#### #1 Повторение перед ускорением

Не хочу никого напугать. Хотя… как я могу напугать того, кто отважно прошел целых два месяца в ударном темпе и решил идти дальше? Добро пожаловать во вторую часть, чемпион! 💪🏼

В этой части мы изучим **JS еще детальнее**, попробуем **новые структуры данных**, научимся **приручать рекурсию** и поймем **асинхронность в JavaScript изнутри**. Кроме детального изучения языка мы возьмемся за **NPM**, установим пакеты сторонних разработчиков и **соберем это** все. Добавим типизации с помощью **TypeScript**, добавим **принципы разработки** и еще лучше поймем **протокол HTTP** и **как работает сеть**. Этих знаний нам точно хватит чтобы написать **собственный чат с общением в реальном времени**. Да, все это мы сделаем в этой части роадмапа, менее чем за 2 месяца.

Уверен что вы готовы, но перед тем как начать все новое мы вернемся к старому на 1 день. **Ваша задача**: повторить все темы, по которым у вас возникали вопросы, пройтись по всем своим заметкам еще раз. Потом открыть список задач прошлой части и убедиться что вы ничего из этого не пропустили, не выкинули из головы и действительно поняли. В этой части у вас не будет недели или двух на раскачку, начнем колбасить с первых дней и закончим только через 8 недель 😤

Будет **интересно**, **весело** и **сложно**

#2 Коллекции и уникальные коллекции

Вы будете редко использовать Set() в повседневной работе и еще реже Map() большинство людей вообще не используют эти структуры, предпочитая им объекты с массивами :(

Но, знать про Map(), а уж тем более про Set() вам точно нужно и вы будете приятно удивлены когда **перепишите список сохраненных городов в вашей погоде с Array на Set**. Перед тем как начать, прочитайте теорию

## Map

Map - это коллекция пар ключ-значение, где ключи и значения могут быть любого типа данных. Map предоставляет мощный способ хранения данных, где каждый ключ может быть связан с уникальным значением.

#### Декларативные методы

Насколько понятные что не нуждаются в пояснениях

const myMap = new Map();

myMap.set("key1", "value1");

myMap.set("key2", "value2");

console.log(myMap.get("key1")); // "value1"

myMap.delete("key2");

console.log(myMap.has("key2")); // false

#### Объекты в качестве ключей

Map позволяет использовать объекты в качестве ключей, что может быть полезно, если вы хотите связать объекты с уникальными значениями или хотите сохранить множество объектов и быстро их искать. В обычных объектах ключи должны быть строками, поэтому, если вы хотите использовать объект в качестве ключа, вам придется преобразовывать его в строку или использовать Symbol.

const user1 = { name: "Alice" };

const user2 = { name: "Bob" };

const userSettings = new Map();

userSettings.set(user1, { theme: "dark", fontSize: 14 });

userSettings.set(user2, { theme: "light", fontSize: 16 });

console.log(userSettings.get(user1)); // выведет { theme: "dark", fontSize: 14 }

#### Гарантия порядка элементов

Map гарантирует, что элементы хранятся в порядке их добавления в коллекцию. Это позволяет использовать Map для хранения итерируемых объектов, сохраняя порядок элементов, что может быть полезно во многих случаях.

const myMap = new Map();

myMap.set("key1", "value1");

myMap.set("key2", "value2");

myMap.set("key3", "value3");

for (const [key, value] of myMap) {

console.log(`${key} = ${value}`);

}

// выведет:

// key1 = value1

// key2 = value2

// key3 = value3

## Set

Set - это коллекция уникальных значений любого типа данных. Set предоставляет простой способ хранения уникальных значений без необходимости проверки наличия дубликатов вручную.

#### Фильтрация дубликатов из массива

// Создаем массив с дублирующими значениями

const myArray = [1, 2, 3, 2, 4, 1];

// Создаем новый Set из массива, удаляя дубликаты

const mySet = new Set(myArray);

// Создаем массив из Set

const filteredArray = Array.from(mySet);

// Выводим результат

console.log(filteredArray); // выведет [1, 2, 3, 4]

#### Удобство работы с уникальными значениями

const favorites = new Set();

favorites.add("JavaScript");

favorites.add("HTML");

favorites.add("CSS");

console.log(favorites.has("JavaScript")); // выведет true

console.log(favorites.has("PHP")); // выведет false

favorites.delete("CSS");

for (const favorite of favorites) {

console.log(favorite);

}

#### Полезно знать

JSON.stringify() не может принять Set, вам нужно “развернуть” его в массив, например:

const favoriteCities = new Set();

JSON.stringify([...favoriteCities]);

Map и Set представляют собой мощные и удобные структуры данных, которые могут быть использованы для хранения коллекций уникальных значений. Они предоставляют специальные методы для добавления, удаления и получения элементов, а также гарантируют порядок хранения элементов и уникальность значений.

Использование Map и Set может значительно улучшить производительность и удобство работы с коллекциями данных, поэтому я рекомендую использовать эти структуры данных вместо объектов и массивов, когда это возможно.

О том почему и как это влияет на производительность вы узнаете позже, это связано со структурами данных.

## Ваша задача:

Добавить Set в “Погоду”, как минимум для списка избранных локаций

#### #3 Ключи, значения и спред

Иногда хочется перебрать значения объекта прямо как массив. Еще чаще нужно получить список ключей из объекта. А иногда, и то, и другое (чтобы превратить объект в Map()) Примеров таких задач может быть достаточно и вы уже сталкивались с некоторыми из них.

### Ключи, значения и записи объектов

Ключи, значения и записи объектов - это основные элементы JavaScript. Ключи и значения в объектах используются для хранения данных. Ключ - это строка, которая используется для идентификации значения, а значение может быть любым типом данных.

#### Keys

Ключи объектов могут быть получены с помощью метода **Object.keys()**. Этот метод возвращает массив ключей, которые есть в объекте.

const person = {

name: "John",

age: 30,

city: "New York"

};

const keys = Object.keys(person);

console.log(keys); // ["name", "age", "city"]

#### Values

Значения объектов могут быть получены с помощью метода **Object.values()**. Этот метод возвращает массив значений, которые есть в объекте.

const person = {

name: "John",

age: 30,

city: "New York"

};

const values = Object.values(person);

console.log(values); // ["John", 30, "New York"]

#### Entries

Записи объектов могут быть получены с помощью метода **Object.entries()**. Этот метод возвращает массив записей, который состоит из массивов, каждый из которых содержит ключ и значение.

const person = {

name: "John",

age: 30,

city: "New York"

};

const entries = Object.entries(person);

console.log(entries); // [["name", "John"], ["age", 30], ["city", "New York"]]

### Rest parameters и spread operator

Rest parameters и spread operator - это новые функциональности в ES6, которые значительно упрощают написание кода.

#### Rest parameters

Rest parameters позволяют функции принимать переменное количество аргументов. Аргументы передаются в виде массива.

function sum(...numbers) {

return numbers.reduce((total, num) => total + num, 0);

}

console.log(sum(1, 2, 3, 4, 5)); // 15

#### Spread operator

Spread operator позволяет "распаковать" массив или объект в местах, где ожидается список аргументов или элементов.

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

const arr3 = [...arr1, ...arr2];

console.log(arr3); // [1, 2, 3, 4, 5, 6]

Также, spread operator может использоваться для создания нового объекта на основе существующего.

const person = {

name: "John",

age: 30,

city: "New York"

};

const newPerson = {...person, city: "San Francisco"};

console.log(newPerson); // { name: "John", age: 30, city: "San Francisco" }

В этом примере мы использовали spread operator **...person**, чтобы скопировать все свойства объекта **person** в новый объект **newPerson**. Затем мы использовали синтаксис **key: value** для замены значения свойства **city**.

### Примеры использования

Рассмотрим примеры, в которых мы можем использовать keys, values, entries, rest parameters и spread operator.

#### Копирование объекта

Мы можем использовать spread operator для копирования объекта в новый объект.

const person = {

name: "John",

age: 30,

city: "New York"

};

const newPerson = {...person};

console.log(newPerson); // { name: "John", age: 30, city: "New York" }

#### Объединение массивов

Мы можем использовать spread operator для объединения двух или более массивов.

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

const arr3 = [...arr1, ...arr2];

console.log(arr3); // [1, 2, 3, 4, 5, 6]

#### Функция с rest parameters

Мы можем использовать rest parameters для создания функции, которая принимает переменное количество аргументов.

function sum(...numbers) {

return numbers.reduce((total, num) => total + num, 0);

}

console.log(sum(1, 2, 3, 4, 5)); // 15

#### Использование entries для итерации по объекту

Мы можем использовать метод **Object.entries()** для итерации по объекту.

const person = {

name: "John",

age: 30,

city: "New York"

};

for (const [key, value] of Object.entries(person)) {

console.log(`${key}: ${value}`);

}

// name: John

// age: 30

// city: New York

#### Использование values для получения среднего значения

Мы можем использовать метод **Object.values()** для получения массива значений объекта, а затем использовать **reduce()** для получения среднего значения.

const ages = {

john: 30,

jane: 25,

bob: 40

};

const ageArray = Object.values(ages);

const averageAge = ageArray.reduce((total, age) => total + age) / ageArray.length;

console.log(averageAge); // 31.666666666666668

### Заключение

Keys, values, entries, rest parameters и spread operator - это важные концепции в JavaScript, которые помогают упростить написание кода и работу с данными. Надеемся, что эта статья помогла вам лучше понять эти концепции и научиться использовать их в своих проектах. Если у вас есть какие-либо вопросы или замечания, не стесняйтесь задавать их!

#### #4 Конструктор

Простая для понимания и одна из главных тем в JS научит вас создавать свои собственные объекты с набором методов и полей

Прочитайте про: [Конструктор](https://learn.javascript.ru/constructor-new) Попробуйте применить это в “TODO” (создание объекта задачи) или “Погоде” (создание объекта города в избранном)

Ваш код будет выглядеть примерно так:

const tasks = new Task(’переписать TODO с конструктором’)

**А есть ли в этом хоть какой-то смысл?**

Да, определенно есть. И не только когда вы пробуете писать в ООП стиле. Даже на примере такого простого объекта как Tasks можно увидеть что для новой задачи вам не нужно передавать ее статус, а это уменьшает кол-во ошибок при создании новых задач, а еще уменьшает кол-во кода. **Уменьшение кол-ва кода - лучший способ для уменьшения кол-ва багов.** Абсолютно серьезно, чем больше шансов для ошибок - тем больше ошибок.

#### #5 Дата

Кто-то из вас уже использовал класс Data в задаче с погодой. Сегодня стоит ближе познакомиться с тем, как работать с датой в JS

Этот класс содержит достаточно много самого разного функционала, запоминать или учить наизусть который, не имеет никакого смысла. Достаточно знать что он есть [Ознакомьтесь с Датой](https://learn.javascript.ru/date) и порефакторите ваш код (в любой из программ)

#### #6 Другие ошибки

Как вы уже знаете, когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Этим занимается встроенный класс Error, который в случае возникновения ошибки в ходе выполнения программы создает и выбрасывает объект ошибки. Мы пока ничего не знаем про классы и то, как они работают, но при этом - в сегодняшнем материале все примеры будут построены именно на них. Не придавайте большого значения тонкостям работы, просто посмотрите на то, как и для чего они могут [использоваться](https://learn.javascript.ru/custom-errors).

А зачем нам это надо?

1. Чтобы все проверки были в одном месте. что-то необходимо поменять, добавить, убавить - меняем это в классе, а не шерстим весь проект, в поисках условного if (![user.name](http://user.name/))
2. Лучше один раз написать корректный класс с проверкой ошибок и потом его только клонировать, чем каждый раз писать логику проверки заново, что в свою очередь может вызвать еще больше ошибок.

После прочтения материала попробуйте попрактиковаться и написать собственный вариант обработки одной из “ошибок”, которая может возникнуть в ходе выполнения вашего кода

#### #7 Выходной

Думаю что эта неделя была простой на фоне предыдущих месяцев. Вы снова заглянули в свои программы, что-то в них сделали лучше, что-то сломали и снова починили.

Это было отличной практикой

#### #8 Структуры данных

Программирование состоит из двух сущностей: Алгоритмы и Структуры данных (данные) Программа - это данные на входе, определенный алгоритм и данные на выходе. Да да, именно этим и занимаются программы день изо дня, преобразуют одни данные в другие по заранее определенному алгоритму. Выбор правильной структуры данных - очень важный навык, от которого зависит как эффективность, так и читаемость/поддерживание вашей программы. Вы уже знаете что для массива из уникальных элементов лучше взять коллекцию Set(), чем обычный массив - иначе придется писать проверки (не говоря про то, что скорость потеряется)

Тоже самое касается и алгоритмов, но о них мы поговорим чуть позже, а сейчас про данные

Посветите этот день следующим структурам ”Массивы”, “Связные списки”, “Очередь”, “Стек”, “Дерево” (только этим и никаким другим) *Можете читать статьи, листать книги и даже смотреть видео. Этот день ваш*

#### #9 Синхронно записанная асинхронность

**async/await** - это синтаксический сахар над промисами, который позволяет написать асинхронный код в более линейном стиле. Код, который выглядит как синхронный, но при этом выполняется асинхронно.

Пример, чтобы проиллюстрировать разницу между промисами и **async/await**. Функция **fetchData()** получает данные из удаленного API и возвращает промис:

function fetchData() {

return fetch('https://api.example.com/data')

.then(response => response.json());

}

С использованием промисов, мы могли бы написать код следующим образом:

fetchData()

.then(data => {

return fetch(`https://api.example.com/another-data/${data.id}`);

})

.then(console.log)

.catch(console.log)

Этот код получает данные из первого API, затем получает данные из другого API с использованием результата первого запроса. Но написание цепочки промисов может стать неудобным и трудоемким, особенно если нужно выполнить множество асинхронных операций последовательно.

С использованием **async/await**, мы можем переписать этот код следующим образом:

Перепишем сначала функцию fetchData:

async function fetchData() {

const response = await fetch(`https://api.example.com/data`);

return await response.json()

}

А теперь все остальное:

async function getData() {

try {

const data = await fetchData();

const responseAnotherData = await fetch(`https://api.example.com/another-data${data.id}`);

const anotherData = await responseAnotherData.json()

console.log(anotherData);

} catch (error) {

console.log(error);

}

}

getData();

Этот код делает то же самое, что и предыдущий, но с использованием **async/await**. Объявляем функцию **getData()** с ключевым словом **async**, чтобы мы могли использовать **await** внутри нее. Затем мы вызываем **fetchData()** с помощью **await** и ждем, пока промис разрешится. После этого вызываем еще один асинхронный запрос с помощью **await**. Если какой-либо из асинхронных запросов завершится с ошибкой, перехватываем и обрабатываем эту ошибку в блоке **catch**.

Заметьте, что вместо использования **then()** для обработки результатов промисов мы используем оператор **await**, который блокирует выполнение функции до тех пор, пока промис не разрешится.

Таким образом, использование **async/await** позволяет писать асинхронный код в более линейном и читаемом стиле, что делает его более легким для понимания и сопровождения.

## В циклах

Использование **async/await** с циклами может быть сложным, так как **await** блокирует выполнение функции до тех пор, пока промис не разрешится, что может привести к замедлению работы цикла. Однако, правильное использование **async/await** с циклами может значительно упростить и улучшить читаемость кода.

#### Использование for...of с async/await

Один из способов использования **async/await** с циклами - это использование **for...of** цикла для итерации по массиву промисов:

async function processData() {

const data = [Promise.resolve(1), Promise.resolve(2), Promise.resolve(3)];

for (const item of data) {

const result = await item;

console.log(result);

}

}

processData();

Здесь мы объявляем функцию **processData()**, которая содержит массив промисов **data**. Затем мы используем цикл **for...of**, чтобы итерироваться по элементам массива **data**, и каждый элемент разрешаем с помощью **await**. Результат разрешения промиса сохраняется в переменной **result**, которая затем выводится в консоль.

#### Использование Promise.all() с async/await

Второй способ использования **async/await** с циклами - это использование метода **Promise.all()** для ожидания разрешения массива промисов:

async function processData() {

const data = [Promise.resolve(1), Promise.resolve(2), Promise.resolve(3)];

const results = await Promise.all(data);

console.log(results);

}

processData();

Здесь мы используем метод **Promise.all()** для ожидания разрешения всех промисов в массиве **data**. Результаты разрешения промисов сохраняются в массиве **results**, который затем выводится в консоль.

#### Ваша задача:

* В своих предыдущих проектах с промисами перепишите как можно больше **.then** на **async/await**
* И не забудьте про **try/catch**
* Обсудите в чате плюсы и минусы такого подхода. Но, имейте ввиду что “правильно” и “не правильно” здесь нет, async await - это всего лишь синтаксический сахар для Promise, который намного удобнее читать, писать, понимать. Но у Promise намного больше возможностей

#### #10 Cобытийный цикл

“Событийный цикл” так же известный как “эвент-луп”, “асинхронность в джс” и “цикл событий”

Один из ключевых механизмов не самого языка, но большинства движков JS (это тоже важно, потому что есть реализации JS без event loop) В первую очередь эта тема нужна чтобы не только понимать как работает асинхронный вызов, но и представлять себе внутреннее устройство этого процесса. С одной стороны знать внутренние механизмы вовсе не обязательно, ведь JS - язык с весьма высоким уровнем абстракции, а все эти подводные камни только забивают голову. С другой стороны, асинхронность - одна из самых сложных, а может даже самая, сложная тема в JS. А если добавить сюда и отрисовку (рендер) в браузере и ввод в виде мыши и клавиатуры 🤯

*А уж как эту тему обожают обмусоливать на собесах - это вы еще узнаете* Тема большая и сложная, не пугайтесь если в голове будет “каша”, если не запомните или не поймете все на 100%. Полное понимание этого паттерна придет к вам с опытом проб и ошибок

Перед тем как читать про [Эвент луп](https://learn.javascript.ru/event-loop), прочитайте статью [про микрозадачи](https://learn.javascript.ru/microtask-queue)

#### #11 Рекурсия

Люди делятся на два типа: Первые, понимают рекурсию Вторые, делятся на два типа..

Еще один шаблон способный разорвать ваши шаблоны в клочья. Еще один супер прием, с помощью которого можно писать очень утонченные и элегантные решения. Главное, не пытаться “просчитать” рекурсию внутри своей головы, лучше уж пусть ваш браузер зависнет, чем вы снова захотите уйти на перерыв

Прочитайте [про рекурсию](https://learn.javascript.ru/recursion)

**Использование рекурсии с async/await**

Другой способ использования **async/await** с циклами - это использование рекурсии для итерации по массиву промисов:

async function processData(data, index = 0) {

if (index === data.length) {

return;

}

const result = await data[index];

console.log(result);

await processData(data, index + 1);

}

processData([Promise.resolve(1), Promise.resolve(2), Promise.resolve(3)]);

Здесь мы используем рекурсивную функцию **processData()**, которая принимает массив промисов **data** и индекс **index** в качестве аргументов. Если индекс равен длине массива, то функция завершается. В противном случае мы разрешаем промис в **data[index]** с помощью **await** и выводим результат в консоль. Затем мы вызываем **processData()** рекурсивно с новым индексом, чтобы обработать следующий промис в массиве.

❗

Важно заметить, что использование рекурсии с **async/await** может привести к проблемам с памятью, если массив слишком большой, поэтому этот метод следует использовать с осторожностью.

*Это сложная тема, не переживайте если не поняли ее полностью всего за один день. Иногда на это нужен месяц или два. Есть много людей, которые программируют вообще не зная о рекурсии. А есть те, которые делятся на два типа..*

#12 Рекурсия: практика

Вы уже познакомились с рекурсией и, возможно, даже решили пару задач.

Я предлагаю сделать вот что: **Возьмите программы которые у вас уже есть и попробуйте переписать как можно больше циклов на рекурсию!**

Эта задача прокачает вас очень мощно. Вы сильно удивитесь насколько по-другому может думать ваш мозг.

#### #13 Отладка кода

Прямо сейчас почти все вы отлаживаете свой код с помощью консоли или даже верстки. Есть куда более продвинутый способ, с рядом преимуществ.

**Ваша задача:** Прочитайте про: [Дебагг](https://learn.javascript.ru/debugging-chrome) Попробуйте обойтись без console.log в ближайшие пару недель.

*Через пару недель вы сильно удивитесь*

#### #13 Отладка кода

Прямо сейчас почти все вы отлаживаете свой код с помощью консоли или даже верстки. Есть куда более продвинутый способ, с рядом преимуществ.

**Ваша задача:** Прочитайте про: [Дебагг](https://learn.javascript.ru/debugging-chrome) Попробуйте обойтись без console.log в ближайшие пару недель.

*Через пару недель вы сильно удивитесь*

#### #14 Выходной

Не самая простая неделя была Хотя асинхронная модель устроена очень просто, особенно когда уже начал понимать зачем она нужна и как работает, а до этого все кажется **магией**.

Очень часто какой-то программист говорит про свой код “не может быть, волшебство какое-то” или “каким-то магическим образом”. Это всего лишь слова, но такие слова ясно дают понять кто управляет ситуацией. Не хочу никого расстраивать, но в программировании нет волшебства : (

Все, что вы поместили в интерпретаторы, будет обработано и исполнено ровно так, как вы указали. У компьютера нет настроения, усталости или желания. Это всего лишь инструмент, который выполняет данные кем-то команды. Компьютер ни в чем не виноват. Вообще никогда. Он может быть сломан, но не виноват. И колдовать он тоже не умеет. А магией обычно называли то, что не могли объяснить, потому что не понимали как оно устроено. Программирование ≠ Магия

#### #15 NPM

NPM (Node Package Manager) -пакетный менеджер в NodeJS. Другими словами, npm - это огромный репозиторий с пакетами (модулями) готового JS кода и, в то же время, инструмент для управления этими модулями.

Модули, пакеты, библиотеки, утилиты и еще с десяток других слов используют когда имеют ввиду кем-то уже написанный и собраный в кучу код, который мы можем переиспользовать\*. \* *да, между “использовать” и “переиспользовать” правда есть разница :)*

Переиспользование готового кода из npm дает **сразу несколько плюсов**:

1. Это безумно ускоряет время разработки. Вы получаете тысячи строк кода за секунды
2. Большие, проверенные временем и сообществом пакеты уже протестированы за вас, что сокращает количество багов
3. Популярные пакеты становятся в некотором смысле стандартом. Переходя из проекта в проект вам не нужно осваивать и понимать новый код раз за разом

**Но есть и минусы**:

1. Не все модули можно использовать в коммерческой разработке
2. Внести изменения в модуль будет куда сложнее (позже вы узнаете о возможных способах это сделать)
3. В NPM нет модерации - потенциально вы можете установить модуль с каким угодно кодом внутри (последствия представьте сами)

Устанавливать npm не придется, так как он уже есть в вашей системе после установки nodeJS

Ваша задача:

1. создать новый проект командой **npm init**
2. перенести в него код погоды (можно просто перенести файлы)

Уже завтра мы начнем добавлять сторонние пакеты

#### #16 Воспользуемся чужим кодом

Вчера вы узнали про npm Сегодня пора начать использовать его возможности и попробовать одну из популярных библиотек.

**date-fns** - библиотека для работы с датой. Содержит кучу готовых функций на самые разные случаи. Но начнем с простого, установите эту библиотеку в ваш проект (который был создан вчера) и поработайте с датой с помощью метода **format**

[Здесь есть документация](https://www.npmjs.com/package/date-fns) *p.s. документация часто бывает только на английском языке, если у вас с ним проблемы - используйте переводчик*

Если у вас ничего не работает - поздравляю, вы все сделали правильно. Дело в том, что такой код можно легко запустить в nodeJS но нельзя запустить в браузере. Для этого нам потребуется простой сборщик (со встроенным сервером, кстати) Пока мы не будем уходить в дебри и что-то настраивать, нам хватит пары команд для тестов и запуска нашего приложения

1. Установите пакет Parcel - **npm install --save-dev parcel**

2. Убедитесь что подключаете модуль <script type="module" src="index.js"></script>

3. Запустите вашу сборку в режиме разработки **npx parcel index.html** (возможно у вас другой путь до index.html)

4. Откройте в браузере и смело идите по адресу, который указан у вас в терминале server is running at http://localhost:1234

#### #17 ESLint

Тут будут аж две задачи в одной. При чем, вторая куда важнее первой и вы уже знаете почему

1. Установить ESLint, настроить пару правил в нем (любые, на ваш вкус)
2. Сделать это самостоятельно

все, что вам нужно знать про ESLint и прочие “вспомогательные средства” [есть здесь](https://eslint.org/docs/user-guide/getting-started) *p.s. еще раз хочу обратить ваше внимание на английский язык, который не должен вас пугать, в Google Chrome есть встроенный переводчик*

#### #18 Печеньки

Куки - это еще одна возможность хранить состояние в браузере (и передавать его между клиентом и сервером) В отличие от localStorage у кук есть несколько серьезных преимуществ (и пара ограничений)

Ваша задача:

1. Изучите эту тему [Куки](https://learn.javascript.ru/cookie) (да, там будет много слов про HTTP - не пугайтесь, если что не ясно, мы к нему скоро вернемся)
2. Обсудите в чате разницу между этими двумя способами
3. Сохраните в куках последний выбранный для погоды город (сейчас он хранится в localStorage) и установите “время жизни”, пусть этот выбранный город стирается через 1 час (для тестов время можно изменить)

p.s. работу с куками можно упростить с помощью библиотеки [js-cookie](https://www.npmjs.com/package/js-cookie) p.s.s. будет лучше если первый раз вы попробуете сделать это без сторонних инструментов

#### #19 Обратный отсчет

Напишите простую программу (с браузерным UI), которая будет принимать дату и отсчитывать кол-во лет, дней и часов до этой даты. Дизайн на вашей совести, главное чтобы был один ввод и один вывод в виде текста

Можно и нужно использовать сторонние библиотеки, лучшие наши практики и вообще все, что вы уже знаете

#### #20 Новые возможности

Посмотрим на “новые” операторы и конструкции в JS Их не так много, но они позволяют выражать свои мысли намного лаконичнее и чище. Прочитав статьи, перепишите ваши программы с применением новых подходов

[**Оператор нулевого слияния (??)**](https://learn.javascript.ru/nullish-coalescing-operator)

[**Оператор опциональной последовательности**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Optional_chaining)

Эти конструкции практически никто не использовал до 2020 года, но теперь они стали частью языка и их можно увидеть в любом свежем проекте.

Имейте ввиду, JS постоянно пополняется новыми и новыми возможностями, следить за новыми версиями не обязательно - вы все равно узнаете об этих новинках. Но можно немного заглядывать в будущее и заранее узнавать что появится в языке, для этого есть целый репозиторий

#### #22 Прототипы

**На прототипах построен весь JS** и на них же работает Наследование

И первое и второе звучит сейчас очень запутанно, кому-то напоминает про ООП, а кому-то страшные истории с собесов. А кто-то вообще не в курсе что все это значит. Но здесь все достаточно просто и логично, прям как в жизни.

Прочитайте про [Прототипное наследование](https://learn.javascript.ru/prototype-inheritance)

**Важно:** Это большая, сложная, напичканная терминами тема, которую вряд ли можно изучить за 1 день. Мы еще вернемся к прототипам, они станут понятнее на практике, как замыкания. Не пытайтесь все запомнить и ответить на каждый вопрос. Главное - понять суть этого механизма и зачем он есть в языке. А пользоваться им вы уже умеете

#### #23 Еще прототипы

Уверен, вам понравилось изучать прототипы. Так что вот вам еще 2 темы:

[Тема 1](https://learn.javascript.ru/function-prototype)

[Тема 2](https://learn.javascript.ru/native-prototypes)

Эту информацию вы будете использовать действительно редко, но она нужна чтобы лучше понимать JS. Не переживайте, если что-то непонятно полностью. Я бы даже сказал что вы можете пропустить эти статьи и, скорее всего, это мало на что повлияет на текущем этапе. Но, неужели вам не интересно что там внутри у вашего любимого языка происходит?)

#### #24 Классы

ООП не будет, без паники :) Просто посмотрим на классы в JS. Просто узнаем о их возможностях и разберем синтаксис. Да, может получится так что вы вообще нигде и никогда не увидите такой код, это не будет сюрпризом в наше время. Но знать об этой конструкции нужно уже с первого дня работы

Прочитайте про [Классы](https://learn.javascript.ru/class)

#### #25 Вызов/Применение/Привязка

Теперь вы уже точно не потеряете ваш this! Есть такие случаи, когда контекст (this) нужно передать явно, буквально указать языку с каким контекстом работать - такая возможность есть

[**Декораторы и переадресация вызова**](https://learn.javascript.ru/call-apply-decorators) [**Привязка контекста к функции**](https://learn.javascript.ru/bind)

Эту скучную теорию вы точно будете использовать в своей работе, но не часто;

#### #26 Практика

Напишите класс **Storage** который будет создавать экземпляры для работы с **localStorage**

Пример: const names = new Storage(’names’); names.get() // возвращает значение для ключа names в localStorage; names.set() // устанавливает значение для ключа names в localStorage; names.clear() // очищает значение для ключа names в localStorage; names.isEmpty() // вернет true если ключ names в localStorage имеет пустое значение (null || undefind);

Создайте несколько экземпляров класса Storage и убедитесь что все они работают правильно

Для класса Storage добавьте пару опций в конструктор

1. чтобы можно было выбирать **local** или **session** storage
2. возможность указать **значение по-умолчанию** (при создании экземпляра)

#### #27 Глобальность

В этой статье, как и среди большинства фронтендеров принято считать что переменные, объявленные в глобальной области видимости, называются глобальными, а все остальные - локальными. Это не совсем так или даже совсем не так, но термины настолько прочно укрепились в сообществе, что проще оперировать ими как есть, чем каждому объяснять как все устроено на самом деле

Прочитайте про [Глобальный объект](https://learn.javascript.ru/global-object)

#### #29 Про сеть

Про сеть веб-разработчику знать положено, но не так много как сис админу (дев опсу). Не так уж важно для нас все что происходит внутри роутеров и по каким проводам бежит (или плывет по волнам) Можно выделить только главные вещи которые стоит понимать Фронтендеру:

1. Первый (прикладной) уровень OSI - это то самое место, где живет HTTP (но о нем позже)
2. Понимать примерную пропускную способность для разных видов соединений (2g-5g, проводной интернет)
3. Понимать что такое ping
4. Понимать что передаваемые нами данные имеют вес (в Кб или Мб)

**Но зачем нам может это понадобиться?**

Первое нужно скорее для общего развития и понимания того, как устроен веб. Не стоит лезть в дебри и разбираться во всех мелочах - вы все равно это забудете за ненадобностью. Второе стоит брать во внимание когда вы решаете на каких устройствах, где и для кого будет открываться ваш сайт. Не лучшим тоном будет показывать пользователям мобильного интернета картинки в 40Мб или фоновое видео в 8к. От третьего напрямую зависит как скоро начнет загружаться ваш контент в браузере пользака, в некоторых случаях это может быть критично (люди редко готовы ждать дольше 3 секунд) Четвертое (вместе со вторым) напрямую повлияет на то как долго браузер будет загружать данные. Следить стоит не только за размером картинок, но и за размером самого приложения

Да, сейчас уже мы все летаем на безумных скоростях с минимальной задержкой, но случаи бывают разные и иногда нужно оптимизировать каждый Кбайт чтобы получить желаемый результат. Вы должны понимать от чего зависит “скорость” вашего приложения

Сегодня идеальный день чтобы набраться знаний по всем четырем пунктам (не тратьте на каждый из них более часа). Используйте любые источники