

# JS Level 3

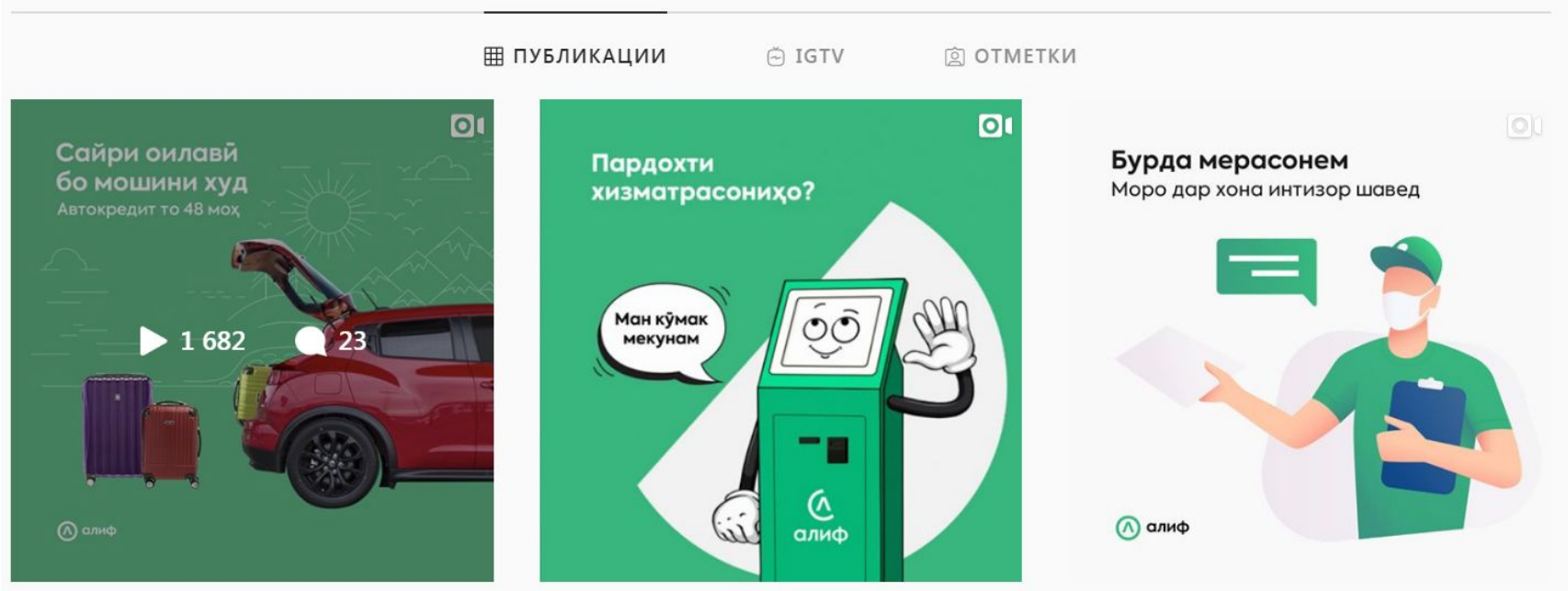
Node.js



# Instagram

Наша с вами задача – создать backend (серверную часть приложения) для социальной сети (frontend для которой мы делали на предыдущих курсах)

В простейшем случае социальная сеть представляет собой набор каких-то элементов (назовём их посты), которые публикуют пользователи:



Скриншот социальной сети Instagram



# Данные

Вы должны понимать, что данные (которые мы видели на скриншоте), а именно:

1. Изображение
2. Ссылка
3. Количество просмотров
4. Количество комментариев

остановимся пока на этих трёх

не хранятся "где-то" в воздухе. Они хранятся на сервере. На фронтенд они (в общем случае) отдаются в виде JSON, при этом для картинок отдаётся ссылка, по которой уже сам браузер (чаще всего) скачивает эту картинку, если мы ссылку подставляем, например, в атрибут `src` объекта [HTMLImageElement](#).

А в каком именно виде они хранятся на сервере – мы сейчас с вами будем выяснять.



# Данные

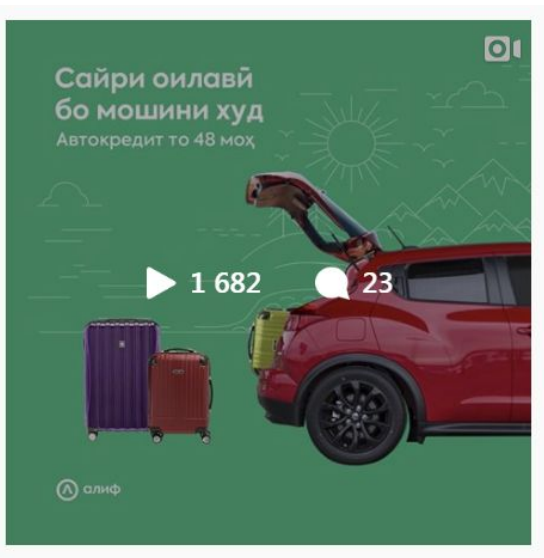
Давайте освежим в памяти некоторые моменты (всё-таки прошло пара месяцев), которые мы разбирали на предыдущих курсах, и посмотрим на особенности Node.js.

Несмотря на то, что ключевые моменты будут вам знакомы, внимательно сравнивайте то, что вы видели в дебаггере браузера, с дебаггером в Node.js. Это позволит вам "прочувствовать" разницу данных окружений.



# Данные

Пока всё, что мы умеем, это использовать конструкцию `console.log` для вывода информации на экран. Давайте попробуем воспользоваться ею, чтобы вывести информацию о первом посте:



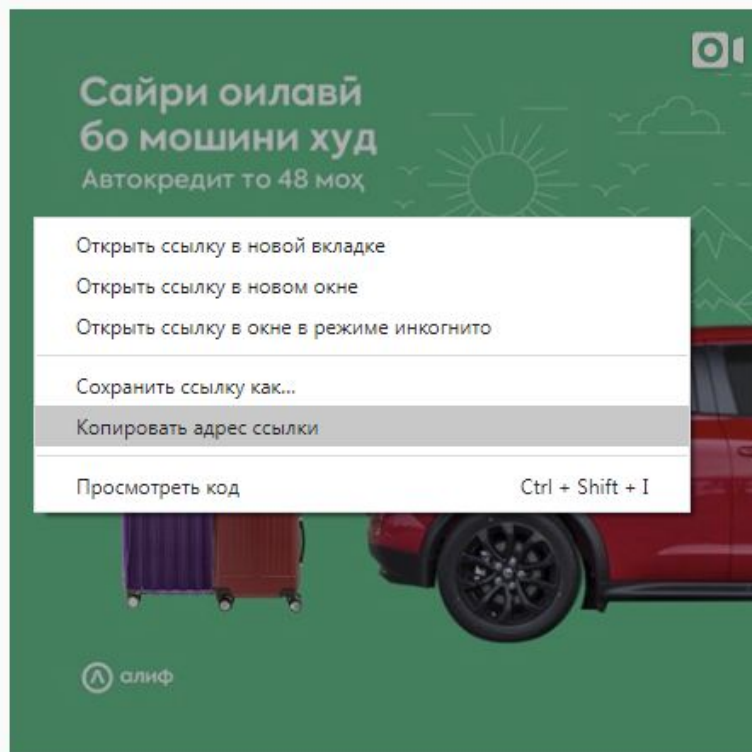
Для этого создадим новый проект (назовём его `social`) и в нём (по традиции файл `main.js`):

```
JS main.js ×
JS main.js
1 console.log('https://www.instagram.com/p/CDMS1vQhJrS/');
2 console.log(1682);
3 console.log(23);
```



# Подсказка

Для того, чтобы скопировать ссылку, необходимо в браузере щёлкнуть правой кнопкой мыши на посте и выбрать **Копировать адрес ссылки**:



# Запуск

Запустим и увидим следующее (нужно как и на прошлой лекции создать конфигурацию запуска):

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
C:\Program Files\nodejs\node.exe c:\projects\social\main.js
Debugger listening on ws://127.0.0.1:50721/40dc3a90-bf5b-4007-bf8b-3b05977ba44c
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
--- Truncated to last 15 messages, set outputCapture to 'all' to see more ---
disconnect...
Process exited with code 0
https://www.instagram.com/p/CDMS1vQhJrS/
1682
23
```



# Данные

А теперь давайте внимательно посмотрим на нашу программу:

```
JS main.js ×   
JS main.js  
1 console.log('https://www.instagram.com/p/CDMS1vQhJrS/');  
2 console.log(1682);  
3 console.log(23);
```

Данные хранятся? Хранятся.

Данные выводятся на экран? Выводятся.

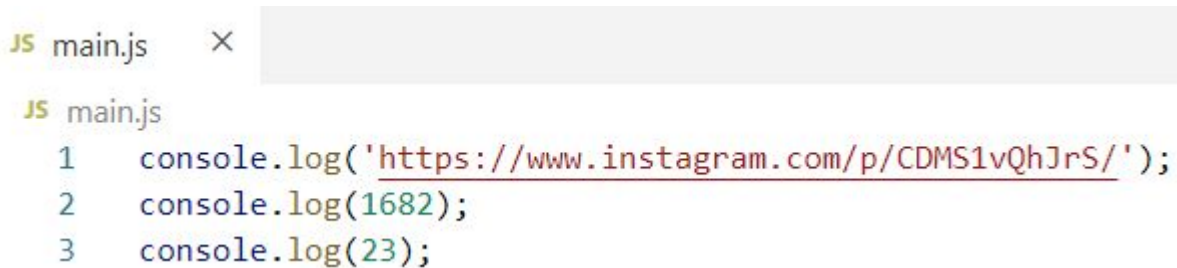
Проблема лишь в том, что не совсем понятно, что это за данные. Например, через неделю или месяц, мы вполне забудем, что такое **1682**, а что такое **23**.





# Данные

Обратите внимание: мы по-разному записали ссылку (в одинарных кавычках) и числа (без кавычек).



```
JS main.js ×  
JS main.js  
1 console.log('https://www.instagram.com/p/CDMS1vQhJrS/');  
2 console.log(1682);  
3 console.log(23);
```

Это потому, что эти данные представляют с собой разные типы: строки и числа. Как на самом деле хранить данные, целиком зависит от того, что вы собираетесь с ними делать. Например, если собираетесь что-то считать (увеличивать/уменьшать на 1 – "лайкать"/"дизлайкать") – тогда лучше числами, если же хотите иметь возможность записывать произвольные значения, например, "HIT", тогда можно и строкой. В нашем конкретном случае – нужны именно числа.



# Refactoring

Выделите всё, что внутри скобок и нажмите **Ctrl + Shift + R**:

```
JS main.js
1 console.log('https://www.instagram.com/p/CDMS1vQhJrS/');
2 console.log(1682);
3 console.log(
4
5
```

Extract to constant in enclosing scope

Extract to function in global scope

Learn more about JS/TS refactorings

Выберите первую опцию (для этого достаточно просто нажать **Enter**) и во всплывающем окошке замените **newLocal** на **views** (что означает просмотры):

```
JS main.js > ...
1 console.log('https://www.instagram.com/p/CDMS1vQhJrS/');
2 const newLocal = 1682;
3 console.log(newLocal);
4 console.log(views
5
6
```

Enter to Rename, Shift+Enter to Preview



# Refactoring

В итоге у вас должно получиться вот так:

```
JS main.js > ...  
1  console.log('https://www.instagram.com/p/CDMS1vQhJrS/');  
2  const views = 1682;  
3  console.log(views);  
4  console.log(23);
```

Проделайте то же самое с оставшимися данными, пока у вас не получится вот так:

```
JS main.js > ...  
1  const url = 'https://www.instagram.com/p/CDMS1vQhJrS/';  
2  console.log(url);  
3  const views = 1682;  
4  console.log(views);  
5  const comments = 23;  
6  console.log(comments);
```

**Важно:** учитесь работать со своим редактором кода (в данном случае VSCode). Это вам сэкономит большое количество времени и убережёт от множества ошибок.



# Refactoring

Refactoring (рефакторинг) - это улучшение существующего кода без изменения функциональности.


Что это значит? Это значит, что если мы запустим нашу программу, то ничего не изменится. Выводится будут всё те же строчки.

Так в чём же мы её улучшили? Её стало проще читать и понимать нам самим. В мире программирования – это самое важное. Если вы не понимаете, что делает программа, вы не сможете её изменить так, чтобы она работала по-другому (а это вам придётся делать очень часто).



# Переменные

Давайте посмотрим, как это всё работает в отладчике. Поставим точку остановки на первую строку:



```
✓ VARIABLES
  ✓ Local
    __dirname: 'c:\projects\social'
    __filename: 'c:\projects\social\main.js'
    comments: undefined
  > exports: {}
  > module: Module {id: '.', exports: {...}, parent: ...
  > require: f require(path) {\n    try {\n      ex...
    url: undefined
    views: undefined
  > this: Object
  > Global
```

```
JS main.js > [?] url
1  const url = 'https://www.instagram.com/p/CDMS1vQhJrS/';
2  console.log(url);
3  const views = '1682';
4  console.log(views);
5  const comments = '23';
6  console.log(comments);
7
8
```

В боковой панели, в разделе **Variables** в блоке **Local** отображаются имена, которые существуют в вашей программе (в локальной области видимости).

Но мы же объявили только три имени, откуда взялось остальное? С этим сейчас разберёмся.



# Переменные

По мере того, как мы будем проходить нашу программу по шагам, значения в блоке **Local** будут заполняться (причём подсвечиваться будет то, что поменялось на предыдущем шаге):

```
✓ VARIABLES
  ✓ Local
    __dirname: 'c:\projects\social'
    __filename: 'c:\projects\social\main.js'
    comments: undefined
    > exports: {}
    > module: Module {id: '.', exports: {...}, ...
    > require: f require(path) {\n    try {\n...
    ! → url: 'https://www.instagram.com/p/CDMS1...'
    views: undefined
    > this: Object
    > Global
```

```
JS main.js > ...
1  const url = 'https://www.instagram.com/p/CDMS1vQhJrS/';
2  console.log(url);
3  const views = 1682;
4  console.log(views);
5  const comments = 23;
6  console.log(comments);
7
8
```



# Переменные

Теперь давайте разбираться с тем, откуда взялись остальные имена, а также что такое **Local** и что такое **Global**.

Как работает Node.js:

1. Он читает файл с нашей программой
2. Разбирает все конструкции, которые там написаны
3. Запускает их на исполнение

Во время исполнения Node.js сам инжектирует (значит подставляет) эти имена ✓, чтобы мы могли их использовать.

Назначение некоторых (с остальными мы познакомимся чуть позже):

- **\_\_dirname** – путь к каталогу, в котором располагается исполняемый файл **main.js**
- **\_\_filename** – путь к файлу, который в данный момент исполняется

```
▼ VARIABLES
  ▼ Local
  ✓ __dirname: 'c:\projects\social'
  ✓ __filename: 'c:\projects\social\main.js'
    comments: undefined
  ✓ > exports: {}
  ✓ > module: Module {id: '.', exports: {...}, parent: ...
  ✓ > require: f require(path) {\n    try {\n      ex...
    url: undefined
    views: undefined
  ✓ > this: Object
    > Global
```



# ECMAScript & Node.js

JavaScript – это язык (набор правил). Эти правила описаны в специальном стандарте, который называется ECMAScript. Сейчас стандарт обновляется каждый год – добавляются новые возможности, а какие-то объявляются устаревшими.

Но язык не может существовать сам по себе. Для того, чтобы язык приносил реальную пользу, нужно какое-то окружение, которое его поддерживает. Например, раньше были компакт-диски (CD), которые поддерживали плееры. Сейчас их заменили флешки. Поэтому если вы даже захотите "послушать", что записано на этом CD, у вас скорее всего не получится, поскольку очень трудно найти устройство, которое поддерживает их.

С языками всё то же самое – нужно окружение, которое позволяет выполнять программы на этом языке.





# ECMAScript & Node.js

Ключевые окружения для JS – это браузер и Node.js (хотя они не единственные: например, для мобильных приложений, которые можно разрабатывать с помощью JS, будет совсем другое окружение).

Так вот это самое окружение может добавлять в язык объекты (те наши переменные, которые объявляли не мы), возможности и конструкции, которые в стандарте не описаны.

Этим и занимается Node.js, браузер и другие окружения (поэтому мы часто будем встречать то, что не описано в стандарте и что добавляли не мы).



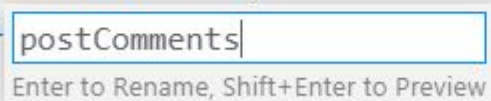
# Имена переменных

Имена переменных принято писать с маленькой буквы, каждое следующее слово начиная с большой. Например, если бы мы писали названия из двух слов, то это выглядело бы вот так:

```
JS main.js > ...
1  const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';
2  console.log(postUrl);
3  const postViews = 1682;
4  console.log(postViews);
5  const postComments = 23;
6  console.log(postComments);
```

Важно: используйте для переименования **F2** (не переименовывайте руками). Для этого поставьте курсор на имя, нажмите **F2**, измените название, а затем нажмите на **Enter**:

```
5  const comments = 23;
6  console.log(postComments)
7
8
```



# Жизнь переменных

Давайте вернёмся к дебаггеру:

```
✓ VARIABLES
  ✓ Local
    __dirname: 'c:\projects\social'
    __filename: 'c:\projects\social\main.js'
    > exports: {}
    > module: Module {id: '.', exports: {...}, ...
      postComments: undefined
      postUrl: undefined
      postViews: undefined
    > require: f require(path) {\n    try {\n...
    > this: Object
  > Global
```

```
JS main.js > [?] postUrl
1  const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';
2  console.log(postUrl);
3  const postViews = 1682;
4  console.log(postViews);
5  const postComments = 23;
6  console.log(postComments);
7
8
```

Панелька **Variables** показывает, что имена **postComments**, **postUrl** и **postViews** уже существуют и им присвоено значение **undefined**.



# Жизнь переменных

Так же, как и в браузерах, в Node.js к переменной можно обращаться только тогда, когда Node.js уже выполнил строку, в которой есть `const` и имя переменной (это называется объявление переменной). Это определяется стандартом ECMAScript, поэтому поведение одинаково и в браузере, и в Node.js.



# Жизнь переменных

Если мы попробуем обратиться раньше, то получим ошибку:

JS main.js > ...

```
1 console.log(postUrl); // обращаемся раньше, чем объявляем
2 const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';
3 const postViews = 1682;
4 console.log(postViews);
5 const postComments = 23;
6 console.log(postComments);
```



--- Truncated to last 15 messages, set outputCapture to 'all' to see more ---  
ReferenceError: postUrl is not defined

```
at Object.<anonymous> (c:\projects\social\main.js:1:13)
at Module._compile (internal/modules/cjs/loader.js:778:30)
at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
at Module.load (internal/modules/cjs/loader.js:653:32)
at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
at Function.Module._load (internal/modules/cjs/loader.js:585:3)
at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
at startup (internal/bootstrap/node.js:283:19)
at bootstrapNodeJSCore (internal/bootstrap/node.js:622:3)
```

Waiting for the debugger to disconnect...

Process exited with code 1



# Комментарии

Часть строки, начинающаяся с `//` (окрашена зелёным цветом) - это комментарий. Комментарий – это возможность программиста оставлять пояснения к какому-то участку своего кода.

JS main.js > ...

```
1 console.log(postUrl); // обращаемся раньше, чем объявляем
2 const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';
3 const postViews = 1682;
4 console.log(postViews);
5 const postComments = 23;
6 console.log(postComments);
```

Комментарии игнорируются Node.js (он делает вид, что не видит их). В JS два вида комментариев:

1. `//` – строчные (от символов `//` и до конца строки)
2. `/* ... */` – блочные (всё, что между символами `/*` и `*/` и есть комментарий)



# Комментарии

В рамках лекций мы будем использовать комментарии, чтобы объяснить некоторые нюансы. Вам же (в вашем коде), в большинстве случаев комментарии не нужны, поскольку считается, что ваша программа должна быть понятна без комментариев.

Чуть позже мы поговорим о том, когда они действительно нужны. Пока же бот будет возвращать вам ДЗ, если найдёт в них комментарии.



# Комментарии

Комментировать можно и участки кода, например, если мы сделаем вот так, то Node.js будет полностью игнорировать первую строку и наша программа будет работать без ошибок:

```
JS main.js > ...  
1  // console.log(postUrl); // обращаемся раньше, чем объявляем  
2  const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';  
3  const postViews = 1682;  
4  console.log(postViews);  
5  const postComments = 23;  
6  console.log(postComments);
```

Закомментированный код – это ещё хуже, чем ненужные комментарии.

Закомментированного кода в ваших работах быть не должно.

Мы рекомендуем вам самостоятельно познакомиться с Git, который позволит вам хранить всю историю вашего проекта, и не нужно будет где-то "про запас" хранить закомментированный код.





# Жизнь переменных

Вернёмся к коду, в котором у нас произошла ошибка:

JS main.js > ...

```
1 console.log(postUrl); // обращаемся раньше, чем объявляем
2 const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';
3 const postViews = 1682;
4 console.log(postViews);
5 const postComments = 23;
6 console.log(postComments);
```



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

C:\Program Files\nodejs\node.exe .\main.js

Process exited with code 1

Uncaught ReferenceError ReferenceError: Cannot access 'postUrl' before initialization

at <anonymous> (c:\projects\social\main.js:1:13)  
at Module.\_compile (internal/modules/cjs/loader:1254:14)  
> at Module.\_extensions..js (internal/modules/cjs/loader:1308:10)  
at Module.load (internal/modules/cjs/loader:1117:32)  
at Module.\_load (internal/modules/cjs/loader:958:12)  
at executeUserEntryPoint (internal/modules/run\_main:81:12)  
at <anonymous> (internal/main/run\_main\_module:23:47)



# ReferenceError

В логе ошибки сбоку отображается, где конкретно произошла ошибка, можно кликнуть и попасть именно в эту точку (в редакторе откроется код):



The screenshot shows the VS Code interface with the 'DEBUG CONSOLE' tab active. The console displays the following output:

```
C:\Program Files\nodejs\node.exe .\main.js
Process exited with code 1
Uncaught ReferenceError: ReferenceError: Cannot access 'postUrl' before initialization
    at <anonymous> (c:\projects\social\main.js:1:13)
    at Module._compile (internal/modules/cjs/loader:1254:14)
    at Module._extensions..js (internal/modules/cjs/loader:1308:10)
    at Module.load (internal/modules/cjs/loader:1117:32)
    at Module._load (internal/modules/cjs/loader:958:12)
    at executeUserEntryPoint (internal/modules/run_main:81:12)
    at <anonymous> (internal/main/run_main_module:23:47)
```

On the right side of the console, there is a link labeled `main.js:1`. A red circle with an arrow points to this link, and the text 'сюда можно кликнуть' (you can click here) is written next to it.



# Важно

Нужно уметь читать ошибки – от того, насколько быстро вы будете понимать, по какой причине произошла ошибка, во многом будет зависеть ваш успех как программиста.

Для перевода текста ошибки вы вполне можете использовать переводчики, например, [Google Translate](#).



# ReferenceError

Теперь давайте вспоминать: **ReferenceError** возникает тогда, когда вы пытаетесь обратиться к имени, которое не доступно (в нашем случае, недоступно оно потому, что будет объявлено только на следующей строке):

JS main.js > ...

```
1 console.log(postUrl); // обращаемся раньше, чем объявляем
2 const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';
3 const postViews = 1682;
4 console.log(postViews);
5 const postComments = 23;
6 console.log(postComments);
```




# ReferenceError

Обычная программа выполняется по строчкам сверху вниз:



```
JS main.js > ...  
1  const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';  
2  console.log(postUrl);  
3  const postViews = 1682;  
4  console.log(postViews);  
5  const postComments = 23;  
6  console.log(postComments);
```

Но если возникает ошибка, Node.js останавливается в том месте, где она произошла и следующие строки просто не выполняются:



```
JS main.js > ...  
1  console.log(postUrl); // обращаемся раньше, чем объявляем  
2  const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';  
3  const postViews = 1682;  
4  console.log(postViews);  
5  const postComments = 23;  
6  console.log(postComments);
```

эти строки не выполняются



# Важно

Возникновение ошибки приводит к "обрушению" нашей программы – т.е. работа программы экстренно завершается.



# TDZ

В дебаггере мы видели, что имя существует и значение у него **undefined**. Всё дело в том, что когда вы объявляете переменную, она "занимает" под себя имя. Но этим именем можно воспользоваться только тогда, когда переменная будет объявлена.

имя существует, но  
воспользоваться ещё —  
нельзя

```
JS main.js > ...  
1 console.log(postUrl); // обращаемся раньше, чем объявляем  
2 const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';  
3 const postViews = 1682;  
4 console.log(postViews);  
5 const postComments = 23;  
6 console.log(postComments);
```

Это как с нашими лекциями: лекция уже существует и загружена в систему, но воспользоваться вы ею не можете, пока не придёт время публикации.

Эта область, в которой имя уже существует, но воспользоваться им нельзя, называется TDZ (Temporal Dead Zone) – временно мёртвая зона.



# const

Исправим ошибку, после чего представим, что кто-то оставил новый комментарий к нашему посту. Значит в программе мы должны увеличить количество комментариев в посте на 1.

```
JS main.js > ...  
1  const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';  
2  console.log(postUrl);  
3  const postViews = 1682;  
4  console.log(postViews);  
5  const postComments = 23;  
6  console.log(postComments);  
7  
8  postComments = postComments + 1;  
9  console.log(postComments);
```

Пока мы выполняем строки 8-9 "просто так". Но уже на следующей лекции они будут выполняться в ответ на действия пользователя (т.е. тогда, когда пользователь заполнит форму комментария и нажмёт на кнопку "Отправить").





# const

Ключевое слово **const** позволяет всего один раз связать имя и значение (чаще говорят присвоить). Например, если мы попробуем выполнить следующий код, то получим ошибку:

```
JS main.js > ...  
1  const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';  
2  console.log(postUrl); // обращаемся раньше, чем объявляем  
3  const postViews = 1682;  
4  console.log(postViews);  
5  const postComments = 23;  
6  console.log(postComments);  
7  
8  postComments = postComments + 1;  
9  console.log(postComments);
```

В случае примитивов это означает, что значение будет неизменно на протяжении всей программы.



# const

Вы уже умеете читать ошибки: здесь написано, что выполняется присваивание "константной переменной". Давайте разбираться, как это должно было работать.

```
Process exited with code 1
```

```
Uncaught TypeError: Assignment to constant variable. ←
```

```
main.js:8
```

```
    at <anonymous> (c:\projects\social\main.js:8:14)  
    at Module._compile (internal/modules/cjs/loader:1254:14)  
    at Module._extensions..js (internal/modules/cjs/loader:1308:10)  
>   at Module.load (internal/modules/cjs/loader:1117:32)  
    at Module._load (internal/modules/cjs/loader:958:12)  
    at executeUserEntryPoint (internal/modules/run_main:81:12)  
    at <anonymous> (internal/main/run_main_module:23:47)
```

Примечание\*: первой строкой на скриншоте вы видите "Process exited with code 1". В большинстве операционных систем (к которым относятся Linux, Windows и MacOS) используется соглашение: любой ненулевой код завершения программы говорит о том, что она завершилась с ошибкой. Вы будете встречать использование этого соглашения повсеместно.



# const

**const** позволяет связать имя со значением всего один раз. И сам следит за тем, чтобы вы потом случайно не связали имя с другим значением (это достаточно частая причина ошибок).

Правильный вариант с **const** выглядел бы вот так:

```
JS main.js > ...
1  const postUrl = 'https://www.instagram.com/p/CDMS1vQhJrS/';
2  console.log(postUrl);
3  const postViews = 1682;
4  console.log(postViews);
5  const postComments = 23;
6  console.log(postComments);
7
8  const updatedPostComments = postComments + 1;
9  console.log(updatedPostComments);
```

Мы создали новую переменную, чтобы хранить результат.



# const

Относитесь к `const` как к одноразовой посуде – её нельзя переиспользовать, если она уже кем-то использована. Это защищает вас от того, чтобы случайно где-то в глубине программы не присвоить имени другого значения.

Кроме того, создание переменных – дёшево. Не стоит бояться создавать новые переменные, если они вам нужны.

Использование `const` считается хорошей практикой.



# Хорошие практики

"Хорошие практики" – это набор рекомендаций, выработанных программистами при написании кода.

Например, компания Airbnb подготовила целое руководство (<https://github.com/airbnb/javascript>), в котором описывает, как лучше писать код на JS.

Не все программисты согласны со всеми пунктами этого руководства (есть и другие руководства). Но в целом, мы будем ориентироваться на них.



# Переменные

Ключевые моменты: переменная – это некоторое имя с привязанным к нему значением. Значение какого типа хранится в переменной определяет то, какие операции с ней (переменной) можно выполнять.

1. Имя
2. Значение
3. Тип\* (определяется значением)

Давайте подробнее поговорим про типы.

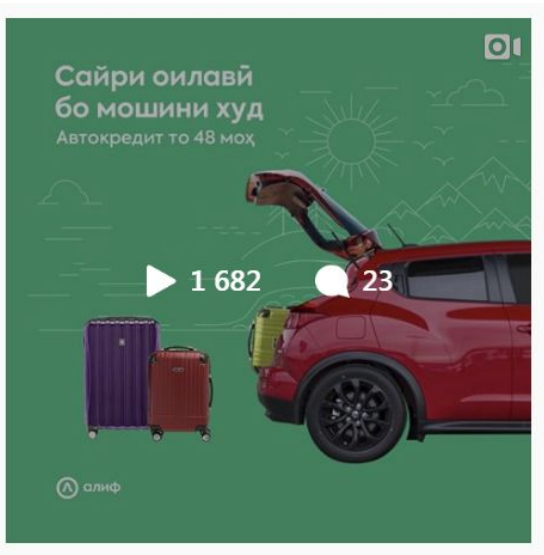


# ОБЪЕКТЫ



# Объекты

Давайте ещё раз внимательно посмотрим на скриншот нашей социальной сети:



В реальной жизни, наш мозг воспринимает то, что мы видим на этом скриншоте, как один "объект".

Это достаточно важный момент, потому что позволяет нам понять, как всё устроено.

Например, когда вы видите человека на улице, вы не говорите "идут руки, ноги, голова, туловище и (длинное перечисление остальных составляющих)". Вы говорите просто: "идёт человек".





# Объекты

Почему мы так делаем? Потому что так удобнее "манипулировать" этими объектами. Удобнее объяснять другому человеку (он скорее всего не поймёт о чём речь и подумает, что вы загадываете ему загадки, если вы скажете "2 руки, 2 ноги, голова и туловище" вместо "человек").

Этот подход, когда мы составляем сложные объекты из более простых называется композиция.



# Объекты

В JS мы точно так же можем собирать объекты из примитивов и даже других объектов. Но давайте сначала посмотрим на то, как объекты можно собирать из примитивов.

Для это перепишем нашу программу следующим образом:

```
JS main.js > ...  
1  const post = {  
2    url: 'https://www.instagram.com/p/CDMS1vQhJrS/',  
3    views: 1682,  
4    comments: 23,  
5  };  
6  console.log(post);
```



# Объекты

Что произошло? Мы создали одну переменную `post`, в которую записали три свойства:

- `url`
- `views`
- `comments`



# Объекты

```
JS main.js > ...  
1  const post = {  
2    url: 'https://www.instagram.com/p/CDMS1vQhJrS/',  
3    views: 1682,  
4    comments: 23,  
5  };  
6  console.log(post);
```

Обратите внимание: теперь мы можем манипулировать этим объектом так же, как в реальной жизни - просто говорить "пост". И распечатывать в консоль (6 строка) целиком весь пост:

```
{ url: 'https://www.instagram.com/p/CDMS1vQhJrS/',  
  views: 1682,  
  comments: 23 }
```



# Объекты

Обратите внимание, свойства пишутся с отступом, чтобы визуально было видно, что они относятся именно к объекту:

```
JS main.js > ...  
1  const post = {  
2    url: 'https://www.instagram.com/p/CDMS1vQhJrS/',  
3    views: 1682,  
4    comments: 23,  
5  };  
6  console.log(post);
```

Если же вы сделали вот так (или так получилось), ничего страшного, нажмите **Alt + Shift + F** и VSCode сам в нужных местах поставит пробелы:

```
JS main.js > ...  
1  const post = {  
2  url: 'https://www.instagram.com/p/CDMS1vQhJrS/',  
3  views: 1682,  
4  comments: 23,  
5  };  
6  console.log(post);
```



# Обращение к свойствам

Теперь возникает вопрос: а как обратиться к свойствам, например, прочитать конкретное свойство или что-то записать в него?

Для этого у нас есть оператор `.` (точка). Используется он достаточно просто – сначала мы пишем имя, в котором хранится наш объект, а затем имя свойства

```
JS main.js > ...  
1  const post = {  
2    url: 'https://www.instagram.com/p/CDMS1vQhJrS/',  
3    views: 1682,  
4    comments: 23,  
5  };  
6  console.log(post.url);
```



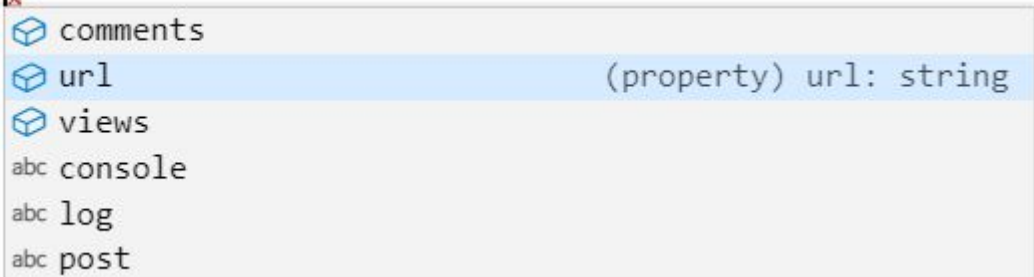
Работает это следующим образом: JS видит имя `post` и находит объект, который привязан к этому имени. Дальше он видит `.url`, и ищет внутри объекта свойство `url`.



# Автоподсказки

Обязательно пользуйтесь автоподсказками: когда вы ставите точку после имени, VSCode сам вам предлагает (если может) имена свойств, вам нужно только выбрать нужное (и не писать его целиком):

```
console.log(post.);
```

A screenshot of the VS Code editor showing an auto-completion dropdown menu. The code in the editor is `console.log(post.);`. The dropdown menu is open, showing a list of suggestions. The first three suggestions are `comments`, `url`, and `views`, each preceded by a blue cube icon. The `url` suggestion is highlighted in blue and includes the text `(property) url: string` to its right. Below these are three suggestions preceded by the text `abc`: `console`, `log`, and `post`.

- comments
- url (property) url: string
- views
- abc console
- abc log
- abc post



# Обращение к свойствам

Для записи нового значения в свойство, достаточно использовать уже знакомый нам оператор `=`:

```
JS main.js > ...  
1  const post = {  
2    url: 'https://www.instagram.com/p/CDMS1vQhJrS/',  
3    views: 1682,  
4    comments: 23,  
5  };  
6  console.log(post.comments);  
7  
8  post.comments = post.comments + 1;  
9  console.log(post.comments);
```

Обратите внимание: `const` означает, что вы не можете в `post` положить другой объект. Но при этом вы можете поменять свойства самого объекта. Это как с вами – вы для своих родителей и друзей один и тот же объект, но ваши свойства могут меняться (например, длина волос или их цвет).





# Обращение к свойствам

JS в некоторых аспектах достаточно лояльно относится к тому, что вы делаете. Например, вы спокойно можете обратиться к свойству, которого не существует:

```
JS main.js > ...  
1  const post = {  
2    url: 'https://www.instagram.com/p/CDMS1vQhJrS/',  
3    views: 1682,  
4    comments: 23,  
5  };  
6  console.log(post.image);
```

Никакой ошибки не будет, мы просто получим значение **undefined** (т.е. не задано).



# Обращение к свойствам

А если мы попробуем записать в несуществующее свойство, то JS "молча" его добавит в наш объект (и начиная со строки, в которой мы это свойство добавили, оно будет доступно):

```
JS main.js > ...
1  const post = {
2    url: 'https://www.instagram.com/p/CDMS1vQhJrS/',
3    views: 1682,
4    comments: 23,
5  };
6  console.log(post.image); // undefined
7
8  post.image = 'https://www.instagram.com/...';
9  console.log(post.image); // 'https://www.instagram.com/...'
10
11 console.log(post); // выведется со свойством image
```



# Обращение к свойствам

Причём JS позволяет удалить свойство (с помощью оператора **delete**), и с этой строки всё будет выглядеть так, как будто в объекте никогда этого свойства не

```
JS main.js > ...
1  const post = {
2      url: 'https://www.instagram.com/p/CDMS1vQhJrS/',
3      views: 1682,
4      comments: 23,
5  };
6  console.log(post.image); // undefined
7
8  post.image = 'https://www.instagram.com/...';
9  console.log(post.image); // 'https://www.instagram.com/...'
10
11 console.log(post); // выведется со свойством image
12
13 delete post.image;
14
15 console.log(post.image); // undefined
16 console.log(post); // выведется без свойства image
```



# Свойства

Вы должны быть аккуратными при обращении к свойствам, которых нет, и добавлению их. Иногда, из-за одной небольшой опечатки в имени свойства, вы будете обращаться не к тому свойству. Это будет приводить к ошибкам.

В некоторых проектах вы будете встречать использование [Object.preventExtensions](#), [Object.seal](#) или [Object.freeze](#) (либо различных библиотек, позволяющих добиться аналогичного эффекта), позволяющих ограничить наши возможности в части того, что мы можем делать с объектом.

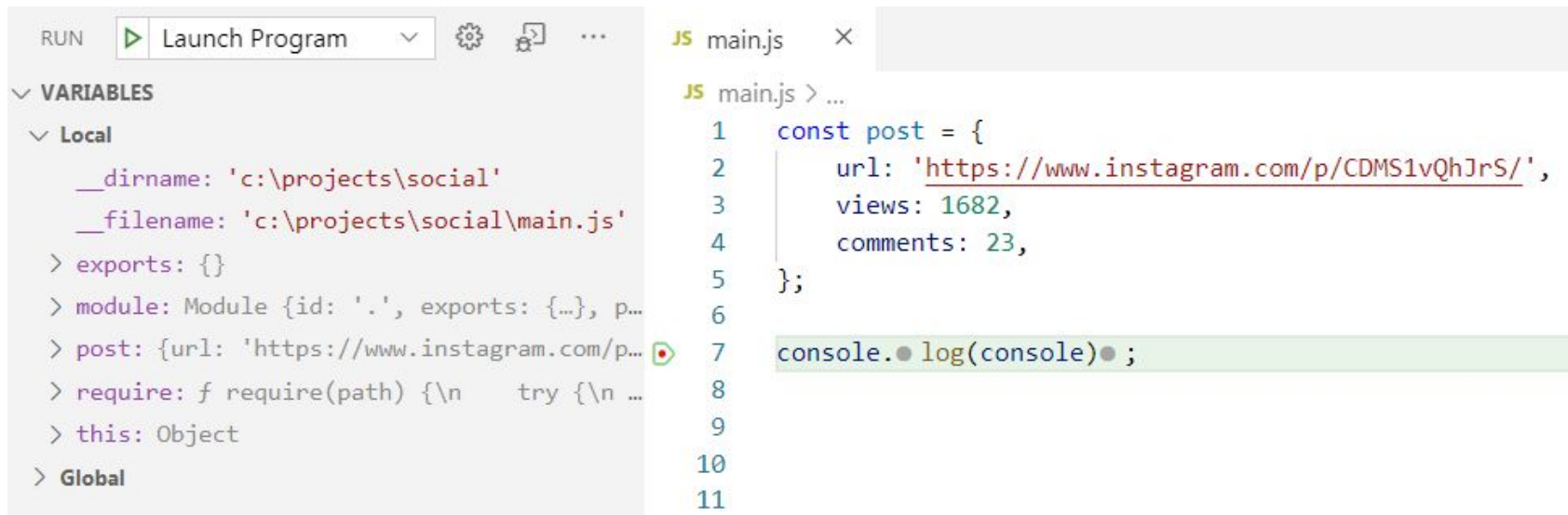


**GLOBAL**



# global

В дебаггере, помимо блока **Local**, есть ещё блок **Global**:



The screenshot shows a JavaScript debugger interface. On the left, the 'VARIABLES' pane is expanded, showing the 'Local' scope. It contains variables: `__dirname` with value `'c:\projects\social'`, `__filename` with value `'c:\projects\social\main.js'`, `exports` with value `{}`, `module` with value `Module {id: '.', exports: {...}, p...`, `post` with value `{url: 'https://www.instagram.com/p/CDMS1vQhJrS/'}`, `require` with value `f require(path) {\n try {\n ...`, and `this` with value `Object`. Below the 'Local' scope, the 'Global' scope is also visible. On the right, the 'main.js' file is open, showing the following code:

```
1  const post = {
2    url: 'https://www.instagram.com/p/CDMS1vQhJrS/',
3    views: 1682,
4    comments: 23,
5  };
6
7  console.log(console);
8
9
10
11
```

Оказывается, в JS есть специальный объект, который называется глобальным (он доступен под именем **globalThis**). Это универсальное имя, которое унифицирует доступ к глобальному объекту в браузере (**window**) и в Node.js (**global**).



# global

И JS работает следующим образом: когда вы пишете любое имя, он сначала ищет его среди тех, которые создали вы (через `const` или `let`), затем смотрит то, что инжектировал Node.js\*, и если не нашёл ни там, ни там, ищет это имя как свойство объекта `globalThis` (тот самый блок `Global`):

```
Global
> Array: f Array()
> ArrayBuffer: f ArrayBuffer()
> Atomics: Atomics {load: f, store: f, add: f, sub: f, and: f}
> BigInt: f BigInt()
> BigInt64Array: f BigInt64Array()
> BigUint64Array: f BigUint64Array()
> Boolean: f Boolean()
> Buffer: f Buffer(arg, encodingOrOffset, length) {\n  showFlagged...
> clearImmediate: f clearImmediate(immediate) {\n  if (!immediate ...
> clearInterval: f clearInterval(timer) {\n  // clearTimeout and c...
> clearTimeout: f clearTimeout(timer) {\n  if (timer && timer._onT...
> console: Console {log: f, debug: f, info: f, dirxml: f, warn: f}
```

Примечание\*: если мы работаем в Node.js



# global

Т.е. если мы напишем вот так (см.скриншот), то это будет то же самое, что мы писали на первой лекции:

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

Как это работает? Мы берём имя **globalThis** (а там хранится объект), ищем среди свойств этого объекта свойство с именем **console** (а там тоже хранится объект), и среди свойств уже объекта **console** ищем свойство с именем **log**.

Но так, конечно же, никто не пишет, потому что зачем писать лишнее (особенно если можно не писать). Хороший программист – ленивый программист, пишет меньше кода\*. Чем меньше кода, тем меньше ошибок.

Примечание\*: меньше кода не значит, что он делает меньше. Это значит, что меньшим количеством кода он может сделать больше полезных функций для продукта.





# global

В глобальном объекте хранятся типы данных, которые описаны в спецификации ECMAScript (сокращённо её называют ES) + те, что туда добавляет сам Node.js.

Например, типы для работы с набором элементов (массивом) или с набором байтов (данных в том виде, в котором они хранятся в памяти компьютера):

```
Global
> Array: f Array()
> ArrayBuffer: f ArrayBuffer()
```

Но среди них мы не найдём ни типов для работы с файлами, ничего такого, о чём мы говорили, когда только начинали говорить про Node.js.

Как же тогда всё устроено?



# MODULES



# modules

В Node.js есть специальная система модулей, как раз и предоставляющих нужные типы данных для каждого конкретного случая: работы с файлами, работы с сетью и т.д. Полный список модулей вы можете увидеть [на странице документации](#):

- File system
- Globals
- HTTP
- HTTP/2
- HTTPS
- Inspector
- Internationalization
- Modules
- Net
- OS
- Path
- Performance hooks
- Policies
- Process

...

**Важно:** эти модули есть в Node.js, но их нет в браузере!

Поэтому вы не сможете их использовать во frontend-приложениях!



# fs

Если мы зайдём внутрь конкретного модуля ([File System](#)) и пролистаем оглавление, то увидим (см. следующий слайд):

1. Название
2. Насколько модуль стабилен (есть экспериментальные модули)
3. Краткое описание
4. Пример использования



# 1 File system

2 Stability: 2 - Stable

Source Code: [lib/fs.js](#)

3 The `node:fs` module enables interacting with the file system in a way modeled on standard POSIX functions.

To use the promise-based APIs:

4 

```
import * as fs from 'node:fs/promises';
```

COPY

CJS ☒ ESM

To use the callback and sync APIs:

```
import * as fs from 'node:fs';
```

COPY

CJS ☒ ESM

All file system operations have synchronous, callback, and promise-based forms, and are accessible using both CommonJS syntax and ES6 Modules (ESM).



# ESM vs CJS

Обратите внимание: в примерах использования вы видите переключатель CJS/ESM:

```
import * as fs from 'node:fs/promises';
```

[COPY](#)

CJS ☒ ESM

В первых лекциях мы будем использовать CJS (в предыдущих курсах у вас был ESM) – по причине того, что он пока более распространён в Node.js проектах:

```
const fs = require('node:fs/promises');
```

[COPY](#)

CJS ☒ ESM

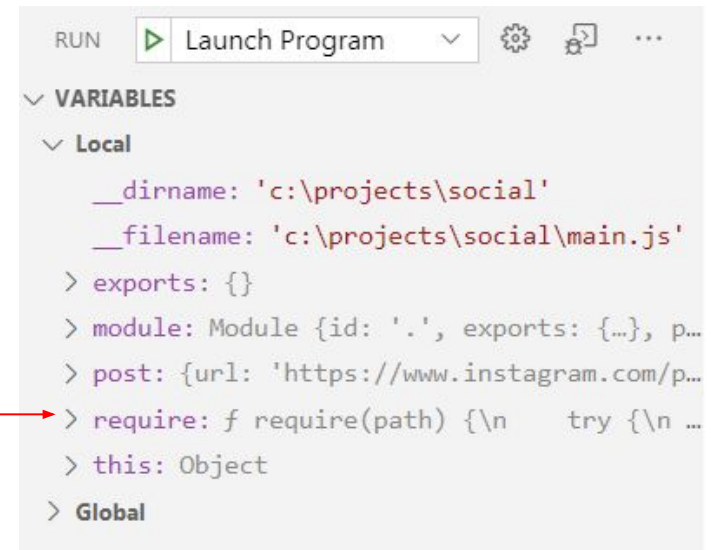
К разговору о системах модулей мы ещё вернёмся чуть позже, когда будем писать собственные модули (делая их доступными для использования в разных системах модулей).



# fs

А теперь вспомним ещё раз панельку дебаггера: в ней мы увидим имя **require** с буковкой **f**.

Что это значит? Это значит, что **require** – это функция.



# Функции

Давайте посмотрим, как использовать функцию **require**:

```
const fs = require('node:fs');
```

1. Чтобы вызвать (запустить на исполнение) функцию, нужно написать имя функции, после него поставить круглые скобки
2. Внутри скобок пишутся входные параметры (в нашем случае - это просто строка с именем модуля)
3. Чтобы получить возвращаемое значение, нужно просто "сохранить" его в переменную (**const** + имя переменной + оператор **=**).

Общепринято, что при подключении модуля через **require** вы сохраняете его в переменную с тем же именем, что и имя модуля (**fs**).





# Promises API

Для части модулей доступно несколько версий API:

1. На основе callback'ов
2. На основе Promise

В случае fs это будет:

1. `require('node:fs')` – на основе callback'ов
2. `require('node:fs/promises')` – на основе Promise

Мы для начала будем использовать то, что основано на callback'ах.



# Promises API

**Важно:** в старых версиях Node.js использовался следующий формат подключения стандартных модулей (вы его до сих пор можете встретить во многих статьях и руководствах):

```
const fs = require('fs');
```

т.е. без префикса **node:**

Сейчас подобный синтаксис не является рекомендуемым, вместо этого необходимо явно прописывать префикс **node:** для всех модулей, входящих в стандартную библиотеку (т.е. перечисленных на странице <https://nodejs.org/api/>).



# Функции

Давайте посмотрим, что мы получим в результате: мы получили объект (объект можно раскрыть, нажав на **>**), внутри которого расположены свойства. Но судя по букровке **f**, эти свойства являются функциями (например, **copyFile** и **copyFileSync**).

```
VARIABLES
  Local
    __dirname: 'C:\projects\social'
    __filename: 'C:\projects\social\main.js'
    > exports: {}
    fs: {appendFile: f, appendFileSync: f, ...
      > _toUnixTimestamp: f toUnixTimestamp(ti...
      > access: f access(path, mode, callback)...
      > accessSync: f accessSync(path, mode) {...
      > appendFile: f appendFile(path, data, o...
      > appendFileSync: f appendFileSync(path,...
      > chmod: f chmod(path, mode, callback) {...
      > chmodSync: f chmodSync(path, mode) {\r...
      > chown: f chown(path, uid, gid, callbac...
      > chownSync: f chownSync(path, uid, gid)...
      > close: f close(fd, callback = defaultC...
      > closeSync: f closeSync(fd) {\r\n fd =...
      > constants: {UV_FS_SYMLINK_DIR: 1, UV_F...
      > copyFile: f copyFile(src, dest, mode, ...
      > copyFileSync: f copyFileSync(src, dest...
```

```
JS main.js > ...
1  const fs = require('node:fs');
2
3  console.log(fs);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

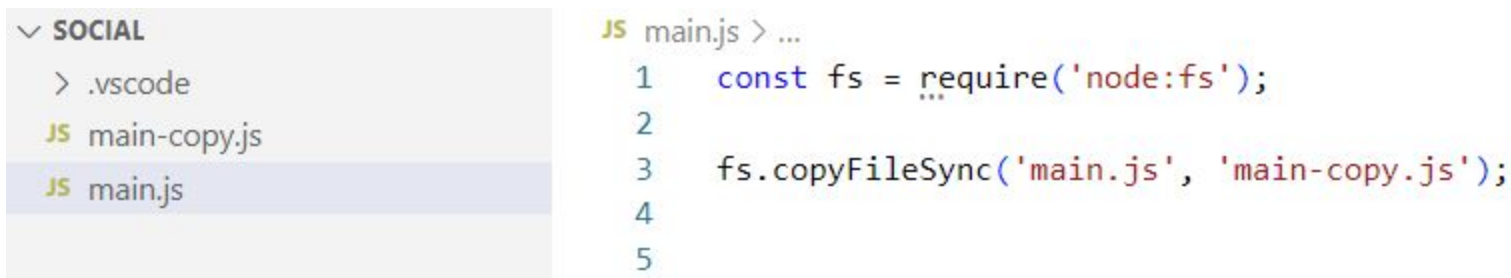
Filter (e.g. text, lexclude)

C:\Program Files\nodejs\node.exe .\main.js



# Функции

Мы с вами будем детально разбираться с модулем `fs` (и другими модулями стандартной библиотеки) в следующей лекции, пока же просто для примера попробуем вызвать функцию, которая сделает копию нашего файла `main.js`:



The image shows a screenshot of the Visual Studio Code interface. On the left, the 'SOCIAL' folder is expanded in the Explorer sidebar, showing files `.vscode`, `main-copy.js`, and `main.js`. The `main.js` file is selected. On the right, the code editor displays the following JavaScript code:

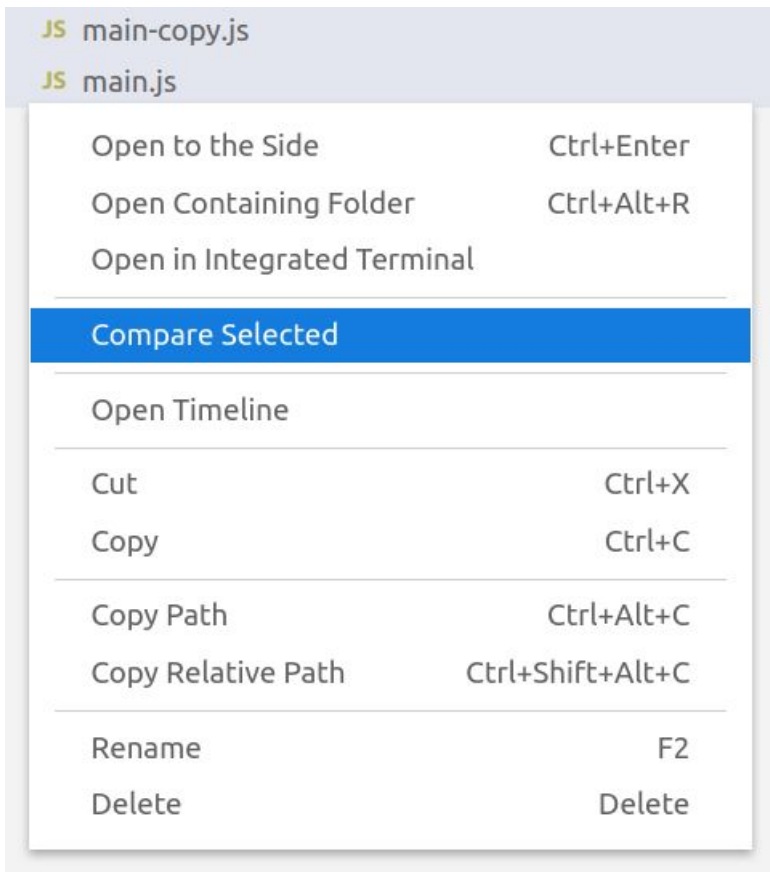
```
JS main.js > ...  
1  const fs = require('node:fs');  
2  
3  fs.copyFileSync('main.js', 'main-copy.js');  
4  
5
```

Обратите внимание: в боковой панели после запуска приложения появится файл `main-copy.js`, содержимое которого точно такое же, как у `main.js`.



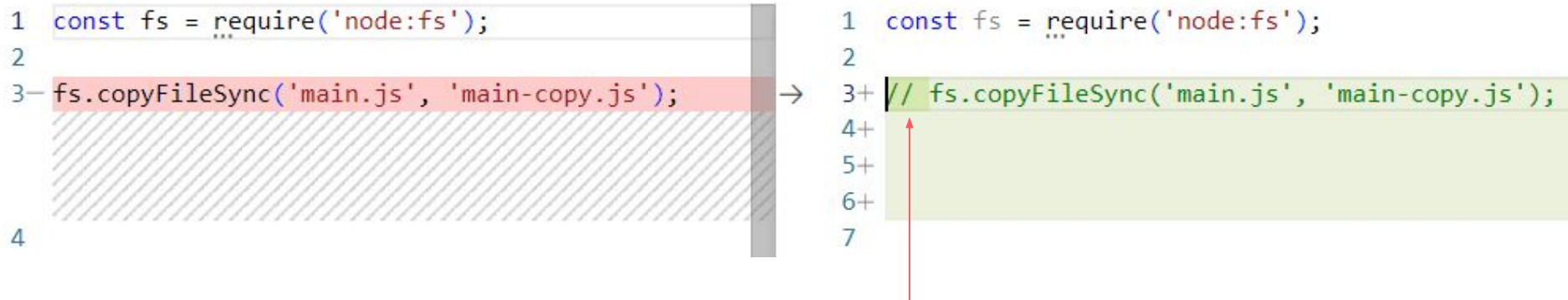
# Функции

Вы можете удостовериться в том, что файлы одинаковые, выбрав файлы (кликайте по их названиям, зажав **Shift**) и опцию **Compare Selected**:



# Функции

Если в файлах будут различия, они подсвелятся вам:



```
1 const fs = require('node:fs');
2
3 fs.copyFileSync('main.js', 'main-copy.js');
4
→
1 const fs = require('node:fs');
2
3+ // fs.copyFileSync('main.js', 'main-copy.js');
4+
5+
6+
7
```

специально добавили комментарий,  
чтобы увидеть различия



# Модули

Таким образом, всё дальнейшее изучение Node.js будет сводиться к трём пунктам:

1. Изучение модулей (какие функции и типы предоставляют, как использовать)
2. Изучение особенностей Node.js
3. Практика (написание реальных приложений и разворачивание их для работы)

Этим мы и займёмся на следующей лекции.



# ИТОГИ





# Итоги

В этой лекции мы обсудили достаточно много важных моментов:

1. Поговорили о переменных, функциях и объектах
2. Поговорили о Node.js и системе модулей, которые и предоставляют необходимую нам функциональность



# ДОМАШНЕЕ ЗАДАНИЕ



# ДЗ 1: os (platform)

Создайте проект аналогично тому, как мы это делали на лекции (включая создание конфигурации запуска). Проект должен располагаться в каталоге `platform` (именно его содержимое и нужно заархивировать).

В Node.js есть [модуль OS](#), который позволяет вам получать информацию об операционной системе, на которой запущено ваше приложение.

Что нужно сделать:

1. Подключить модуль `os` (так же, как мы делали с модулем `fs`)
2. Вызвать функцию `platform` (эта функция не требует входных параметров, поэтому вызываете с пустыми скобками)
3. Результат вызова функции сохраняете в переменную с именем `platform`
4. Выводите значение этой переменной через `console.log`



# ДЗ №2: os (username)

Создайте проект аналогично тому, как мы это делали на лекции (включая создание конфигурации запуска). Проект должен располагаться в каталоге **username** (именно его содержимое и нужно заархивировать). В Node.js есть [модуль OS](#), который позволяет вам получать информацию об операционной системе, на которой запущено ваше приложение.

Что нужно сделать:

1. Подключить модуль **os** (так же, как мы делали с модулем **fs**)
2. Вызвать функцию **userInfo** (эта функция не требует входных параметров, поэтому вызываете с пустыми скобками)
3. Результат вызова функции сохраняете в переменную с именем **userInfo**, функция вернёт вам объект
4. Вам нужно изучить объект, который возвращается в п.3 и вывести на экран поле из этого объекта, в котором содержится имя пользователя (через **console.log**)



Спасибо за внимание

**alif skills**

2023г.

