

# РАБОТА С ФАЙЛАМИ



**АЛЕКСЕЙ КУЛАГИН** / НЕТОЛОГИЯ



# АЛЕКСЕЙ КУЛАГИН



[al.qlogin@gmail.com](mailto:al.qlogin@gmail.com)



[al\\_qlogin](#)



[fb.me/alex.kulagin](https://fb.me/alex.kulagin)



# ПЛАН ЗАНЯТИЯ

1. [Создание модулей](#)
2. [Чтение файлов](#)
3. [Асинхронные операции](#)
4. [Техники борьбы с колбэк-адом](#)



# СОЗДАНИЕ МОДУЛЕЙ

# СОЗДАДИМ ПРОСТОЙ МОДУЛЬ

В файле `random.js` реализуем функцию генерации случайного числа:

```
1  const random = (min, max) => {  
2    min = Math.ceil(min);  
3    max = Math.floor(max);  
4    max = Math.floor(Math.random() * (max - min + 1));  
5    return max + min;  
6  };
```

## ЭКСПОРТИРУЕМ НАШУ ФУНКЦИЮ

Для этого нужно присвоить её в `module.exports`

```
1  const random = (min, max) => {  
2    min = Math.ceil(min);  
3    max = Math.floor(max);  
4    max = Math.floor(Math.random() * (max - min + 1));  
5    return max + min;  
6  };  
7  
8  module.exports = random;
```

`module` — специальная переменная, доступная в любом модуле, которая содержит информацию о модуле: имя файла, дочерние модули, родительский модуль.

## В НЕКОТОРЫХ СЛУЧАЯХ ПРОМЕЖУТОЧНЫЕ ПЕРЕМЕННЫЕ НЕ НУЖНЫ

Можно присвоить сразу в `module.exports`

```
1 module.exports = (min, max) => {  
2   min = Math.ceil(min);  
3   max = Math.floor(max);  
4   max = Math.floor(Math.random() * (max - min + 1));  
5   return max + min;  
6 };
```

В свойстве `exports` объект, который модуль экспортирует, и который будет доступен при подключении.

# ИСПОЛЬЗОВАНИЕ МОДУЛЯ

Создадим файл `index.js`, который будет использовать нашу функцию генерации случайных чисел:

```
1 let numbers = [];  
2 for (let i = 0; i < 5; ++i) {  
3   let number = random(1, 100);  
4   numbers.push(number);  
5 }  
6 console.log(numbers);
```

Сейчас этот код выдает ошибку: `ReferenceError: random is not defined`



## ПОДКЛЮЧИМ МОДУЛЬ И ОПРЕДЕЛИМ `random`

Для подключения модуля используется функция `require`. Она принимает путь к файлу модуля и возвращает объект `module.exports` этого модуля.

```
1  const random = require('./random');
2  let numbers = [];
3  for (let i = 0; i < 5; ++i) {
4      let number = random(1, 100);
5      numbers.push(number);
6  }
7  console.log(numbers); // [ 33, 7, 23, 11, 95 ]
```

Укажем относительный путь к модулю в `require`. Расширения `.js`, `.json` и `.node` можно опустить

# ПОДКЛЮЧЕННЫЙ МОДУЛЬ МОЖНО ИСПОЛЬЗОВАТЬ НА ЛЕТУ

Так как `require` возвращает ровно то, что присвоено в `module.exports`

```
1 | let number = require('./random')(100, 200);  
2 |  
3 | console.log(number); // 178
```

# ЭКСПОРТ НЕСКОЛЬКИХ СУЩНОСТЕЙ

Допустим, мы хотим создать функции фильтров для массивов в `filters.js`

```
1  const odd = number => number % 2;  
2  const even = number => !(number % 2);  
3  
4  module.exports = {  
5      odd,  
6      even  
7  };
```

# ИСПОЛЬЗУЕМ НАШИ ФИЛЬТРЫ

Можно подключить весь модуль

```
1  const filters = require('./filters');  
2  const odd = [1, 2, 3, 4, 5, 6].filter(filters.odd);  
3  console.log(odd); // [ 1, 3, 5 ]
```

А можно присвоить только нужные свойства

```
1  const oddFilter = require('./filters').odd;  
2  const odd = [1, 2, 3, 4, 5, 6].filter(oddFilter);  
3  console.log(odd); // [ 1, 3, 5 ]
```

Или так

```
1  const { even } = require('./filters');  
2  const evenNumbers = [1, 2, 3, 4, 5, 6].filter(even);  
3  console.log(evenNumbers); // [ 2, 4, 6 ]
```

# ИЗБАВИМСЯ ОТ ЛИШНИХ ПЕРЕМЕННЫХ В НАШЕМ МОДУЛЕ

Сразу добавить все нужные свойства в `module.exports`

```
1 module.exports.odd = number => number % 2;  
2 module.exports.even = number => !(number % 2);
```

Чтобы сократить повторения, можно использовать `exports` — ссылка на `module.exports`


```
1 exports.odd = number => number % 2;  
2 exports.even = number => !(number % 2);
```

# ПРИСВОЕНИЕ В `exports` НЕ ДАСТ НУЖНОГО РЕЗУЛЬТАТА

Так как это просто ссылка на `module.exports`

```
1 | const random = require('./random');  
2 | exports = random(1, 100);
```

Файл `main.js`

```
1 | const num = require('./lib/number');  
2 |  
3 | console.log(num); // 
```

# СВЯЗЬ `module.exports` И `exports`

Хорошо показана в этом псевдокоде:

```
1  const require = path => {  
2    const module = { exports: {} };  
3  
4    ((module, exports) => {  
5      // Код вашего модуля, расположенного в path  
6    })(module, module.exports);  
7    return module.exports;  
8  };
```

## ПЕРЕМЕННЫЕ, ОБЪЯВЛЕННЫЕ В МОДУЛЕ, НЕ ДОСТУПНЫ ПРИ ПОДКЛЮЧЕНИИ

Создадим файл `private.js`:

```
1 let number = 0;  
2 exports.set = value => number = value;  
3 exports.get = () => number;
```

Не экспортированные переменные становятся «приватными»:

```
1 const value = require('./private');  
2 value.set(99);  
3 console.log(number); // undefined  
4 console.log(value.get()); // 99
```



# ПОДКЛЮЧАЕМ МОДУЛЬ, КОТОРЫЙ НИЧЕГО НЕ ЭКСПОРТИРУЕТ

Модуль `info.js`:

```
1 | console.log('Информация о приложении');
```

Наше приложение `index.js`:

```
1 | if (!process.argv[3]) {  
2 |     require('./info');  
3 | }
```

Ошибок не выдается. Сама функция `require` вернет пустой объект `{}`. Но код модуля выполнится.

`process` — еще один глобальный объект, содержит информацию о текущем запущенном процессе. `process.argv` — аргументы этого процесса.

# ПОДКЛЮЧЕНИЕ JSON-ФАЙЛОВ

Создадим файл `number-list.json`:

```
1 | [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

И используем эти данные в нашем коде с фильтрами:

```
1 | const { even } = require('./filters');  
2 | const numbers = require('./number-list');  
3 | const evenNumbers = numbers.filter(even);  
4 | console.log(evenNumbers); // [ 0, 2, 4, 6, 8 ]
```

При подключении `.json` файла создается JavaScript-объект, содержащий данные из файла. Преобразовывать ничего не нужно.

## ИТОГО ПРО `require`

- Для подключения собственного модуля `require` и путь к файлу модуля.
- Расширения файла `.js`, `.json` и `.node` можно опускать.
- Подключенный модуль можно использовать на лету или получить из него только нужные свойства и методы.
- Можно подключать модули, которые ничего не экспортируют. Получим пустой объект. Но код модуля выполнится. Иногда это все, что нам нужно.

## ИТОГО ПРО `module.exports`

- Для экспорта присвойте нужную сущность в `module.exports`.
- Для задания свойств можно использовать ссылку `exports`.
- Подключенные модули кешируются.
- Переменные, объявленные внутри модуля, недоступны за его пределами. Доступно только то, что присвоено в `module.exports`.



# СИСТЕМНЫЕ МОДУЛИ

## ДЛЯ ПОДКЛЮЧЕНИЯ ТОЖЕ ИСПОЛЬЗУЕМ `require`

Главное отличие: в `require` мы передаем не путь, а название системного модуля.

```
1  const fs = require('fs');  
2  
3  content = fs.readFileSync('data.txt');  
4  console.log(content);
```

Например, весь функционал взаимодействия с файловой системой реализован в модуле `fs`, и чтобы его использовать, нам нужно его подключить.

# МЫ МОЖЕМ ПРИСВОИТЬ МОДУЛЬ В ЛЮБУЮ ПЕРЕМЕННУЮ

Функция `require` по сути возвращает то, что является модулем. Это не обязательно объект. И мы присваиваем результат в переменную, или можем его тут же использовать:

```
1  const fileSystem = require('fs');  
2  
3  content = fileSystem.readFileSync('data.txt');  
4  console.log(content);
```



# АСИНХРОННЫЕ ФУНКЦИИ





## АСИНХРОННЫЕ ФУНКЦИИ – ОСНОВА NODE.JS

- Такая функция выполняется сразу.
- Но для получения результата требуется какое-то время.
- Поэтому после выполнения функции результат еще не доступен.
- Поэтому в такую функцию мы передаем нашу функцию, которая обычно называется функцией обратного вызова, или колбэком.
- Когда результат будет готов, система вызовет наш колбэк, и передаст в него результат.

## ПРИМЕР АСИНХРОННОЙ ФУНКЦИИ – `setTimeout`

```
1  setTimeout(() => {  
2    console.log('Прошло 5 секунд');  
3  }, 5000);  
4  console.log('Система не ожидает 5 секунд');  
5  console.log('Она продолжает выполнять код программы дальше');  
6  console.log('Когда 5 секунд пройдет, она вызовет нашу функцию');
```

Результат выполнения:

```
$ node timeout
```

```
Система не ожидает 5 секунд
```

```
Она продолжает выполнять код программы дальше
```

```
Когда 5 секунд пройдет, она вызовет нашу функцию
```

```
Прошло 5 секунд
```

# ИМИТИРУЕМ ДЛИТЕЛЬНОЕ ПОЛУЧЕНИЕ ДАННЫХ С ПОМОЩЬЮ `setTimeout`

```
1 function getDataAsync(callback) {  
2   setTimeout(() => {  
3     let numbers = require('number-list');  
4     callback(null, numbers);  
5   }, 1500);  
6 }  
7 const { even } = require('./filters');  
8 getDataAsync((error, numbers) => {  
9   if (error) throw error;  
10  const evenNumbers = numbers.filter(even);  
11  console.log(evenNumbers); // [ 0, 2, 4, 6, 8 ]  
12 });
```

## РАЗБЕРЕМ ПОЛУЧИВШИЙСЯ ПРИМЕР

- `getDataAsync` не возвращает данные, код подвис на полторы секунды.
- Когда данные «готовы», вызывается переданная функция `callback`.
- Стандартный паттерн организации асинхронного кода с функциями обратного вызова: первым аргументом передается ошибка, а остальные — уже данные. Если ошибок нет, передаётся `null` или `false`.
- Мы не можем обработать числа сразу. Нам нужно дождаться их получения.
- Поэтому весь наш код в колбэке.



# ПАТТЕРН «ОШИБКА + РЕЗУЛЬТАТ»

## ИСХОД ДЛИТЕЛЬНОЙ ОПЕРАЦИИ – ОШИБКА

Любой длительный процесс может завершиться либо с ошибкой, либо успешно. В случае, когда ожидание завершилось ошибкой, нам нужно передать информацию об ошибке:

```
1 function getDataAsync(callback) {  
2     setTimeout(() => {  
3         callback({  
4             code: 500,  
5             message: 'Сервер не доступен'  
6         });  
7     }, 1500);  
8 }
```

## ИСХОД ДЛИТЕЛЬНОЙ ОПЕРАЦИИ – УСПЕХ

В случае, когда ожидание завершилось успешно, нам нужно передать данные:

```
1 function getDataAsync(callback) {  
2   setTimeout(() => {  
3     let numbers = require('number-list');  
4     callback(null, numbers);  
5   }, 1500);  
6 }
```

## АРГУМЕНТЫ ФУНКЦИИ ОБРАТНОГО ВЫЗОВА

Для того, чтобы передача информации о состоянии и результат длительной операции были однообразны, используют такой подход:

- Первым аргументом передают информацию об ошибке либо `null`, если ошибки нет.
- Вторым и последующими аргументами передают результат. Который имеет смысл, если ошибки не случилось.



# АРГУМЕНТЫ ФУНКЦИИ ОБРАТНОГО ВЫЗОВА

При таком подходе легко реализовать код функции обратного вызова, просто понимая, что мы ждем массив чисел:

```
1  getDataAsync((error, numbers) => {  
2    if (error) throw error;  
3    const evenNumbers = numbers.filter(even);  
4    console.log(evenNumbers); // [ 0, 2, 4, 6, 8 ]  
5  });
```



# РАБОТА С ФАЙЛАМИ

## МЕТОД `readFile` АСИНХРОННЫЙ

Это означает, что система не будет ожидать завершения операции и продолжит работу.

```
1  const fs = require('fs');
2  const conf = { encoding: 'utf8' };
3  console.log('Читаем из файла ...');
4  fs.readFile('./hello.txt', conf, (err, content) => {
5      if(err) return console.error(err);
6      console.log('Содержимое файла:');
7      console.log(content);
8  });
9  console.log('Файл прочитан и содержимое выведено!');
```

## РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРИМЕРА КОДА

```
> node readfile.js
```

Читаем из файла ...

**Файл прочитан и содержимое выведено!**

Содержимое файла:

Привет, Мир!

Как видите, сообщение о том, что файл прочитан, вывелось до того, как вывелось содержимое файла. По сути, асинхронные вызовы работают как `setTimeout`, только колбэк вызывается не через определенное время, а после завершения операции.

## МЕТОД `readFile` АСИНХРОННЫЙ

Поэтому, если мы хотим выполнить какое-то действие после выполнения асинхронной функции, то это действие нужно делать в функции обратного вызова.

```
1  const fs = require('fs');
2  const conf = { encoding: 'utf8' };
3  console.log('Читаем из файла ...');
4  fs.readFile('./hello.txt', conf, (err, content) => {
5      if(err) return console.error(err);
6      console.log('Содержимое файла: ');
7      console.log(content);
8      console.log('Файл прочитан и содержимое выведено!');
9  });
```

## ПОЛУЧАЕМ СПИСОК ФАЙЛОВ В ПАПКЕ

```
1  const fs = require('fs');
2
3  fs.readdir('./', (err, files) => {
4    if(err) return console.error(err);
5    files.forEach(file => console.log(file));
6  });
```

## ПОЛУЧАЕМ ИНФОРМАЦИЮ ОБ ОБЪЕКТЕ

```
1  const fs = require('fs');
2
3  fs.stat('./hello', (err, stats) => {
4    if(err) return console.error(err);
5    if (stats.isFile()) {
6      console.log('./hello.txt is file');
7    }
8    if (stats.isDirectory()) {
9      console.log('./hello.txt is directory');
10   }
11   ....
12 });
```

Здесь `stats` – объект типа `fs.Stats`

## ПРОВЕРИМ ПРАВА НА ЗАПИСЬ В ФАЙЛЕ

```
1  const fs = require('fs'), path = 'hello.txt';
2
3  fs.access(path, fs.constants.W_OK, err => {
4    if (err) {
5      return console.log('нет прав на запись %s', path);
6    }
7    console.log('есть права на запись %s', path);
8  });
```

1. `fs.constants.F_OK` — есть доступ на чтение/запись/исполнение
2. `fs.constants.R_OK` — есть доступ на чтение
3. `fs.constants.W_OK` — есть доступ на запись
4. `fs.constants.X_OK` — есть доступ на исполнение



## ЗАПИСЫВАЕМ ДАННЫЕ В ФАЙЛ

```
1  const fs = require('fs');
2  const text = 'Привет, Нетология!';
3  fs.writeFile('./hello.txt', text, err => {
4    if (err) throw err;
5    console.log('Файл сохранен');
6  });
```

## СОЗДАЕМ ПАПКУ

```
1  const fs = require('fs')
2  const name = './neto';
3  fs.mkdir(name, err => {
4    if (err) throw err;
5    console.log('Папка создана');
6  });
```


## ПЕРЕИМЕНУЕМ ФАЙЛ

```
1  const fs = require('fs');
2  fs.rename('./hello.txt', './hw.txt', err => {
3    if (err) throw err;
4    console.log('Файл переименован');
5  });
```

Аналогично можно переименовать или переместить файл или папку.

## ДОПИШЕМ ДАННЫЕ В ФАЙЛ

```
1  const fs = require('fs');
2  const conf = { encoding: 'utf8' };
3  const text = 'Новые данные';
4  fs.appendFile('./hw.txt', text, conf, err => {
5    if (err) throw err;
6    console.log('Файл дополнен');
7  })
```



# ПОСЛЕДОВАТЕЛЬНОСТЬ АСИНХРОННЫХ ВЫЗОВОВ

# ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ НЕЗАВИСИМЫХ ОПЕРАЦИЙ

Нет никакой гарантии, что файлы будут созданы в том порядке, как был вызван метод `writeFile`

```
1  const fs = require('fs');
2  const names = ['Иван', 'Олег', 'Екатерина'];
3  names.forEach((name, i) => {
4      fs.writeFile(`./name${i}.txt`, name, err => {
5          if (err) throw err;
6          console.log(`Файл name${i}.txt сохранен`);
7      });
8  });
```

# ПОСЛЕДОВАТЕЛЬНОЕ ВЫПОЛНЕНИЕ ЗАВИСИМЫХ ОПЕРАЦИЙ

Что, если нам сначала нужно создать папку, а потом создать файл в ней?

```
1  const fs = require('fs'), opt = { encoding: 'utf8' };
2  fs.mkdir('./data', err => {
3    if (err) throw err;
4    console.log('Папка создана!');
5  });
6  fs.writeFile('./data/my.txt', 'Привет!', opt, err => {
7    if (err) throw err;
8    console.log('Файл создан!');
9  });
```


Если мы просто последовательно вызовем `mkdir` и `writeFile`, нет никаких гарантий, что файл создастся после того, как будет создана папка. Он может быть создан и раньше. И это приведет к ошибке.

## ВЫПОЛНЯТЬ ЗАВИСИМОЕ ДЕЙСТВИЕ НУЖНО В КОЛБЕКЕ

```
1  const fs = require('fs'), opt = { encoding: 'utf8' };
2  fs.mkdir('./data', err => {
3    if (err) throw err;
4    console.log('Папка создана!');
5    fs.writeFile('./data/my.txt', 'Привет!', opt, err => {
6      if (err) throw err;
7      console.log('Файл создан!');
8    });
9  });
```

А это приводит к огромной вложенности функций обратного вызова. И абсолютно нечитаемому коду.





# ТЕХНИКИ БОРЬБЫ С КОЛБЭК-АДОМ

## ИМЕНОВАННЫЕ ФУНКЦИИ

```
1  const fs = require('fs'), opts = { encoding: 'utf8' };
2  fs.mkdir('./data', handleDirReady);
3  function handleDirReady(err) {
4      if (err) throw err;
5      console.log('Папка создана!');
6      const filename = './data/test.txt';
7      fs.writeFile(filename, 'Привет!', opts, handleSave);
8  }
9  function handleSave(err) {
10     if (err) throw err;
11     console.log('Файл создан!');
12 }
```

## ИЗМЕНИТЬ УРОВЕНЬ АБСТРАКЦИИ

```
1  const fs = require('fs'), opts = { encoding: 'utf8' };
2  function saveTo(dirname, filename, data){
3      fs.mkdir(dirname, handleDirReady);
4      function handleDirReady(err) {
5          if (err) throw err;
6          console.log('Папка создана!');
7          const filepath = `${dirname}/${filename}`;
8          fs.writeFile(filepath, data, opts, handleSave);
9      }
10     function handleSave(err) {
11         if (err) throw err;
12         console.log(`Файл ${filename} создан!`);
13     }
14 }
15 module.exports = {saveTo};
```

---

# PROMISE



***Promise** – объект использующийся для асинхронных операций.*

*Он представляет значение, которое может быть доступно сразу, или в будущем или вообще никогда*

# PROMISE КАК ОБЕЩАНИЕ ЧТО-ЛИБО СДЕЛАТЬ

Когда папка будет реально создана Promise разрешится значением  
`dirname`

```
1 function mkdir(dirname){
2   return new Promise((done, fail)=>{
3     fs.mkdir(dirname, err => {
4       if (err) {
5         fail(err);
6       } else {
7         done(dirname);
8       }
9     })
10  })
11 }
```

# PROMISE КАК ОБЕЩАНИЕ ЧТО-ЛИБО СДЕЛАТЬ

Когда данные в файл будут сохранены Promise разрешится значением

Данные сохранены в файл #имя файла#

```
1 function writeFile(filepath, data){
2   return new Promise((done, fail) => {
3     fs.writeFile(filepath, data, opts, err => {
4       if (err) {
5         fail(err);
6       } else {
7         done(`Данные сохранены в файл ${filepath}`);
8       }
9     })
10  })
11 }
```

# ПОСЛЕДОВАТЕЛЬНОЕ ВЫПОЛНЕНИЕ ЗАВИСИМЫХ ОПЕРАЦИЙ С PROMISE

Промисы можно связывать создавая цепочки.

```
1 mkdir( './data' )  
2   .then(path => mkdir( `${path}/files` ))  
3   .then(path => writeFile( `${path}/my.txt`, 'Привет!' ))  
4   .then(result => console.log(result))  
5   .catch(error => console.error(error))
```

# ПОСЛЕДОВАТЕЛЬНОЕ ВЫПОЛНЕНИЕ ЗАВИСИМЫХ ОПЕРАЦИЙ С PROMISE

Данный пример можно прочитать следующим образом:

- создать `./data`
- потом (если успешно) создать `./data/files`
- потом (если успешно) записать Привет! в `./data/files/my.txt`
- потом (если успешно) вывести в консоль результат работы `writeFile`, а в случае ошибки вывести ее в консоль



# ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ ОПЕРАЦИЙ С PROMISE

Для связывания параллельных операций в один Promise используется `Promise.all([массив промисов])`, который разрешается массивом результатов каждого промиса

```
1 mkdir( './data' )
2   .then(path => mkdir( `${path}/files` ))
3   .then(path => Promise.all([
4       writeFile( `${path}/file1.txt`, 'Привет птичка!' ),
5       writeFile( `${path}/file2.txt`, 'Привет рыбка!' )
6   ]))
7   .then(result => console.log(result))
8   .catch(error => console.error(error))
```



**НЕТОЛОГИЯ**  
центр онлайн-образования

**Задавайте вопросы и напишите отзыв о лекции!**

**АЛЕКСЕЙ КУЛАГИН**



[al.kulagin@gmail.com](mailto:al.kulagin@gmail.com)



[al\\_kulagin](https://t.me/al_kulagin)



[fb.me/alex.kulagin](https://fb.me/alex.kulagin)