

JS Level 3

Node.js



Предисловие

На прошлых лекциях мы начали знакомиться с Node.js, поговорили о том, что возможности, предоставляемые Node.js, отличаются от тех, что доступны в браузере (хотя некоторые вещи, вроде `console.log`) будут похожи.

В этой лекции ключевой задачей будет ответить на следующий вопрос: какие возможности предоставляет Node.js. Этот вопрос разбивается на две части:

1. Поддержка самого языка JS
2. Стандартная библиотека Node.js



Релизный цикл

И поддержка возможностей JS, и доступные возможности стандартной библиотеки очень сильно зависят от версии Node.js. [На странице проекта](#) можно увидеть сроки поддержки разных версий:

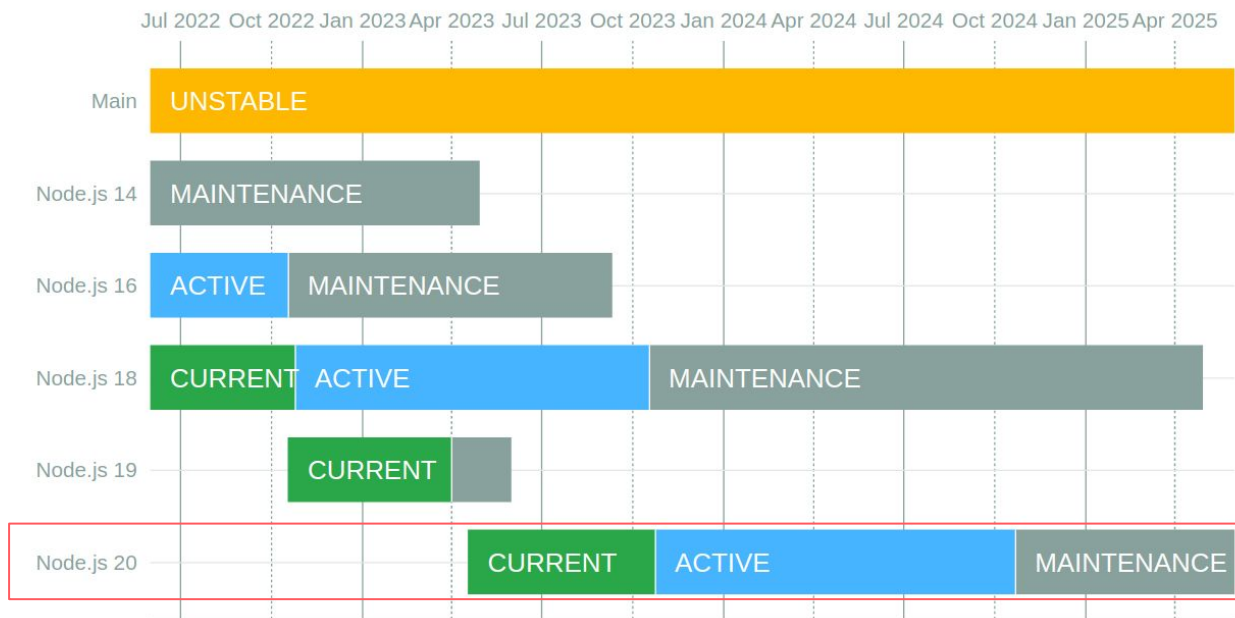
Release	Status	Codename	Initial Release	Active LTS Start	Maintenance Start	End-of-life
16.x	Maintenance	Gallium	2021-04-20	2021-10-26	2022-10-18	2023-09-11
18.x	LTS	Hydrogen	2022-04-19	2022-10-25	2023-10-18	2025-04-30
20.x	Current		2023-04-18	2023-10-24	2024-10-22	2026-04-30



Релизный цикл

В Node.js используется следующее соглашение: каждая чётная версия (мы с вами используем 20.x) проходит следующую последовательность этапов:

- Current – версия в разработке, в неё добавляются новые изменения, не нарушающие совместимость
- Active LTS – версия с длительной поддержкой, баг-фиксы, обновления
- Maintenance – версия с поддержкой (баг-фиксы и обновления безопасности)



Релизный цикл

Какие конкретно [возможности языка JavaScript](#) поддерживаются в какой версии можно увидеть в соответствующей [таблице совместимости](#):

<div><div>COMPAT</div><div>NODE</div></div> <div>Node.js ES2023 Support</div> <div>kangax's compat-table applied only to Node.js</div>		<div> Learn more</div>	<div>Nightly!</div> <div><div>21.0.0</div><div>20.4.0</div><div>19.1.0</div><div>18.16.1</div><div>17.9.1</div><div>16.10.0</div><div>16.8.0</div><div>16.5.0</div><div>16.3.0</div><div>16.0.0</div></div> <div><div>100% complete</div><div>100% complete</div><div>100% complete</div><div>100% complete</div><div>33% complete</div><div>33% complete</div><div>33% complete</div><div>33% complete</div><div>33% complete</div><div>33% complete</div></div>										
<div>Array find from last</div>													
<div>Array.prototype.findLast</div>			<div>?</div>	<div>Yes</div>	<div>Yes</div>	<div>Yes</div>	<div>Yes</div>	<div>Error</div>	<div>Error</div>	<div>Error</div>	<div>Error</div>	<div>Error</div>	<div>Error</div>
<div>Array.prototype.findLastIndex</div>			<div>?</div>	<div>Yes</div>	<div>Yes</div>	<div>Yes</div>	<div>Yes</div>	<div>Error</div>	<div>Error</div>	<div>Error</div>	<div>Error</div>	<div>Error</div>	<div>Error</div>



Релизный цикл

А поддерживаемые модули и их статус в конкретной версии можно найти на соответствующей [странице документации](#):

Stability overview

API	Stability
Assert	(2) Stable
Async hooks	(1) Experimental
Asynchronous context tracking	(2) Stable
Buffer	(2) Stable
Child process	(2) Stable
Cluster	(2) Stable
Console	(2) Stable
Crypto	(2) Stable
Diagnostics Channel	(2) Stable
DNS	(2) Stable
Domain	(0) Deprecated

Stable – можно использовать

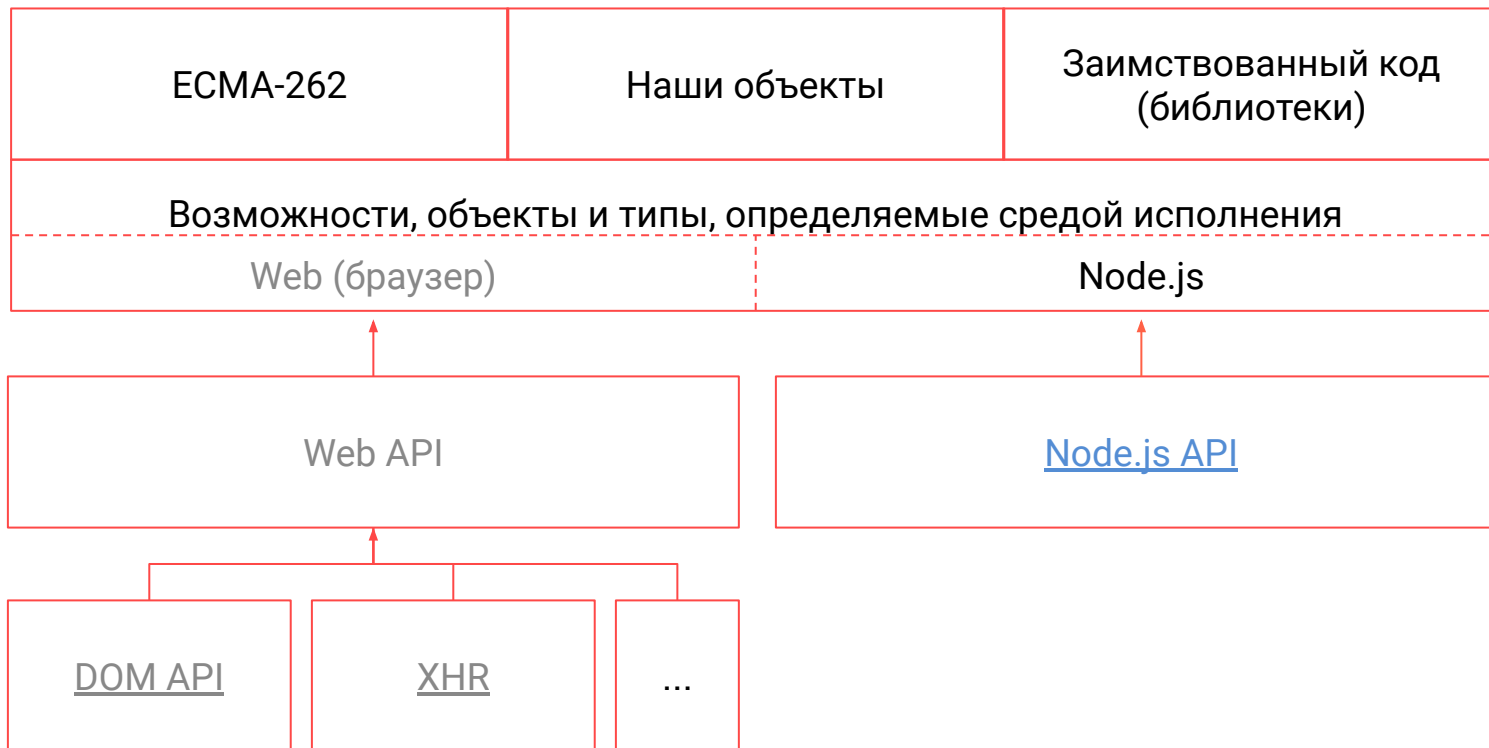
Experimental – стоит
ознакомиться, но пока
использовать не стоит

Deprecated – устарело,
использовать не стоит



Общая картина

Чтобы уложить у себя в голове общую картину, давайте нарисуем, что нам в итоге доступно (что мы можем использовать) в Node.js и как это связано с браузером:

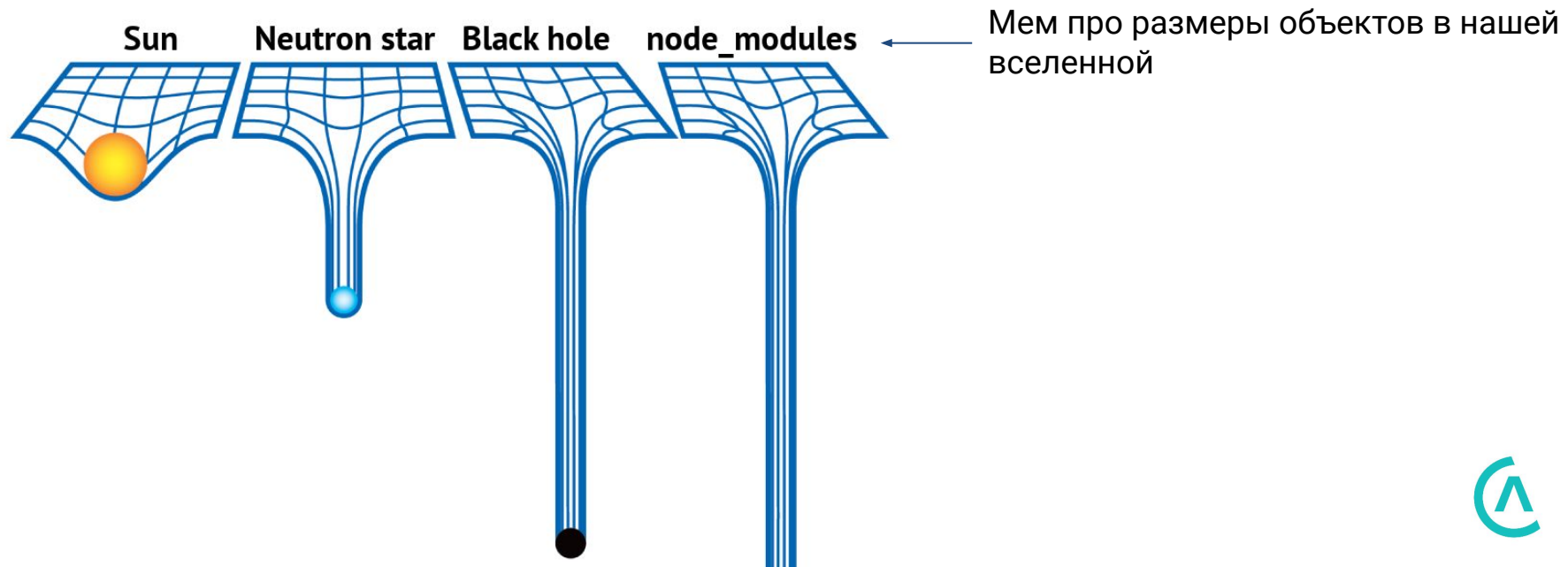


Стандартная библиотека



Стандартная библиотека

Теперь, когда у нас есть понимание общей картины, мы можем приступить к изучению стандартной библиотеки. Знание стандартной библиотеки очень важно: многие Node.js-разработчики плохо знают стандартную библиотеку (к сожалению) и сразу бегут устанавливать из npm сторонний пакет, который реализует нужную им функциональность (хотя она может быть доступна в самой Node.js "из коробки"), что приводит к следующему:



Стандартная библиотека

Перечислим ключевые модули стандартной библиотеки, которых мы так или иначе будем касаться в рамках курса (с остальными предлагаем вам ознакомиться самостоятельно):

1. Buffer – работа с бинарными данными
2. Child process – возможность запускать процессы и коммуницировать с ними
3. Crypto – криптография
4. Events – инфраструктура для создания событий
5. File system – работа с файловой системой
6. HTTP, HTTP/2, HTTPS – работа с семейством протоколов HTTP/HTTPS
7. Net – работа с сетью (на уровне TCP)
8. OS – взаимодействие с операционной системой
9. Path – работа с путями
10. Process – управление текущим процессом



Стандартная библиотека

- 11. Permissions – управление правами процесса
- 12. Stream – работа с потоками данных
- 13. Query String – работа с параметрами URL'a
- 14. URL – работа с URL
- 15. Worker Threads – работа с потоками (threads) ОС



Подключение модулей

Для получения функциональности, расположенной в конкретном модуле, необходимо подключить этот модуль одним из двух способов:

1. CommonJS (CJS): `const fs = require('node:fs');`
2. ESM: `import fs from 'node:fs';` (про ESM будем говорить чуть позже)

Важно: некоторые имена (помимо глобальных имён из стандарта ECMAScript), доступны напрямую без импорта соответствующих модулей (например, `process`, `URL`, `URLSearchParams`, `Buffer` и т.д.)



ESM

Варианты включения ESM:

- расширение файлов – `*.mjs`
- `type: "module"` в `package.json`
- `--input-type=module`



FS



fs

На прошлой лекции мы начали знакомиться с [модулем fs](#), который позволяет работать с файловой системой (а именно файлами и каталогами).

Если посмотреть на его документацию, то мы увидим несколько ключевых вещей:

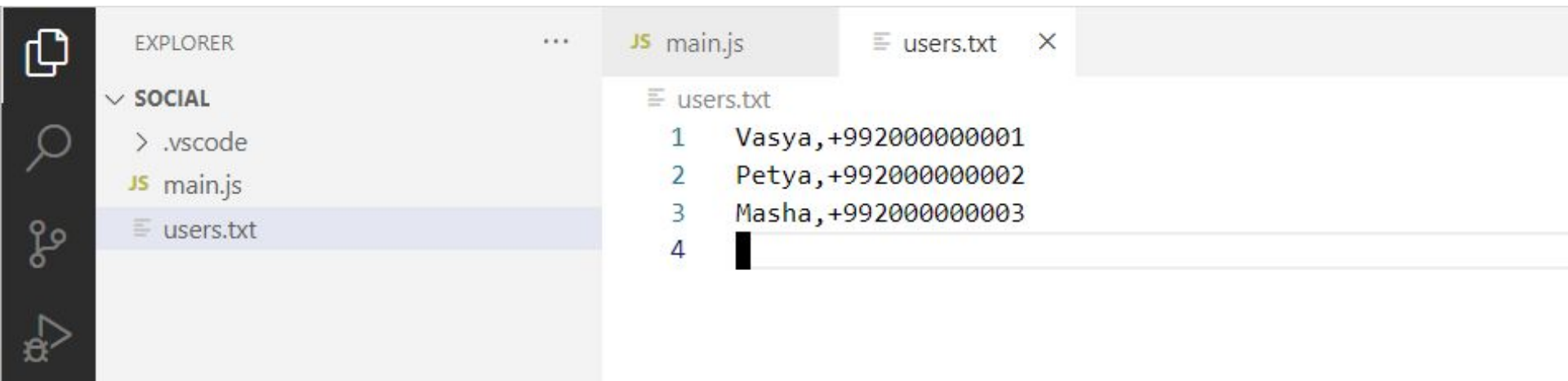
1. Есть какие-то классы (которые нам нужно будет обсудить)
2. Есть функции (вроде `readFile`, `readFileSync`)
3. Есть целый раздел `fs Promises API`

Давайте начнём разбираться со второго пункта.



fs

Пока мы не научились работать с базами данных, давайте будем хранить пользователей нашей сети в обычном файлике (обычно это выгрузка из базы данных):



Это обычный текстовый файл, создаёте вы его так же, как и `main.js`, только называете `users.txt`. Соответственно, мы хотим "вытащить" из этого файла всех пользователей и разослать им SMS.



fs

Модуль `fs` нам предлагает для чтения файлов специальную функцию `readFileSync`, которая позволяет прочитать файл целиком.

Что значит целиком? Для того, чтобы ответить на этот вопрос, нам нужно разобраться с тем, как вообще работают приложения на вашем компьютере.



Устройство компьютера

Если упрощённо представлять себе компьютер, то он будет выглядеть вот так:

процессор

оперативная память
(ОЗУ)

накопитель
(жёсткий диск или SSD)

сеть
(Wi-Fi, кабель, 3/4/5G)



Устройство компьютера

- Процессор – это часть компьютера, которая умеет выполнять команды (например, суммировать и т.д.)
- ОЗУ – это часть компьютера, которая хранит данные, необходимые процессору для вычислений (её обычно не очень много и она очень быстрая и дорогая)
- Накопитель – это часть компьютера, которая отвечает за хранение данных (её обычно много, она не очень быстрая и более дешёвая, чем ОЗУ)
- Сеть – это часть компьютера, которая позволяет общаться нашему компьютеру с другими компьютерами



Запуск приложения

Наша программа (`main.js`) хранится на жёстком диске, как и сам файл `node.exe`. Пока они просто хранятся – ничего не происходит. Но когда вы запускаете в VSCode вашу программу на исполнение, то происходит следующая последовательность операций:

1. Операционная система создаёт специальную сущность, которая называется процесс
2. В рамках этого процесса файл `node.exe` загружается с жёсткого диска в ОЗУ
3. Процессор начинает выполнять код из `node.exe` (который на шаге 1 загружен в память)
4. `node.exe` загружает с жёсткого диска файл `main.js` в ОЗУ
5. Процессор с помощью `node.exe` выполняет код, который написан в `main.js`



Процесс

Процесс – это просто сущность, у которой есть следующие атрибуты:

1. Исполняемый код (программа)
2. Выделенная процессу память (ОЗУ)
3. и т.д.

Например, в менеджере задач Windows мы можем увидеть следующее (вы не увидите, т.к. процесс очень быстро завершится):

Имя образа	ИД п...	Выделен...	Счетч...	Прочита...	Записан...	Путь к образу	Командная строка	Описание
jusched.exe	1076	3 572 КБ	5	2 181 139	230 824	C:\Program Files\Common Files\Java\J...	"C:\Program File...	Java Update Scheduler
node.exe	2604	8 468 КБ	7	161	0	C:\Program Files\nodejs\node.exe	node main.js	Node.js: Server-side JavaScript
ONENOTEM.EXE	1548	1 228 КБ	3	72 556	0	C:\Program Files\Microsoft Office\Offic...	"C:\Program File...	Microsoft OneNote Quick Launcher



Процесс

Процесс в большинстве случаев живёт только до тех пор, пока есть код, который нужно исполнять. Например, если наша программа закончена, то и процесс завершится.

За процессом следит сама операционная система: она его создаёт, даёт открывать ему файлы*, выделяет память и даёт исполнять код на процессоре (ведь только процессор умеет исполнять код).

Примечание*: на самом деле, сам процесс не может открывать файлы. Он может только попросить операционную систему открыть файл, а там уже она сама решает – можно его открыть или нет.



Чтение файлов

Возвращаясь к нашему вопросу с чтением файла. Целиком значит, что весь файл загружается в ОЗУ. И это очень проблематично, если файл весит несколько гигабайт. Поэтому будьте с этим аккуратны, не пытайтесь за один раз (за вызов одной функции) прочитать сразу весь файл в память.

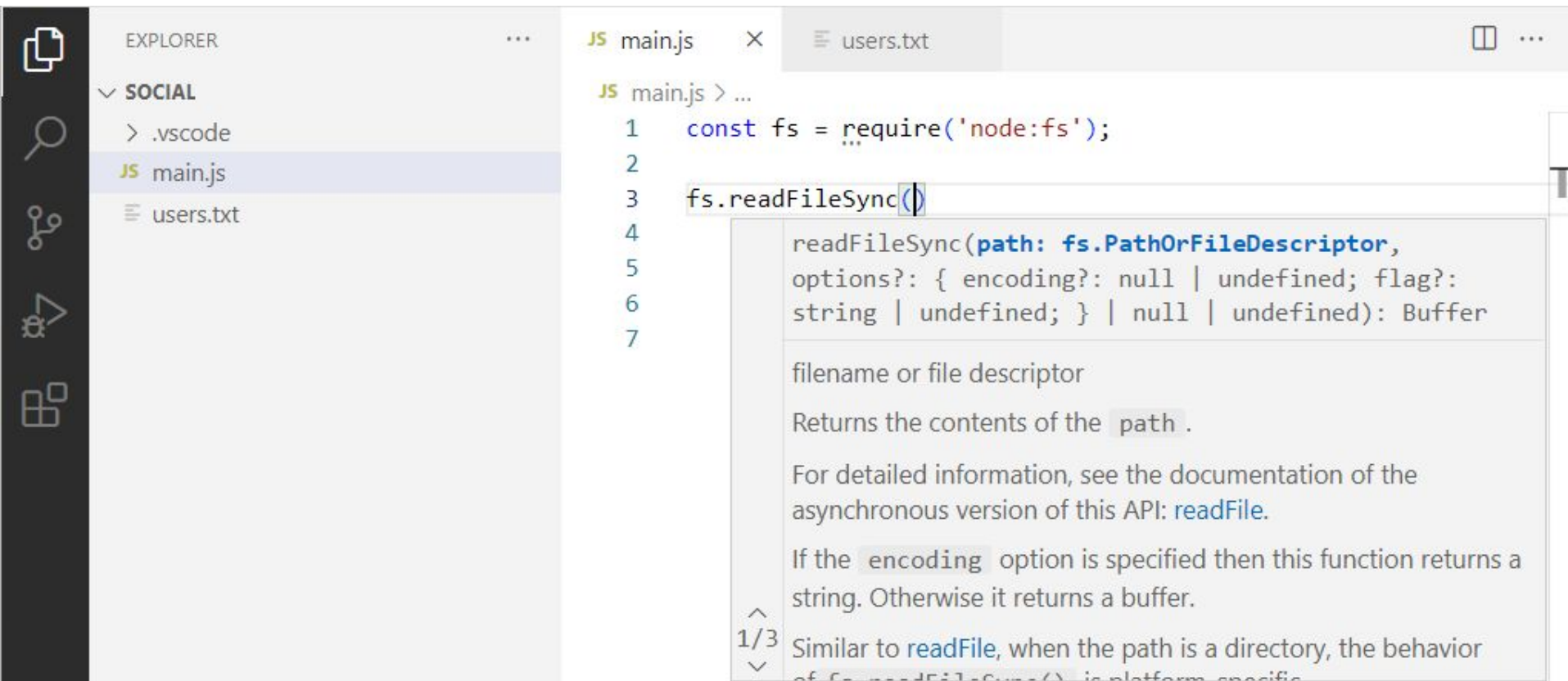
Поскольку файл `users.txt` пока* у нас не очень большой, мы вполне его можем прочитать с помощью `readFileSync`.

Примечание*: потом, конечно же придётся переписать нашу программу.



fs

Когда вы начнёте набирать вызов функции, VSCode будет вам подсказывать аргументы этой функции:



fs

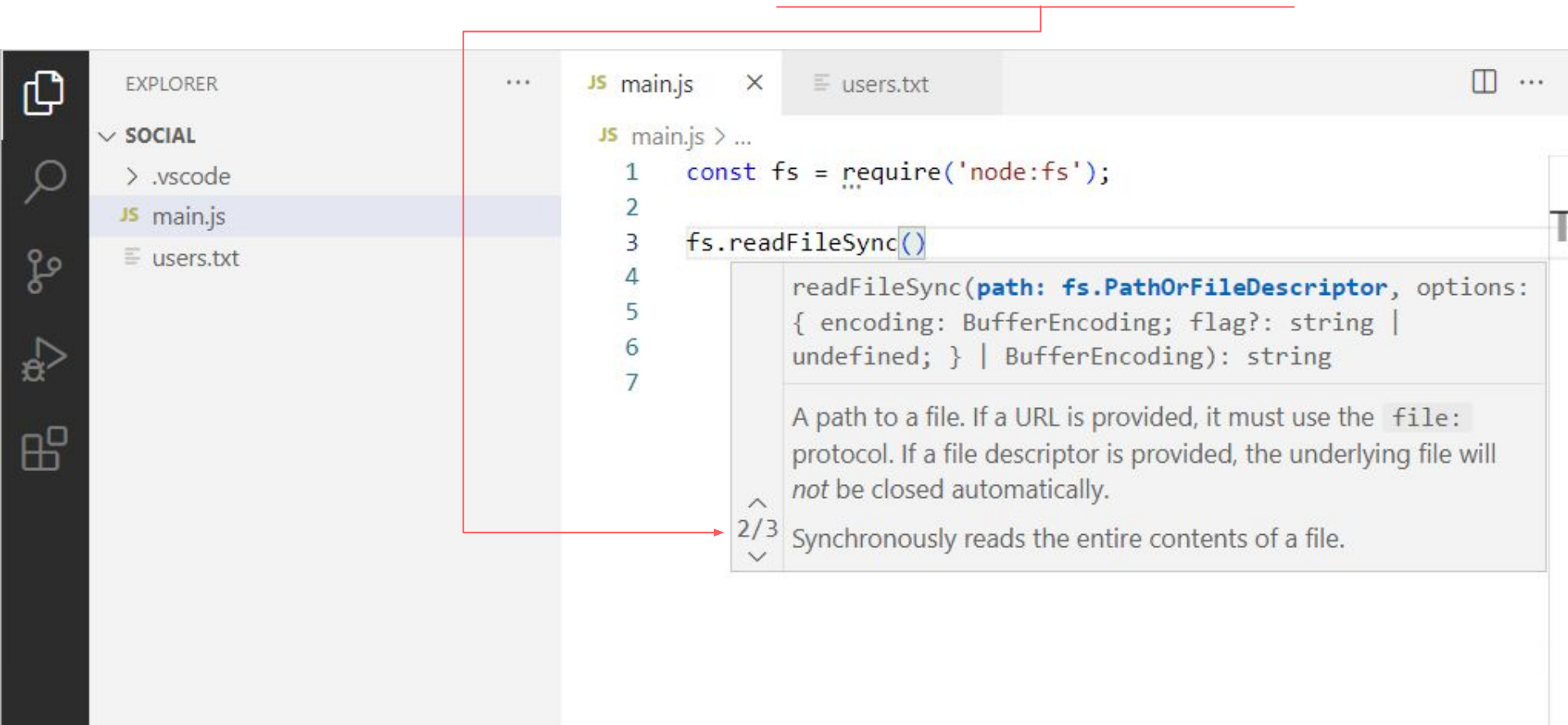
Здесь важно запомнить следующие моменты:

1. Сначала пишется имя параметра, а затем – допустимые типы, т.е. `path` может быть `fs.PathOrFileDescriptor`
2. То, что с вопросительным знаком (`options?`) - не обязательно, т.е. можно не указывать
3. В самом конце после двоеточия пишется тип, который возвращает функция (`Buffer`)
4. `1/3` означает, что если вы нажмёте на стрелку вниз на клавиатуре, то увидите другие варианты вызова функции (это возможно благодаря тому, что в JS функции могут принимать произвольное количество аргументов любого типа)



fs

Если нажать на стрелку вниз на клавиатуре, мы перейдём к следующему варианту вызова функции (всего их 3 – это показано внизу):



fs

Что значит `fs.PathOrFileDescriptor`? Об этом можно почитать на странице [документации модуля fs](#):

Node.js v20.5.0 | [► Table of contents](#) | [► Index](#) | [► Other versions](#) | [► Options](#)

`fs.readFileSync(path[, options])`

► History

- `path` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `options` `<Object>` | `<string>`
 - `encoding` `<string>` | `<null>` **Default:** `null`
 - `flag` `<string>` See [support of file system flags](#). **Default:** `'r'`.
- **Returns:** `<string>` | `<Buffer>`

Returns the contents of the `path`.



fs

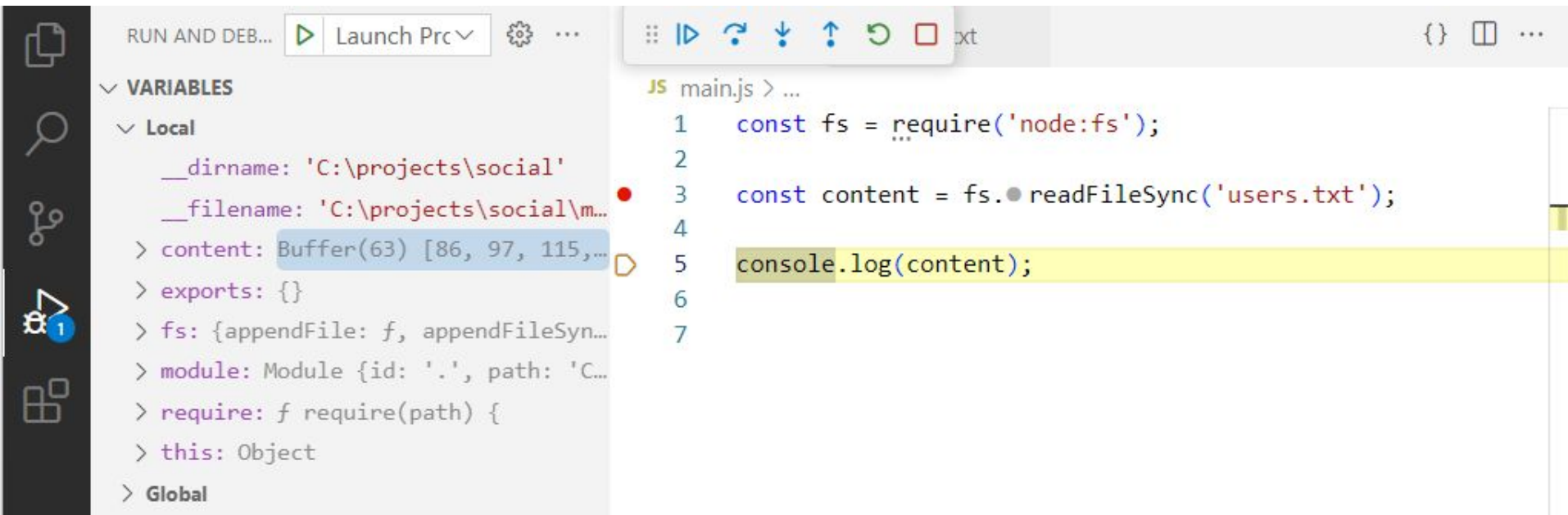
Либо с помощью клавиши **F12** "попереходить" по типам:

```
27  /**
28   * Valid types for path values in "fs".
29   */
30  export type PathLike = string | Buffer | URL;
31  export type PathOrFileDescriptor = PathLike | number;
```



Чтение файлов

Давайте попробуем что-то прочесть, выбрав первый вариант и указав только необходимые параметры:



The screenshot shows the Visual Studio Code editor with a JavaScript file named `main.js` and the integrated terminal. The code in `main.js` is as follows:

```
1 const fs = require('node:fs');
2
3 const content = fs.readFileSync('users.txt');
4
5 console.log(content);
6
7
```

The terminal output shows the result of `console.log(content)`:

```
> content: Buffer(63) [86, 97, 115, ...]
> exports: {}
> fs: { appendFile: f, appendFileSyn...
> module: Module { id: '.', path: 'C...
> require: f require(path) {
> this: Object
> Global
```

Обратите внимание, мы читали текстовый файл, а в ответ получили **Buffer(63)**.

Давайте разбираться с тем, что это такое.



Хранение информации

Наверняка вы слышали про биты и байты. Давайте разберёмся, что это такое.

Информация на различных носителях (например, жёстком диске) хранится в виде специальных ячеек, в которых может быть только два значения – **1** или **0**.

Соответственно, мы можем брать последовательность этих ячеек и придумывать какой последовательности какая информация соответствует. Поскольку длинные последовательности очень сложно читать, решили их разделить на байты (последовательность из 8-ми бит).

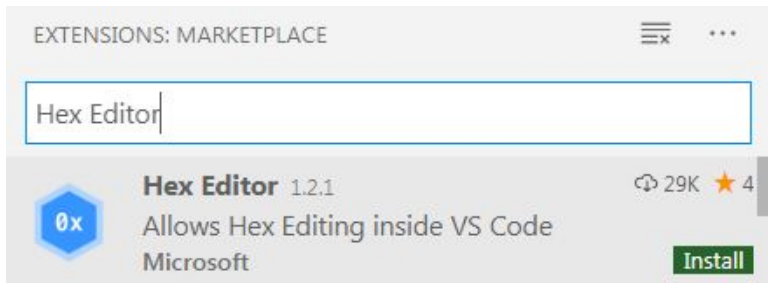
Например, возьмём и договоримся, что последовательности **11010000 10010010** (2 байта) соответствует большая буква **В**. И так с остальными буквами и цифрами.



Хранение информации

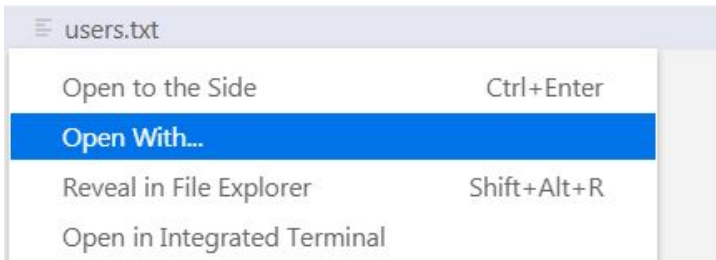
Почему мы так решили? Просто придумали и договорились со всеми остальными, что будет именно так. Компьютеру – всё равно, он хранит только биты. Мы же можем интерпретировать эти биты так, как нам захочется.

Зайдите в панельку расширений VSCode и установите Hex Editor от Microsoft:



Хранение информации

Кликните на файле **users.txt** правой кнопкой мыши и выберите Open With:



В выпадающем окошке выберите Hex Editor:



Хранение информации

Для удобства чтения все биты делятся на четвёрки и кодируются буквами и цифрами:

0000 - 0, 0001 - 1, 0010 - 2, 0011 - 3, 0100 - 4, 0101 - 5, 0110 - 6, 0111 - 7, 1000 - 8, 1001 - 9

1010 - A, 1011 - B, 1100 - C, 1101 - D, 1110 - E, 1111 - F

users.txt	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	DECODED TEXT	DATA INSPECTOR
00000000	56 61 73 79 61 2C 2B 39 39 32 30 30 30 30 30 30	V a s y a , + 9 9 2 0 0 0 0 0 0	8 bit Binary 01010110
00000010	30 30 31 0D 0A 50 65 74 79 61 2C 2B 39 39 32 30	0 0 1 . . P e t y a , + 9 9 2 0	Int8 86
00000020	30 30 30 30 30 30 30 30 32 0D 0A 4D 61 73 68 61 2C	0 0 0 0 0 0 0 2 . . M a s h a ,	UInt8 86
00000030	2B 39 39 32 30 30 30 30 30 30 30 30 33 20 0D 0A	+ 9 9 2 0 0 0 0 0 0 0 0 0 3 . .	Int16 24918
00000040	+	+	UInt16 24918
			Int24 7561558
			UInt24 7561558
			Int32 2037604694
			UInt32 2037604694
			Int64 4119435081321505110
			UInt64 4119435081321505110
			UTF-8 v
			UTF-16 情

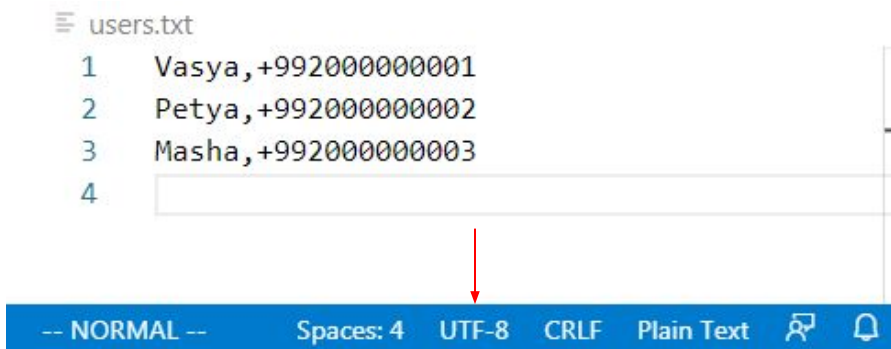
Таким образом, 56 – это 01010110, соответствует букве V.

Откуда это соответствие берётся?



UTF8

На самом деле, просто сели и составили большую таблицу, где каждой букве, цифре и иному знаку соответствует определённая последовательность бит. Таблица эта называется кодировкой. В нашем конкретном случае используется UTF-8:



```
users.txt
1 Vasya,+9920000000001
2 Petya,+9920000000002
3 Masha,+9920000000003
4
```

--- NORMAL --- Spaces: 4 UTF-8 CRLF Plain Text

Т.е. VSCode сохранил наш файл в этой кодировке. А когда он его будет читать (в виде байт) он будет эти байты преобразовывать в знаки в соответствии с этой таблицей.



UTF8

Поскольку мы ничего не говорим Node.js о том, в какой кодировке читать наш файл, он и возвращает нам не строку (тип `string`), а набор байт, за который и отвечает существующий в Node.js тип `Buffer` (важно: это тип Node.js, а не JS).



Типы данных

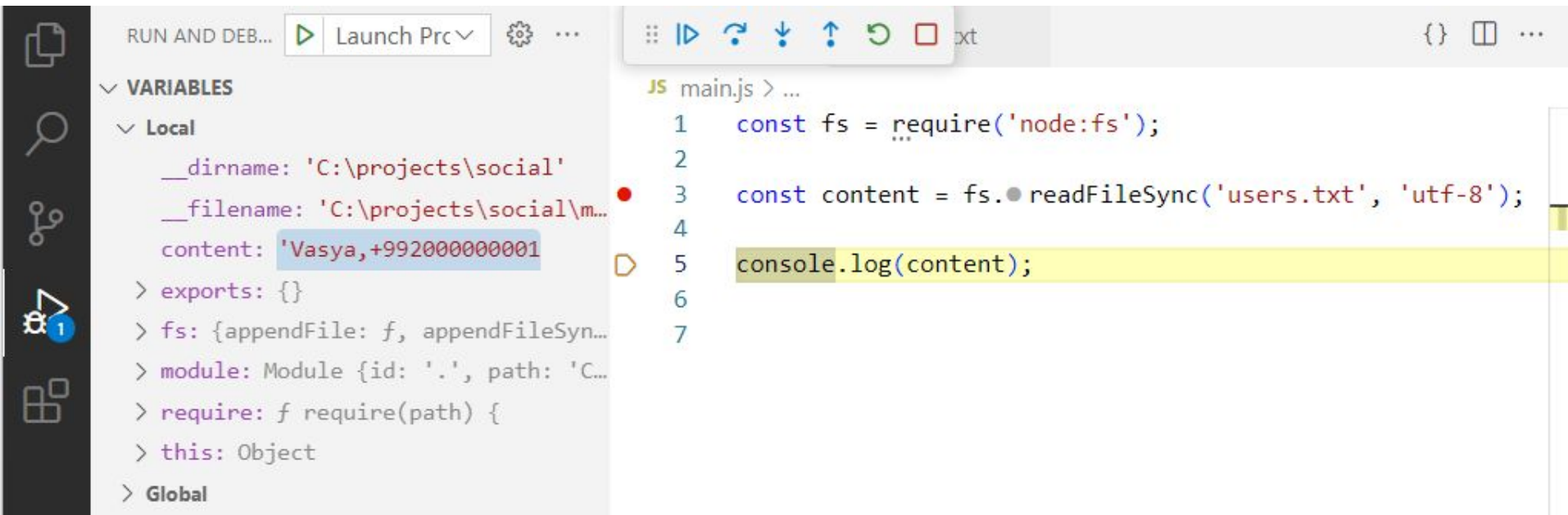
На всякий случай напомним вам о существующих типах данных:

1. Примитивы:
 - a. **Boolean** – true/false
 - b. **Null** – null (отсутствие объекта*)
 - c. **Undefined** – undefined (незначенное значение)
 - d. **Number** – число
 - e. **String** – строка
 - f. **Symbol** – символы
 - g. **BigInt** – числа для больших вычислений
2. Всё остальное, что не попало в п.1 – это объекты:
 - a. **Object** – объект ← **Buffer** относится сюда



UTF8

Давайте попробуем прочитать всё в виде строки. В этом нам поможет третья* форма вызова функции **readFileSync**:



The screenshot shows the Visual Studio Code interface. On the left, the 'VARIABLES' panel is expanded, showing the 'Local' scope. The variable 'content' is highlighted, showing its value as 'Vasya,+992000000001'. The 'fs' object is also visible. On the right, the 'main.js' file is open, showing the following code:

```
1 const fs = require('node:fs');
2
3 const content = fs.readFileSync('users.txt', 'utf-8');
4
5 console.log(content);
6
7
```

Примечание*: на самом деле, в любой из трёх форм можно указать кодировку, просто в этой форме достаточно её передать в виде строки, а в предыдущих двух нужно было передавать объект со свойством.



UTF8

Теперь наши данные прочитались уже строкой, а не **Buffer**. И мы можем с ними работать как со строкой (хотя и с **Buffer** тоже можно работать и конвертировать в строку).

Теперь посмотрим в самый низ файла в редакторе:

```
JS main.js  users.txt
users.txt
1 Vasya,+9920000000001
2 Petya,+9920000000002
3 Masha,+9920000000003
4
```

STATUS BAR: Ln 4, Col 1 | Spaces: 4 | UTF-8 | CRLF | Plain Text

UTF8

Q: но что это за странные символы **CRLF**?

A: поскольку всё хранится в виде байт, то перенос строки (следующий текст пишется с новой строки) тоже кодируется в виде байт.



Escape Symbols

В языках программирования, чтобы удобно записывать подобные конструкции, придумали escape-последовательности (или символы):

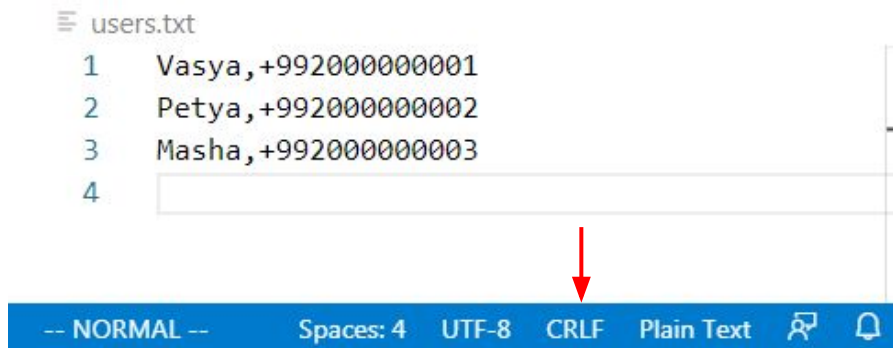
- `\n` – перевод строки (переход на новую строку)
- `\r` – возврат каретки (так было в печатных машинках)
- и т.д.

Возврат каретки – это перемещение курсора влево до конца строки, после чего `\n` означает переход на новую строку. Их комбинация используется в ОС Windows для обозначения переноса строки.



\r\n

В редакторах эта последовательность обозначается как **CRLF** (Carriage Return – **\r**, Line Feed – **\n**):



В ОС Linux и Mac обычно используется только **LF** (**\n**). С этим мы ещё столкнёмся, пока же нам важно только то, что у пользователя при печати никаких **\r\n** не будет, а будут переносы строки:

```
C:\Program Files\nodejs\node.exe c:\project\mobile\main.js
Debugger listening on ws://127.0.0.1:51588/52f120a8-2af4-4859-9606-3d00669df3ae
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Vasya,+9920000000001
Petya,+9920000000002
Masha,+9920000000003
```



Записи

Теперь неплохо бы разделить наши данные типа `string` на строки и из каждой строки выделить имя пользователя и его телефон.

Давайте разбираться, как это сделать.



ПРИМИТИВЫ VS ОБЪЕКТЫ



Типы данных

Напомним типы данных в JS:

1. Прimitives:
 - a. **Boolean** – true/false
 - b. **Null** – null (отсутствие значения*)
 - c. **Undefined** – undefined
 - d. **Number** – число
 - e. **String** – строка
 - f. **Symbol** – символы
 - g. **BigInt** - числа для больших вычислений
2. Всё остальное, что не попало в п.1 - это объекты:
 - a. **Object** – объект



Примитивы

Примитивы представляют из себя неизменяемые значения.

Q: что это значит?

A: это значит, что если вы создали один раз в коде строку, например, `'+992XXXXXXXXXX'`, то поменять вы её уже не сможете:

```
let phone = '+992XXXXXXXXXX';  
// это не изменение! мы создали новый примитив и положили его в phone  
phone = '+992YYYYYYYYYY';
```

Т.е. мы можем в переменную положить новое значение, но старое поменять нельзя. Это работает для всех примитивов.

Сам примитив – это просто значение. У него нет свойств, как у объектов.



Примитивы

Когда переменные, которые в данный момент содержат примитивное значение присваиваются другой переменной, то другая переменная также содержит примитивное значение:

```
let phone = '+992XXXXXXXXX';  
let copy = phone; // в copy теперь тоже +992XXXXXXXXX
```

```
phone = '+992YYYYYYYYY';  
console.log(phone); // +992YYYYYYYYY  
console.log(copy); // +992XXXXXXXXX
```

Важно: эти два имени после строки `copy = phone` уже никак не связаны (потому что примитивное значение поменять нельзя). И то, что мы положим в `phone` другое значение никак не затронет.



Примитивы

Зачем нужны примитивы? Они позволяют быть JS более эффективным языком, т. к. требуют гораздо меньше ресурсов для работы чем объекты.



Объекты

Объекты более сложные конструкции, они могут иметь свойства и являются изменяемыми (если их специальным образом не настроить):

```
const person = {  
  name: 'Vasya',  
  phone: '+992XXXXXXXX',  
};
```

```
person.phone = '+992YYYYYYYY';
```

```
console.log(person); // { name: 'Vasya', phone: '+992YYYYYYYY' }
```

Т.е. мы записали в свойство **phone** объекта новое примитивное значение. Условно говоря, мы "поменяли объект", назначив его свойству новое значение.



Объекты

Поскольку мы можем изменять свойства объектов, то при присваивании мы можем это делать через обе переменных:

```
const person = {  
  name: 'Vasya',  
  phone: '+992XXXXXXXX',  
};
```

```
const copy = person; // copy и person указывают на один объект (его свойства можно менять)  
copy.phone = '+992YYYYYYYY';
```

```
console.log(person); // { name: 'Vasya', phone: '+992YYYYYYYY' }
```



Параметры

Эта же схема (про примитивы и объекты) работает и тогда, когда вы передаёте их в качестве аргументов в функцию: примитивам ничего не будет, поскольку они не изменяемые, а вот свойства объекта можно изменить:

```
function change(primitive, object) {  
  primitive = 100; // no effect, мы просто положили в имя другое значение  
  object.phone = '+992YYYYYYYY'; // мы изменили свойство объекта, который доступен по этому имени  
  object = null; // no effect, мы просто положили в имя другое значение  
}  
  
const bonus = 1000;  
const person = {  
  name: 'Vasya',  
  phone: '+992XXXXXXXX',  
};  
  
change(bonus, person);  
console.log(bonus); // 1000  
console.log(person); // { name: 'Vasya', phone: '+992YYYYYYYY' }
```



Свойства

Это всё хорошо, но как нам это поможет в решении нашей задачи? Давайте вспомним: у нас есть объекты, у которых есть полезные свойства, в которых лежат функции.

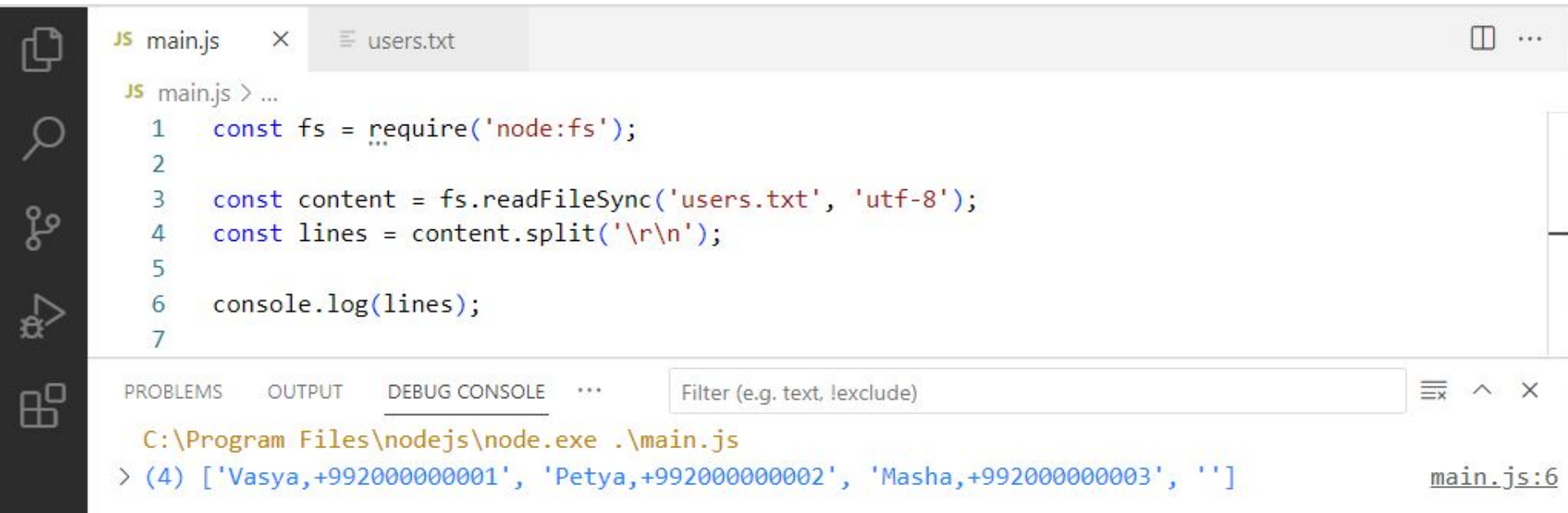
Такие свойства, в которых лежат функции, мы будем называть методами. У нас есть замечательный объект `console`, у которого есть метод `log`, который и позволяет нам выводить информацию в консоль.

Было бы здорово, если бы у нас существовали какие-то методы (или функции), которые позволяли бы работать со строками, например, разбивать одну строку на несколько по какому-нибудь символу (или группе символов).



Свойства

Давайте посмотрим на пример, а потом обсудим:



The screenshot shows a VS Code editor window with two tabs: 'main.js' and 'users.txt'. The 'main.js' tab is active, displaying the following JavaScript code:

```
JS main.js > ...  
1  const fs = require('node:fs');  
2  
3  const content = fs.readFileSync('users.txt', 'utf-8');  
4  const lines = content.split('\r\n');  
5  
6  console.log(lines);  
7
```

Below the editor, the 'DEBUG CONSOLE' panel is open, showing the command executed and its output:

```
C:\Program Files\nodejs\node.exe .\main.js  
> (4) ['Vasya,+992000000001', 'Petya,+992000000002', 'Masha,+992000000003', '']
```

The output is displayed on line 6 of 'main.js'.



Wrappers

На самом деле, когда мы используем переменные, в которых хранятся примитивные значения, или литералы (например, **29.9**) в виде объектов (например, обращаемся к свойствам), JS делает забавную вещь: он создаёт на базе этих примитивов временные объекты, которые обладают нужными свойствами, читает значения этих свойств (либо вызывает функции, если в свойстве лежит функция), после чего возвращает результат (а GC – garbage collector уже уничтожит объект).

Понятное дело, что примитив от этого не изменится, но это позволяет использовать всю мощь объектов тогда, когда это нужно. А самое главное – это всё происходит "за сценой", т.е. полностью без нашего участия.



Литералы

Первое, о чём стоит рассказать из текста предыдущего слайда - это о том, что такое литералы.

Литерал - это просто какое-то значение, записанное в коде (число, строка, true/false и т.д.):

```
console.log('Hello, Node.js!');
```

литерал

Так во если мы поставим после литерала точку, JS создаст объект, обратиться к свойству и т.д. (всё как на предыдущем слайде):

```
const words = 'Hello, Node.js!'.split(' ');  
console.log(words);
```



Литералы

Работает это для всех литералов, кроме чисел. Поскольку точка в числах является разделителем целой и вещественной части. Поэтому числа лучше сначала заключать в круглые скобки, а потом ставить точку: `(23).имяСвойства`

Хотя будет работать и так: `23..имяСвойства`.



Wrappers

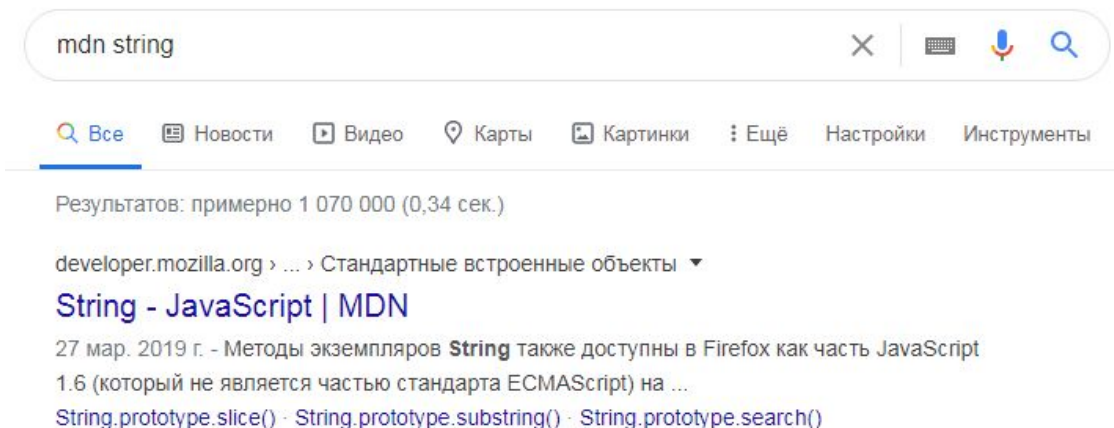
Осталось понять только одно: откуда JS берёт эти свойства? Как он узнаёт, что у строк есть метод `split`?

Всё дело в том, что для всех примитивов уже определены так называемые обёртки (wrapper'ы). Они называются так же как название типов, т.е. для строк - это будет `String`.



Wrappers

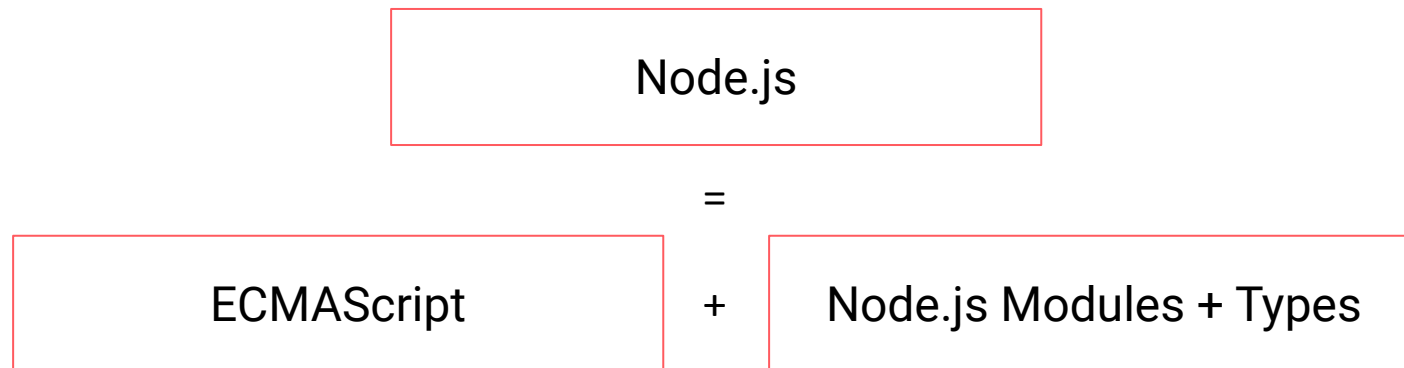
Когда вы в Google забиваете **mdn string**, вы первым делом попадаете на страницу описания (лучше, конечно, читать на английском - более актуальная и полная информация):



MDN

MDN (Mozilla Developer Network) – крупнейший информационный портал по JS и веб-технологиям. Но сразу возникает вопрос – мы же программируем для Node.js, почему документацию мы смотрим на MDN?

Если отвечать кратко, то используемый нами инструмент сейчас выглядит следующим образом:



MDN

Т.е. Node.js предоставляет нам реализацию самого языка JS (в котором и определены все стандартные типы) и свои собственные типы, например **Buffer**.

Поэтому за документацией на то, что относится к самому языку, мы идём на MDN, а на то, что относится к Node.js – на сайт Node.js.



MDN

Поехали дальше. На странице документации вы увидите вот такой список:

String

Свойства

String.length

Методы

...

String.prototype.replace()

String.prototype.replaceAll()
[Перевести]

String.prototype.search()

String.prototype.slice()

String.prototype.small()

String.prototype.split()

String.prototype.startsWith()

String.prototype.strike()

String.prototype.sub()

Нас будут интересовать свойства и методы (в которых есть **prototype**).

О **prototype** мы с вами говорили уже на предыдущих курсах. Поэтому для нас важно следующее: всё, что написано в "Свойства" и "Методы" (с **prototype**) мы можем использовать на наших примитивах типа **String**.

То, что помечено  использовать не нужно.



MDN

Давайте кликнем на `String.prototype.split()` и посмотрим, что внутри:

Сводка

Метод `split()` разбивает объект `String` на массив строк путём разделения строки указанной подстрокой.

Синтаксис

```
str.split([separator[, limit]])
```

Параметры

`separator`

Необязательный параметр. Указывает символы, используемые в качестве разделителя внутри строки. Параметр `separator` может быть как строкой, так и `регулярным выражением`. Если параметр опущен, возвращённый массив будет содержать один элемент со всей строкой. Если параметр равен пустой строке, строка `str` будет преобразована в массив символов.

`limit`


Необязательный параметр. Целое число, определяющее ограничение на количество найденных подстрок. Метод `split()` всё равно разделяет строку на каждом сопоставлении с разделителем `separator`, но обрезает возвращаемый массив так, чтобы он содержал не более `limit` элементов.

Единственное, что здесь стоит отметить, это то, что **необязательные** параметры здесь заключены в квадратные скобки (а не указаны с **?** как в VSCode).



MDN

Важно: ваша успешность во многом будет зависеть от того, насколько хорошо вы умеете пользоваться инструментами, которые встроены в сам язык.

Поэтому не поленитесь и прочитайте документацию на все методы **String**, которые без  (и с **prototype**).



МАССИВЫ



Массивы

В документации сказано, что если мы передаём строку-разделитель (а мы использовали `\r\n` для разделения), то в качестве результата функция вернёт нам массив данных.



Массивы

Давайте посмотрим на массивы в дебаггере:

The screenshot shows the VS Code interface with a JavaScript file named `main.js` open. The code contains the following lines:

```
1 const fs = require('node:fs');
2
3 const content = fs.readFileSync('users.txt', 'utf-8');
4 const lines = content.split('\r\n');
5
6 console.log(lines);
7
```

The line `console.log(lines);` is highlighted in yellow, and a breakpoint is set at line 6. The left sidebar shows the **VARIABLES** panel with the following structure:

- Local**
 - `__dirname`: 'C:\projects\soci...
 - `__filename`: 'C:\projects\soc...
 - `content`: 'Vasya,+992000000001'
 - `exports`: {}
 - `fs`: {appendFile: f, appendFi...
 - lines**: (4) ['Vasya,+99200000...']
 - `0`: 'Vasya,+992000000001'
 - `1`: 'Petya,+992000000002'
 - `2`: 'Masha,+992000000003'
 - `3`: ''
 - `length`: 4
 - `[[Prototype]]`: Array(0)

The bottom panel shows the **DEBUG CONSOLE** with the command `C:\Program Files\nodejs\node.exe .\main.js` entered.

Массив – это объект (потому что всё, что не примитив – это объект), у которого некоторые свойства пронумерованы от 0 и до количества элементов - 1.



Доступ к свойствам

И, напомним, что к числовым свойствам мы просто так обращаться не можем:

```
const content = fs.readFileSync('users.txt', 'utf-8');  
  
const lines = content.split('\r\n');  
console.log(lines.length);  
console.log(lines.0); // сигнализирует об ошибке
```



Доступ к свойствам

На самом деле, для того, чтобы получить доступ к свойствам объектов, существует три формы записи:

1. `имяПеременной.имяСвойства` – если имя свойства соответствует правилам для имён переменных
2. `имяПеременной[имяСвойства]` – если имя свойства число
3. `имяПеременной['имяСвойства']` – универсальный способ (имя свойства может быть любым, включая варианты для п.1 и п.2)

Конечно же, если имя свойства хранится в переменной, то мы можем использовать вариант №2.



Доступ к элементам

0, 1 и 2 – не являются допустимыми именами для переменных, поэтому вариант 1 мы использовать не можем, а вот варианты 2 и 3 – вполне:

```
const content = fs.readFileSync('users.txt', 'utf-8');
```

```
const lines = content.split('\r\n');  
console.log(lines.length);  
console.log(lines[0]);  
console.log(lines['0']);
```



Доступ к элементам

Это всё хорошо, но мы же можем не знать, сколько в файле строк (а, соответственно, в массиве - элементов). Не создавать же на каждый элемент переменную:

```
const content = fs.readFileSync('users.txt', 'utf-8');  
  
const lines = content.split('\r\n');  
const firstRecord = lines[0];  
const secondRecord = lines[1];  
// .. много-много раз  
// не делайте так!
```



Доступ к элементам

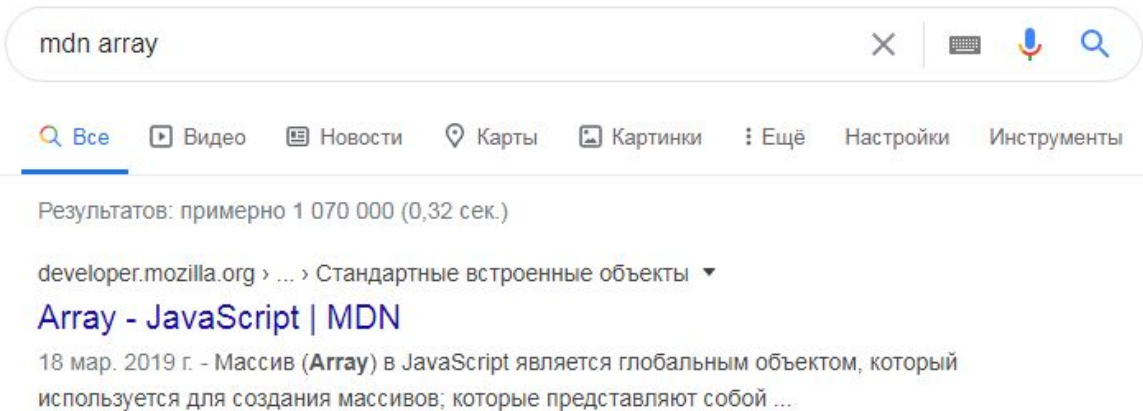
Давайте вернёмся к нашей задаче и вспомним следующую вещь: мы хотели для каждой строки из нашего файла (а строка представляет собой информацию о пользователе) отправить SMS.

Что значит отправить SMS? Это значит, выполнить набор каких-то действий, который мы назвали "Отправить SMS". Т.е. на самом деле, мы хотим для каждой строки из нашего файла выполнить какую-то функцию!



Методы

У массивов тоже есть набор полезных методов, которые можно найти, забив в Google "mdn array":



forEach

Метод `forEach()` выполняет указанную функцию один раз для каждого элемента в массиве.

Фактически, то, что нам нужно.

Осталось разобраться с тем, что такое `callback`.

Синтаксис

```
arr.forEach(function callback(currentValue, index, array) {  
    //your iterator  
},[, thisArg]);
```

Параметры

`callback`

Функция, которая будет вызвана для каждого элемента массива. Она принимает от одного до трех аргументов:

`currentValue`

Текущий обрабатываемый элемент в массиве.

`index` | Необязательный

Индекс текущего обрабатываемого элемента в массиве.

`array` | Необязательный

Массив, по которому осуществляется проход.

`thisArg`

Необязательный параметр. Значение, используемое в качестве `this` при вызове функции `callback`.

У массивов ещё очень много полезных методов, с ними мы познакомимся чуть позже.



CALLBACK




Функции

В JS функции – это объекты, которые можно вызывать. Функцию можно сохранить в переменную, можно передавать как аргумент в другую функцию (точно так же, как мы передавали числа, объекты).

Но самое главное, если в каком-то имени хранится функция, то эту функцию можно вызвать с помощью оператора `()`.

```
function handleRecord(record) {  
  console.log(record);  
}
```

нет круглых скобок, значит не вызываем


`const func = handleRecord; // не вызываем функцию, а просто сохраняем в переменной`
`func('Vasya,+992XXXXXXXXX'); // а вот здесь уже вызываем`



Callback

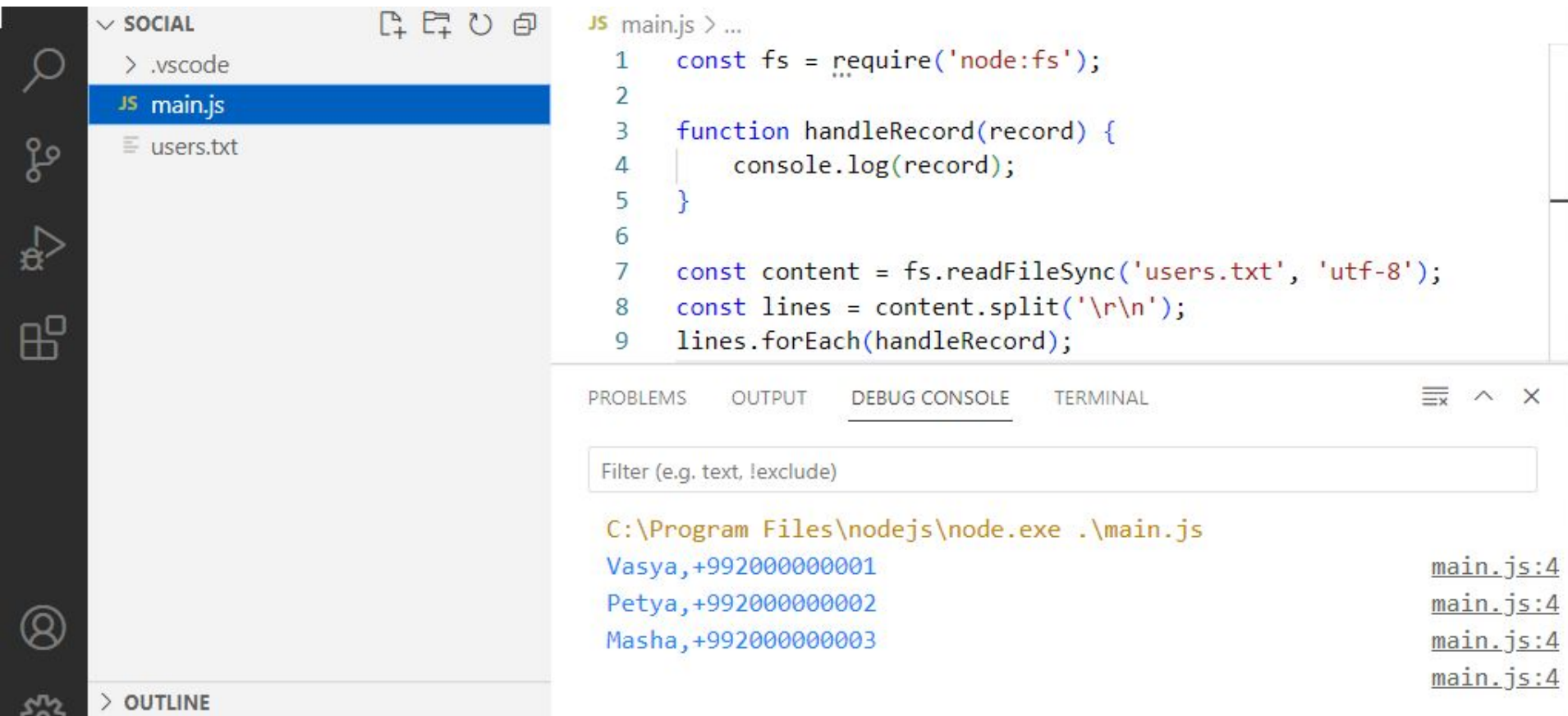
Такой подход с передачей функции в другую функцию (для последующего вызова) называют callback'и – функции обратного вызова. Вместо того, чтобы самим вызывать функцию – вы просто передаёте её название и вашу функцию вызывают тогда, когда нужно.

Так мы и поступим с **forEach** – мы отдадим ему нашу функцию, которую нужно вызвать для каждого элемента, а он пусть сам её и вызывает.



Callback

Зачем это всё было нужно? Затем, что мы название этой функции можем передать в метод **forEach**, который и будет вызывать нашу функцию для каждого элемента:



The screenshot shows the Visual Studio Code editor interface. On the left, the Explorer sidebar shows a project named 'SOCIAL' with files '.vscode', 'main.js', and 'users.txt'. The 'main.js' file is selected and open in the editor. The code in 'main.js' is as follows:

```
1  const fs = require('node:fs');
2
3  function handleRecord(record) {
4      console.log(record);
5  }
6
7  const content = fs.readFileSync('users.txt', 'utf-8');
8  const lines = content.split('\r\n');
9  lines.forEach(handleRecord);
```

Below the editor, the 'DEBUG CONSOLE' tab is active, showing the output of the program. The output is:

```
C:\Program Files\nodejs\node.exe .\main.js
Vasya,+9920000000001
Petya,+9920000000002
Masha,+9920000000003
```

On the right side of the output, the source file and line number for each log statement are listed:

```
main.js:4
main.js:4
main.js:4
main.js:4
```

Callback

Можете поставить точку остановки внутри `handleRecord` и удостовериться, что он будет вызван 4 раза. Но почему 4?



Пустая строка

Всё потому, что у нас есть пустая строка, в которой нет данных:

```
users.txt
1  Vasya,+9920000000001
2  Petya,+9920000000002
3  Masha,+9920000000003
4
```

Нам нужно её аккуратно удалить (руками), чтобы осталось только три строки.

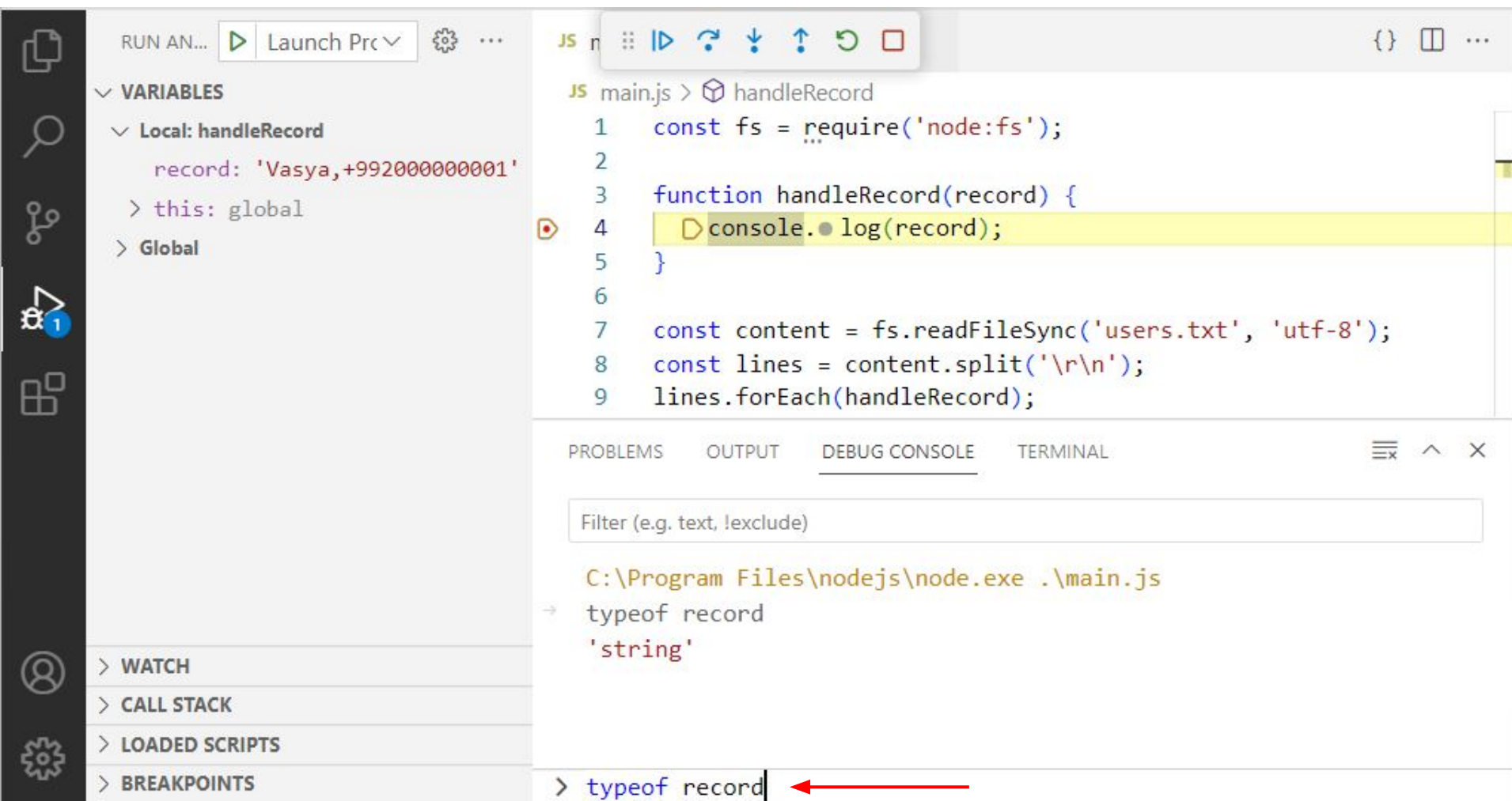
Тогда проблем не будет.

Либо, вы можете посмотреть на методы [trim](#) у строки и [map/filter](#) у массива, которые позволят нам отсечь пустые строки:

```
const lines = content
    .split('\r\n')
    .map(o => o.trim())
    .filter(o => o !== "");
```



Во время отладки в VSCode в нижней части расположена строка, в которую вы можете вводить код. Этот код будет исполняться в той области видимости, в которой вы сейчас находитесь:



ИТОГОВЫЙ КОД

SMS мы отправлять пока не умеем, поэтому просто будем выводить номер телефона в консоль:



```
JS main.js × users.txt
JS main.js > handleRecord
1  const fs = require('node:fs');
2
3  function handleRecord(record) {
4      const parts = record.split(',');
5      const phone = parts[1];
6      console.log(phone);
7  }
8
9  const content = fs.readFileSync('users.txt', 'utf-8');
10 const lines = content.split('\r\n');
11 lines.forEach(handleRecord);
12
```



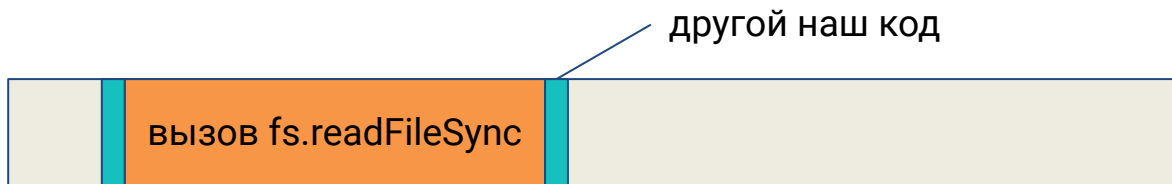
ASync



Blocking IO

Тот код, который мы с вами написали, он неплох для небольшой утилиты (вспомогательного приложения). Но для больших решений не пригоден.

Почему? Потому что мы с вами использовали `readFileSync`. Что означает суффикс `sync`? Он означает, что пока Node.js читает файл, никакой другой наш JS код не может быть выполнен (т.е. мы не попадём на следующую строку за `readFileSync`):



Это как кафе, в котором официант один на всё заведение и одновременно работает и официантом, и поваром. Вы ему сделали заказ (прочитать файл), он ушёл готовить. В этот момент ни вы не можете ещё что-то заказать, ни другие посетители.



Blocking IO

Такие вызовы, связанные с вводом-выводом (IO – чтение/запись с диска или сети), которые ожидают завершения работы и не дают исполняться другому коду, называют блокирующими (blocking IO).



Blocking IO

Мы же хотим с вами писать сервера, которые обслуживают тысячи пользователей, поэтому такой подход нам не совсем подходит.



Non-Blocking IO

Почему бы не попробовать сделать по-другому? Пусть задача официанта будет заключаться в том, чтобы собрать заказы, отнести их на кухню и приносить их с кухни, когда всё будет готово.

Обратите внимание: **приносить их с кухни, когда всё будет готово.** Что это значит? Это значит, что кто-то вызовет официанта тогда, когда блюдо будет готово. Т.е. мы снова приходим к callback'ам. У официанта будет метод "забрать готовое блюдо", которое на кухне будут вызывать тогда, когда будет готово блюдо.

Такие вызовы называют неблокирующими (non-blocking).



Non-Blocking IO

Так вот функции, которые без суффикса **sync** или которые в **fsPromises** они как раз позволяют такое осуществить:

`fs.readFile(path[, options], callback)`

► History

- `path` `<string> | <Buffer> | <URL> | <integer>` filename or file descriptor
- `options` `<Object> | <string>`
 - `encoding` `<string> | <null>` Default: `null`
 - `flag` `<string>` See [support of file system flags](#). Default: `'r'`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `data` `<string> | <Buffer>`

Asynchronously reads the entire contents of a file.



Non-Blocking IO

В данном случае callback должен принимать два аргумента:

1. **error** – если при чтении произошла ошибка (например, вы пытаетесь прочитать файл, которого не существует)
2. **data** – данные, которые были прочитаны (**Buffer**, если вы не указали кодировку, **string** – если указали)

Это общая практика в Node.js – первым аргументом всегда указывать параметр, в который "кладётся" ошибка (если его значение не равно **null**, значит, случилась ошибка).



Non-Blocking IO

JS main.js > ...

```
1  const fs = require('node:fs');
2
3  function handleRecord(record) {
4      const parts = record.split(',');
5      const phone = parts[1];
6      console.log(phone);
7  }
8
9  function handleFile(error, data) {
10     const lines = data.split('\r\n');
11     lines.forEach(handleRecord);
12 }
13
14 console.log('before call readFile');
15 fs.readFile('users.txt', 'utf-8', handleFile);
16 console.log('after call readFile');
```

Если вы посмотрите на консоль, вы увидите, что **'after call readFile'** будет вызван раньше, чем вывод телефонов. Потому что "официант поставил задачу" и пошёл обслуживать других клиентов (16-ая строка), а когда файл будет прочитан – он пойдёт выполнять функцию **handleFile**.



Как это работает?

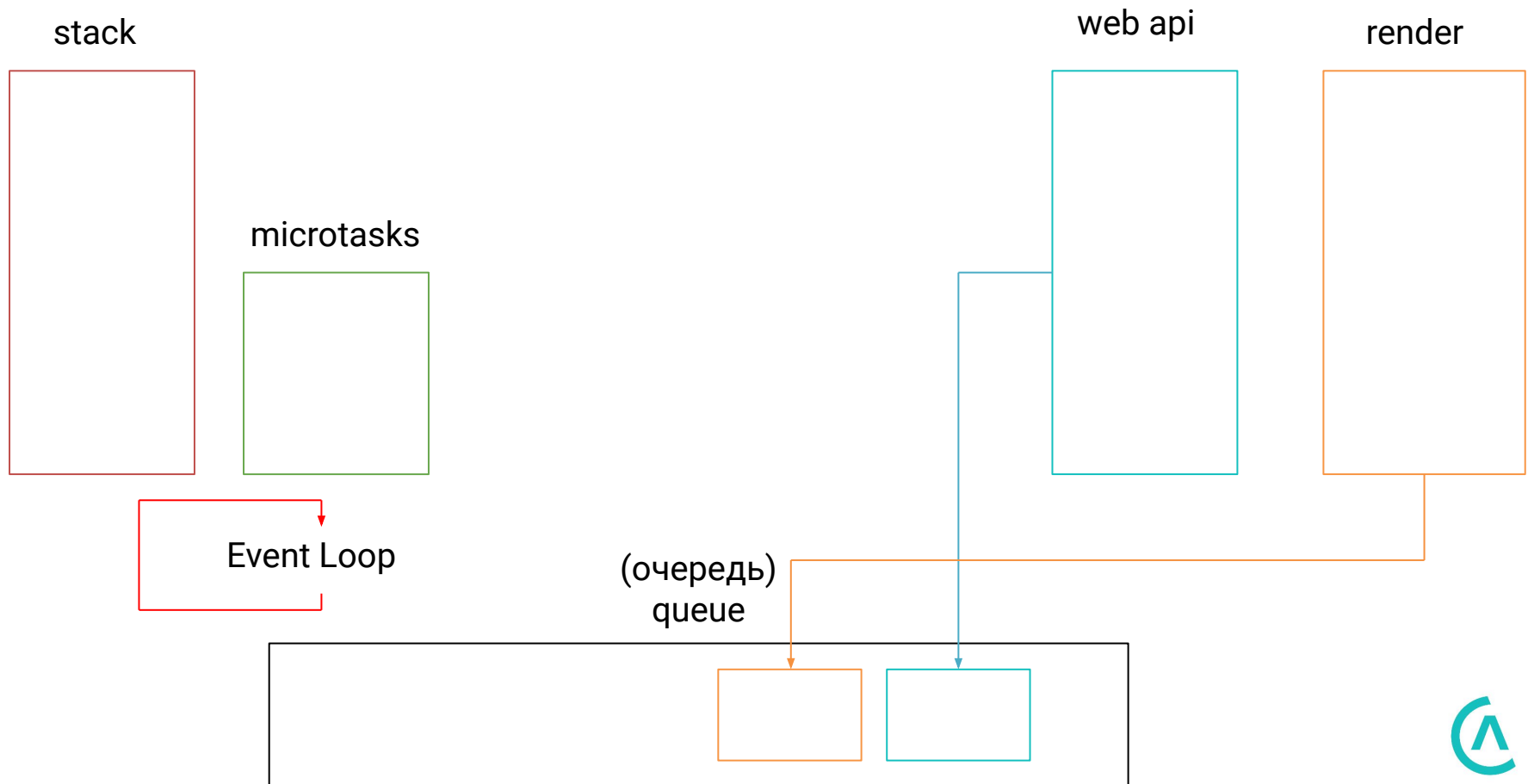
```
JS main.js > ...  
1  const fs = require('node:fs');  
2  
3  function handleRecord(record) {  
4      const parts = record.split(',');  
5      6 const phone = parts[1];  
6      console.log(phone);  
7  }  
8  
9  3 ⏳ function handleFile(error, data) {  
10     5 const lines = data.split('\r\n');  
11     lines.forEach(handleRecord);  
12 }  
13  
14 console.log('before call readFile');  
15 fs.readFile('users.txt', 'utf-8', handleFile);  
16 console.log('after call readFile');
```

Node.js сверху до низу выполняет нашу программу – стрелки слева (не заходя в функции, так как их ещё не вызвали), а дальше не завершает работу приложения, а ждёт, пока не отработает чтение файла. Как только чтение файла отработает, будет вызвана функция **handleFile** (стрелки справа). После того, как для Node.js работы не останется (не осталось кода, который нужно исполнить), работа всей программы будет завершена.



Event Loop

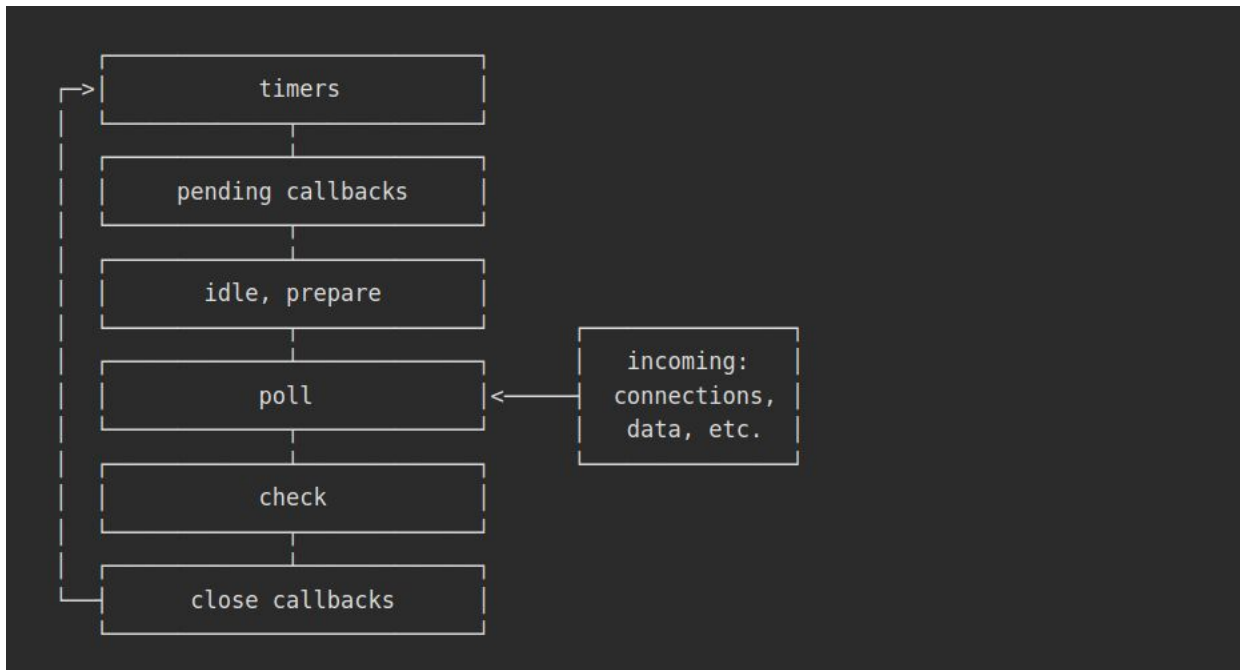
Теперь о том, как это работает: в своё время мы проходили Event Loop в браузере, где он устроен был следующим образом:



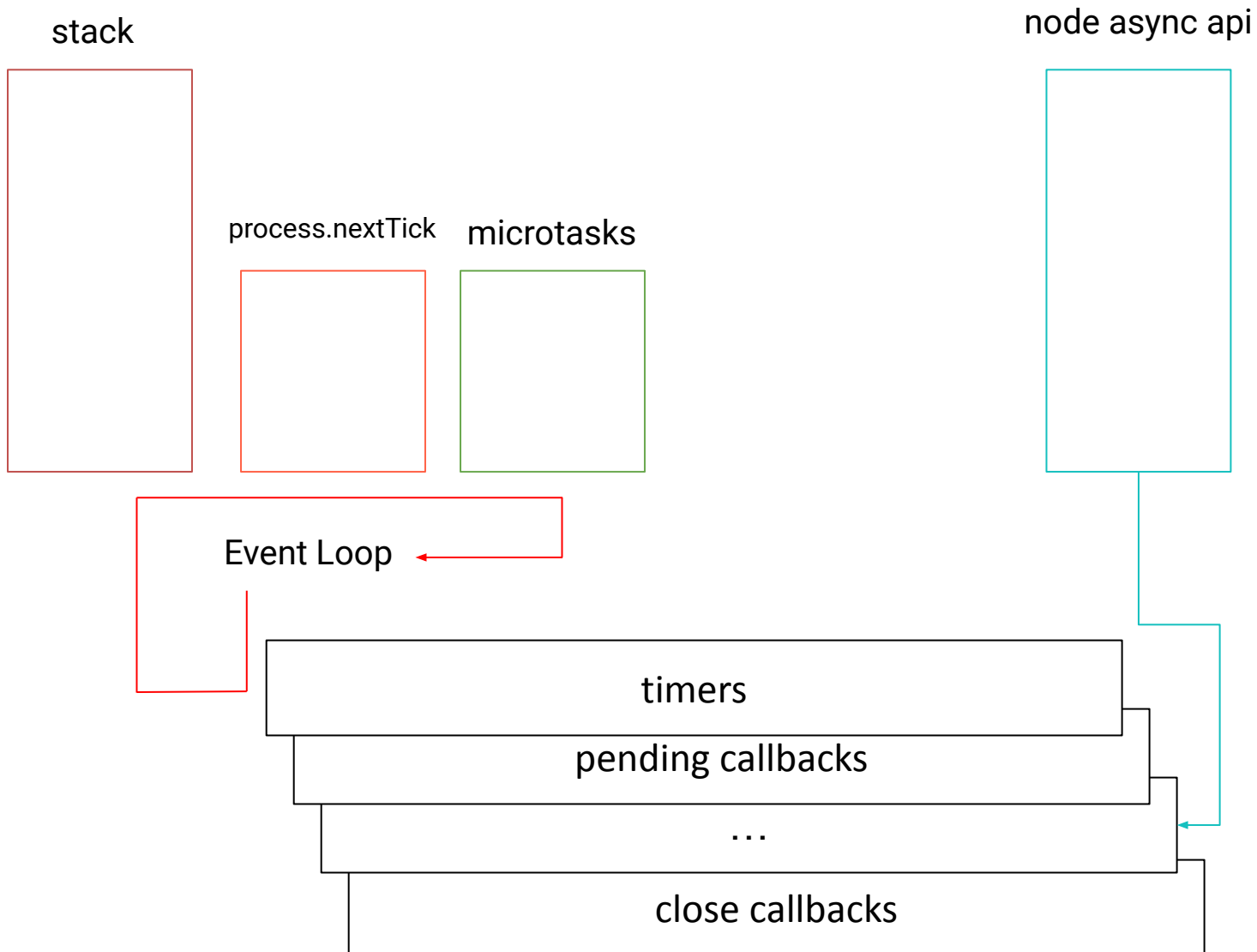
Event Loop

В Node.js всё немного по-другому:

1. Никакого render нет (т.к. отрисовывать нечего)
2. Помимо **microtasks** (callback'и Promise) есть ещё доп.очередь **process.nextTick**
3. Основная очередь не одна, а несколько:



Event Loop



Event Loop

Т.е. сначала обрабатывается синхронный код, затем очереди `process.nextTick` и `microtasks`, а потом поочерёдно (при этом перед переходом к следующей очереди обрабатывается синхронный код, `process.nextTick` и `microtasks`):

- timers
- pending callbacks
- ...
- check
- close callbacks

Результаты вызова асинхронного API встают в соответствующую очередь. Если же мы используем синхронное API (вроде `readFileSync`), то ждём завершения этого вызова в стеке (это считается плохой практикой).



Promise

Помимо стандартного чтения файла (с использованием callback'ов) есть и подход на базе **Promises** (его мы проходили в предыдущих курсах):

```
JS main.js > ...
1  const fs = require('node:fs');
2
3  function handleRecord(record) {
4      const parts = record.split(',');
5      const phone = parts[1];
6      console.log(phone);
7  }
8
9  function handleFile(data) {
10     const lines = data.split('\r\n');
11     lines.forEach(handleRecord); // не вызываем, а передаём название, forEach сам вызовет
12 }
13
14 console.log('before call readFile');
15 fs.promises.readFile('users.txt', 'utf-8').then(handleFile);
16 console.log('after call readFile');
```



Promise

Напоминаем, что **Promise** можно использовать только тогда, когда результат может вернуться всего один раз (в нашем случае – прочитали файл или нет).



Асинхронность

Это и называется асинхронность – наш JS код продолжает выполняться, пока платформа Node.js занимается чтением файла (она берёт это на себя). В нашей небольшой программе это не добавляет сколько бы то ни было плюсов, а наоборот, усложняет саму программу.

Чтобы почувствовать все плюсы асинхронности, нам необходимо от рассмотрения простых примеров с одним файлом (или одним пользователем) перейти к рассмотрению примеров с обработкой многих файлов (или многих пользователей).

Этим мы и займёмся на следующей лекции.



ИТОГИ



Итоги

В этой лекции мы обсудили достаточно много важных моментов:

1. Поговорили о функциях и о том, как хранятся данные, что такое кодировки
2. Поговорили о примитивах и рассмотрели работу с обёрткой `String`
3. Рассмотрели основы асинхронности и EventLoop Node.js



ДОМАШНЕЕ ЗАДАНИЕ



ДЗ 1: Tracks

Используйте пример с урока для того, чтобы вычитать из файла информацию о музыкальных треках и вывести в консоль:

≡ music.txt

```
1 No:1|Title:Foreword|Duration:0:13
2 No:2|Title:Don't Stay|Duration:3:07
3 No:3|Title:Somewhere I Belong|Duration:3:33
4 No:4|Title:Lying from You|Duration:2:55
5 No:5|Title:Hit the Floor|Duration:2:44
6 No:6|Title:Easier to Run|Duration:3:24
7 No:7|Title:Faint|Duration:2:42
8 No:8|Title:Figure.09|Duration:3:17
9 No:9|Title:Breaking the Habit|Duration:3:16
```

Для первого трека через **console.log** должно быть выведено (для остальных по аналогии):

1

Foreword

0:13

Проект должен храниться в каталоге **tracks**



ДЗ №2: Export

Вы уже научились читать из файла, пришло время научиться писать в файл. Что нужно сделать: изучите [документацию на функцию writeFileSync](#) и сохраните в файл **export.txt** следующие данные (это массив из строк):

```
JS main.js > ...  
1  const users = [  
2      '1,Vasya,+9920000000001',  
3      '2,Petya,+9920000000002',  
4      '3,Masha,+9920000000003',  
5  ];
```

После записи файл должен выглядеть следующим образом:

```
≡ users.txt  
1  1,Vasya,+9920000000001  
2  2,Petya,+9920000000002  
3  3,Masha,+9920000000003
```

Скорее всего, вам придётся дополнительно изучить метод массива **join**.



ДЗ №3: Async

Реализуйте предыдущую задачу на базе [writeFile](#), а не [writeFileSync](#).



Спасибо за внимание

alif skills

2023г.

