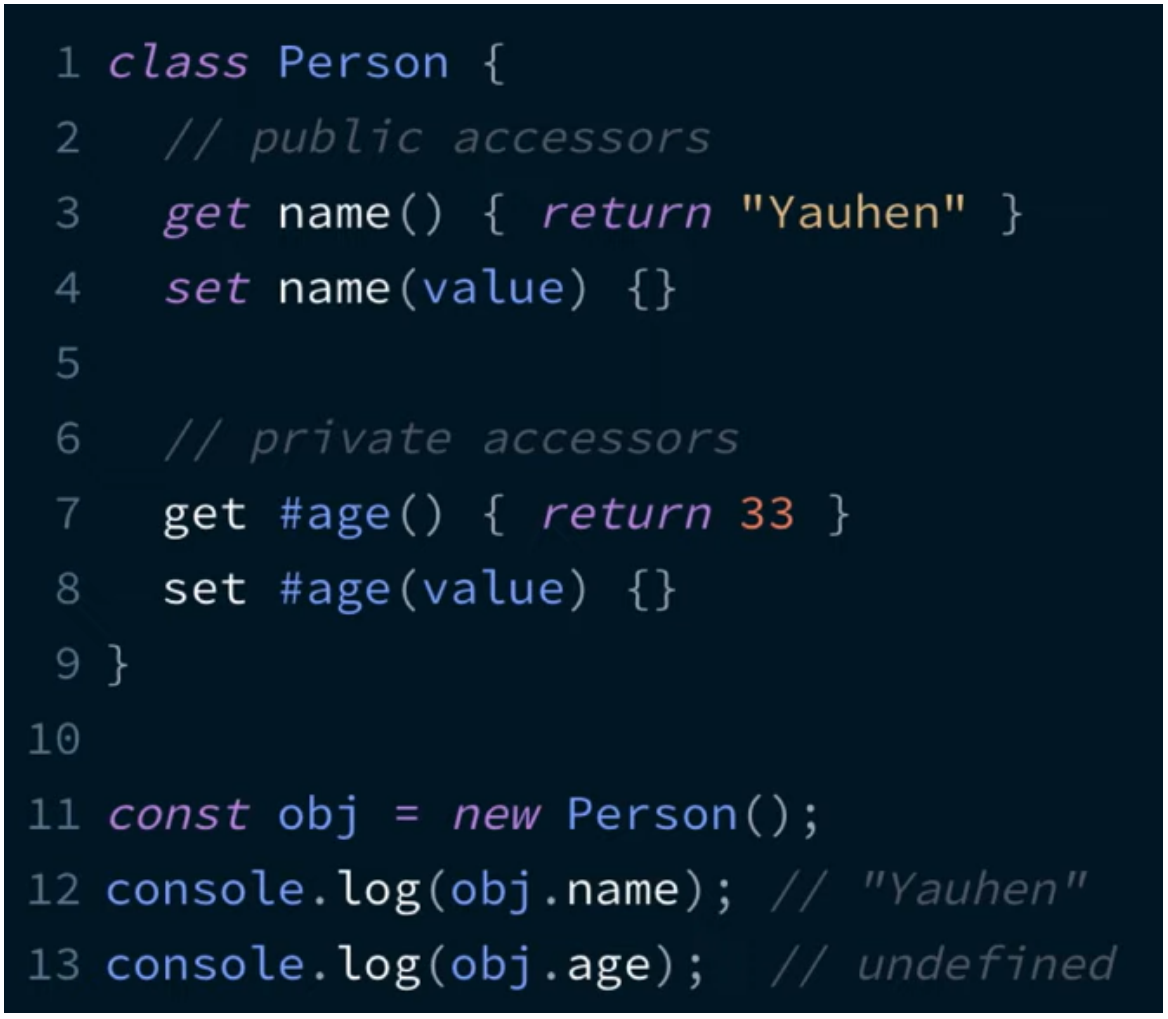
Аксессоры - это операторы, которые предоставляют доступ к свойству объекта.

В JavaScript это методы get и set, get для получения свойства, set для его установки.

Аксессоры нужны чтобы сделать свойство класса более приватными.

В EcmaScript 2021, появилась возможность делать аксессоры приватными с помощью знака `#` перед именем.

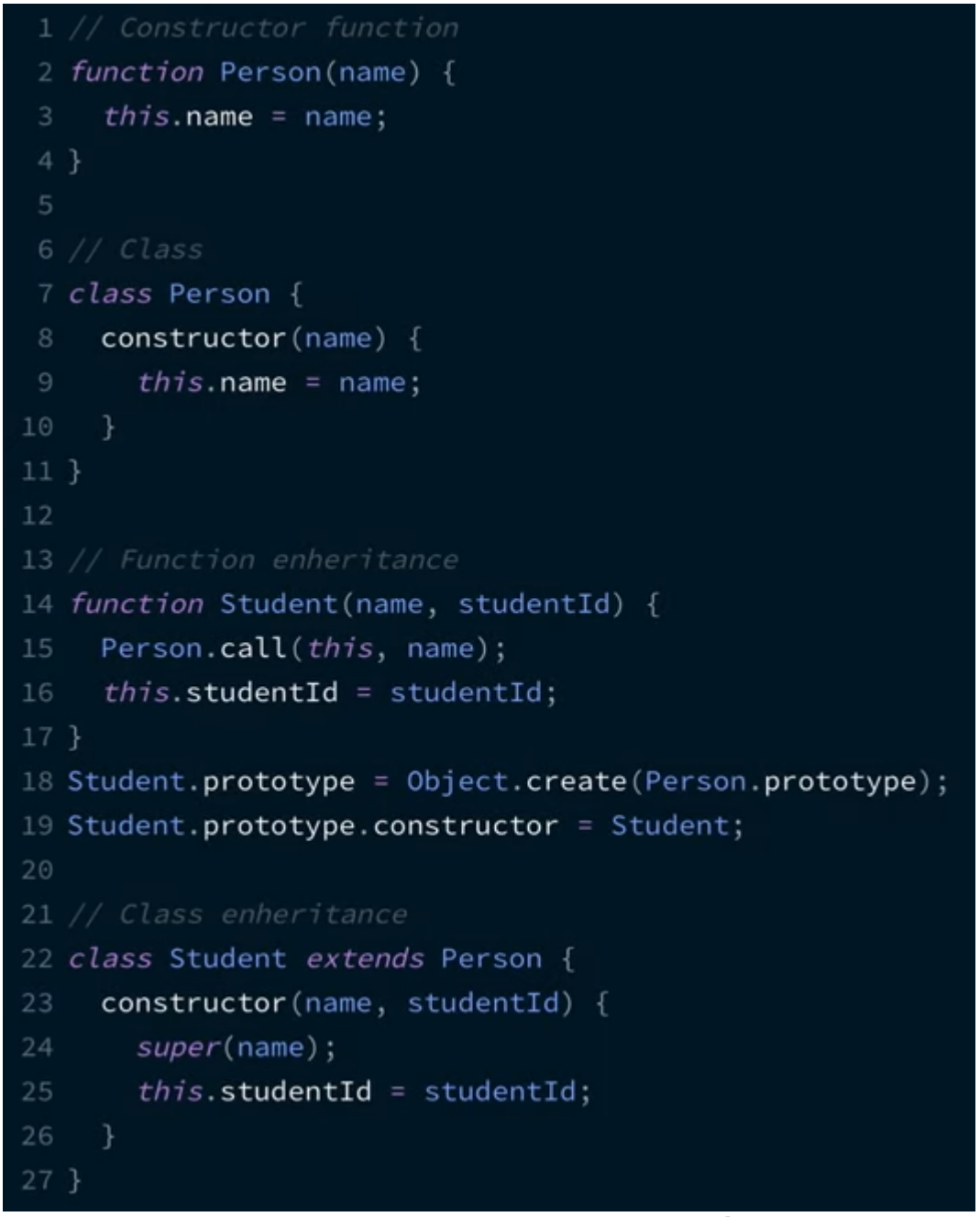
Также приватными можно сделать любые свойства и методы класса.



```
class Person {
  // public accessors
  get name() { return "Yauhen" }
  set name(value) {}

  // private accessors
  get #age() { return 33 }
  set #age(value) {}
}

const obj = new Person();
console.log(obj.name); // "Yauhen"
console.log(obj.age);  // undefined
```

▼ Разница между ES6-классами и конструкторами функций?

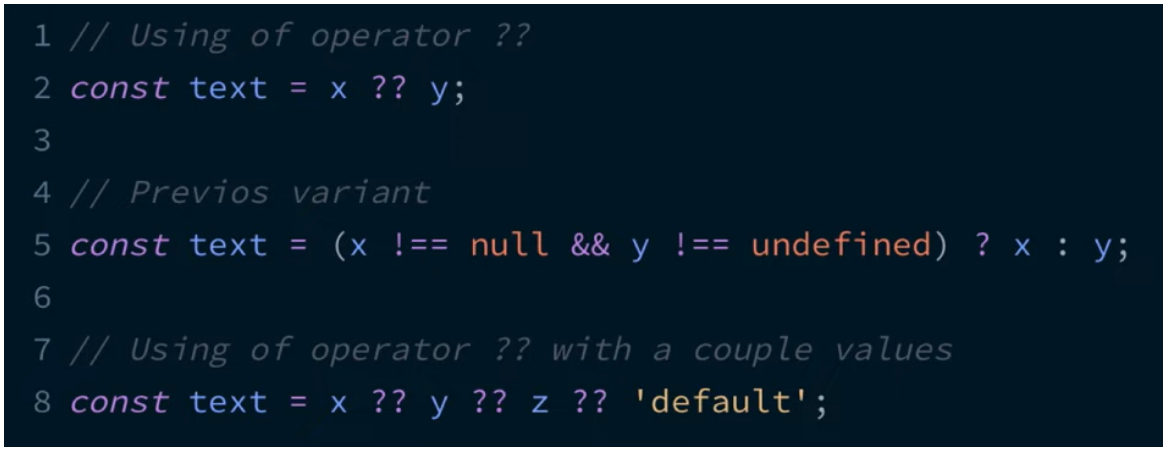
- **Синтаксис**: ES6-классы используют ключевое слово `class`, а конструкторы функций - ключевое слово `function`.
- **Наследование**: в ES6-классах наследование осуществляется с помощью ключевого слова `extends`, а в конструкторах функций - с помощью прототипного наследования.
- **Методы**: методы в ES6-классах объявляются как функции внутри класса, а в конструкторах функций - как свойства прототипа.
- **Конструктор**: в ES6-классах конструктор объявляется с помощью метода `constructor`, а в конструкторах функций - сама функция.



```
// Constructor function
function Person(name) {
  this.name = name;
}

// Class
class Person {
  constructor(name) {
    this.name = name;
  }
}

// Function enheritance
function Student(name, studentId) {
  Person.call(this, name);
  this.studentId = studentId;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

// Class enheritance
class Student extends Person {
  constructor(name, studentId) {
    super(name);
    this.studentId = studentId;
  }
}
```

▼ Что такое оператор нулевого слияния (`??`)?

Оператор нулевого слияния (??) - это новый оператор, добавленный в ECMAScript 2020, который возвращает первый операнд, если он не равен null или undefined, иначе возвращает второй операнд.

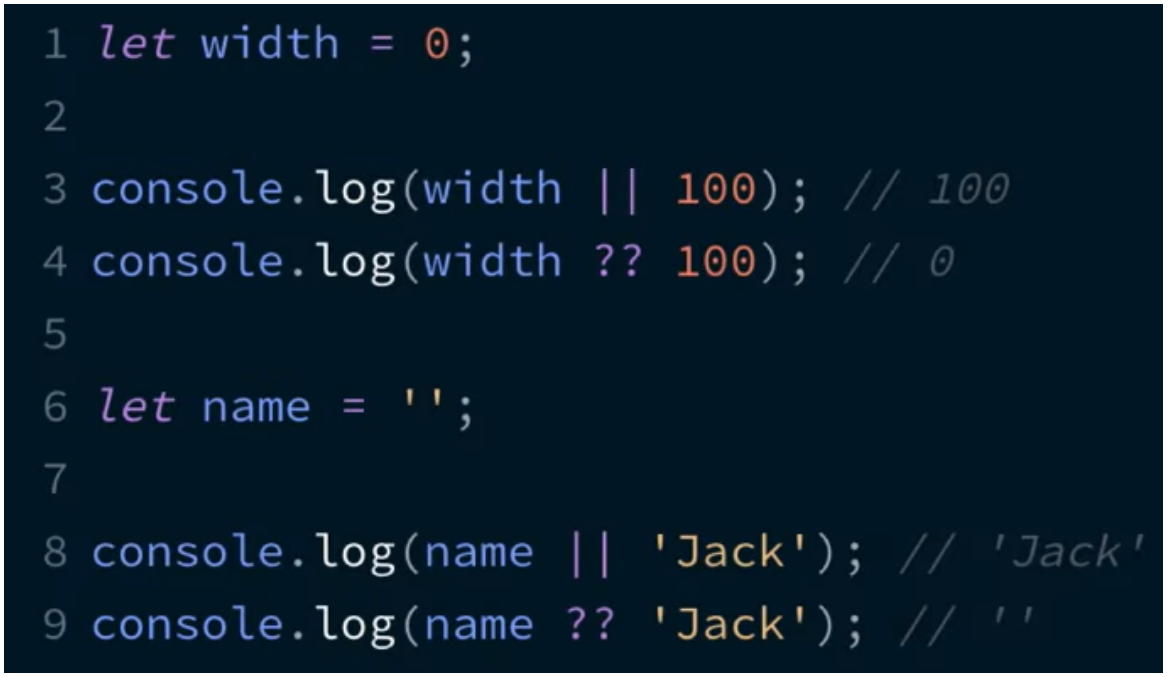


```
// Using of operator ??
const text = x ?? y;

// Previos variant
const text = (x !== null && y !== undefined) ? x : y;

// Using of operator ?? with a couple values
const text = x ?? y ?? z ?? 'default';
```

```
let a = null;
let b = 10;
let c = a ?? b; // c будет равно 10, так как a равно null
```

▼ В чём отличие оператора нулевого слияния (??) и оператора “ИЛИ” (||)?

Оператор || возвращает первый операнд, если он приводится к true, иначе возвращает второй операнд. Оператор ?? возвращает первый операнд, если он не равен null или undefined, иначе возвращает второй операнд. Таким образом, оператор || может вернуть ложное значение, если первый операнд равен false, 0 или пустой строке, тогда как оператор ?? не будет возвращать ложное значение в таких случаях.



```
let width = 0;

console.log(width || 100); // 100
console.log(width ?? 100); // 0

let name = '';

console.log(name || 'Jack'); // 'Jack'
console.log(name ?? 'Jack'); // ''
```

▼ Назовите основные методы и свойства работы с коллекцией `Map`?

- **set**(key, value) - добавляет новую пару ключ-значение в коллекцию.
- **get**(key) - возвращает значение, связанное с указанным ключом.
- **has**(key) - проверяет, существует ли указанный ключ в коллекции.
- **delete**(key) - удаляет пару ключ-значение из коллекции по указанному ключу.
- **clear**() - удаляет все пары ключ-значение из коллекции.
- **size** свойство, которое возвращает количество пар ключ-значение в коллекции.

▼ Назовите основные методы и свойства работы с коллекцией `Set`?

- **add**(value) - добавляет новый элемент в коллекцию.
- **delete**(value) - удаляет указанный элемент из коллекции.
- **has**(value) - проверяет, существует ли указанный элемент в коллекции.
- **clear**() - удаляет все элементы из коллекции.
- **size** свойство, которое возвращает количество элементов в коллекции.

▼ Как осуществить перебор элементов в коллекциях `Map` и `Set`?

Подробнее тут

- .keys(), для Set возвращает значения
- .values()
- .entries(), используется по умолчанию в for of, для Set возвращает пару [значение. значение]
- .forEach()
- for...of

▼ Proxy и Reflect

Подробнее тут

Прокси – это обёртка вокруг объекта, которая «по умолчанию» перенаправляет операции над ней на объект, но имеет возможность перехватывать их.

Проксировать можно любой объект, включая классы и функции.

Синтаксис:

```
let proxy = new Proxy(target, {/* ловушки */});
```

...Затем обычно используют прокси везде вместо оригинального объекта `target`. Прокси не имеет собственных свойств или методов. Он просто перехватывает операцию, если имеется соответствующая ловушка, а иначе перенаправляет её сразу на объект `target`.

Мы можем перехватывать:

- Чтение (`get`), запись (`set`), удаление (`deleteProperty`) свойства (даже несуществующего).
- Вызов функции (`apply`).
- Оператор `new` (ловушка `construct`).
- И многие другие операции (полный список приведён в начале статьи, а также в документации).

Это позволяет нам создавать «виртуальные» свойства и методы, реализовывать значения по умолчанию, наблюдаемые объекты, функции-декораторы и многое другое.

Мы также можем оборачивать один и тот же объект много раз в разные прокси, добавляя ему различные аспекты функциональности.

Reflect API создано как дополнение к Proxy. Для любой ловушки из `Proxy` существует метод в `Reflect` с теми же аргументами. Нам следует использовать его, если нужно перенаправить вызов на оригинальный объект.

Прокси имеют некоторые ограничения:

- Встроенные объекты используют так называемые «внутренние слоты», доступ к которым нельзя проксировать. Однако, ранее в этой главе был показан один способ, как обойти это ограничение.

- То же самое можно сказать и о приватных полях классов, так как они реализованы на основе слотов. То есть вызовы проксированных методов должны иметь оригинальный объект в качестве `this`, чтобы получить к ним доступ.
- Проверка объектов на строгое равенство `===` не может быть перехвачена.
- Производительность: конкретные показатели зависят от интерпретатора, но в целом получение свойства с помощью простейшего прокси занимает в несколько раз больше времени. В реальности это имеет значение только для некоторых «особо нагруженных» объектов.

TypeScript

▼ Что такое TypeScript?

TypeScript - это язык программирования, который расширяет возможности JavaScript, добавляя в него статическую типизацию, классы, интерфейсы и другие возможности.

▼ Основные компоненты TypeScript?

1. Статическая типизация - возможность указывать типы данных для переменных, функций и объектов.
2. Классы - возможность определять классы и использовать наследование.
3. Интерфейсы - возможность определять интерфейсы и использовать их для описания структуры объектов.
4. Дженерики - возможность создания обобщенных типов данных и функций.
5. Декораторы - возможность добавления дополнительной функциональности к классам и функциям.

▼ Назовите особенности TypeScript?

1. TypeScript является строго типизированным языком программирования, что позволяет выявлять ошибки на этапе компиляции.
2. TypeScript поддерживает функциональное программирование, что позволяет писать более чистый и модульный код.
3. TypeScript поддерживает ES6 и более новые версии JavaScript, что позволяет использовать новые возможности языка.
4. TypeScript имеет широкую поддержку в инструментах разработки, таких как Visual Studio Code, WebStorm и других.
5. TypeScript может быть использован для разработки приложений на любой платформе, включая браузеры, сервера и мобильные

устройства.

6. TypeScript обеспечивает лучшую читаемость и поддерживаемость кода благодаря его строгой типизации и возможности описывать интерфейсы.
7. TypeScript позволяет создавать более безопасный код благодаря проверкам типов на этапе компиляции.

▼ Плюсы использования TypeScript?

1. Уменьшение количества ошибок на этапе компиляции благодаря строгой типизации.
2. Возможность использования функционального программирования для более чистого и модульного кода.
3. Поддержка новых возможностей языка JavaScript.
4. Широкая поддержка в инструментах разработки.
5. Возможность использования на любой платформе.
6. Улучшение читаемости и поддерживаемости кода благодаря строгой типизации и возможности описывать интерфейсы.
7. Создание более безопасного кода благодаря проверкам типов на этапе компиляции.

▼ Минусы использования TypeScript?

1. Дополнительное время на изучение языка и его особенностей.
2. Необходимость компиляции кода перед запуском, что может замедлить процесс разработки.
3. Ограничения в использовании некоторых функций JavaScript.
4. Необходимость использования дополнительных инструментов для работы с TypeScript (например, компилятор).
5. Не все библиотеки и фреймворки поддерживают TypeScript.

▼ Типы в TypeScript?

- boolean (логический тип)
- number (числовой тип)
- string (строковый тип)

- array (массив)
- tuple (кортеж)
- enum (перечисление)
- any (любой тип)
- void (отсутствие значения)
- null и undefined
- never (тип, который не имеет значений)
- object (не примитивный тип данных)

▼ Что такое декораторы?

Декораторы - это функции, которые используются для изменения поведения классов, методов, свойств и параметров во время выполнения. Они позволяют добавлять дополнительную функциональность к существующему коду без его изменения.

▼ Поддерживает ли TypeScript перегрузку функций?

Да, TypeScript поддерживает перегрузку функций. Это означает, что вы можете определить несколько версий функции с разными параметрами и типами возвращаемого значения, и TypeScript будет выбирать правильную версию функции на основе переданных аргументов.

▼ Разница между типом (`type`) и интерфейсом (`interface`)?

1. Интерфейсы могут быть расширены и реализованы, а типы - нет.
2. Интерфейсы могут наследовать свойства и методы других интерфейсов, а типы - нет.
3. Типы могут быть использованы только для определения формы объектов, а интерфейсы могут быть использованы для определения формы объектов, классов и функций.
4. Интерфейсы могут иметь перегруженные методы, а типы - нет.
5. Использование интерфейсов более распространено в сообществе TypeScript, чем использование типов.

▼ Что такое JSX в TypeScript? Какие режимы JSX поддерживает TypeScript?

JSX - это синтаксис расширения языка JavaScript, который позволяет использовать XML-подобный синтаксис для создания элементов пользовательского интерфейса. В TypeScript JSX используется для описания типов React-компонентов.

▼ Что такое директивы с тремя наклонными чертами (Triple-Slash Directives), их типы?

Директивы с тремя наклонными чертами (Triple-Slash Directives) - это специальные комментарии в начале файла TypeScript, которые используются для указания определенных настроек компиляции или подключения дополнительных файлов.

1. /// <reference path="..."/> - используется для указания пути к файлу, который должен быть включен в компиляцию. Эта директива может быть использована, если файлы TypeScript разбиты на несколько модулей и один модуль зависит от другого.
2. /// <reference types="..."/> - используется для указания типовых определений, которые должны быть использованы в файле. Эта директива может быть использована, если файлы TypeScript используют сторонние библиотеки, для которых уже существуют типовые определения.
3. /// <reference lib="..."/> - используется для указания библиотеки, которая должна быть включена в компиляцию. Эта директива может быть использована, если файлы TypeScript используют функциональность, которая не является частью стандартной библиотеки JavaScript.
4. /// <amd-module name="..."/> - используется для указания имени модуля AMD, который будет создан при компиляции файла.
5. /// <amd-dependency path="..." /> - используется для указания зависимостей модуля AMD, которые должны быть загружены перед загрузкой текущего модуля.
6. /// <jsx .../> - используется для указания режима JSX-компиляции (preserve или react).
7. /// <tsconfig .../> - используется для указания пути к файлу конфигурации TypeScript (tsconfig.json), который должен быть использован при компиляции файла.

▼ Что такое внешние объявления переменных (ambient declaration) в TypeScript?

Внешние объявления переменных (ambient declaration) в TypeScript - это способ объявления типов и интерфейсов для сторонних библиотек или модулей, которые не имеют типовых определений в TypeScript.

declare var $: any;

Здесь мы объявляем переменную $ как внешнюю, используя ключевое слово declare. Поскольку мы не знаем тип переменной $, мы указываем тип any.

▼ Разница между абстрактным классом (abstract class) и интерфейсом (interface)?

Абстрактный класс - это класс, который не может быть создан напрямую, а может только наследоваться другими классами. Он может содержать абстрактные методы, которые не имеют реализации в самом классе, но должны быть реализованы в подклассах. Абстрактные классы могут также иметь обычные методы и свойства.

Интерфейс - это набор абстрактных методов и свойств, которые должны быть реализованы классами, которые реализуют этот интерфейс. Интерфейсы не могут содержать реализацию методов или свойств, они описывают только сигнатуру методов и свойств.

Основная разница между абстрактным классом и интерфейсом заключается в том, что абстрактный класс может иметь реализацию методов и свойств, а также может содержать общую функциональность для всех подклассов, тогда как интерфейс описывает только сигнатуру методов и свойств и не может содержать реализацию. Также класс может наследовать только один абстрактный класс, но может реализовывать несколько интерфейсов.

▼ Какие элементы ООП поддерживаются в TypeScript?

В TypeScript поддерживаются все основные элементы ООП, такие как классы, объекты, наследование, полиморфизм, инкапсуляция, абстрактные классы и интерфейсы. TypeScript также поддерживает модификаторы доступа (public, private, protected), статические свойства и методы, конструкторы и деструкторы. Кроме того, TypeScript позволяет использовать дженерики для создания универсальных классов и функций.

▼ Модификаторы доступа в TypeScript?

1. **public** свойство или метод доступны из любого места в коде, включая внешний код и наследующие классы.
2. **private** свойство или метод доступны только внутри класса, в котором они определены.
3. **protected** свойство или метод доступны только внутри класса, а также в классах, которые наследуют этот класс.
4. **readonly** только для чтения.

▼ Разница между внутренним (Internal Module) и внешним модулями (External Module)?

В TypeScript внутренний модуль (Internal Module) - это пространство имен (namespace), которое может содержать классы, интерфейсы, функции и другие объекты. Они могут быть использованы только внутри файла, где они определены.

Внешний модуль (External Module) - это файл, который экспортирует свои объекты для использования в других файлах. Они могут быть загружены с помощью системы модулей.

▼ Что такое декораторы в TypeScript?

Декораторы в TypeScript - это специальные функции, которые могут быть применены к классам, методам, свойствам и параметрам. Они используются для добавления дополнительной функциональности к объектам во время выполнения.

▼ Как TypeScript поддерживает необязательные и дефолтные параметры в функции?

TypeScript поддерживает необязательные и дефолтные параметры в функции.

Необязательные - ?

Дефолтные - =

▼ Что такое перечисление (`enum`)?

Перечисление (enum) в TypeScript - это набор именованных констант, которые могут быть использованы в коде. Они представляют собой

удобный способ определения ограниченного набора значений, которые могут использоваться в приложении.

▼ Для чего в TypeScript используется `NoImplicitAny`?

NoImplicitAny - это параметр компилятора TypeScript, который указывает, что все типы должны быть явно указаны в коде.

▼ Разница между типами “Объединение” (`|`) и “Пересечение” (`&`)?

Тип "Объединение" (|) используется для объединения двух или более типов. Если значение имеет хотя бы один из этих типов, то оно будет соответствовать объединенному типу. Например, тип "string | number" означает, что значение может быть либо строкой, либо числом.

Тип "Пересечение" (&) используется для определения типа, который соответствует значениям, которые имеют все перечисленные типы. Например, тип "string & number" означает, что значение должно быть одновременно строкой и числом.

▼ Что такое общие типы (`generic`) в TypeScript?

Общие типы (generic) в TypeScript позволяют создавать функции, классы и интерфейсы, которые могут работать с различными типами данных без необходимости явного указания конкретных типов.

▼ Какие области видимости доступны в TypeScript?

Такие же как в JS.

▼ Что такое `.map` файл, как и зачем его использовать?

.map файл (source map) - это файл, который содержит информацию о соответствии исходного кода и скомпилированного кода. Он используется для отладки исходного кода в инструментах разработки, которые работают с скомпилированным кодом, например, в браузерных инструментах разработчика.

▼ Можно ли использовать TypeScript в серверной разработке?

Да, TypeScript может быть использован в серверной разработке. TypeScript может быть использован для написания серверных приложений на Node.js, а также для создания API и веб-сервисов. TypeScript обеспечивает статическую типизацию, что может улучшить безопасность и надежность серверного приложения.

▼ Для чего в TypeScript используют ключевое слово `declare`?

Ключевое слово declare используется в TypeScript для объявления типов, которые не имеют своей реализации в текущем файле, но будут использоваться в других файлах или библиотеках.

▼ Разница между типами `void`, `never` и `unknown`?

Тип void используется для функций, которые не возвращают значения. Тип never используется для функций, которые никогда не завершаются (например, функция, которая всегда выбрасывает исключение). Тип unknown используется для значений, о типе которых мы не знаем на момент компиляции. Он похож на тип any, но более безопасен, так как TypeScript не позволяет присваивать значения типа unknown переменным других типов без явного приведения типов.

▼ Как вы отлавливаете ошибки в TypeScript коде?

1. Компиляция TypeScript кода с помощью TypeScript компилятора (tsc). Компилятор выдаст ошибки и предупреждения о типах и других проблемах в коде.
2. Использование интегрированных сред разработки (IDE) с поддержкой TypeScript, таких как Visual Studio Code, WebStorm, Atom и другие. Они обычно выделяют ошибки в коде, подсвечивая их красным цветом.
3. Использование сторонних инструментов для статического анализа кода, таких как TSLint или ESLint с плагинами для TypeScript. Они также могут выделять ошибки в коде и предлагать исправления.
4. Запуск тестовых сценариев для проверки правильности работы программы. Тесты могут помочь выявить ошибки, которые не были обнаружены при компиляции или статическом анализе кода.

Привет, вот и настал момент для этой открытки :) Очень надеюсь, что ты сейчас счастлива.
Начало 2024 года было конечно тяжелым. Нам с тобой предстояло пройти еще много препятствий и испытаний , но я уверен, что мы справились. Знаю, в какие то моменты тебе было очень трудно и может даже невыносимо, знаю что ты винила себя и возможно жалела о своих поступках, но то как ты все это перенесла я будут помнить всегда.
Но вот прошел год и уже все позади, мы вместе, и я безумно счастлив

быть с тобой. Думаю, теперь все твои сомнения уже точно развеяны и ты понимаешь на все сто процентов как я тебя люблю.

Привет, вот и настал момент для этой открытки :)

Начало 2024 года было, конечно, тяжёлым. Нам с тобой предстояло пройти ещё много препятствий и испытаний, но я уверен, что мы справились. Знаю, в какие-то моменты тебе было очень трудно и, может быть, даже невыносимо. Знаю, что ты винила себя и, возможно, жалела о своих поступках, но то, как ты всё это перенесла, я буду помнить всегда. Знай, что как бы тебе трудно не было, я буду всегда готов тебе помочь.

Но вот прошёл год, и уже всё позади. Мы вместе, и я безумно счастлив быть с тобой. Думаю, теперь все твои сомнения уже точно развеяны, и ты понимаешь на все сто процентов, как я тебя люблю. Я до сих пор не могу поверить, что смог встретить любовь всей моей жизни.

Впереди нас ждёт ещё множество прекрасных моментов, и я с нетерпением жду, чтобы их разделить с тобой. А любые трудности мы теперь уж точно сможем преодалеть.

Моей любимой.

Привет, вот и настал момент для этой открытки :)

Начало 2024 года было, безусловно, тяжёлым. Нам с тобой предстояло пройти ещё множество препятствий и испытаний, но я уверен, что мы справились. Знаю, что в какие-то моменты тебе было очень трудно. Наверное, даже иногда, невыносимо. Знаю, что ты винила себя и, возможно, жалела о своих поступках. Но то, как ты всё это перенесла, я буду помнить всегда. Хочу, чтобы ты знала: как бы трудно тебе ни было, я всегда буду рядом и готов помочь.

И вот, прошёл год, и всё позади. Мы вместе, и я безумно счастлив быть рядом с тобой. Уверен, что теперь все твои сомнения развеяны, и ты на все сто процентов понимаешь, как сильно я тебя люблю. Я до сих пор не могу поверить, что мне посчастливилось встретить любовь всей моей жизни.

Впереди нас ждёт множество прекрасных моментов, которые я с нетерпением жду разделить с тобой. А любые трудности мы теперь точно сможем вместе преодолеть.

Моей любимой

React

▼ Что такое React?

React - это JavaScript библиотека для создания пользовательских интерфейсов.

Фреймворк это несколько библиотек и определенная заложенная архитектура.

Реакт инструмент для построения интерфейсов.

Основная философия это компонентный подход.

Отвечает за view в модели MVP.

▼ Перечислите особенности React?

1. Использование виртуального DOM для оптимизации производительности.
2. Поддержка SSR через NEXT.js когда разметка подготавливатеся на сервере и в готовом виде передается на клиент.
3. Возможность использования JSX для более удобного написания кода.

▼ Что Такое `JSX`?

JSX - javascript xml, это расширение синтаксиса JavaScript, которое позволяет использовать HTML-подобный синтаксис для создания элементов интерфейса в React. JSX позволяет объединить JavaScript и HTML в одном файле, что упрощает создание компонентов и улучшает читаемость кода.

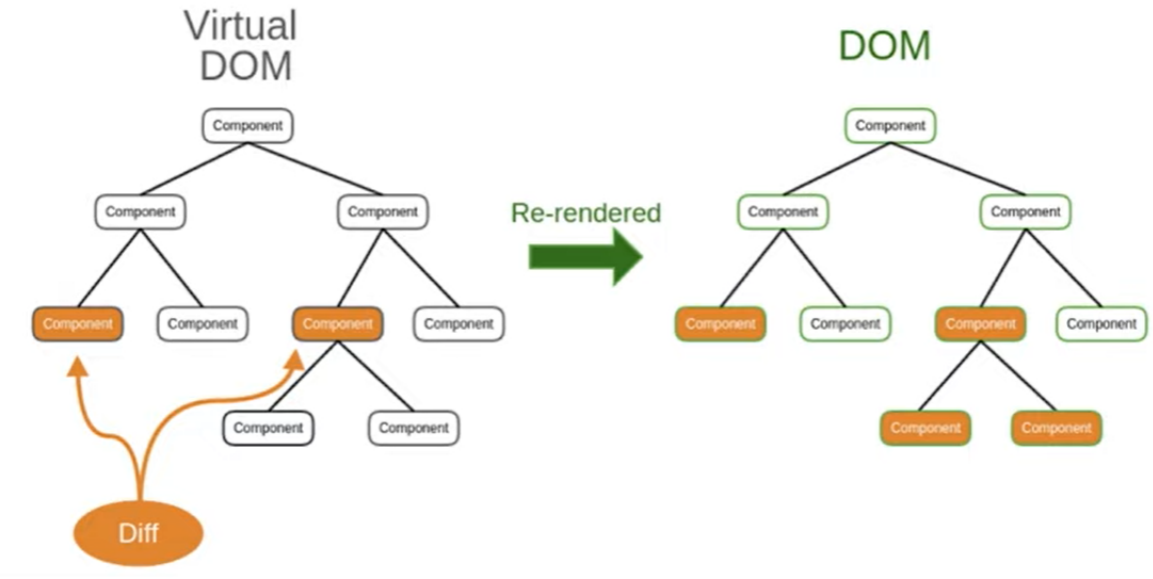
Требует транспиялции чтобы мог быть прочитан браузером.

```jsx
const Search = ({ value, onChange, onClick, onKeyPress }) => (
  // JSX syntax
  <div className={styles.search}>
    <input
      className={styles.input}
      type="text"
      onChange={onChange}
      onKeyPress={onKeyPress}
      value={value}
    />
    <Icon className={styles.searchSubmit} flag="fa" name="search" onClick={onClick} />
  </div>
);
```

▼ Что такое Virtual DOM? Как он работает с React?

Виртуальный DOM (Virtual DOM) - легковесная копия обычного DOM.

React использует виртуальный DOM для оптимизации производительности. При изменении компонента, React создает новое дерево виртуального DOM, сравнивает его с предыдущим деревом и определяет, какие элементы нужно обновить. Затем React обновляет только те элементы, которые изменились. Это позволяет сократить количество операций с реальным DOM и повысить скорость работы приложения.



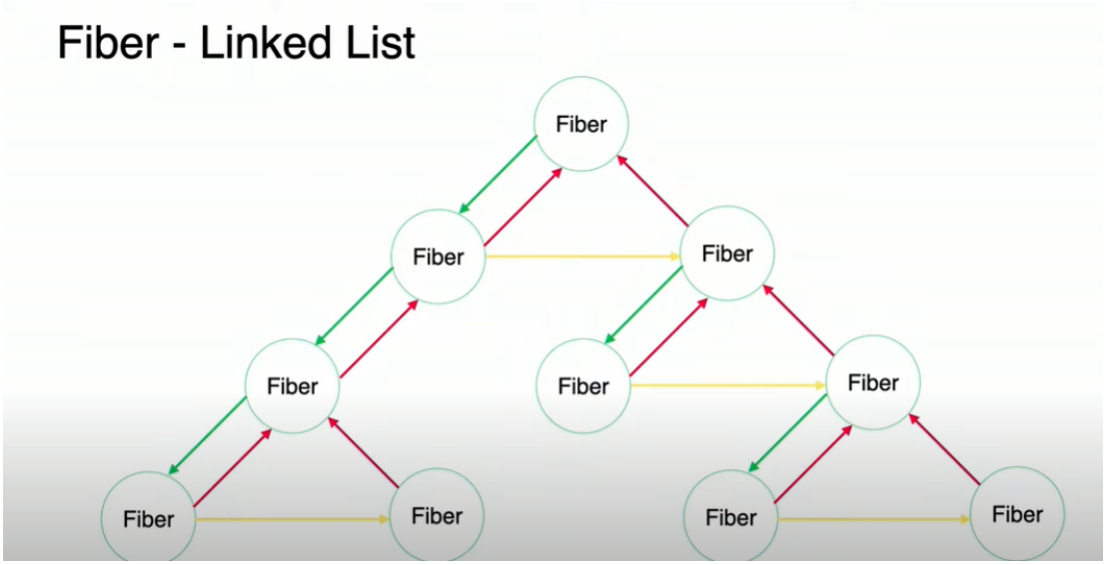
Две фазы:

1. Рендеринг и реконсилейшен

- Рендеринг - это процесс создания виртуального DOM на основе текущего состояния.
- Реконсиляция - это процесс сравнения предыдущего виртуального DOM с новым и выявления различий между ними.
- для каждого элемента создается файбер нода, которая знает о состоянии элемента и хранит ссылку на него. На всем жизненном компоненте у нас один файбер. Он удаляется лишь когда элемент размонтируется.

Файбер это ДЖС объект, имеет ссылку на элемент, у него есть поле stateNode где хранится состояние и инфа о пропсах. Знает дочерние элементы, родители, siblings (брат сестра)

Файберы хранятся в структуре связанных списков.



файберы связаны между собой разными отношениями.

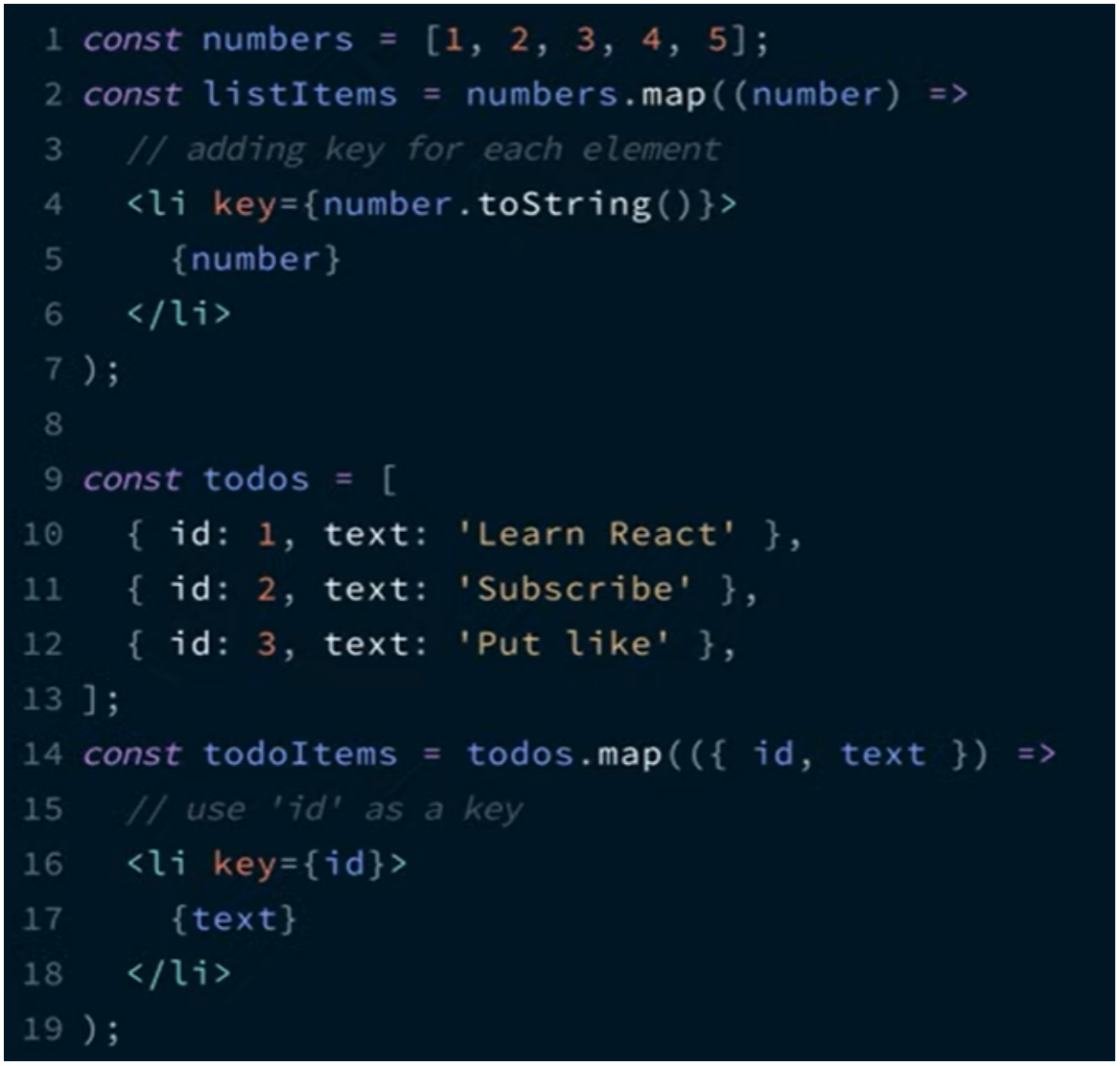
- родитель - ребенок.
- ребенок-родитель.
- ребенок-ребенок

2. фаза - коммит

- работа по внесению изменений. Она непрерываемая чтобы отобразились все изменения.
- сначала вносятся изменения в ДОМ, потом вызываются эффекты, т.к. эффекты могут потребовать взаимодействие с ДОМом.

▼ Для чего нужен атрибут `key` при рендере списков?

Атрибут key используется при рендере списков в React для определения уникальности каждого элемента списка. Когда происходит изменение списка, React использует ключи для определения, какие элементы были добавлены, удалены или перемещены. Без ключей React может некорректно обновлять элементы списка, что может привести к ошибкам в работе приложения. Использование ключей позволяет React эффективно обновлять только те элементы списка, которые действительно изменились, что повышает производительность и уменьшает нагрузку на браузер.



```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  // adding key for each element
  <li key={number.toString()}>
    {number}
  </li>
);

const todos = [
  { id: 1, text: 'Learn React' },
  { id: 2, text: 'Subscribe' },
  { id: 3, text: 'Put like' },
];
const todoItems = todos.map(({ id, text }) =>
  // use 'id' as a key
  <li key={id}>
    {text}
  </li>
);
```

▼ Что такое `PureComponent`?

PureComponent - это компонент в React, который автоматически реализует метод **shouldComponentUpdate**, который поверхностно

сравнивает новые и старые пропсы и состояние компонента и определяет, нужно ли перерисовывать компонент. Используется для решения проблемы рендера от родителя.

При сложных типах данных в пропсах и состоянии может не сработать т.к. сравнивает поверхностно.

```
class Count extends PureComponent {
  state = {
    count: 10,
  }

  handleClick = () => {
    this.setState(({ counter }) => ({
      counter: ++counter,
    }))
  }

  render(){
    return (
      <div className="count">
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.handleClick}>+1</button>
      </div>
    );
  }
}

export default Count;
```

▼ Что такое Компонент высшего порядка (Higher-Order Component/HOC)?

Higher-Order Component (HOC) - это компонент в React, который принимает другой компонент и возвращает новый компонент с дополнительными функциональными возможностями не изменяя исходный код.

Это не часть API, а паттерн, который представляет собой функцию которая принимает в себя компонент(функцию), расширяет его

функциональность не изменяя исходный код и внутреннюю реализацию, затем возвращает новый компонент.

```
const isEmpty = (prop) => (
  prop === null ||
  prop === undefined ||
  (prop.hasOwnProperty('length') && prop.length === 0) ||
  (prop.constructor === Object && Object.keys(prop).length === 0)
);

const LoadingHOC = (loadingProp) => (WrappedComponent) => {
  return class LoadingHOC extends Component {
    render() {
      return isEmpty(this.props[loadingProp]) ?
      <div className="loader" />
      : <WrappedComponent {...this.props} />;
    }
  }
}

export default LoadingHOC;
```

▼ Разница между управляемыми (controlled) и не управляемыми (uncontrolled) компонентами?

Управляемые компоненты (controlled components) - это компоненты, у которых значения состояний и свойств контролируются React. С помощью вызова setState или хука useState вызываемых внутри обработчиков элементов форм. Благодаря этому данные которые хранятся внутри управляемых элементов могут быть доступны за их пределами.

Неуправляемые компоненты (uncontrolled components) - это компоненты, значения которых контролируются DOM-элементами, а не React. Они изолированы от реакт.

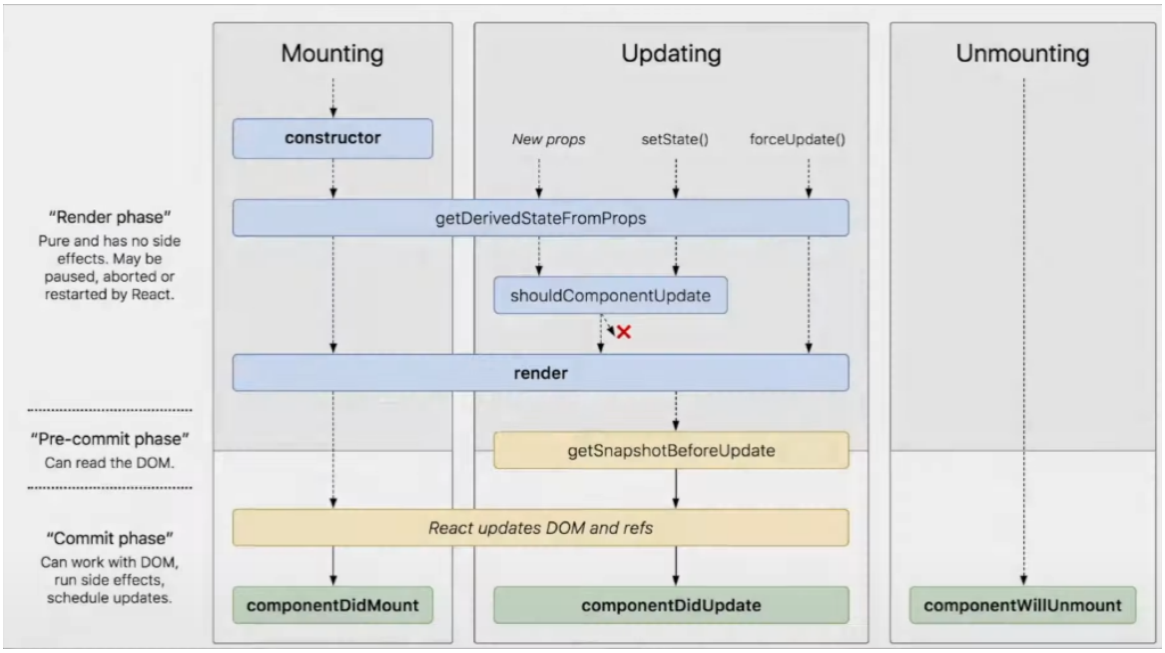
Основная разница между управляемыми и неуправляемыми компонентами заключается в том, что управляемые компоненты позволяют контролировать значения компонента из родительского компонента, тогда как неуправляемые компоненты работают независимо от React и контролируются DOM-элементами.

▼ Стадии жизненного цикла компонента в React?

1. **Mounting** (создание компонента) - компонент создается и вставляется в DOM.
2. **Updating** (обновление компонента) - компонент обновляется при изменении состояния или пропсов.
3. **Unmounting** (удаление компонента) - компонент удаляется из DOM.
4. **Error Handling** (обработка ошибок) - обработка ошибок в компоненте в классовых компонентах.

Также фазы:

- render - Не имеет побочных эффектов, может быть приостановлена, прервана или перезапущена реактом.
- pre-commit - Можно производить чтение DOM.
- commit - Может работать с DOM, выполнять побочные эффекты, назначать обновления.



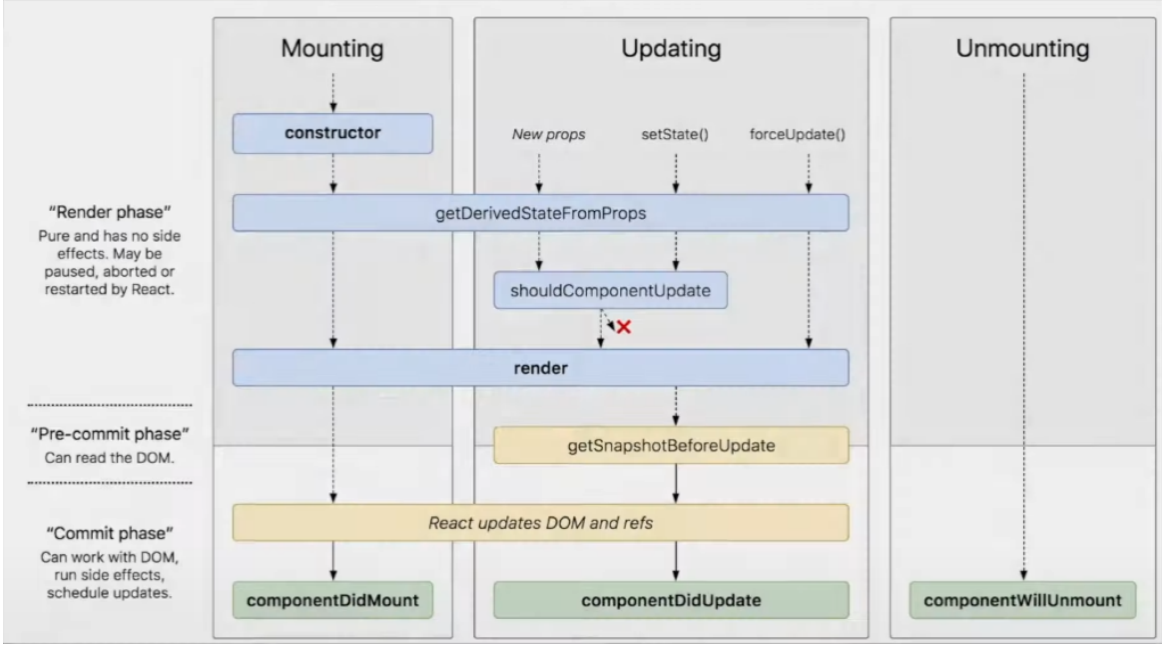
▼ Методы жизненного цикла компонента в React?

1. **Mounting (Создание)**:
   - `constructor()`: Метод конструктор вызывается первым при создании экземпляра компонента. Используется для инициализации состояния (`state`) и привязки методов к экземпляру.

- `static getDerivedStateFromProps()`: Этот статический метод вызывается перед каждым рендерингом, как при монтировании, так и при обновлении компонента, и позволяет компоненту обновить его внутреннее состояние на основе новых свойств (`props`).
- `render()`: (ОБЯЗАТЕЛЬНЫЙ!) Он отвечает за возвращение JSX-разметки, которая будет отображена в браузере.
- `componentDidMount()`: Метод вызывается сразу после того, как компонент был отрендерен в DOM. Используется для инициализации данных, подписки на события и других действий, которые требуют доступа к DOM.

2. **Updating (Обновление)**:
   - `static getDerivedStateFromProps()`: Подробнее описан выше.
   - `shouldComponentUpdate()`: Метод вызывается перед обновлением компонента и позволяет определить, нужно ли React обновлять компонент или нет. Возвращает булево значение (`true` - обновить, `false` - не обновлять).
   - `render()`: Подробнее описан выше.
   - `getSnapshotBeforeUpdate()`: Метод вызывается прямо перед изменением DOM после рендеринга. Позволяет компоненту захватить текущее состояние DOM (например, положение прокрутки) перед тем, как произойдет обновление.
   - `componentDidUpdate()`: Метод вызывается после обновления компонента и после того, как обновленный компонент был отрендерен в DOM. Используется для выполнения дополнительных действий после обновления компонента, например, для обновления состояния на основе новых свойств или для работы с DOM после его изменения.
3. **Unmounting (Размонтирование)**:
   - `componentWillUnmount()`: Метод вызывается перед тем, как компонент будет удален из DOM. Используется для выполнения дополнительных действий перед удалением компонента, например, для отмены подписок на события или очистки ресурсов.

4. **Error Handling (Обработка ошибок)**:
    - `static getDerivedStateFromError()`: статический метод вызывается, когда дочерний компонент бросает ошибку. Позволяет компоненту обновить свое состояние на основе ошибки.
    - `componentDidCatch()`: Метод вызывается после того, как дочерний компонент бросил ошибку. Используется для логирования ошибок или отправки их на сервер.



▼ Что делает метод `shouldComponentUpdate`?

Метод `shouldComponentUpdate()` является методом жизненного цикла классовых компонентов. Он вызывается перед повторным рендерингом компонента и позволяет контролировать, должен ли React обновить компонент.

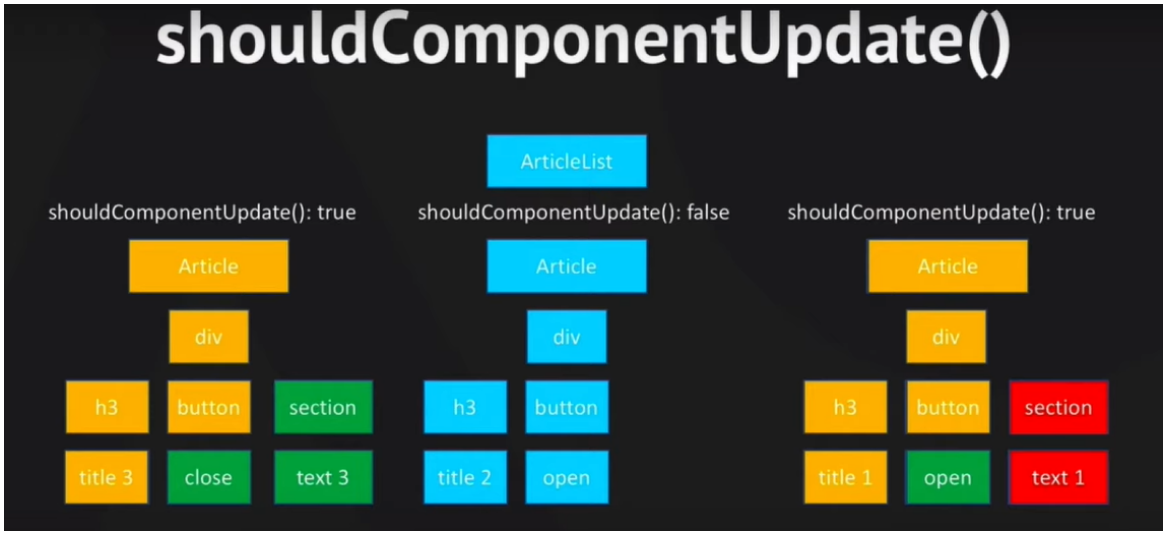
Возвращает булиновое значение. Если компоненту не требуется обновление, он возвращает
`false`, чтобы предотвратить повторный рендеринг и повысить производительность приложения. Это особенно полезно, когда компонент имеет сложное состояние или рендерится часто, но его отображение не зависит от изменений входных данных или состояния.

Метод **shouldComponentUpdate** принимает два аргумента: **nextProps** и **nextState**. Они содержат новые значения свойств и состояния

компонента соответственно. Метод сравнивает текущие и новые значения свойств и состояния, и определяет, нужно ли обновлять компонент.

```
class ExampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    // Проверяем, изменились ли входные данные
    if (this.props.someProp !== nextProps.someProp) {
      return true; // Обновляем компонент
    }

    // Проверяем, изменилось ли состояние
    if (this.state.someState !== nextState.someState) {
      return true; // Обновляем компонент
    }

    return false; // Не обновляем компонент
  }

  render() {
    // Рендеринг компонента
  }
}
```

Аналог это **useEffect** с массивом зависимостей.



▼ Что такое React Reconciliation?

React Reconciliation - это рекурсивный алгоритм React, используемый для того, чтобы отличить текущее дерево элементов от нового для определения частей, которые нужно будет заменить.

Примерный алгоритм такой

1. **Изменение состояния или пропсов:**
   Когда происходит изменение состояния/пропсов в React-компоненте, React создает новое виртуальное дерево элементов (**Virtual DOM**) для представления обновленного состояния.
2. **Сравнение предыдущего и нового деревьев:**
   React берет предыдущее виртуальное дерево и новое виртуальное дерево, затем начинает сравнивать их, начиная с корневых элементов.
3. **Сравнение типов элементов:**
   Сначала React сравнивает типы элементов в предыдущем и новом деревьях. Если типы разные, React полностью удаляет старый элемент и все его дочерние элементы из реального DOM.
4. **Сравнение дочерних элементов:**
   Если типы элементов совпадают, React рекурсивно сравнивает дочерние элементы обоих деревьев.
5. **Определение изменений:**
   В процессе сравнения React определяет, какие части дерева изменились. Это может включать в себя добавление, удаление или обновление компонентов.
6. **Пакет обновлений (Batch Updates)**:

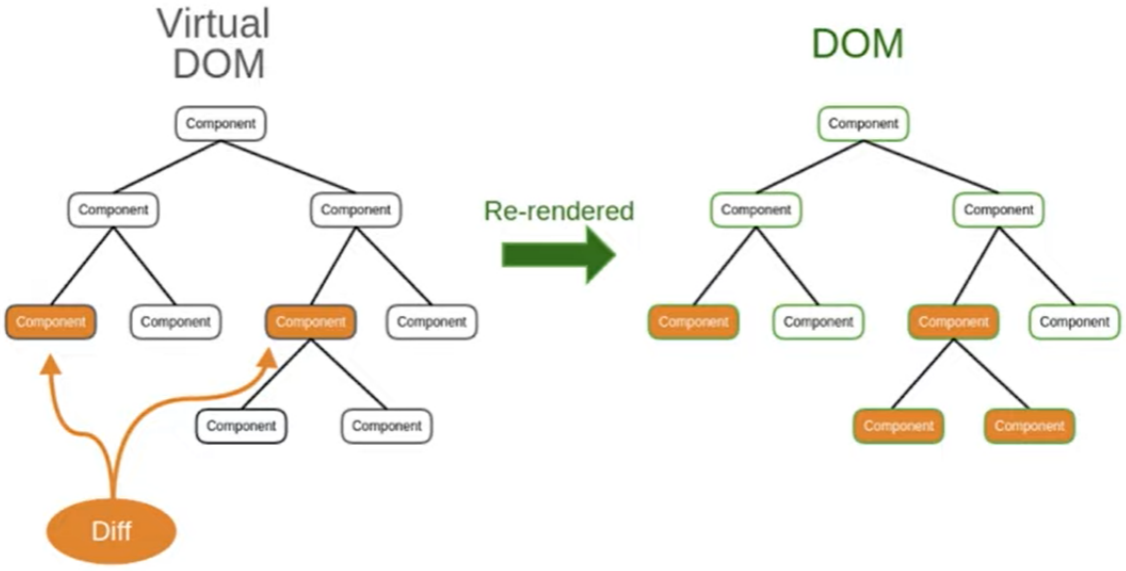
   React группирует изменения, чтобы минимизировать количество манипуляций DOM и повысить производительность. Это означает, что React не обновляет DOM после каждого изменения, а собирает все изменения и применяет их одновременно.
7. **Применение изменений:**
   React применяет эти изменения к реальному DOM. Если компонент был изменен, React обновит его атрибуты и содержимое. Если компонент был добавлен, React добавит его в DOM. Если компонент был удален, React удаляет его из DOM.

В результате этого процесса React обновляет только те части DOM, которые действительно изменились, а не перерисовывает всю страницу заново.

При сравнении двух React DOM-элементов одного типа, React смотрит на атрибуты обоих, сохраняет лежащий в основе этих элементов DOM-узел и обновляет только изменённые атрибуты.

По умолчанию при рекурсивном обходе дочерних элементов DOM-узла React одновременно проходит по обоим спискам потомков и создаёт мутацию, когда находит отличие.

Когда у дочерних элементов есть ключи, React воспользуется ими, чтобы сопоставить потомков исходного дерева с потомками последующего дерева.



▼ Что такое портал (`Portal`)?

Портал - это компонент, который мы можем рендерить в DOM вне иерархии нашего компонента

Создается при помощи метода **React.createPortal(children, container)**.

Типичные примеры использования порталов:

- Модальные и диалоговые окна.
- Всплывающие подсказки.
- Всплывающие визитки.
- Загрузчики.

Они как правило не зависят от разметки и появляются поверх любого лейаута.

```
const modalRoot = document.getElementById('modal-root');

class Modal extends Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }
  componentDidMount() {
    modalRoot.appendChild(this.el);
  }
  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }
  render() {
    // Portal creation
    return ReactDOM.createPortal(
      this.props.children,
      this.el,
    );
  }
}
```

▼ Что такое контекст (`Context`)?

Контекст (Context) - это механизм в React, который позволяет передавать данные глубоко вниз по иерархии компонентов без необходимости явно передавать их через пропсы каждому компоненту.

Контекст создается с помощью метода **React.createContext(defaultValue)**, где **defaultValue** - значение по умолчанию для контекста. Затем, любой компонент может получить доступ к контексту с помощью метода **Context.Consumer** или использовать **Context.Provider** для изменения значения контекста.

С версии 16.8 с появлением хуков, для этой задачи используется хук **useContext**

```
// (1) File context.js
import React from 'react';
export const ThemeContext = React.createContext('main-theme');

// (2) File App.js
// Importing ThemeContext
import { ThemeContext } from './context';
// Importing useContext hook
import React, { useContext } from 'react';

export const App = () => {
  // Assign the result of the hook work to a variable
  const theme = useContext(ThemeContext);

  return <App theme={theme} />;
};
```

▼ Что такое React хуки (Hooks)?

React хуки (Hooks) - специальные функции в React, которые расширяют возможности функциональных компонентов.

Правила хуков:

- Хуки доступны только в функциональных компонентах или в других хуках. Нельзя использовать хуки в классовых компонентах.
- Хуки должны вызываться только на верхнем уровне функционального компонента. Не вызывайте хуки внутри циклов, условий или вложенных функций. Порядок вызова хуков должен оставаться постоянным между рендерами компонента.
- Необходимо следить за порядком вызова хуков, так как он может повлиять на их работу.
- Имена хуков должны начинаться с префикса `use`, чтобы React мог отличить хуки от обычных функций.
- Нельзя использовать хуки в функциях, которые вызываются на сервере (например, при генерации статических страниц).

Основные хуки:

- useContext
- useState

- useEffect
- useRef
- useReducer
- useLayoutEffect
- useCallback
- useMemo
- useImperativeHandle
- useDebugValue
- пользовательские хуки

▼ Хуки (Подробнее)

▼ useState

Хук React, который позволяет вам добавить переменную состояния в ваш компонент.

Асинхронный.

Подробнее

```
import { useState } from 'react';

function MyComponent() {
    const [state, setState] = useState(initialState);
    // ...
}
```

Параметры:

- `initialState` - Значение, которое будет первоначальным. Это может быть значение любого типа, но для функций предусмотрено особое поведение. Этот аргумент игнорируется после первоначального рендеринга.

Если вы передадите функцию в качестве `initialState`, она будет рассматриваться как ***инициализирующая функция***. Она должна быть чистой, не принимать никаких аргументов и возвращать значение любого типа. React будет вызывать вашу функцию

инициализатора при инициализации компонента и сохранять возвращаемое значение в качестве начального состояния.

Возвращает:

Массив, содержащий два значения:

1. Текущее состояние. Во время первого рендера оно будет соответствовать переданному вами `initialState`.
2. Функция `set`, которая позволяет обновить состояние до другого значения и вызвать повторный рендеринг.

▼ useEffect

Подробнее

Предоставляет возможность выполнения побочных эффектов в функциональных компонентах АСИНХРОННО, обычно вызывается после отрисовки компонента. Побочные эффекты включают в себя такие операции, как загрузка данных с сервера, подписка на события DOM, изменение заголовка документа и т. д. `useEffect` выполняет эти действия после каждого рендера компонента. Является заменой хуков жизненного цикла в классовых компонентах (componentDidMount, componentDidUpdate, componentWillUnmount).

```
useEffect(callback, dependencies?)
```

Параметры

- `callback`: Функция с логикой вашего Эффекта.
- `dependencies`: (**опциональные**)Список всех зависимостей**.** Список зависимостей должен иметь постоянное количество элементов и быть написан по типу `[dep1, dep2, dep3]`. React будет сравнивать каждую зависимость с предыдущим значением, используя сравнение `Object.is`.

  Если не передать массив зависимостей, то функция, переданная в `useEffect`, будет выполняться после каждого рендера компонента и после каждого обновления его состояния. Это означает, что `useEffect` будет запускаться при каждом

обновлении компонента.

Если передать пустой массив зависимостей, то функция, переданная в `useEffect`, будет выполняться только при монтировании компонента, а также при его размонтировании. Это аналог `componentDidMount` и `componentWillUnmount` в классовых компонентах.

Возвращает:

`useEffect` возвращает `undefined` или функцию очистки.

- Когда включен строгий режим, React будет **проводить один дополнительный цикл настройки+очистки** только в режиме разработки перед первой реальной настройкой. Это стресс-тест, который гарантирует, что ваша логика очистки "отражает" вашу логику настройки и что она останавливает или отменяет все, что делает настройка. Если это вызывает проблему, реализуйте функцию очистки.

```
useEffect(() => {
  const timerId = setInterval(() => {
    // Логика, выполняемая каждый интервал времени
  }, 1000);

  // Функция очистки, которая будет вызвана при размо
  return () => {
    clearInterval(timerId); // Отменяем таймер при ра
  };
}, []);
```

- Если некоторые из ваших зависимостей являются объектами или функциями, определенными внутри компонента, есть риск, что они **приведут к тому, что Эффект будет перезапускаться чаще, чем нужно.**
- Если ваш Эффект не был вызван взаимодействием (например, щелчком мыши), React позволит браузеру сначала нарисовать обновленный экран, прежде чем запустить ваш Эффект. Если ваш Эффект делает что-то визуальное

(например, позиционирует всплывающую подсказку), и задержка заметна (например, она мерцает), замените `useEffect` на `useLayoutEffect`.

- Даже если ваш Эффект был вызван взаимодействием (например, щелчком), **браузер может перерисовать экран до обработки обновлений состояния внутри вашего Эффекта.** Обычно это то, что вам нужно. Однако, если вы должны запретить браузеру перерисовывать экран, вам нужно заменить `useEffect` на `useLayoutEffect`.

▼ useLayoutEffect

Подробнее

Это версия `useEffect`, которая срабатывает перед тем, как браузер перерисовывает экран.
Код внутри `useLayoutEffect` и все запланированные обновления состояния **блокируют браузер от перерисовки экрана (**РАБОТАЕТ СИНХРОННО**).** При чрезмерном использовании это делает ваше приложение медленным. Когда это возможно, предпочитайте `useEffect`.

▼ useCallback

Используется для мемоизации функций и предотвращения их повторного создания при каждом рендере компонента. Мемоизированная функция так же запоминает свое окружение.

```
const memoizedCallback = useCallback(callback, dependenc
```

Параметры:

- `callback` - функция, которую нужно мемоизировать.
- `dependencies` (необязательный параметр) - массив зависимостей. Если передан, `useCallback` будет возвращать мемоизированную версию функции, которая будет пересоздаваться только при изменении значений в этом массиве.

Возвращает:

- При первоначальном рендере `useCallback` возвращает переданную вами функцию `fn`.
- При последующих рендерах она либо вернет уже сохраненную функцию `fn` из последнего рендера (если зависимости не изменились), либо вернет функцию `fn`, которую вы передали во время этого рендера.

Подробнее про использование

```
import React, { useState, useCallback } from 'react';

const ChildComponent = React.memo(({ onClick }) => {
  console.log('ChildComponent рендерится');
  return <button onClick={onClick}>Нажми меня</button>;
});

const ParentComponent = () => {
  const [count, setCount] = useState(0);

  // Мемоизация функции handleClick при помощи useCallba
  const handleClick = useCallback(() => {
    setCount(prevCount => prevCount + 1);
  }, []);

  console.log('ParentComponent рендерится');

  return (
    <div>
      <p>Вы нажали {count} раз</p>
      {/* Передача мемоизированной функции handleClick в
      <ChildComponent onClick={handleClick} />
    </div>
  );
};

export default ParentComponent;
```

Кейсы использования:

- **Передача функций в дочерние компоненты -** Если компонент передает колбэк-функцию в качестве пропса в дочерний компонент, используйте `useCallback`, чтобы предотвратить ненужные перерисовки дочернего компонента при изменении родительского состояния.
- **Оптимизация мемоизированных компонентов -** Когда используется мемоизация компонентов с помощью `React.memo`, функции, передаваемые в пропсах, также должны быть мемоизированы. В этом случае `useCallback` может быть использован для мемоизации этих функций.
- **Оптимизация эффектов зависимости -** Если колбэк используется как зависимость внутри эффекта (`useEffect`), использование `useCallback` сможет предотвратить лишние повторные запуски эффекта при каждом рендере компонента.
- **Повторное использование функций обработчиков событий -** Когда обработчики событий передаются в элементы JSX, `useCallback` может использоваться для повторного использования функций обработчиков событий и предотвращения их пересоздания при каждом рендере компонента.

▼ useMemo

Подробнее

Используется для мемоизации вычислений в функциональных компонентах. Это позволяет избегать повторного вычисления значений при каждом рендере компонента, что может повысить производительность приложения.

```
const cachedValue = useMemo(calculateValue, dependencies
```

Параметры:

- `calculateValue`: Функция, вычисляющая значение, которое вы хотите кэшировать. Она должна быть чистой, не принимать аргументов и возвращать значение любого типа.

- `dependencies`: массив зависимостей. Если передан, `useMemo` будет пересчитывать значение только при изменении значений в этом массиве.

Возвращает:

- При первоначальном рендере `useMemo` возвращает результат вызова `calculateValue` без аргументов.
- При последующих рендерах он либо вернет уже сохраненное значение из последнего рендера (если зависимости не изменились), либо снова вызовет `calculateValue` и вернет результат, который вернул `calculateValue`.

В строгом режиме React будет **вызывать вашу функцию вычисления дважды**, чтобы помочь вам найти случайные примеси Это поведение только для разработки и не влияет на производство. Если ваша функция вычисления чиста (как и должно быть), это не должно повлиять на вашу логику. Результат одного из вызовов будет проигнорирован.

▼ useContext

Используется для получения доступа к значению контекста изнутри функционального компонента.

Подробнее

```
const value = useContext(SomeContext);
```

Параметры:

- Объект контекста, который был создан с помощью функции `React.createContext()`.

Возвращает:

- `useContext` возвращает значение контекста для вызывающего компонента. Оно определяется как `value`, переданное ближайшему `Context.Provider` над вызывающим компонентом в

дереве. Если такого провайдера нет, то возвращаемое значение будет `defaultValue`, которое вы передали в `createContext` для этого контекста.

Ограничения

- Вызов `useContext()` в компоненте не влияет на провайдеров, возвращаемых из *этого же* компонента. Соответствующий `<Context.Provider>` **должен находиться *выше*** компонента, выполняющего вызов `useContext()`.
- React **автоматически перерисовывает** все дочерние компоненты, использующие определенный контекст, начиная с провайдера, получившего другое `value`. Пропуск повторных рендеров с помощью `memo` не мешает дочерним компонентам получать свежие значения контекста.

▼ useRef

Подробнее

Используется для создания изменяемых значений внутри компонента и сохранения ссылок на DOM-элементы. Он позволяет сохранять значения между рендерами компонента и обновлять их без вызова повторного рендера. **useRef()** также может использоваться для хранения промежуточных значений и функций внутри компонента. Это делает его полезным инструментом для управления состоянием и поведением компонентов в React.

```
const ref = useRef(initialValue);
```

Параметры:

- `initialValue` (необязательный параметр) - начальное значение для созданной ссылки.

Возвращает:

- Объект с одним свойством `current`: Изначально оно установлено на `initialValue`, которое вы передали.

```
import React, { useRef } from 'react';

const ExampleComponent = () => {
  const inputRef = useRef(null);

  const handleClick = () => {
    // Установка фокуса на поле ввода при клике на кнопк
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Установить фокус</bu
    </div>
  );
};

export default ExampleComponent;
```

В этом примере мы используем `useRef`, чтобы создать ссылку `inputRef`, которая затем привязывается к элементу `<input>`. Когда мы кликаем на кнопку, вызывается функция `handleClick`, которая устанавливает фокус на поле ввода, используя свойство `.current` ссылки `inputRef`.

Важно отметить, что изменения свойства `.current` не вызывают перерисовку компонента. Поэтому `useRef` часто используется для сохранения данных между рендерами без вызова перерисовки компонента, или для доступа к DOM-узлам без необходимости использования `document.getElementById` или `document.querySelector`.

▼ useReducer

Используется для управления состоянием компонентов. В отличие от `useState`, который предоставляет простое локальное состояние для

функциональных компонентов, `useReducer` используется для управления более сложными состояниями и выполнения действий (actions) на этом состоянии.

```
const [state, dispatch] = useReducer(reducer, initialSta
```

Параметры

- `reducer`: Функция **reducer**, определяющая, как обновляется состояние. Она должна быть чистой, принимать в качестве аргументов состояние и действие и возвращать следующее состояние. Состояние и действие могут быть любого типа.
- `initialState`: Начальное состояние, которое будет передано в редюсер при первом рендеринге компонента. Это может быть значение любого типа.
- **опционально** `init`: Функция инициализатора, которая должна вернуть начальное состояние. Если она не указана, то начальное состояние устанавливается в `initialState`. В противном случае начальное состояние устанавливается в результат вызова `init(initialArg)`.

```
import React, { useReducer } from 'react';

// Начальное состояние
const initialState = { count: 0 };

// Редюсер - функция, которая принимает текущее состояни
// и возвращает новое состояние на основе этого действия
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return { count: 0 };
    default:
      throw new Error();
```

```
    }
};

// Компонент, использующий useReducer
const Counter = () => {
  // Используем хук useReducer для управления состоянием
  // Передаем редюсер и начальное состояние
  const [state, dispatch] = useReducer(reducer, initialS

  // Функции для увеличения и уменьшения счетчика
  const increment = () => {
    dispatch({ type: 'increment' });
  };

  const decrement = () => {
    dispatch({ type: 'decrement' });
  };

  const decrement = () => {
    dispatch({ type: 'reset' });
  };

  // Возвращаем JSX компонента
  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};

export default Counter;
```

Возвращает:

Массив, содержащий два значения:

- Текущее состояние. Во время первого рендера оно устанавливается в `init(initialArg)` или `initialArg` (если нет `init`).

- Функция `dispatch`, которая позволяет обновить состояние до другого значения и вызвать повторный рендеринг.

▼ useImperativeHandle

Подробнее

Используется для настройки того, какие функции или значения дочерний компонент может предоставить своему родителю через `ref`. Это полезно, если вам нужно, чтобы родительский компонент мог управлять определенными аспектами поведения дочернего компонента, но без необходимости открывать доступ ко всем его методам и свойствам.

```
useImperativeHandle(ref, createHandle, dependencies?)
```

Параметры

- `ref`: Ссылка на ref, переданная через `forwardRef`.
- `createHandle`: Функция, которая не принимает аргументов и возвращает хэндл ссылки, которую вы хотите раскрыть. Этот ref handle может иметь любой тип. Обычно вы возвращаете объект с методами, которые вы хотите раскрыть.
- **опционально** `dependencies`: Список зависимостей.

Возвращает:

- `undefined`.

```
const CustomInput = forwardRef((props, ref) => {
  const [value, setValue] = useState('');

  // Определение метода focus(), который будет доступен
  useImperativeHandle(ref, () => ({
    focus: () => {
      console.log('Input focused');
    },
    // Для примера, возврат текущего значения поля
    getValue: () => value
```

```
  }));

  return (
    <input
      type="text"
      value={value}
      onChange={(e) => setValue(e.target.value)}
      placeholder="Enter text"
    />
  );
});


const ExternalComponent = () => {
  const inputRef = useRef(null);

  const handleClick = () => {
    // Вызываем метод focus() на компоненте CustomInput
    inputRef.current.focus();
  };

  const handleGetValue = () => {
    // Получаем текущее значение поля через метод getVal
    const value = inputRef.current.getValue();
    console.log('Current value:', value);
  };

  return (
    <div>
      <CustomInput ref={inputRef} />
      <button onClick={handleClick}>Focus Input</button>
      <button onClick={handleGetValue}>Get Value</button
    </div>
  );
};
```

▼ useTransition

Позволяет обновлять состояние без блокировки пользовательского интерфейса.

```
const [isPending, startTransition] = useTransition();
```

Параметры:

- Не принимает никаких параметров.

Возвращает:

Массив, содержащий два значения:

- `isPending` - булево значение, которое указывает, находится ли компонент в процессе перехода.
- `startTransition` - функция, которая запускает переход между состояниями. Вызов **startTransition** помечает компонент для перехода, но не блокирует пользовательский интерфейс, пока переход не завершится.

Как это работает простым языком:

Посмотри!!!

▼ useDeferredValue

Подробнее

Позволяет отложить обновление состояния компонента до момента, когда он неактивен или до завершения других приоритетных задач.

```
const deferredValue = useDeferredValue(value);
```

Параметры:

- Значение, которое нужно обновить.

Возвращает:

- Во время первоначального рендеринга возвращаемое отложенное значение будет таким же, как и предоставленное вами значение. Во время обновления React сначала попытается

выполнить повторный рендеринг со старым значением (поэтому вернет старое значение), а затем попытается выполнить повторный рендеринг в фоновом режиме с новым значением (поэтому вернет обновленное значение).

Предупреждения:

Значения, которые вы передаете в `useDeferredValue`, должны быть либо примитивными значениями (такими как строки и числа), либо объектами, созданными вне рендеринга. Если вы создадите новый объект во время рендеринга и сразу передадите его в `useDeferredValue`, он будет отличаться при каждом рендеринге, что приведет к ненужным повторным рендерам фона.

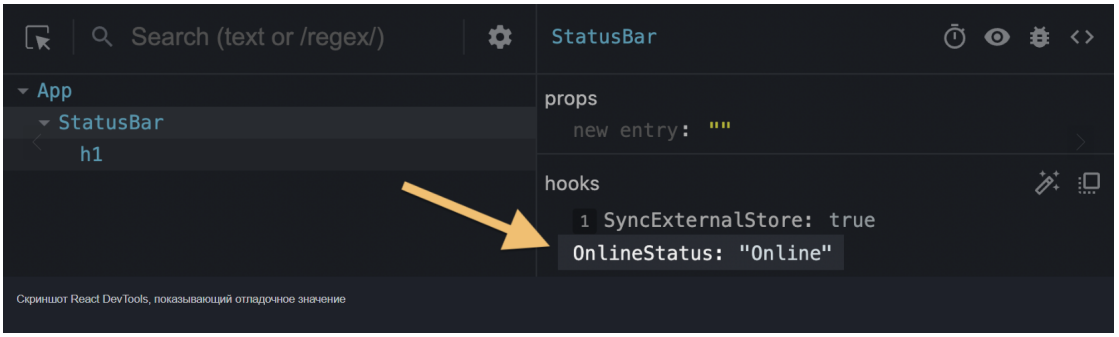
Посмотри

▼ useDebugValue

Подробнее

Позволяет добавить метку к пользовательскому хуку в **React DevTools**.

```
useDebugValue(value, format?)
```



Параметры:

- `value`: Значение, которое вы хотите отобразить в React DevTools. Оно может иметь любой тип.
- `format` **(опционально)** : Функция форматирования. При осмотре компонента React DevTools вызовет функцию форматирования

с `value` в качестве аргумента, а затем отобразит возвращенное форматированное значение (которое может иметь любой тип). Если вы не укажете функцию форматирования, будет отображено исходное `value`.

Возвращает

- Ничего не возвращает.

▼ Разница между `useEffect()`, `componentDidMount()` и `useLayoutEffect()?`

- **componentDidMount()** вызывается только один раз после того, как компонент был добавлен в DOM, но перед отрисовкой в браузере. Он используется для выполнения действий, которые должны произойти только один раз при монтировании компонента, например, загрузка данных с сервера.
  На основе `render` создается виртуальное дерево, но перед тем как отдать виртуальное дерево на отрисовку в браузер, вызывается `componentDidMount` и блокирует отрисовку в браузере, если внутри есть код блокирующий поток.
- **useEffect()** вызывается после каждого рендера компонента (включая первый рендер), т.е. после того как интерфейс отрисован на экране. Он может быть использован для выполнения действий при каждом рендере компонента или для эмуляции методов жизненного цикла, которые вызываются только один раз. Например, **useEffect()** может быть использован для выполнения действий при изменении пропсов или состояния компонента.
  Сначала на основе `return` создается виртуальное дерево, далее оно отдается на отрисовку в браузер и только после этого вызывается функция переданная в `useEffect` (Выполняется асинхронно)
- **useLayoutEffect()** если нам нужно выполнить какой-то код до отрисовки в браузере, нам предоставили хук `useLayoutEffect`, интерфейс которого полностью совпадает с `useEffect`, но по очередности выполнения полностью совпадает с `componentDidMount` (Выполняется синхронно).

```
// Class component
import React, { Component } from 'react';

export default class MyComponent extends Component {
  componentDidMount() {
    // executing code after components is mounted
  }

  render() {
    return (<div>test</div>);
  }
}

// Function component
import React, { useEffect } from 'react';

export const MyComponent = () => {
  useEffect(() => {
    // executing code after components is mounted
  }, []);

  return (<div>test</div>);
};
```

▼ Преимущества хуков?

Преимущества хуков в React заключаются в том, что они позволяют использовать состояние и другие функциональные возможности React в функциональных компонентах.

Хуки позволяют разделять логику компонента на более мелкие и переиспользуемые части, что улучшает читаемость и поддерживаемость кода. Легче тестировать отдельные функции компоненты.

Также использование хуков позволяет избежать проблем с **this**, которые могут возникать при использовании классовых компонентов.

Наконец, хуки предоставляют возможность более гибко управлять жизненным циклом компонента и его состоянием, что упрощает разработку и отладку приложений.

Сложная логика может быть вынесена в кастомный хук.

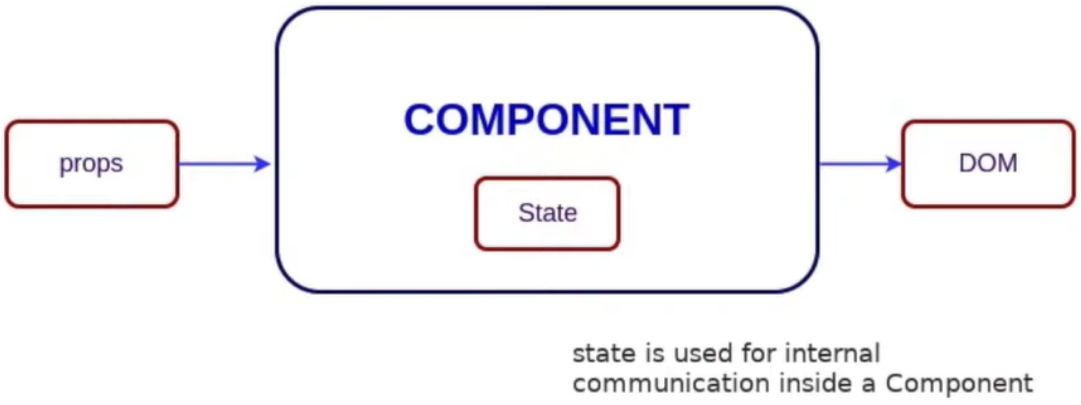
▼ Недостатки хуков?

- Правила, или ограничения на их использование
- Не покрывают getSnapshotBeforeUpdate и componentDidCatch
- Проблемы с перформансом при неверном использовании

▼ Разница между состоянием(`state`) и пропсами(`props`)?

Props - это аргументы, которые передаются компоненту извне. Нужны для получения данных в компоненте из вне, вызов изменения состояния.

Состояние (state) - это объект, который содержит данные, которые могут изменяться в течении жизненного цикла компонента. Инкапсулирован внтури компонента и управляется только изнутри.



▼ Что такое React Fiber?

Посмотри!!!

React Fiber - это новый механизм реконсиляции, который был введен в React 16. Этот механизм был разработан для улучшения производительности и возможностей асинхронного обновления пользовательского интерфейса.

Основные характеристики React Fiber:

1. **Асинхронность**: React Fiber позволяет React выполнять реконсиляцию частями, разбивая процесс на множество небольших задач. Это позволяет React более гибко управлять приоритетами и обрабатывать асинхронные события, такие как анимации или сетевые запросы, без блокировки интерфейса.
2. **Приоритеты и возобновляемость**: Fiber вводит понятие приоритетов задач, что позволяет React определять, какие части интерфейса требуют немедленного обновления, а какие могут быть отложены. Это также позволяет приостанавливать и возобновлять процесс реконсиляции, что полезно для более гладких анимаций и интерактивности.
3. **Расширяемость и поддержка синхронной реконсиляции**: В ходе разработки Fiber была предпринята работа по расширению возможностей реконсиляции в React. Хотя асинхронность - основная черта Fiber, он также поддерживает синхронную реконсиляцию, что обеспечивает совместимость с существующим кодом и позволяет более просто разрабатывать и отлаживать приложения.

Fiber Node это JS объект, который хранит ссылку на элемент! Если элемент меняется из-за изменения пропсов или состояния, но остается на том же месте, **Fiber Node** остается прежним, в нем меняется ссылка на элемент. **Fiber Node** удаляется, если размонтируется компонент.

Хранит в себе:

- инфо с помощью какой функции был создан этот элемент
- тип элемента
- ссылку на дочерний элемент (только один!!! даже если их несколько)
- ссылку на родителя
- ссылку на одноуровневого родственника

- состояние
- пропсы

▼ Что такое фрагмент (`Fragment`)? Почему фрагмент лучше, чем `div`?

Фрагмент (Fragment) - это компонент React, который позволяет группировать несколько элементов без создания лишних узлов в DOM. Он позволяет рендерить несколько элементов как один компонент без использования обертывающего элемента, такого как div.

Фрагмент лучше, чем **div**, потому что он не создает дополнительных узлов в DOM, что может привести к нежелательным последствиям для производительности и доступности.

▼ Что такое синтетические события в React?

Синтетические события в React - это кроссбраузерная обертка для нативных событий браузера, то есть любые события с которыми работает Реакт являются не нативными, а всего лишь обертками. Данный API полностью аналогичен браузерному, однако синтетические события работают одинаково во всех браузерах, что помогает в кроссбраузерности т.к. не требуются фолбеки или полифилы.

▼ Что такое React-ссылка (`ref`)? Как создать ссылку?

Подробнее тут

React-ссылка (ref) - это объект, который позволяет получить доступ к DOM-элементу или экземпляру компонента в React, возвращает ссылку на элемент. Практически аналог метода getElementById

В большинстве случаев следует избегать применения ссылок.

Как создать:

- React.createRef() - классовые компоненты

```
class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.inputRef = React.createRef(); // Создаем ссылку
  }
```

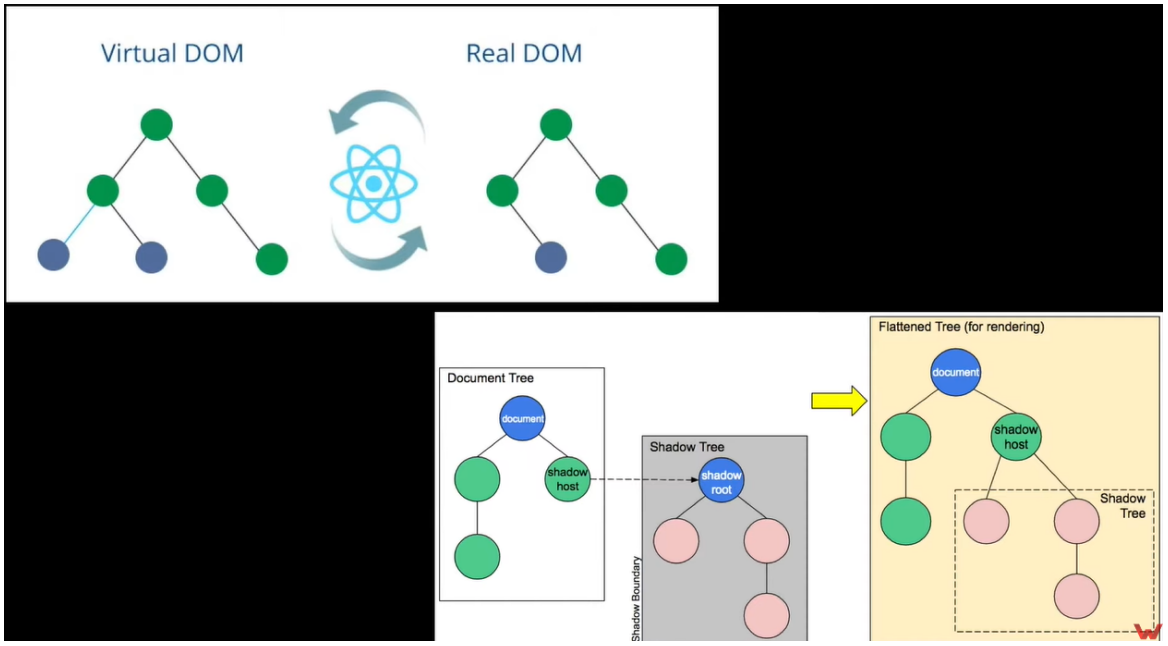
```
  componentDidMount() {
    this.inputRef.current.focus(); // Устанавливаем фокус н
  }

  render() {
    return (
      <div>
        <input type="text" ref={this.inputRef} /> {/* Присв
      </div>
    );
  }
}
```

- useRef() - функциональные компоненты (Подробнее)

▼ Разница между теневым (Shadow) и виртуальным (Virtual) DOM?

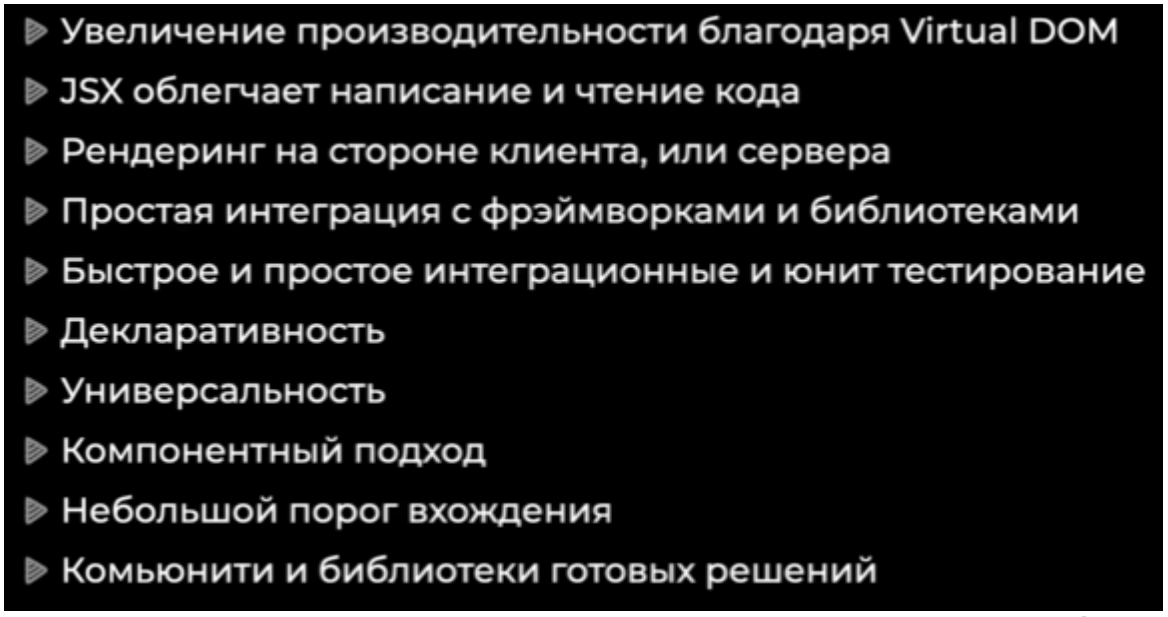
Теневой (Shadow) DOM - это изолированный участок дерева DOM, часть web API, которое может быть создано и использовано внутри компонента веб-приложения. Он позволяет создавать компоненты с собственным стилем и поведением, которые могут быть вложены в другие компоненты без конфликтов стилей и поведения за счет того что они изолированы от дерева DOM.

Virtual DOM - не является частью web API, копия всего DOM.



▼ Назовите преимущества использования React?

1. **Быстродействие** благодаря использованию виртуального DOM, React обеспечивает высокую производительность и быстрое обновление интерфейса.
2. **Масштабируемость** React позволяет создавать масштабируемые приложения, которые легко поддерживать и расширять.
3. **Повторное использование компонентов** React позволяет создавать компоненты, которые можно повторно использовать в разных частях приложения.
4. **Удобство разработки** React предоставляет удобный и интуитивно понятный API для создания пользовательского интерфейса.
5. **Поддержка сообщества** React имеет большое сообщество разработчиков, которые активно работают над улучшением фреймворка и созданием новых инструментов и библиотек.



▼ Что такое условный рендеринг (Conditional Rendering)? Как его выполнить?

Условный рендеринг - это способ отображения компонентов или элементов в зависимости от определенных условий или состояний приложения. Это позволяет создавать динамические пользовательские интерфейсы, которые изменяются в соответствии с различными сценариями. Например, если пользователь аутентифицирован, отображается один компонент, а если нет, то другой.

Оператор `if` и цикл `for` не являются выражениями в JavaScript, поэтому их нельзя непосредственно использовать в JSX. Вместо этого, вы можете окружить ими JSX-код.

Т.к. JSX не поддерживает **if else** то реализуется через логические условия или тернарным оператором.

```jsx
const Component = ({ name, address }) => (
  <>
    <h2>{name}</h2>
    {address && <p>{address}</p>}
  </>
);

const Component = ({ name, address }) => (
  <>
    <h2>{name}</h2>
    {address
      ? <p>{address}</p>
      : <p>{'Address is not available'}</p>
    }
  </>
);
```

▼ Что такое компонент-переключатель (Switching Component)?

Компонент-переключатель (Switching Component) - это компонент в React, который отображает один из нескольких компонентов в зависимости от определенного условия. Он может использоваться для выполнения условного рендеринга на основе нескольких условий.

Основная идея, это структура в виде объекта в котором по ключам доступны компоненты. Получая пропсы, он вытягивает из них значения этого ключа, после чего по этому значению возвращает нужный компонент которому с помощью спред оператора передаются все пропсы.

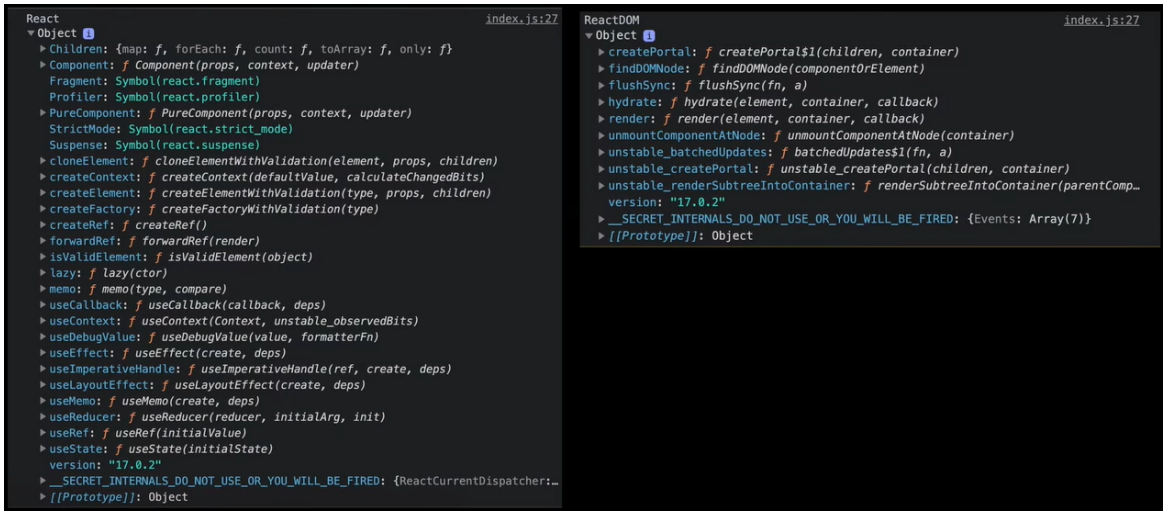
```jsx
import HomePage from './HomePage';
import AboutPage from './AboutPage';
import ServicesPage from './ServicesPage';
import ContactPage from './ContactPage';

const PAGES = {
  home: HomePage,
  about: AboutPage,
  services: ServicesPage,
  contact: ContactPage
}

const Page = (props) => {
  const Handler = PAGES[props.page] || ContactPage;

  return <Handler {...props} />;
}

Page.propTypes = {
  page: PropTypes.oneOf(Object.keys(PAGES)).isRequired,
}
```

▼ Разница между `React` и `ReactDOM`?

React - это библиотека JavaScript, которая позволяет создавать компоненты пользовательского интерфейса и управлять ими. Она предоставляет инструменты для создания компонентов, обработки событий и управления состоянием приложения. Т.е. предоставляет универсальные методы для создания компонентов.

ReactDOM - это библиотека, которая используется для рендеринга React-компонентов в браузере. Она предоставляет методы для взаимодействия с DOM-элементами и управления ими.

Таким образом, React используется для создания компонентов пользовательского интерфейса, а ReactDOM - для их отображения в браузере.

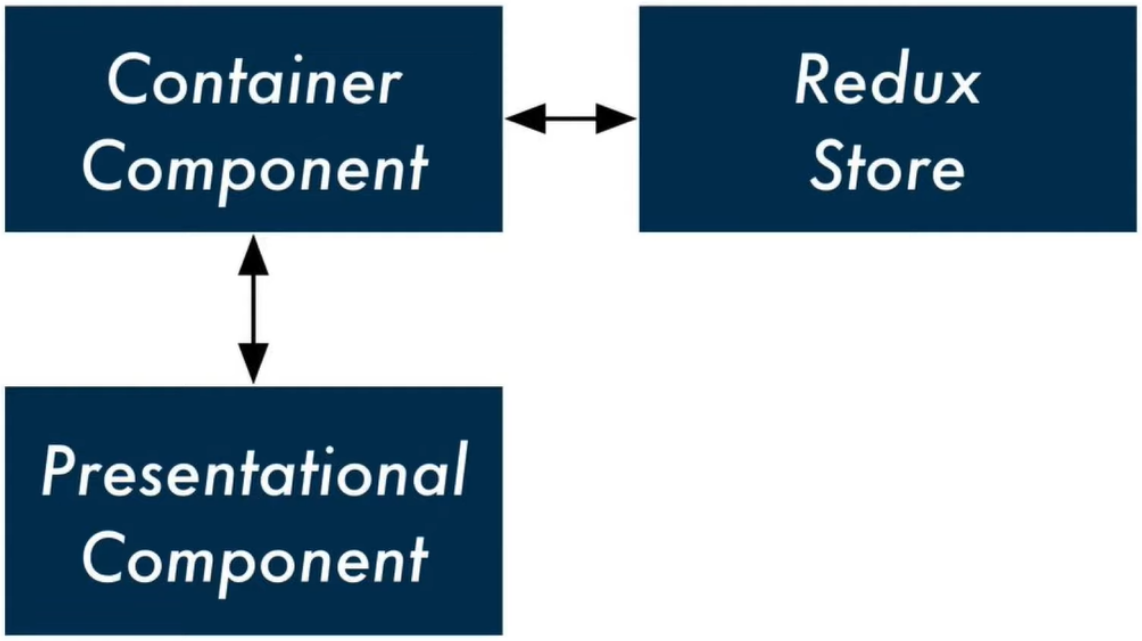


▼ Разница между компонентом и контейнером?

Компонент - это независимая часть пользовательского интерфейса, которая может быть использована повторно в разных частях приложения. Компонент может содержать HTML-разметку, CSS-стили и JavaScript-логику. Он может быть функциональным или классовым.

Контейнер - это компонент, который используется для управления другими компонентами и передачи им данных. Контейнер может содержать логику для работы с данными, обработки событий и управления состоянием приложения. Он может быть классовым компонентом или функцией высшего порядка (HOC). Сами по себе как правило не генерируют контент, этим занимаются их дочерние элементы.

Таким образом, компоненты являются независимыми частями пользовательского интерфейса, а контейнеры используются для управления другими компонентами и передачи им данных.



▼ Как React обрабатывает, или ограничивает использование пропсов определенного типа?

PropTypes - это библиотека, которая позволяет определить типы пропсов для компонентов. Она предоставляет набор функций-валидаторов для различных типов данных (строки, числа, объекты и т.д.), которые могут быть использованы для проверки типов передаваемых пропсов. Если пропс имеет неверный тип, PropTypes выдаст предупреждение в консоли браузера.

.defaultProps - дефолтные пропсы

```jsx
import React from 'react';
import PropTypes from 'prop-types';

export const Person = ({ firstName, lastName, country }) => (
  <>
    <h2>{firstName} {lastName}</h2>
    {country && <p>Country: {country}</p>}
  </>
);

Person.propTypes = {
  firstName: PropTypes.string.isRequered,
  lastName: PropTypes.string.isRequered,
  country: PropTypes.string,
};

Person.defaultProps = {
  country: '',
};
```

▼ Что такое строгий режим в React? Его преимущества?

Строгий режим (Strict Mode) в React - это режим работы, который позволяет находить распространенные ошибки в компонентах на ранних стадиях разработки.

Преимущества строгого режима в React:

1. Компоненты будут перерендериваться дополнительно для поиска ошибок, вызванных нечистым рендерингом.
2. Компоненты будут перезапускать эффекты дополнительно, чтобы найти ошибки, вызванные отсутствием очистки эффектов.
3. Компоненты будут проверяться на использование устаревших API.

Предупреждения

Не существует способа отказаться от строгого режима внутри дерева, обернутого в `<StrictMode>`. Это дает уверенность в том, что все компоненты внутри `<StrictMode>` проверены.

Строгий режим включает дополнительные проверки, предназначенные только для разработчиков, для всего дерева компонентов внутри компонента `<StrictMode>`. Эти проверки помогут вам найти распространенные ошибки в ваших компонентах на ранних стадиях разработки.

Чтобы включить режим Strict Mode для всего приложения, оберните корневой компонент компонентом `<StrictMode>` при его рендеринге:

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(
    <StrictMode>
        <App />
    </StrictMode>
);
```

▼ Что такое «бурение пропсов» (Prop Drilling)? Как его избежать?

«Бурение пропсов» (Prop Drilling) - это ситуация, когда компоненты передают пропсы через несколько уровней вложенности, чтобы достичь компонента, который их действительно нуждается.

Чтобы избежать «бурения пропсов», можно использовать следующие подходы:

1. Контекст (Context) - это механизм, который позволяет передавать данные глубоко вложенным компонентам без необходимости явно передавать пропсы через каждый компонент. Однако, не следует злоупотреблять контекстом, так как это может усложнить понимание кода и ухудшить его читаемость.
2. Стейт менеджеры: Redux, MobX, Zustand
3. Redux - это библиотека для управления состоянием приложения, которая позволяет хранить данные в глобальном хранилище и получать к ним доступ из любого компонента приложения. Redux также предоставляет механизмы для обновления данных и уведомления компонентов об изменениях.

4. Использование более высокоуровневых компонентов (Higher-Order Components) - это специальные компоненты, которые принимают другие компоненты в качестве пропсов и могут обрабатывать их, не передавая их дальше по иерархии компонентов.

```
// Props drilling:
const Component = () => (
  <LevelOne title="simple title" />
);

const LevelThree = ({ title }) => <h1>{title}</h1>;
const LevelTwo = ({ title }) => <LevelThree title={title} />;
const LevelOne = ({ title }) => <LevelTwo title={title} />;


// Using hook & context:
// (1) File context.js
export const titleContext = React.createContext('simple title');

// (2) File LevelThree.js
import { titleContext } from './context';
import React, { useContext } from 'react';

export const LevelThree = () => {
  const title = useContext(titleContext);
  return <h1>{title}</h1>;
};
```

▼ Что такое «опрос» (Polling)? Как его реализовать в React?

«Опрос» (Polling) - это процесс периодического запроса данных с сервера для обновления информации на странице без необходимости перезагрузки страницы.

В React опрос можно реализовать с помощью использования функции **setInterval()** и **useEffect** для периодического вызова функции, которая отправляет запрос на сервер и обновляет состояние компонента с полученными данными.

```jsx
import React, { useState, useEffect } from 'react'

const IntervalExample = () => {
  const [data, setData] = useState(null);

  const getItems = () => {
    fetch('api-url')
        .then(result => result.json())
        .then(result => setData(result));
  };

  useEffect(() => {
    const interval = setInterval(() => getItems(), 1000);
    return () => clearInterval(interval);
  }, []);

  return (
    <div className="App">
      <Component data={data} />
    </div>
  );
};
```

▼ Разница между элементом и компонентом?

Компонент (Component) - это функция или класс, которая описывает, как должна выглядеть и вести себя часть интерфейса. Компонент может состоять из одного или нескольких элементов, а также содержать логику для обработки событий и изменения состояния. Имеет состояние, принимает пропсы, имеет методы жизненного цикла и может использовать хуки. Возвращает элементы react.

Элемент (Element) - это часть виртуального DOM, которая представляет собой описание компонента или HTML-тега. Элементы не могут быть напрямую отображены на странице, они используются для создания компонентов.

Представляет собой JavaScript объект с четырьмя основными свойствами:

- **type** - html тег или ссылка на компонент

- **key** - используется для определения уникальности элемента в списке.
- **ref** - ссылка на узел DOMили экземпляр компонента react
- **props** - объект со свойствами

React элементы

- Элементы представляют собой простые объекты JavaScript, которые описывают, что вы хотите увидеть на экране.
- Они являются наименьшими строительными блоками React-приложений.
- Элементы представляют собой виртуальное представление DOM-узлов или других компонентов, которые будут отображены на экране.
- Элементы можно создавать с помощью функции React.createElement() или JSX-синтаксиса.

Пример элемента:

```
const element = <div>Hello, world!</div>;
```

React компоненты

- Компоненты - это функции или классы, которые принимают пропсы (props) и возвращают элементы (или другие компоненты), описывающие, как компонент должен выглядеть на экране.
- Они используются для создания переиспользуемых и изолированных блоков пользовательского интерфейса.
- Компоненты могут быть функциональными (функциональные компоненты) или классовыми (классовые компоненты).
- Функциональные компоненты представляют собой простые функции JavaScript, которые принимают пропсы в качестве аргумента и возвращают элементы.
- Классовые компоненты - это классы JavaScript, расширяющие React.Component, и они содержат метод render(), который

возвращает элементы.

Пример функционального компонента:

```
const MyComponent = (props) => {
    return <div>Hello, {props.name}!</div>;
};
```

▼ Что такое `ReactDOMServer`?

ReactDOMServer - это часть библиотеки React, которая предоставляет утилиты для рендеринга компонентов React на стороне сервера (Server-Side Rendering) и для работы с HTML вне браузера. Эти утилиты позволяют разработчикам использовать React для генерации HTML на сервере, а также для работы с HTML в окружениях, где нет доступа к браузерным DOM API, таких как Node.js.

Пример на CommonJS:

```
const React = require('react');
const ReactDOMServer = require('react-dom/server');

// Определение простого компонента React
const MyComponent = (props) => {
  return <div>Hello, {props.name}!</div>;
};

// Рендеринг компонента в виде строки HTML с помощью render
const htmlString = ReactDOMServer.renderToString(<MyCompone

console.log(htmlString);
```

После выполнения этого кода в переменной `htmlString` будет содержаться следующая строка HTML:

```
<div data-reactroot="">Hello, John!</div>
```

▼ Что такое предохранители (Error Boundaries)?

Предохранители — это классовые компоненты React, которые отлавливают ошибки JavaScript в любом месте деревьев их дочерних

компонентов, сохраняют их в журнале ошибок и выводят запасной UI вместо рухнувшего дерева компонентов.

Предохранители не поймают ошибки в:

- обработчиках событий (подробнее);
- асинхронном коде (например колбэках из **setTimeout** или **requestAnimationFrame**);
- серверном рендеринге (Server-side rendering);
- самом предохранителе (а не в его дочерних компонентах).

```
// Компонент предохранителя
class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  // Обработка ошибок
  componentDidCatch(error, info) {
    console.error('Error caught by Error Boundary:', error,
    this.setState({ hasError: true });
  }

  render() {
    if (this.state.hasError) {
      // Возвращаем запасной UI в случае ошибки
      return <div>Something went wrong.</div>;
    }
    // Рендерим дочерние компоненты
    return this.props.children;
  }
}

// Компонент, который иногда выбрасывает ошибку
class ComponentWithError extends Component {
  render() {
    // Генерируем ошибку в случае, если пропс shouldThrowEr
```

```
    if (this.props.shouldThrowError) {
      throw new Error('An error occurred in ComponentWithEr
    }
    // Возвращаем нормальный рендер компонента
    return <div>No error occurred in ComponentWithError.</d
  }
}

// Родительский компонент, который использует предохранител
class ParentComponent extends Component {
  render() {
    return (
      <div>
        <h1>Parent Component</h1>
        <ErrorBoundary>
          {/* Компонент с ошибкой */}
          <ComponentWithError shouldThrowError={true} />
        </ErrorBoundary>
        <hr />
        <h2>After Error Boundary</h2>
        <ComponentWithError shouldThrowError={false} />
      </div>
    );
  }
}

export default ParentComponent;
```

▼ Что такое «ленивая» (Lazy) функция?

«Ленивая» (Lazy) функция в React - это функция, которая загружает компоненты только тогда, когда они действительно нужны, а не заранее. Это позволяет ускорить начальную загрузку приложения и уменьшить объем передаваемого кода. В React для создания «ленивых» функций используется функция `**lazy()**`, которая принимает функцию импорта компонента, который будет загружаться только при необходимости.

Пример использования ленивой загрузки компонента в React с помощью `React.lazy()` и `Suspense`:

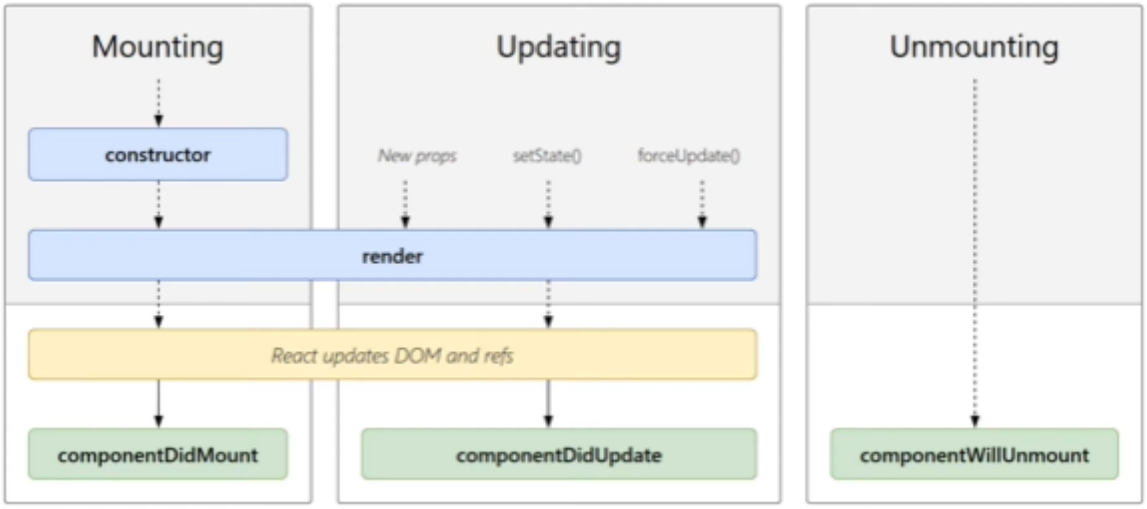
```jsx
import React, { Suspense } from 'react';

// Определение компонента, который будет загружен лениво
const LazyComponent = React.lazy(() => import('./LazyCompor

function App() {
  return (
    <div>
      <h1>Пример ленивой загрузки компонента в React</h1>
      {/* Используем Suspense для отображения заглушки во в
      <Suspense fallback={<div>Loading...</div>}>
        {/* Обертка для лениво загружаемого компонента */}
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

▼ Разница между рендерингом и монтированием?

Рендеринг (rendering) - это процесс создания виртуального дерева DOM на основе компонентов React. Во время рендеринга React создает виртуальное дерево DOM, которое представляет собой копию реального дерева DOM, но без фактического изменения элементов на странице.

Повторный рендеринг получает инфу об уже смонтированном компоненте.

Монтирование (mounting) - это процесс добавления виртуального дерева DOM на страницу. Когда React создает виртуальное дерево DOM, он еще не добавлен на страницу, т.е. первый рендеринг компонента и внедрение его в DOM . Этот процесс называется монтированием. Во время монтирования React создает реальный дерево DOM и добавляет его на страницу.



▼ Что такое `children`?

Подробнее тут

Children (дети) в React - это свойство компонента, которое позволяет передавать другие компоненты или элементы внутрь текущего компонента в качестве дочерних элементов. Children могут быть переданы через JSX или через пропсы.

```
function ParentComponent({ children }) {
  return (
    <div className="parent">
      {/* Дочерние компоненты, переданные через props.child
      {children}
    </div>
  );
}

function App() {
  return (
    <ParentComponent>
      <ChildComponent /> {/* Передаем компонент в качестве
    </ParentComponent>
  );
}
```

В этом примере `ParentComponent` принимает `children` через свои **props** и просто отображает их внутри себя. `App` использует `ParentComponent` и передает ему `ChildComponent` в качестве `children`.

`children` может представлять собой любой валидный JSX:

- другие компоненты
- строки
- числа
- выражения и так далее.

Все, что расположено между открывающим и закрывающим тегами компонента, считается его `children`.

▼ Что такое события указателя (Pointer Events)?

События указателя (Pointer Events) - это API, которое позволяет отслеживать действия пользователя с помощью указателя на устройствах сенсорных экранов, мышей и стилусов. Оно предоставляет единый интерфейс для обработки событий указателя независимо от типа устройства, что упрощает разработку кросс-платформенных приложений.

Некоторые из основных событий указателя включают:

1. `pointerdown`: происходит при нажатии на указатель (например, нажатие мыши или касание пальцем).
2. `pointerup`: происходит при отпускании указателя.
3. `pointermove`: происходит при движении указателя.
4. `pointerover`: происходит, когда указатель перемещается над элементом.
5. `pointerout`: происходит, когда указатель покидает элемент.

Event	On Event Handler	Description
pointerover	onpointerover	Fired when a pointer is moved into an element's hit test boundaries.
pointerenter	onpointerenter	Fired when a pointer is moved into the hit test boundaries of an element or one of its descendants, including as a result of a pointerdown event from a device that does not support hover (see `pointerdown`).
pointerdown	onpointerdown	Fired when a pointer becomes *active buttons state*.
pointermove	onpointermove	Fired when a pointer changes coordinates. This event is also used if the change in pointer state can not be reported by other events.
pointerup	onpointerup	Fired when a pointer is no longer *active buttons state*.
pointercancel	onpointercancel	A browser fires this event if it concludes the pointer will no longer be able to generate events (for example the related device is deactivated).
pointerout	onpointerout	Fired for several reasons including: pointer is moved out of the hit test boundaries of an element; firing the pointerup event for a device that does not support hover (see pointerup); after firing the `pointercancel` event (see `pointercancel`); when a pen stylus leaves the hover range detectable by the digitizer.
pointerleave	onpointerleave	Fired when a pointer is moved out of the hit test boundaries of an element. For pen devices, this event is fired when the stylus leaves the hover range detectable by the digitizer.
gotpointercapture	ongotpointercapture	Fired when an element receives pointer capture.
lostpointercapture	onlostpointercapture	Fired after pointer capture is released for a pointer.

▼ Что такое инверсия наследования (Inheritance Inversion)?

Инверсия наследования (inheritance inversion) - это `HOC`, который выглядит следующим образом:

```
const inheritanceInversionHOC = (WrappedComponent) => {
  return class extends WrappedComponent {
    render() {
      return super.render()
    }
  }
}
```

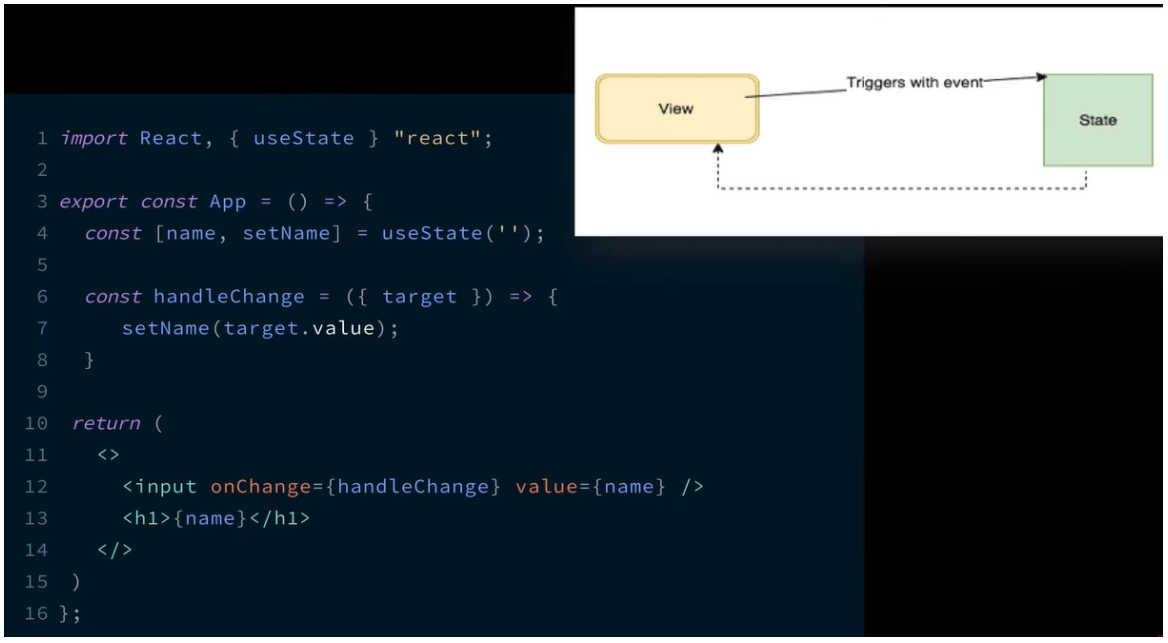
Мы возвращаем класс, *расширяющий* `WrappedComponent`. Данная техника называется инверсией наследования, поскольку вместо расширения некоторого класса-усилителя (enhancer) с помощью `WrappedComponent`, последний сам пассивно расширяется. Отношения между ними напоминают *инверсию*.

▼ Как в React реализовать двустороннее связывание данных?

Короче это управляемый компонент, когда у нас стейт меняет то, что мы видим на экране, а изменение того, что мы видим на экране меняет стейт.

Двустороннее связывание данных означает следующее:

- Данные, которые мы изменяем в представлении, обновляют состояние.
- Данные в состоянии обновляют представление.




```jsx
import React, { useState } "react";

export const App = () => {
  const [name, setName] = useState('');

  const handleChange = ({ target }) => {
     setName(target.value);
  }

 return (
   <>
     <input onChange={handleChange} value={name} />
     <h1>{name}</h1>
   </>
 )
};
```

▼ Разница между классовым и функциональным компонентами?

Классовые компоненты - это компоненты, наследуются от базового класса **React.Component** и имеют свое состояние (**state**) и методы жизненного цикла (lifecycle methods). Они обычно использовались для более сложной логики и управления состоянием.

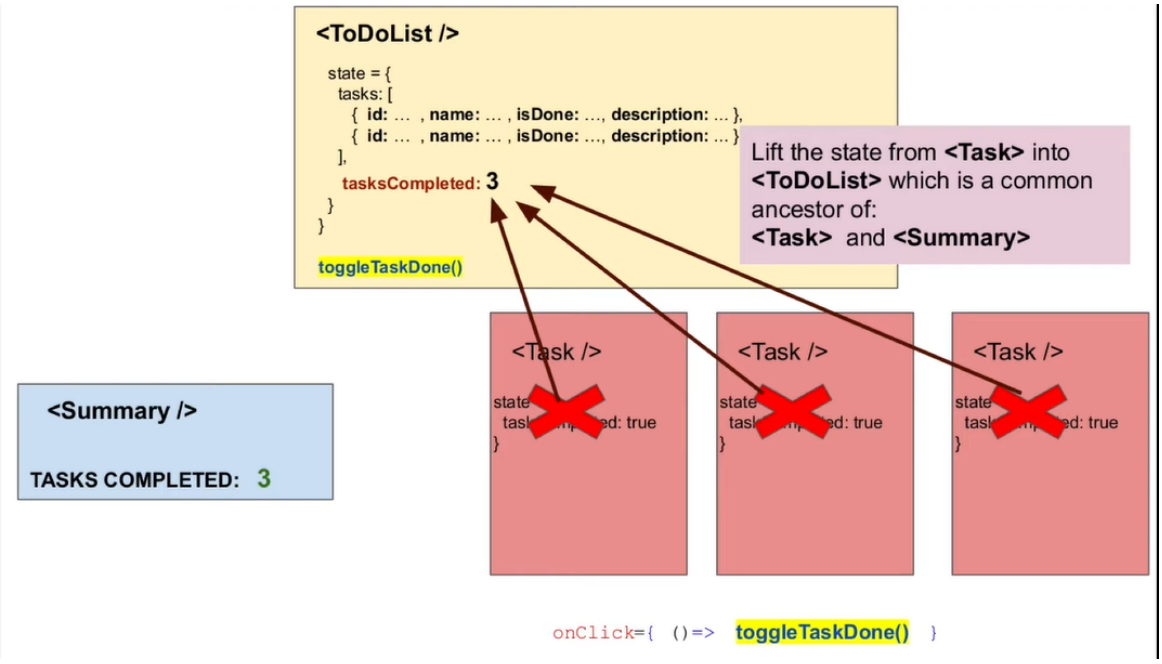
Функциональные компоненты - это компоненты, являются простыми функциями, принимающими пропсы и возвращающими JSX. Они не имеют состояния и методов жизненного цикла, но могут использоваться для простых компонентов без необходимости управления состоянием.

В React 16.8 были добавлены хуки (hooks), которые позволяют функциональным компонентам иметь состояние и использовать методы жизненного цикла, делая их более мощными и гибкими.

▼ Что такое поднятие состояния вверх (Lifting State Up)?

Поднятие состояния вверх (Lifting State Up) - это паттерн в React, который позволяет передавать состояние от дочерних компонентов к родительским. Это делается путем определения общего состояния на

более высоком уровне и передачи обработчиков для его изменения в дочерние компоненты через пропсы.



▼ Разница между `createElement()` и `cloneElement()`?

Метод **createElement()** используется для создания новых элементов.

Принимает:

- **Тип элемента:** HTML-тег (например, `'div'`, `'span'`, `'h1'`, и т. д.), или функциональный компонент. Если функциональный компонент, то это должна быть ссылка на компонент, а не его вызов.
- **Объект свойств:** Свойства представлены в виде пар ключ-значение, где ключ - это имя свойства, а значение - его значение
- **Дочерние элементы:** Дополнительные аргументы метода, которые являются дочерними элементами создаваемого элемента. Это может быть любое количество дочерних элементов, включая строки, числа, другие React-элементы или массивы React-элементов.

```
const element = React.createElement(
  'div', // Тип элемента (div)
  { className: 'example' }, // Объект свойств
  'Hello, world!' // Дочерние элементы (children)
);
```

Метод **cloneElement()** используется для клонирования существующих элементов React и изменения их свойств.

Принимает:

- **Элемент для клонирования (element)**: React-элемент, который вы хотите скопировать.
- **Новые свойства (props)**: Объект, содержащий свойства, которые вы хотите применить к скопированному элементу. Эти свойства заменят любые существующие свойства в исходном элементе.
- **Дочерние элементы (children)** (необязательно): Дополнительные аргументы, которые будут добавлены как дочерние элементы к скопированному элементу.

```
const existingElement = <div className="old-class">Hello, w

const clonedElement = React.cloneElement(
  existingElement, // Элемент для клонирования
  { className: 'new-class' } // Новые свойства
);

// Результат: <div className="new-class">Hello, world!</div
```

▼ Как реализовать однократное выполнение операции при начальном рендеринге?

- componentDidMount()
- useEffect(() ⇒ {}, [])

▼ Как отрендерить HTML код в React-компоненте?

Как использовать innerHTML в React?

Использование dangerouslySetInnerHTML: Если вам нужно вставить HTML-код, который вы получили из внешнего источника (например, из API), вы можете использовать `dangerouslySetInnerHTML`. Однако, будьте осторожны с этим подходом, так как он может быть уязвим для атак XSS, если HTML-код не безопасен.

```jsx
import React from 'react';

function MyComponent() {
  const htmlCode = '<p>This is HTML code.</p>';
  return <div dangerouslySetInnerHTML={{ __html: htmlCode }}
}

export default MyComponent;
```

▼ Зачем в `setState()` нужно передавать функцию?

Обновление состояния в классовых компонентах это асинхронная операция, и оно обновляется не сразу после вызова **setState**.

В **setState()** нужно передавать функцию с предыдущим состоянием в качестве аргумента, чтобы гарантировать корректность обновления состояния компонента в случаях, когда новое состояние зависит от предыдущего состояния.

▼ Для чего предназначен метод `registerServiceWorker()` в React?

Метод registerServiceWorker() в React предназначен для регистрации сервис-воркера, который позволяет создавать оффлайн-версии приложений и улучшать производительность загрузки страниц.

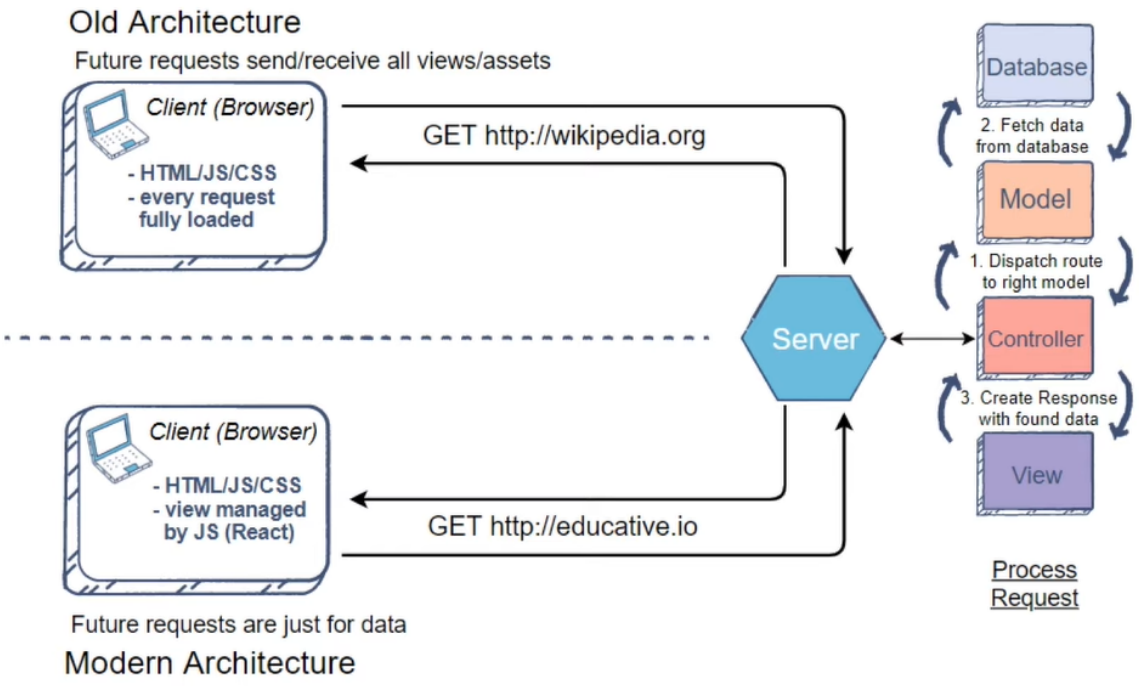
Сервис воркер - это веб API позволяющее записывать файлы приложения в кэш и возвращать их из него при отсутствии соединения с интернетом перехватывая запросы и проверяя соединение с интернетом.

```jsx
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

▼ Чем React Router отличается от обычной маршрутизации?

React Router представляет собой библиотеку для маршрутизации в React-приложениях, которая позволяет создавать динамические маршруты без перезагрузки всей страницы, при этом запоминая историю и позволяя по ней перемещаться.

Она отличается от обычной маршрутизации тем, что работает на клиентской стороне и не требует обращения к серверу для получения новой страницы т.к. в SPA всего один html файл внутри которого вместо переходов по страницам просто перерисовываются компоненты.



▼ Какие хуки были добавлены в React Router версии 5?

1. **useHistory** - позволяет получить доступ к объекту истории браузера, который можно использовать для перехода на другие страницы или изменения URL используя методы push, go, и т.д.
2. **useLocation** - позволяет получить текущий URL и другую информацию о текущем маршруте. Дает доступ к параметрам запроса и к полному адресу.
3. **useParams** - позволяет получить параметры маршрута, переданные в URL, например, идентификатор товара или пользователя.
4. **useRouteMatch** - позволяет получить информацию о текущем маршруте и его совпадении с заданным шаблоном маршрута. Нужен для построения внутренних маршрутов.

```
// useHistory();
const history = useHistory();
history.push('/profile');

// useLocation();
const location = useLocation();
const currentPath = location.pathname;
const searchParams = new URLSearchParams(location.search);

// useParams();
const { name } = useParams();

// useRouteMatch();
const match = useRouteMatch();
<Route path={`${match.url}/login`}></Route>;
```

▼ Как передавать пропсы в React Router?

Можно передавать пропсы через атрибут `component` или `render` компонента `<Route>`:

Это называется рендер пропсы, когда компонент передается через пропсы, который потом и отрисовывается.

```
import React from 'react';
import { Route } from 'react-router-dom';
import MyComponent from './MyComponent';

const App = () => {
  const myProp = 'Hello, world!';

  return (
    <div>
      <Route path="/example" component={() => <MyComponent
    </div>
  );
};

export default App;
```

▼ Что такое Reselect и как он работает?

Посмотри

Reselect - это библиотека для управления состоянием в приложениях, основанных на **Redux**, в экосистеме **React**. Это инструмент, который позволяет создавать селекторы (selectors) для извлечения данных из хранилища Redux с помощью мемоизации.

Селектор вычисляется повторно только при изменении его аргументов.

Если изменяются не связанные с ним данные, то селектор не вычисляется.

Reselect предоставляет функцию **createSelector**, которая позволяет создавать мемоизированные селекторы.

```
import { createSelector } from 'reselect';

// Селекторы для извлечения данных из состояния Redux
const getUsers = state => state.users;
const getFilter = state => state.filter;

// Создание селектора с помощью createSelector
const getUsersFiltered = createSelector(
  [getUsers, getFilter], // массив селекторов, данные котор
  (users, filter) => {
    // Логика фильтрации пользователей
    return users.filter(user => user.name.includes(filter))
  }
);

// Пример использования в компоненте React
import React from 'react';
import { useSelector } from 'react-redux';

const UserList = () => {
  // Использование селектора для получения отфильтрованного
  const filteredUsers = useSelector(getUsersFiltered);

  return (
    <div>
```

```
        <h2>User List</h2>
        <ul>
          {filteredUsers.map(user => (
            <li key={user.id}>{user.name}</li>
          ))}
        </ul>
      </div>
    );
};

export default UserList;
```

▼ Назовите основную цель React Fiber?

Основная цель React Fiber - улучшение производительности и возможностей рендеринга в React приложениях. Он представляет собой переработанную архитектуру внутреннего движка React.

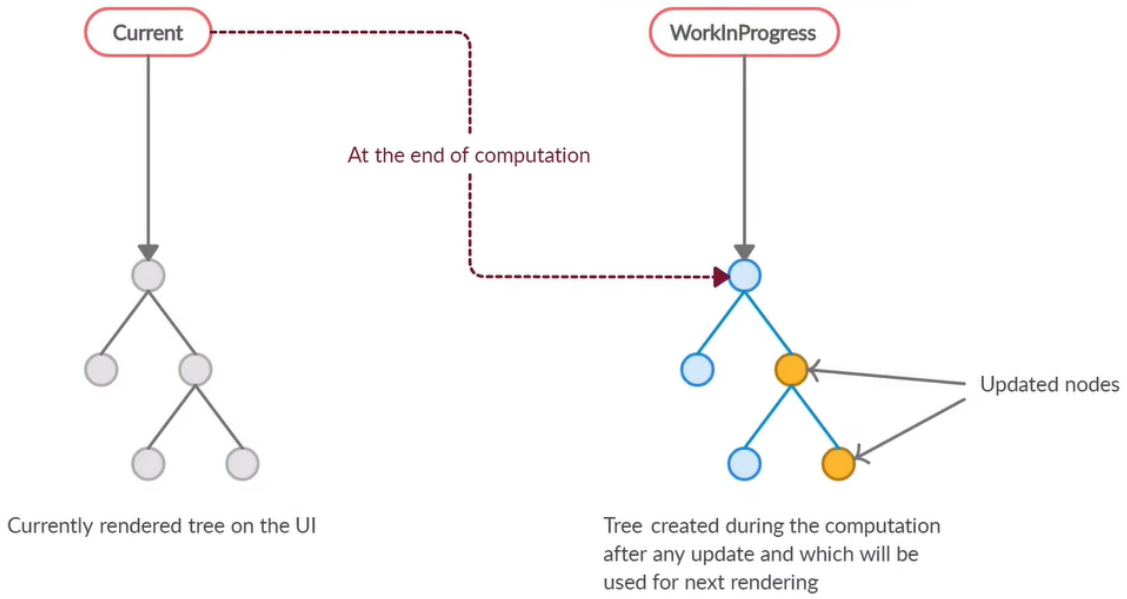
- приостанавливать работу и возвращаться к ней позже.
- назначить приоритет разным типам работы.
- переиспользовать результат ранее выполненной работы.
- прервать работу, если результат ее больше не нужен.
- Возможность разбить прерываемую работу на куски.
- Возможность расставлять приоритеты, перебазировать и повторно использовать незавершенную работу.
- Возможность переключения между родителями и детьми для поддержки layout в react
- Возможность возврата нескольких элементов из render.
- Улучшенная обработка ошибок или error boundaries

Его особенностью является инкрементный рендеринг - в контексте React, инкрементный рендеринг обычно означает, что обновления в виртуальном DOM и реальном DOM применяются частично и поэтапно, в отличие от того, чтобы ожидать, пока все обновления будут готовы, и применять их сразу.

Цель повысить производительность для:

- анимаций

- макетов
- жестов



▼ Какие типы данных может возвращать метод `render`?

Метод **render** в React компоненте должен возвращать только один корневой элемент, который может быть одним из следующих типов данных:

1. React элемент (React elements)
2. Массивы (Arrays)
3. Фрагменты
4. Portals
5. Строки и числа (Strings and numbers) - в виде текстовых узлов.
6. Booleans и null (Booleans and null) - ничего не отображается, но используются для условного рендеринга.

▼ Разница между `memo` и `useMemo`?

`memo` - это HOC, позволяет пропустить повторный рендеринг компонента, если его пропсы неизменны.

`useMemo` - это хук React, позволяющий кэшировать результат вычисления между повторными рендерингами если массив зависимостей не изменился.

▼ Что такое синтетические события (SyntheticEvent) в React?

Посмотри

Это обертка над нативными событиями, созданная React, для обеспечения кросс-браузерной совместимости и упрощения работы событий в веб-приложениях.

Раньше одни и те же события в браузерах могли называться по разному, React решил эту проблему, предоставив унифицированный интерфейс для работы с событиями.

Примеры синтетических событий в React:

- `onClick`: событие клика
- `onChange`: событие изменения значения
- `onSubmit`: событие отправки формы
- `onKeyDown`, `onKeyUp`, `onKeyPress`: события нажатия клавиш
- `onFocus`, `onBlur`: события фокуса и потери фокуса и т.д.

▼ Является ли React реактивным?

НЕТ!!!

▼ Техники оптимизации React?

1. Использование **shouldComponentUpdate()** для оптимизации рендеринга компонентов.
2. Использование **PureComponent**, который автоматически реализует **shouldComponentUpdate()** для компонента.
3. Использование мемоизации с помощью **React.memo()** для кеширования результатов функций и избегания повторных вычислений.
4. Использование ключей (**keys**) для ускорения процесса рендеринга списков.
5. Использование ленивой загрузки **(lazy loading)** для отложенной загрузки компонентов и уменьшения начальной нагрузки на приложение.

6. Оптимизация изображений и других ресурсов, чтобы уменьшить размер файлов и время загрузки страницы.
7. Использование библиотеки **React Profiler** для анализа производительности компонентов и выявления узких мест.

- Немутабельная структура данных
- Function/Stateless Components и React.PureComponent
- shouldComponentUpdate
- React.Fragments
- Избегать онлайновых функций в render
- Throttling and Debouncing
- Использовать "key" для отрисовки элементов массива
- Использование Reselect в Redux
- Мемоизация компонентов (memo, useMemo, useCallback)
- Использование Web Workers
- Разбиение на “чанки”

▼ Лучшие практики безопасности в React?

1. Использование библиотеки **PropTypes** для проверки типов пропсов и избегания ошибок типизации.
2. Использование контролируемых компонентов для управления состоянием форм и избегания уязвимостей XSS.
3. Использование библиотеки **React Helmet** для управления метаданными страницы и избегания уязвимостей SEO, а так же для добавления безопасных заголовков HTTP, таких как Content-Security-Policy и X-Frame-Options.
4. Использование библиотеки **React Router** для управления маршрутизацией и избегания уязвимостей **CSRF**.
5. Использование библиотеки React Context для передачи данных между компонентами без необходимости использования пропсов и избегания уязвимостей XSS.
6. Использование библиотеки React-Intl для локализации приложения и избегания уязвимостей XSS, связанных с неправильным форматированием текста.

- Дефолтная XSS защита
- Санитайзинг (dangerouslySetInnerHTML, dompurify)
- dangerouslySetInnerHTML
- SSR: renderToString, renderToStaticMarkup, ReactDOMServer
- URL проверка
- Проверка уязвимостей в зависимостях
- JSON.stringify + RegEx
- Использование стабильной версии React
- Конфигурирование линтера (Husky)

▼ Что триггерит перерендер компонента?

1. Перерендер родителя
2. Изменения в контексте
3. Изменения стейта через функцию setState
4. Изменение пропсов, но ТОЛЬКО в том случае, если:
    - shouldComponentUpdate
    - используется React.memo

State management

▼ Redux

▼ Что такое Flux?

Flux - это паттерн управления состоянием приложения, самой популярной реализацией **Flux** является **Redux**, особенно в контексте приложений на **React**. Представляет собой методологию организации кода, направленную на управление данными в приложении и обеспечение предсказуемости состояния интерфейса.

Flux не является конкретной реализацией или библиотекой, это скорее набор общих принципов, которые могут быть реализованы различными способами в зависимости от потребностей приложения.

Основные принципы Flux:

1. **Однонаправленный поток данных**: Данные движутся в одном направлении через приложение. Это означает, что данные изменяются только в одном месте и все изменения проходят через один центральный управляющий объект (обычно называемый "стор"), что обеспечивает предсказуемость и легкость отладки.
2. **Дискретные источники данных (Actions)**: Изменения данных в системе могут быть инициированы только с помощью объектов - действий (**Actions**). Компоненты пользовательского интерфейса не могут изменять данные напрямую, они могут только создавать действия, которые оповещают о необходимости изменения данных.
3. **Централизованное управление (Store)**: Состояние приложения хранится в централизованных объектах данных, называемых "хранилищами" (**Stores**). Компоненты могут подписываться на изменения в хранилищах и обновлять свое состояние в соответствии с изменениями данных.

4. **Изменения данных через Dispatcher**: Для обработки действий и обновления состояния хранилищ **Flux** использует объект, называемый "диспетчер" (**Dispatcher**). Он централизует обработку действий и гарантирует последовательность обновлений состояния.

▼ Что такое Redux? Ключевые принципы Redux?

Подробнее тут

Redux — это библиотека для управления состоянием (**state management**) в приложениях, написанных на языке **JavaScript** или других языках, компилирующихся в **JavaScript**. Он часто используется в связке с библиотеками пользовательского интерфейса, такими как **React**, но может использоваться и независимо. Она является реализацией паттерна **Flux**.

3 принципа Redux:

1. Единственный источник правды! Состояние всего приложения сохранено в дереве объектов внутри одного хранилища (**Store**).
2. Состояние только для чтения! Единственный способ изменять состояние — это применять **Action** обьект, который описывает, что случится.
3. Мутации написаны как чистые функции.

Основные идеи Redux включают в себя:

- **Store** - объект, который хранит всю информацию о состоянии приложения и является единственным источником истины.
- **Actions -** объекты, которые описывают изменения состояния.
- **Reducers -** чистые функции, которые принимают предыдущее состояние и действие, и возвращают новое состояние.
- **Dispatch** - метод, который отправляет действие в **store** для изменения состояния.
- **Middleware -** функции, которые могут изменять или перехватывать действия перед тем, как они достигнут **reducers**.

- **Immutable data -** данные, которые не могут быть изменены напрямую, а только созданы новые копии с изменениями.
- **Предсказуемость (Predictability)** - **Redux** позволяет легко отслеживать изменения состояния и предсказывать, как они будут влиять на приложение.

▼ Разница между Redux и Flux?

Подробнее тут

- в редакс неизменяемое состояние
- в редакс есть мидлвары
- в редакс один стор, а во флакс много сторов
- во флакс есть диспетчер в котором указаны зависимости между сторами

В **Flux** несколько хранилищ поддерживаются для каждого приложения как одноэлементного объекта. С другой стороны, в Redux обычно поддерживается одно хранилище для каждого приложения (но вы можете создать больше, если требуется для сложных сценариев).

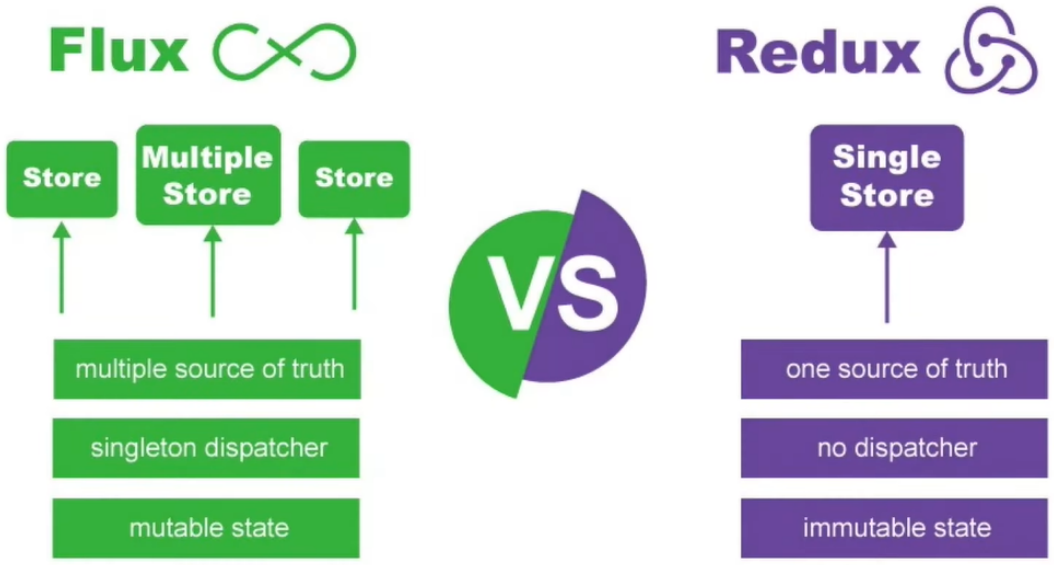
В **Flux** логика того, что должно быть выполнено на основе действия, записывается в хранилище, а в Redux - в редюсере. Может быть несколько редьюсеров, но их нужно объединить в один корневой редьюсер.

Другое самое большое отличие заключается в том, что **Redux** поддерживает неизменное состояние, в то время как состояние **Flux** является изменяемым. Наличие истории состояний в **Redux** помогает легко реализовать определенные задачи, такие как отмена и повтор.

Масштабируемость Flux достигается за счет гибкости добавления необходимого количества хранилищ при добавлении дополнительных функций. Redux справляется с этим, поддерживая единое общее состояние для всех компонентов.

Таким образом, основные отличия между **Flux** и **Redux** заключаются в том, что **Flux** – это архитектурный шаблон, в то время как **Redux** – это библиотека, которая строится на идеях **Flux**. **Redux** также

добавляет новые возможности и использует иммутабельные объекты для управления состоянием.



▼ Что такое «единственный источник истины» (Single Source of Truth)?

Принцип "единственного источника истины" - это концепция, используемая в контексте управления состоянием данных в приложениях. Он подразумевает, что в приложении должен существовать только один централизованный источник данных, который хранит всю необходимую информацию о состоянии приложения.

В контексте веб-разработки, особенно с применением библиотек управления состоянием, таких как Redux, принцип SSOT означает, что все состояние приложения должно храниться в единственном объекте (Redux store). Этот объект состояния является единственной истинной версией состояния приложения, и все компоненты приложения должны обращаться к этому источнику данных для доступа к состоянию и для его обновления.

▼ Как создать и использовать Store?

```
import { createStore } from 'redux';

const store = createStore(counterReducer);
```

```
export default store;
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store'; // Импортируем созданное хр
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

▼ Что такое редьюсер (Reducer)?

Редьюсер (Reducer) - это чистая функция в **Redux**, которая принимает предыдущее состояние и действие (**action**), и возвращает новое состояние. Действие может содержать полезные данные, **payload**.

Она не изменяет предыдущее состояние, а создает новый объект состояния. Редюсеры используются для обновления состояния в **store** в ответ на действия, отправленные через метод **dispatch**.

Основная задача редьюсера состоит в том, чтобы обрабатывать действия, которые происходят в приложении, и вносить изменения в состояние на основе этих действий.

```
const initialState = {
  count: 0
};

// Пример редьюсера для обновления состояния счетчика
function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
```

```
      return { ...state, count: state.count + 1 };
    case 'DECREMENT':
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
}
```

▼ Разница между React State и Redux State?

React State - это локальное состояние компонента, которое управляется и изменяется только внутри этого компонента. Оно может быть использовано для хранения и обновления данных, которые не нужно передавать между компонентами. Или передавать состояние через пропсы нижестоящим компонентам.

Redux State - это глобальное состояние, которое хранится в store и может быть использовано в любом компоненте приложения. Оно позволяет управлять состоянием приложения централизованно и предотвращает передачу данных через пропсы между компонентами.

▼ Как выглядит поток данных в Redux-приложении?

Подробнее тут

Супер наглядно

Событие на странице создает обьект **action** → **action** передается в **dispatch** → **dispatch** передает **action** в **middleware** (опционально) → **action** попадает в функцию **reducer** → **reducer** возвращает новый обьект состояния и передает его в **state** → **state** обновляется → перересовывается **UI**

▼ Плюсы и минусы Redux?

Плюсы Redux:

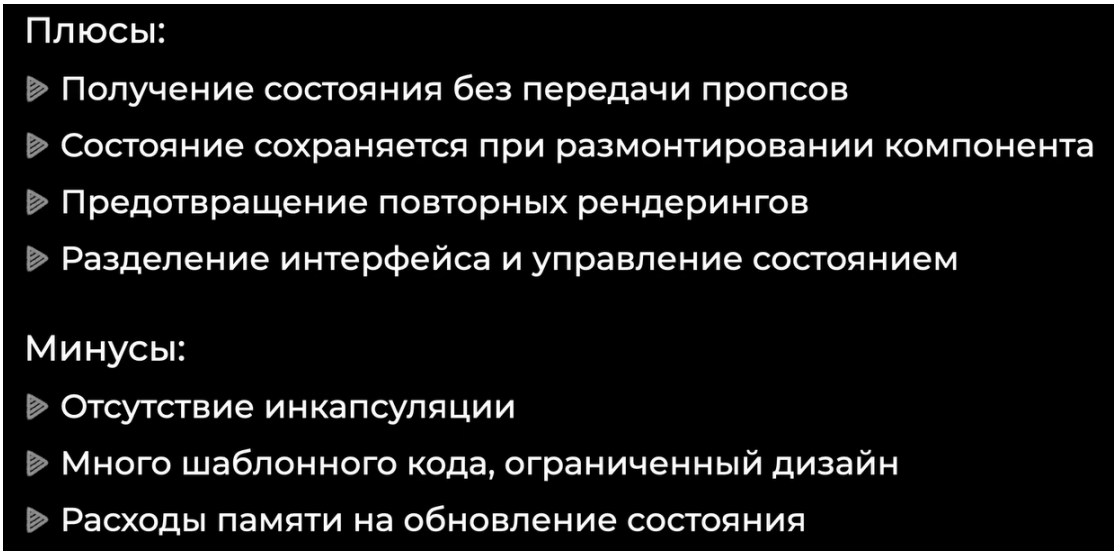
- **Предсказуемость состояния:** Состояние приложения хранится в единственном объекте (хранилище), что обеспечивает предсказуемость и прозрачность изменений.
- **Управление сложным состоянием:** Redux упрощает управление сложным состоянием приложения, особенно когда у приложения

большое количество компонентов, которые должны иметь доступ к одним и тем же данным.

- **Удобство тестирования:** Отделение состояния от компонентов делает код более тестируемым, что облегчает написание и поддержку модульных тестов.
- **Отличная поддержка для разработки инструментов:** Redux имеет развитую экосистему инструментов для отладки, мониторинга и разработки.
- Возможность использования **middleware** для обработки асинхронных действий.

Минусы Redux:

- Дополнительный слой абстракции может усложнить разработку для начинающих разработчиков.
- Некоторые задачи могут быть решены проще без использования Redux.
- Необходимость создания множества файлов для реализации Redux-архитектуры.



▼ Что такое Middleware

Middleware — это функция, которая расширяет функциональность **dispatch**. Эта функция принимает **store** (но не весь, а только **dispatch** и **getState.**) и возвращает функцию, которая принимает **next** (ссылку на следующую **middleware** или **редьюсер**) и возвращает еще одну функцию, которая принимает **action**.

Как создать:

```
function foo(store) {
    return function(next) {
        return function(action) {
            // код мидлвары
            return next(action)
        }
    }
}


const foo = store => next => action => {
    return next(action)
}

// В функцию передается не весь стейт, а только dispatch
// т.е. можно написать так:
const foo = ({dispatch, getState}) => dispatch => action
    return dispatch(action)
}
```

Процесс выглядит следующим образом:

- **Middleware** получает **store**.
- **Middleware** возвращает функцию, которая получает **next → next** возвращает следующую **middleware** в цепочке, если такая существует. Если **next** является последней **middleware** в цепочке, она возвращает функцию редьюсера.
- Внутри этой функции, **middleware** возвращает еще одну функцию, которая получает **action**.
- В этой функции **middleware** может выполнить свою логику до или после передачи действия следующему **middleware** или **редьюсеру**, используя **next**(**action**).

▼ Redux Toolkit

Дока

Redux Toolkit - это пакет, облегчающий работу с `Redux`. Он создан с целью уменьшения сложности кода Redux, улучшения производительности и снижения объема написанного кода.

Решает три главных проблемы Redux:

- Слишком сложная настройка хранилища (**store**)
- Для того, чтобы заставить `Redux` делать что-то полезное, приходится использовать дополнительные пакеты
- Слишком много шаблонного кода (boilerplate)

Основные возможности и функции:

▼ configureStore

Функция, которая упрощает создание хранилища Redux. Автоматически включает в себя множество полезных настроек, таких как **middleware** для обработки асинхронных действий, поддержка **devtools** и другие.

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';
import usersReducer from './usersReducer';

const store = configureStore({
  reducer: {
    counter: counterReducer,
    users: usersReducer,
  }
});

export default store;
```

Также допускается передавать объект с несколькими редукторами, в этом случае `configureStore()` автоматически вызывает `combineReducers()`.

Обратите внимание, что это работает только для одного уровня вложенности. Если требуются вложенные редукторы, придется вызывать `combineReducers()` самостоятельно.

▼ createSlice

Позволяет создавать "срезы" (**slices**) состояния и связанные с ними редюсеры и действия в едином месте. Это упрощает структурирование кода и уменьшает объем повторяющегося кода.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    increment: state => {
      state.value += 1;
    },
    decrement: state => {
      state.value -= 1;
    },
    reset: state => {
      state.value = 0;
    }
  }
});

export const { increment, decrement, reset } = counterSl
export default counterSlice.reducers;
```

▼ createAction

Используется для создания действий (**actions**) Redux. Автоматически создает объекты действий, что упрощает создание и управление действиями.

```
import { createAction } from '@reduxjs/toolkit';

// Создание действия для инкремента счетчика
export const increment = createAction('counter/increment

// Создание действия для декремента счетчика
export const decrement = createAction('counter/decrement

// Создание действия для сброса счетчика
export const reset = createAction('counter/reset');

// Дополнительные параметры могут быть переданы вторым а
export const addValue = createAction('counter/addValue',
  return {
    payload: value
  };
});
```

В примере выше создаются действия для инкремента, декремента и сброса счетчика. Дополнительно, показано создание действия `addValue`, которое принимает параметр `value` и передает его в поле `payload`.

Функция `createAction` принимает два аргумента:

1. **Type (тип действия)**: Это строка, которая представляет тип действия. Он обычно записывается в формате `domain/eventName`, где `domain` - это область (или сфера) применения действия, а `eventName` - это название самого действия.
2. **Payload Creator (создатель payload)** (необязательно): Это функция, которая принимает аргументы и возвращает объект, который становится `payload` в созданном действии. В случае если этот аргумент не передан, действие будет создано с пустым `payload`.

▼ createReducer

Предоставляет удобный способ создания редьюсеров (**reducers**) с помощью функции `createReducer`, которая позволяет определять редьюсеры в виде таблицы сопоставления.

```
import { createReducer } from '@reduxjs/toolkit';
import { increment, decrement, reset } from './actions';

// Начальное состояние счетчика
const initialState = {
  value: 0
};

// Создание редьюсера
const counterReducer = createReducer(initialState, {
  // Обработчик для инкремента счетчика
  [increment.type]: (state) => {
    state.value += 1;
  },

  // Обработчик для декремента счетчика
  [decrement.type]: (state) => {
    state.value -= 1;
  },

  // Обработчик для сброса счетчика
  [reset.type]: (state) => {
    state.value = 0;
  }
});

export default counterReducer;
```

▼ createAsyncThunk

Функция `createAsyncThunk` позволяет создавать асинхронные действия, которые упрощают работу с асинхронными запросами и обновлением состояния в Redux.

```
import { createAsyncThunk } from '@reduxjs/toolkit';
import axios from 'axios';

export const fetchUserById = createAsyncThunk(
  'users/fetchByIdStatus',
  async (userId, thunkAPI) => {
    const response = await axios.get(`https://jsonplaceh
    return response.data;
  }
);
```

▼ createEntityAdapter

Предоставляет удобный способ управления данными в нормализованном формате. Он автоматически создает селекторы (**selectors**) и редьюсеры для работы с данными.

```
import { createEntityAdapter, createSlice } from '@redux

const usersAdapter = createEntityAdapter();

const usersSlice = createSlice({
  name: 'users',
  initialState: usersAdapter.getInitialState(),
  reducers: {
    userAdded: usersAdapter.addOne,
    usersLoaded: usersAdapter.setAll,
    userUpdated: usersAdapter.updateOne,
    userRemoved: usersAdapter.removeOne
  }
});

export const {
  selectById: selectUserById,
  selectAll: selectAllUsers
} = usersAdapter.getSelectors(state => state.users);
```

```
export const { userAdded, usersLoaded, userUpdated, user
export default usersSlice.reducer;
```

React Query

Дока

Видео

React Query - библиотека для получения, кэширования, синхронизации и обновления "серверного" состояния в React-приложениях.

Плюсы:

- Упрощение запросов.
- Встроенное кэширование данных.
- Автоматическая обработка ошибок.
- Автоматические запросы на сервер для актуализации данных (например через определенный интервал времени или при фокусировании на приложении).
- Инвалидация кэша и его автоматическое обновление. Инвалидация - это процесс обозначения кэшированных данных как устаревших или недействительных.

Основные хуки:

useQuery

Позволяет выполнить запрос к серверу для получения данных.

```
const { data } = useQuery('todos', fetchTodos, {});
// queryKey === ['todos']
```

Принимает:

- Строку, представляющую ключ запроса. При передаче строки в качестве ключа запроса, она преобразуется в массив с единственным элементом.
- Возможно передать массив в качестве ключа запроса. Это позволяет создавать более сложные ключи запроса, которые могут включать в себя несколько параметров.

```
const { data, isLoading, isError } = useQuery(['userDa
```

В этом примере ключ запроса представлен в виде массива, где первый элемент - строка `'userData'`, а второй элемент - идентификатор пользователя `userId`. Такой подход позволяет создавать уникальные ключи запроса, которые зависят от различных параметров.

- Функцию или промис, выполняющий запрос к серверу.
- Обьект опций (не обязательный):

```
{
  cacheTime,
  enabled,
  initialData,
  initialDataUpdatedAt
  isDataEqual,
  keepPreviousData,
  notifyOnChangeProps,
  notifyOnChangePropsExclusions,
  onError,
  onSettled,
  onSuccess,
  queryKeyHashFn,
  refetchInterval,
  refetchIntervalInBackground,
  refetchOnMount,
  refetchOnReconnect,
  refetchOnWindowFocus,
  retry,
  retryOnMount,
  retryDelay,
  select,
  staleTime,
  structuralSharing,
  suspense,
```

```
  useErrorBoundary
}
```

Возвращает:

Обьект содержащий всю информацию о запросе:

```
{
  data,
  dataUpdatedAt,
  error,
  errorUpdatedAt,
  failureCount,
  isError,
  isFetched,
  isFetchedAfterMount,
  isFetching,
  isIdle,
  isLoading,
  isLoadingError,
  isPlaceholderData,
  isPreviousData,
  isRefetchError,
  isStale,
  isSuccess,
  refetch,
  remove,
  status,
}
```

- `status` - возможные значения:
  - `idle` - возможно только при инициализации запроса с `enabled: false` и отсутствии начальных данных
  - `loading` - запрос находится на стадии выполнения
  - `error` - выполнение запроса завершилось ошибкой
  - `success` - выполнение запроса завершилось успешно. Соответствующее свойство `data` - это данные, полученные

из успешного запроса, или, при `enabled: false`, начальные данные

- производные логические значения из переменной `status`:
  - `isIdle`
  - `isLoading`
  - `isError`
  - `isSuccess`
  - `data` - последние успешно разрешенные данные для запроса (по умолчанию `undefined`)
  - `error` - объект ошибки для запроса (по умолчанию `null`)
  - `isFetching` - `true`, если запрос находится в процессе выполнения (включая выполнение в фоновом режиме)
  - `refetch` - функция для ручного выполнения повторного запроса `remove` - функция для удаления запроса из кэша

▼ useQueries

Использоваться для выполнения нескольких запросов.

```
const results = useQueries([
  { queryKey: ['post', 1], queryFn: fetchPost },
  { queryKey: ['post', 2], queryFn: fetchPost },
]);
```

Принимает:

- Массив объектов с настройками запроса

Возвращает:

- Массив результатов выполнения этих запросов.

▼ useInfiniteQuery

Позволяет загружать данные пачками (пагинация) и автоматически управлять загрузкой следующих страниц данных.

Полезен, когда необходимо загружать большие объемы данных частями.

```
const { data, fetchNextPage, isFetchingNextPage, hasNe
    getNextPageParam: (lastPage) => lastPage.nextCurs
});
```

Принимает:

- Строку, представляющую ключ запроса. При передаче строки в качестве ключа запроса, она преобразуется в массив с единственным элементом.
- Возможно передать массив в качестве ключа запроса. Это позволяет создавать более сложные ключи запроса, которые могут включать в себя несколько параметров.
- Вторым аргументом принимает функцию или промис, выполняющий запрос к серверу для получения следующей порции данных.
- Третьим аргументом принимает объект с настройками, включая опции пагинации и другие параметры, такие как `getNextPageParam`, который определяет, как извлечь параметр следующей страницы из данных текущей страницы.

Возвращает:

- **data** - Значение данных, полученных в результате запроса.
- **data.pages** - Массив, содержащий все страницы
- **data.pageParams** - Массив, содержащий все параметры страницы
- **fetchNextPage** - Функция, которая выполняет запрос для получения следующей порции данных.
- **fetchPreviousPage** - аналогична функции `fetchNextPage`, но в отношении предыдущей страницы
- **hasPreviousPage** - Имеется ли предыдущая страница для получения
- **hasNextPage** - Имеется ли следующая страница для получения

- **isFetchingNextPage**: Булевое значение, указывающее на то, выполняется ли в данный момент запрос для получения следующей страницы данных.

▼ useMutation

Используется для выполнения мутаций на сервере, таких как создание, обновление или удаление данных. “Аналог” POST/PUT/DELTE.

```
const { mutate, isLoading, isError } = useMutation(mut
  mutationKey,
  onError,
  onMutate,
  onSettled,
  onSuccess,
  useErrorBoundary,
});
```

Принимает:

- **mutationFn** - функция, выполняющая асинхронную задачу и возвращающая промис.
- Объект **variables** для мутирования.

Возвращает:

- Обьект, содержащий всю информацию о запросе:

```
{
  data,
  error,
  isError,
  isIdle,
  isLoading,
  isPaused,
  isSuccess,
  mutate,
  mutateAsync,
```

```
  reset,
  status,
}
```

- `mutate` - функция мутации, которая может вызываться с переменными для запуска мутации и, опционально, для перезаписи настроек, переданных в `useMutation`
- `mutateAsync` - функция, аналогичная `mutate`, но возвращающая промис
- `status` - возможные значения:
  - `idle`
  - `loading`
  - `error`
  - `success`
- `data` - последние успешно разрешенные данные для запроса (по умолчанию `undefined`)
- `error` - объект ошибки для запроса (по умолчанию `null`)
- `reset` - функция для очистки внутреннего состояния мутации

▼ useQueryClient

Предоставляет доступ к экземпляру **QueryClient**, который содержит методы для управления кэшем запросов.

```
import { QueryClient } from 'react-query';

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: Infinity,
    },
  },
});

await queryClient.prefetchQuery('posts', fetchPosts);
```

Принимает обьект настроек:

Настройки:

- `queryCache` - кэш запроса, к которому подключен данный клиент
- `mutationCache` - кэш мутации, к которому подключен данный клиент
- `defaultOptions` - настройки по умочанию для всех запросов и мутаций

▼ RTK Query

Видео

Видео

Дока

RTK Query - это необязательное дополнение, включенное в пакет Redux Toolkit, и его функциональность построена поверх других API в Redux Toolkit. Мощный инструмент для запросов и кэширования данных.

Черпает вдохновение из других инструментов, которые стали пионерами в решениях для извлечения данных, таких как Apollo Client, React Query, Urql и SWR, но добавляет уникальный подход к дизайну своего API

Ключевые возможности RTK Query:

1. **Управление состоянием данных**: RTK Query позволяет управлять состоянием данных в приложении, автоматически обновляя его при изменении данных на сервере и обеспечивая консистентность данных в приложении.
2. **Кэширование данных**: Он автоматически кэширует данные, полученные из удаленного сервера, и предоставляет механизмы для управления кэшем, включая инвалидацию кэшированных данных.
3. **Генерация кода на основе схемы API**: RTK Query может генерировать код запросов и селекторов (**selectors**) на основе

схемы API, что упрощает использование API и снижает вероятность ошибок.

4. **Оптимистичные обновления**: Он поддерживает оптимистичные обновления, позволяя мгновенно обновлять интерфейс в ответ на действия пользователя и асинхронно отправлять запрос на сервер для фактического обновления данных.
5. **Интеграция с Redux Toolkit**: RTK Query интегрирован с Redux Toolkit, что обеспечивает совместимость с другими функциями и инструментами Redux Toolkit.
6. **Автоматическое обновление данных**: Он автоматически обновляет данные в кэше при выполнении запросов к серверу и обновлении данных на сервере.

Основные API:

▼ createApi

Функция, которая создает экземпляр API с набором эндпоинтов и конфигурацией. Она принимает объект конфигурации, включающий в себя определения эндпоинтов и опции для настройки поведения API.

Принимает:

- Объект конфигурации, который определяет настройки API и его эндпоинты.

Возвращает:

Объект API, который содержит следующие свойства:

- `reducerPath`: Путь к редьюсеру API.
- `endpoints`: Объект, содержащий функции хуков для выполнения запросов и мутаций.
- `reducer`: Редьюсер, который содержит логику обработки действий, связанных с запросами API.
- `middleware`: Middleware, который необходимо добавить к Redux для работы с RTK Query.

```
import { createApi, fetchBaseQuery } from '@reduxjs/tool

const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  endpoints: (builder) => ({
    getUsers: builder.query({
      query: () => 'users'
    }),
    createUser: builder.mutation({
      query: (newUser) => ({
        url: 'users',
        method: 'POST',
        body: newUser
      })
    })
  })
});

const { useGetUsersQuery, useCreateUserMutation } = api;
export { useGetUsersQuery, useCreateUserMutation };
```