

# Потоки

**Поток (stream)** – это концепция, которая была сначала реализована в UNIX системах для передачи данных из одной программы в другую в операциях ввода/вывода. Это позволяет каждой программе быть очень специализированной в том, что она делает – быть независимым модулем

Потоки позволяют **обмениваться** данными **небольшими частями** (чанки, chunk), что в свою очередь дает возможность в своей работе не расходовать много памяти

Распространенная задача – **парсинг файла большого объема**. Например, в текстовом файле с данными логов нужно найти строку, содержащую определенный текст. Вместо того, чтобы файл полностью загрузить в память, и потом начать разбирать в нем строки в поисках нужной, можно его считывать небольшими порциями. Тем самым не занимаем память сверх необходимого, а лишь столько памяти, сколько нужно для буферизации считанных данных. Как только найдется требуемую запись, сразу прекратить дальнейшую работу

# Потоки

Для работы с потоками в Node.js используется модуль **stream**

Потоки делятся на четыре типа:

1. **Readable** – для чтения данных
2. **Writable** – для записи данных
3. **Duplex** – комбинация двух предыдущих типов, при этом процесс чтения и записи происходит независимо друг от друга
4. **Transform** – разновидность Duplex потоков, которые могут изменять данные при их записи и чтении в/из потока (чаще используется как промежуточное звено в цепочке передачи данных)

**Все потоки** являются наследниками **EventEmitter**, т.е. во всех потоках используется событийная система

## Потоки. Чтение

---

Пример наследника типа `Readable`, который представляет потоковое чтение данных из массива

Для реализации читающего потока нужно реализовать метод `_read()`, который определяет какие данные нужно поместить в очередь для чтения

Для добавления данных в очередь используется метод родителя `push()`

Когда **все данные будут прочитаны**, в метод `push()` передается параметр `null`

# Потоки. Чтение

Создадим файл **StreamArray.js**:

```
const stream = require("stream");

class StreamArray extends stream.Readable {
  constructor(array) {
    super({objectMode: true});
    this._array = array;
  }
  _read() {
    this._array.forEach((value) => {
      this.push(value);
    });
    this.push(null);
  }
}

module.exports = StreamArray;
```

# Потоки. Чтение

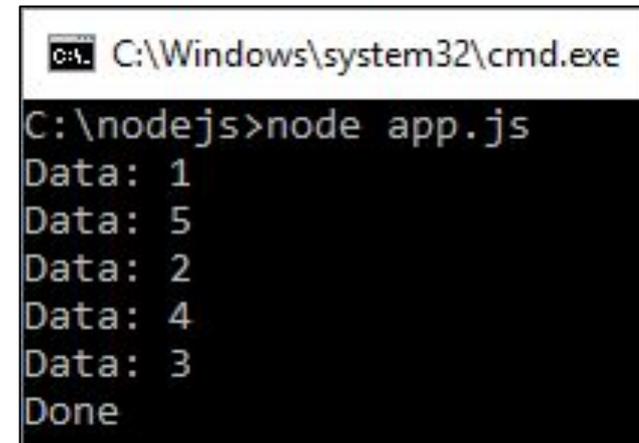
Для работы с потоками типа `Readable` **нужно добавить обработчики** на его **события**, основные события:

- **data** – генерируется каждый раз, когда поток возвращает часть считанных данных
- **end** – генерируется когда все данные были прочитаны

Создадим файл приложения **app.js**:

```
const StreamArray = require("./StreamArray");

let sa = new StreamArray([1, 5, 2, 4, 3]);
sa.on("data", (chunk) => {
  console.log(`Data: ${chunk}`);
});
sa.on("end", () => {
  console.log("Done");
});
```



```
C:\Windows\system32\cmd.exe
C:\nodejs>node app.js
Data: 1
Data: 5
Data: 2
Data: 4
Data: 3
Done
```

Запуск **app.js**:

# Потоки. Запись

Пример наследника типа `Writable`, который выводит данные в консоль

Для реализации записывающего потока нужно реализовать метод `_write()`, параметры:

- **chunk** – порция данных для записи
- **encoding** – кодировка
- **callback** – функция обратного вызова, которую необходимо вызывать в конце обработки порции данных (знак того, что текущая порция данных обработана)

# Потоки. Запись

Создадим файл **ConsoleWriter.js**:

```
const stream = require("stream");

class ConsoleWriter extends stream.Writable {
  constructor() {
    super({objectMode: true});
  }
  _write(chunk, encoding, callback) {
    console.log(`Data: ${chunk}`);
    callback();
  }
}

module.exports = ConsoleWriter;
```

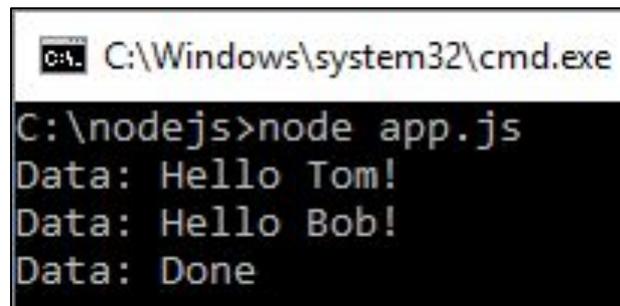
# Потоки. Запись

Для работы с потоками типа Writable используется метод **write()** для записи данных и метод **end()** для окончания записи

Файл **app.js**:

```
const ConsoleWriter = require("../ConsoleWriter");  
  
let cw = new ConsoleWriter();  
cw.write("Hello Tom!");  
cw.write("Hello Bob!");  
cw.end("Done");
```

Запуск **app.js**:



```
C:\Windows\system32\cmd.exe  
C:\nodejs>node app.js  
Data: Hello Tom!  
Data: Hello Bob!  
Data: Done
```

# Потоки. Преобразование

Пример наследника типа Transform, который преобразует данные в разные типы

Для реализации преобразовывающего потока нужно реализовать метод **\_transform()**, параметры:

- **chunk** – порция данных для преобразования
- **encoding** – кодировка
- **callback** – функция обратного вызова, которую необходимо вызывать после преобразования порции данных

Для **возврата преобразованных данных** используется метод родителя **push()**

# Потоки. Преобразование

Создадим файл **TypeTransform.js**:

```
const stream = require("stream");

class TypeTransform extends stream.Transform {
  constructor(type) {
    super({objectMode: true});
    this._type = type;
  }
  _transform(chunk, encoding, callback) {
    let res = this._type.call(null, chunk);
    this.push(res);
    callback();
  }
}

module.exports = TypeTransform;
```

# Потоки. Преобразование

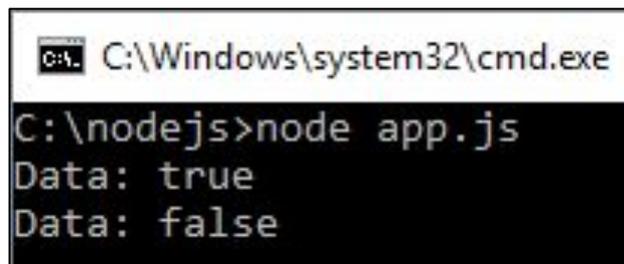
Для работы с потоками типа Transform используется метод **write()** для **передачи** исходных данных и **событие data** для **получения** обработанных данных

Файл **app.js**:

```
const TypeTransform = require("../TypeTransform");

let tt = new TypeTransform(Boolean);
tt.on("data", (chunk) => {
  console.log(`Data: ${chunk}`);
});
tt.write("Hello Tom!");
tt.write("");
```

Запуск **app.js**:

A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.exe". The command prompt shows the following text:

```
C:\nodejs>node app.js
Data: true
Data: false
```

# Потоки. Каналы

**Канал (pipe)** – механизм, который **связывает** поток для чтения и поток для записи и позволяет сразу считать из потока чтения в поток записи

Задача записи в поток данных, считанных из другого потока, является довольно распространенной, и в этом случае **каналы позволяют сократить объем кода**. Для работы с каналами используется метод `pipe()`

Файл **app.js**:

```
const StreamArray = require("./StreamArray");
const ConsoleWriter = require("./ConsoleWriter");
const TypeTransform = require("./TypeTransform");

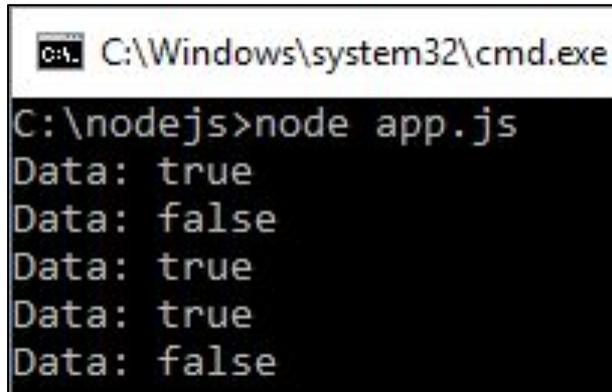
let sa = new StreamArray([1, 0, 1, 1, 0]);
let cw = new ConsoleWriter();
let tt = new TypeTransform(Boolean);

sa.pipe(tt).pipe(cw);
```

# Потоки. Каналы

```
/* СВЯЗЫВАНИЕ ПОТОКОВ БЕЗ КАНАЛОВ
sa.on("data", (chunk) => {
    tt.write(chunk);
});
tt.on("data", (chunk) => {
    cw.write(chunk);
});
*/
```

Запуск **app.js**:

A screenshot of a Windows command prompt window. The title bar shows the path C:\Windows\system32\cmd.exe. The command prompt shows the following text:

```
C:\nodejs>node app.js
Data: true
Data: false
Data: true
Data: true
Data: false
```

## Пример использования потоков для работы с файловой системой

Для создания потока для **записи** используется метод **createWriteStream()**, в который передается название файла. Если такого файла нет, то он создается

Для создания потока для **чтения** используется метод **createReadStream()**, в который также передается название файла. В качестве опционального параметра здесь передается кодировка, что позволит сразу при чтении кодировать считанные данные в строку в указанной кодировке

# ПОТОКИ

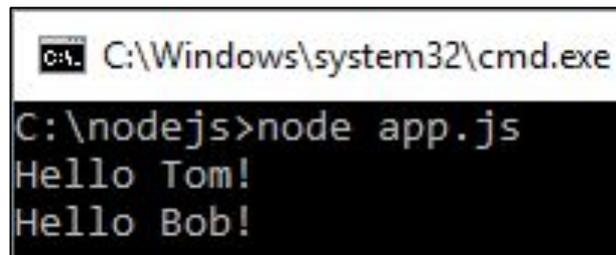
Файл **app.js**:

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

let writableStream = fs.createWriteStream("./hello.txt");
writableStream.write("Hello Tom!");
writableStream.write("Hello Bob!");
writableStream.end();

let readableStream = fs.createReadStream("./hello.txt",
"utf8");
readableStream.on("data", (chunk) => {
  console.log(chunk);
});
```

Запуск **app.js**:

A screenshot of a Windows command prompt window. The title bar shows the path C:\Windows\system32\cmd.exe. The command prompt shows the following text: C:\nodejs>node app.js, followed by two lines of output: Hello Tom! and Hello Bob!.

```
C:\Windows\system32\cmd.exe
C:\nodejs>node app.js
Hello Tom!
Hello Bob!
```

## Пример использования каналов для **создания копии файла**

Файл **app.js**:

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

let readableStream = fs.createReadStream("./hello.txt",
"utf8");
let writableStream =
fs.createWriteStream("./copyHello.txt");

readableStream.pipe(writableStream);
```

Только работой с файлами функциональность потоков не ограничивается, также имеются **сетевые** потоки, потоки **шифрования**, **архивации** и т.д., но общие принципы работы с ними будут те же, что и у файловых потоков