Обработка ошибок, замыкания и асинхронность

Василий Петров
Разработчик Python, JavaScript

Василий Петров

О спикере:

- Стаж работы в IT более 25 лет
- Разрабатывал корпоративные приложения
- Руководил проектами и IT-подразделениями
- Руководил собственным бизнесом
- Участвует в различных проектах с применением Python и JavaScript

Вспоминаем прошлое занятие

Вопрос: как осуществляется поиск методов и свойств объектов в JavaScript, если они отсутствуют в самом объекте?

Вспоминаем прошлое занятие

Вопрос: как осуществляется поиск методов и свойств объектов в JavaScript, если они отсутствуют в самом объекте?

Ответ: методы и свойства объектов ищутся в цепочке прототипов

Вспоминаем прошлое занятие

Вопрос: какой паттерн функционального программирования можно использовать для подсчёта суммы по полям объектов, хранящихся в массиве?

Вспоминаем прошлое занятие

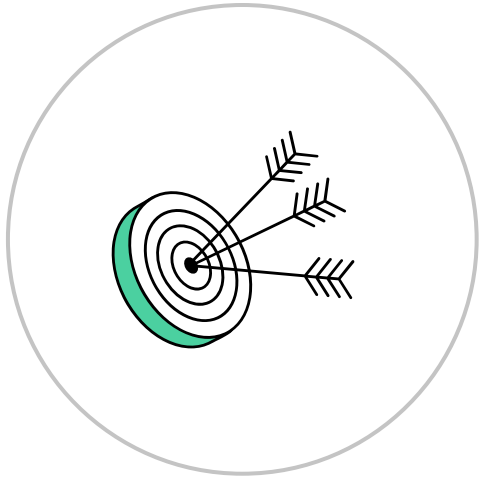
Вопрос: какой паттерн функционального программирования можно использовать для подсчёта суммы по полям объектов, хранящихся в массиве?

Ответ: функция reduce():

```
function totalSum (goods) {
  return goods.reduce((good, acc) =>
    acc+good.price*good.amount)
}
```

Цели занятия

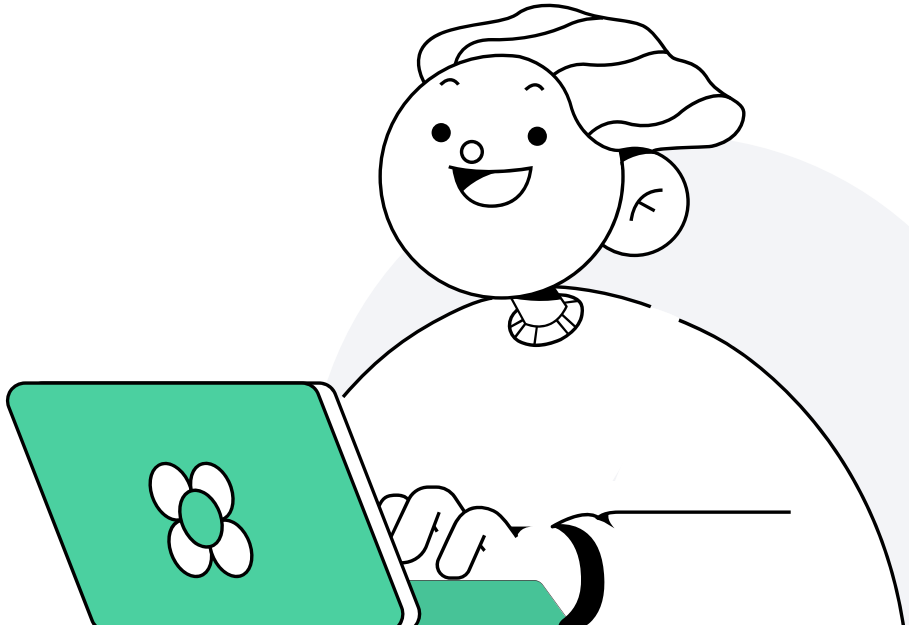
- Узнаем про обработку ошибок времени выполнения
- Познакомимся с понятием замыкания
- Познакомимся с механизмами асинхронности в JavaScript
- Изучим использование колбэков
- Изучим использование промисов
- Изучим современный асинхронный синтаксис с **async/await**



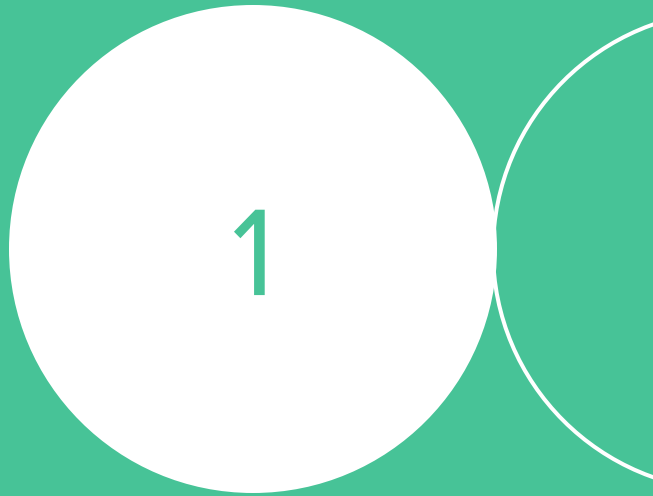
План занятия

1. Обработка ошибок в JavaScript
2. Замыкания и их применение
3. Асинхронное выполнение кода, event loop
4. Колбэк-функции (callback functions)
5. Промисы (Promise)
6. Современный синтаксис async/await
7. Итоги
8. Домашнее задание

*Нажми на нужный раздел для перехода



Обработка ошибок в JavaScript



Ошибки в коде

Виды ошибок:

1. Синтаксические ошибки требуют исправления до запуска кода
2. Ошибки времени выполнения, предусмотренные алгоритмом
3. Ошибки времени выполнения, не предусмотренные разработчиком
4. Логические ошибки разработчика, некорректный алгоритм

Мы будем говорить об обработке ошибок типов 2, 3 и 4 (частично)

Обработка ошибок, предусмотренных алгоритмом

Такие ошибки должны быть заложены в алгоритмы в виде проверок корректности данных и получаемых результатов.

Обработка ошибок не должна приводить к некорректному состоянию программы и данных. Попытка удалить из корзины отсутствующий товар приводит к выводу в журнал соответствующего сообщения и не меняет состояние корзины:

```
function deleteGood(goodId) {
     const idx = basket.findIndex((good) => good.id == goodId)
     if (idx >= 0) {
          basket.splice(Idx, 1);
     } else {
          console.log(`Функция deleteGood: Не найден товар с id = ${goodId}`);
     }
}
```

Обработка ошибок, не предусмотренных алгоритмом

«Мудрые разработчики понимают, что исправление одной ошибки всегда вносит в код несколько новых»

Обработка непредусмотренных ошибок зависит от принятой стратегии и ответственности кода.

Иногда лучшая стратегия — позволить программе завершиться с ошибкой и перезапуститься автоматически или с оповещением разработчика.

Ответственный серверный код должен содержать обработку непредусмотренных ошибок, чтобы обеспечить корректное состояние данных и продолжить обработку параллельных запросов

Механизм обработки ошибок в JavaScript

Похож на конструкцию обработки исключений **try/except/finally** в Python:

```
function div(a, b) {
     try {
          return a/b
     } catch (e) {
          console.log(‘Деление на 0!’);
     } finally {
          console.log(‘Если вы видите это, что-то пошло не так.’);
     }
}
```

В отличие от Python, блок **catch** может быть только один. Фильтровать разные виды исключений необходимо по содержимому переданного объекта-исключения.

Сюрприз: в JavaScript деление на 0 не приводит к исключению, результат равен специальному значению Infinity

Механизм обработки ошибок в JavaScript

Объект исключения содержит информацию об ошибке:

- name — название ошибки, например, ReferenceError
- message — текст сообщения об ошибке
- stack — информация о состоянии кода в момент ошибки для отладки (stack trace)

Name используется для фильтрации ошибок по типам:

```
if (e.name === ReferenceError) { … }
```

Message — для вывода информации об ошибке в человекочитаемом формате.

Stack — для поиска места возникновения ошибки в коде (модуль и номер строки кода), а также последовательности вызовов функций в стеке

Генерация исключений

Исключения можно явно генерировать в коде для обработки через механизм **try/catch**. Для этого используется специальный оператор:

```
throw <объект-исключение>
```

В качестве объекта-исключения можно передать любые данные, даже примитивного типа. Но лучше передавать объект с полями **name** и **message**.

Существуют стандартные классы для объектов-исключений:

```
let error = new Error(message);
// или
let error = new ReferenceError(message);
// …
throw error
```

Можно создавать свои классы, наследуя их от стандартных. Это хорошая практика

Замыкания и их применение

2

Замыкание

Вложенные функции имеют доступ к контексту функций, в которых они описаны:

```
function calc(a, b, op) {
     let result;
     function add() {
          result = a+b;
     }
     switch (op) {
          case ‘+’: add();
                    return result;
     …
     }
}
```

Однако внешние функции не имеют доступа к контексту вложенных

Замыкание

Вложенные функции создаются с контекстом внешней функции, это можно использовать для создания замыкания (closure):

```
function getAdder(a) {
     return function add(b) {
          return a+b;
     }
}

const incr = getAdder(1)
const decr = getAdder(-1)

console.log(incr(1));
console.log(incr(decr(1)));
```

При этом в замыкании сохраняется доступ к параметрам и переменным контекста, в котором оно создано

Демонстрация

Давайте посмотрим, как можно использовать замыкания в JavaScript

Асинхронное выполнение кода, event loop

3

Асинхронное программирование

- В отличие от Python, реализовано в JavaScript, начиная с самой первой версии, в виде встроенных механизмов

Асинхронное программирование

- В отличие от Python, реализовано в JavaScript, начиная с самой первой версии, в виде встроенных механизмов
- В браузере все события, генерируемые пользователем, должны обрабатываться параллельно с другим кодом, чтобы не вызывать зависание (freeze) пользовательского интерфейса страницы

Асинхронное программирование

- В отличие от Python, реализовано в JavaScript, начиная с самой первой версии, в виде встроенных механизмов
- В браузере все события, генерируемые пользователем, должны обрабатываться параллельно с другим кодом, чтобы не вызывать зависание (freeze) пользовательского интерфейса страницы
- В серверном коде асинхронное программирование используется для параллельной обработки запросов без запуска отдельного потока или процесса

Асинхронное программирование

- В отличие от Python, реализовано в JavaScript, начиная с самой первой версии, в виде встроенных механизмов
- В браузере все события, генерируемые пользователем, должны обрабатываться параллельно с другим кодом, чтобы не вызывать зависание (freeze) пользовательского интерфейса страницы
- В серверном коде асинхронное программирование используется для параллельной обработки запросов без запуска отдельного потока или процесса
- Весь код выполняется движком JavaScript в одном потоке, даже если мы обслуживаем 10 тыс. запросов в секунду

Асинхронное программирование

- В отличие от Python, реализовано в JavaScript, начиная с самой первой версии, в виде встроенных механизмов
- В браузере все события, генерируемые пользователем, должны обрабатываться параллельно с другим кодом, чтобы не вызывать зависание (freeze) пользовательского интерфейса страницы
- В серверном коде асинхронное программирование используется для параллельной обработки запросов без запуска отдельного потока или процесса
- Весь код выполняется движком JavaScript в одном потоке, даже если мы обслуживаем 10 тыс. запросов в секунду
- Существуют также возможности параллельного программирования (Workers в браузере, потоки и процессы в Node.js), но основная парадигма — асинхронный код

Цикл событий event loop

Асинхронное выполнение кода реализовано в браузере и Node.js через цикл событий:

- в каждой итерации последовательно обрабатываются различные источники событий, которые накапливаются в очередях

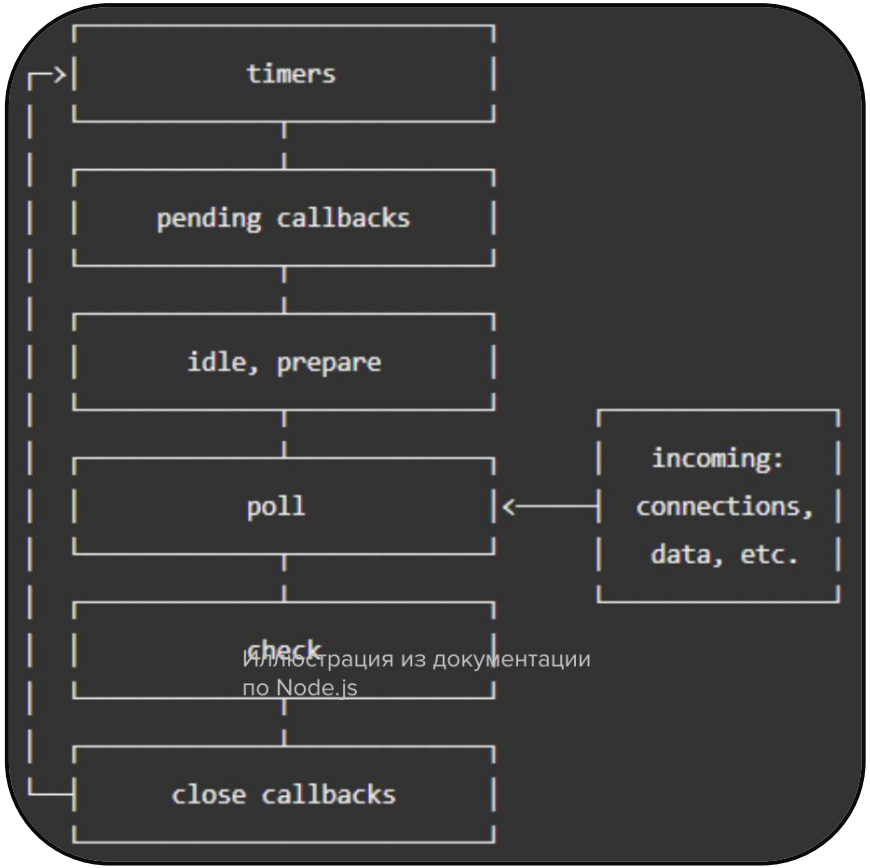


Иллюстрация из документации по Node.js

Иллюстрация из документации по Node.js

Цикл событий event loop

Асинхронное выполнение кода реализовано в браузере и Node.js через цикл событий:

- в каждой итерации последовательно обрабатываются различные источники событий, которые накапливаются в очередях
- каждый обработчик должен выполниться полностью до того, как следующий может быть вызван

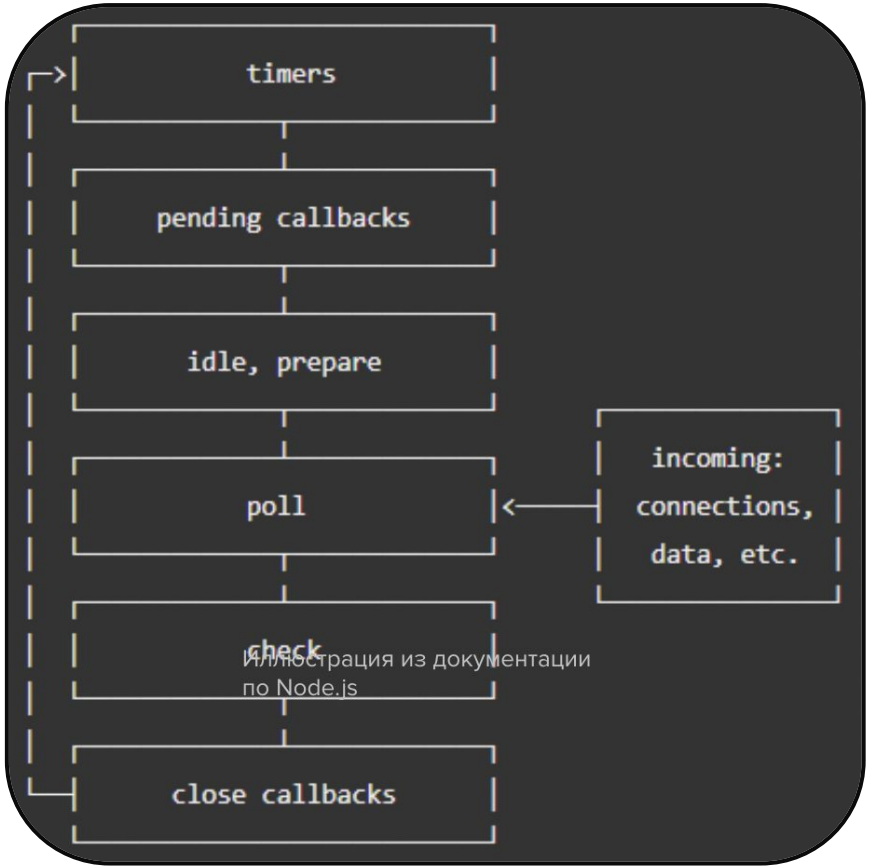


Иллюстрация из документации по Node.js

Иллюстрация из документации по Node.js

Цикл событий event loop

Асинхронное выполнение кода реализовано в браузере и Node.js через цикл событий:

- в каждой итерации последовательно обрабатываются различные источники событий, которые накапливаются в очередях
- каждый обработчик должен выполниться полностью до того, как следующий может быть вызван
- поступающие события попадают в очередь и их обработчики вызываются в соответствующей событию фазе цикла

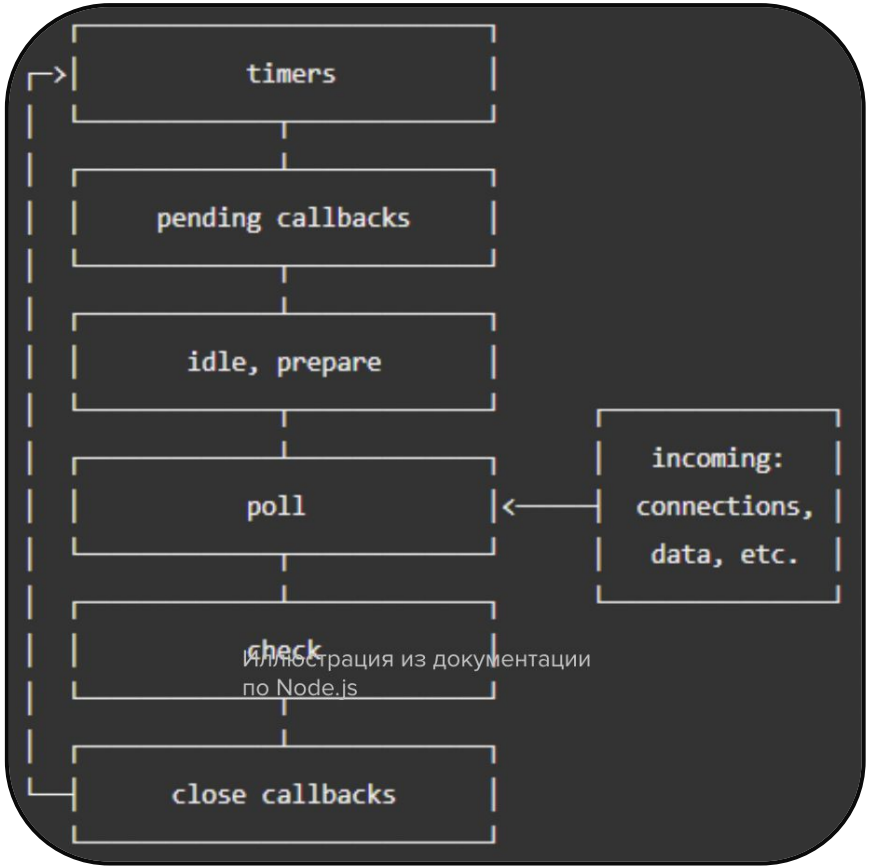


Иллюстрация из документации по Node.js

Иллюстрация из документации по Node.js

Цикл событий event loop

Асинхронное выполнение кода реализовано в браузере и Node.js через цикл событий:

- в каждой итерации последовательно обрабатываются различные источники событий, которые накапливаются в очередях
- каждый обработчик должен выполниться полностью до того, как следующий может быть вызван
- поступающие события попадают в очередь и их обработчики вызываются в соответствующей событию фазе цикла
- браузерный цикл событий отличается в наборе фаз и видов событий, но логика такая же

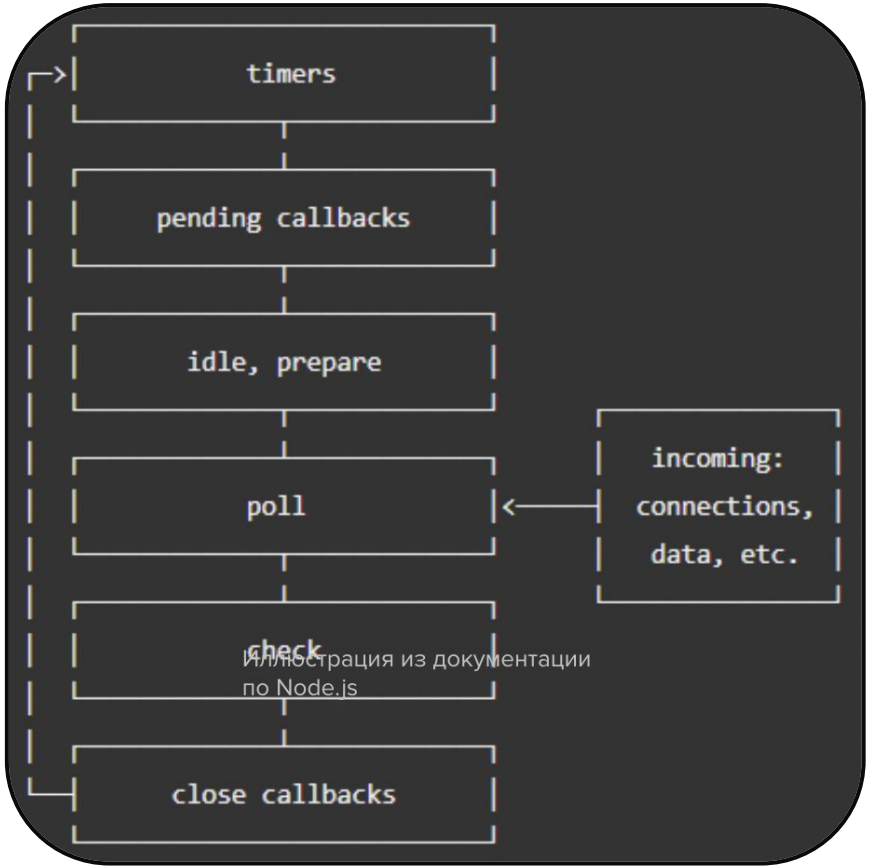


Иллюстрация из документации по Node.js

Иллюстрация из документации по Node.js

Колбэк-функции (callback functions)

Асинхронный код в браузере

Обработчики пользовательских событий (нажатия на кнопки и ввод данных в поля ввода) вызываются через механизм колбэк-функций (callback function):

```
button.onclick = function (event) {
     console.log(‘Вы нажали на кнопку!’);
}
```

Но что произойдёт, если в это время выполняется другой код?

```
while (true) {}
```

Интерфейс страницы зависнет, т. к. параллельное выполнение разного кода невозможно. Браузер будет ждать, пока цикл завершится, и только потом сможет вызвать обработчик события для кнопки

Сложности разработки асинхронного кода

Рассмотрим пример асинхронного чтения файла с последующей обработкой формата JSON:

```
let fs = require('fs');
let data;
fs.readFile('data.json', 'utf-8', (err, dataRead) => {
  data = JSON.parse(dataRead);
  console.log(data);
})
console.log(data);
```

? **Что будет выведено в консоль в последней строке?**

Сложности разработки асинхронного кода

Рассмотрим пример асинхронного чтения файла с последующей обработкой формата JSON:

```
let fs = require('fs');
let data;
fs.readFile('data.json', 'utf-8', (err, dataRead) => {
  data = JSON.parse(dataRead);
  console.log(data);
})
console.log(data);
```

? **Что будет выведено в консоль в последней строке?**
(undefined)

Сложности разработки асинхронного кода

Рассмотрим пример асинхронного чтения файла с последующей асинхронной записью в другой файл:

```
let fs = require(‘fs’);
let data;
fs.readFile(‘data.json’, ‘utf-8’, (err, dataRead) => {
     fs.writeFile(‘newData.json’, dataRead, ‘utf-8’, (err) => {
          if (err) { console.log(‘Ошибка записи файла!’); }
          else { console.log(‘Файл записан.’); }
     })
})
```

Последовательные асинхронные вызовы с колбэками приводят к аду колбэков (callback hell) в коде

Промисы (Promise)

Промисы

Аналогичны future в библиотеке asyncio для Python. Введены в стандарте ES6 (ES2015) для решения проблемы callback hell

```
const readPromise = new Promise((resolve, reject) => {
     fs.readFile(‘data.json’, ‘utf-8’, (err, data) => {
          resolve(data);
     })
})

readPromise.then((data) => {
     console.log(JSON.parse(data);
})
```

Цепочки промисов

Промисы можно обрабатывать по цепочке, чтобы последовательно выполнять асинхронные операции.

```
fs.promises.readFile(‘data.json’, ‘utf-8)
     .then((data) => fa.promises.writeFile(‘newData.json’, ‘utf-8’))
     .then(() => console.log(‘Данные записаны.’));
```

Для этого необходимо из обработчика промиса вернуть новый промис: промисифицированные функции из стандартной библиотеки возвращают промис

Цепочки промисов и обработка ошибок

В цепочку обработки промисов можно встраивать обработчики ошибок и блок **finally**:

```
fs.promises.readFile(‘data.json’, ‘utf-8)
     .catch((err) => console.log(‘Ошибка чтения файла:’, err))
     .then((data) => fa.promises.writeFile(‘newData.json’, ‘utf-8’))
     .catch((err) => console.log(‘Ошибка записи файла:’, err))
     .then(() => console.log(‘Данные записаны.’))
     .finally(() => console.log(‘Все кончилось.’))
```

Блоки catch не обязательно вставлять после каждой операции.
В этом случае ошибка будет попадать в ближайший обработчик catch ниже в цепочке

Промисификация функций

Часто нам доступны библиотечные функции, реализованные на колбэках
Их можно легко промисифицировать:

```
const readFilePromise = function (name, encoding) {
     return new Promise((resolve, reject) => {
          fs.readFile(name, encoding, (err, data) {
               if (err) reject(err)
               resolve(data);
          })
     })
}
```

Но есть способ проще. Функция встроенного модуля util:

```
const readFilePromise = util.promisify(fs.readFile);
```

Её можно использовать только для функций, имеющих формат вызова callback last, error first

Современный синтаксис async/await

6

Async/await

Упрощённый синтаксис, введённый в стандарте ES2017. Аналогичен тому, что используется в Python.

Внутри промисы, которые используют колбэки, поэтому необходимо изучать разные механизмы

```
async function readFile(name) {
     const data = await fs.promises.readFile(name, ‘utf-8);
     return data;
}

let data = readFile(‘data.json’);
data = JSON.parse(data);
console.log(data);
```

Async/await

Вызов с **await** можно применять только внутри функции, определённой как **async**. Нельзя использовать в глобальном контексте.

Чтобы это ограничение обойти, можно использовать самовызываемую функцию (IIFE):

```
let data;
(async () => {
     data = await fs.promises.readFile(name, ‘utf-8);
     data = JSON.parse(data);
     console.log(data);
})()
console.log(data);
```

Обратите внимание на скобки вокруг определения функции, они обязательны. Второй вызов console.log выведет undefined

Обработка ошибок с async/await

Для обработки ошибок необходимо использовать конструкцию обработки исключений **try/catch**:

```
let data;
(async () => {
     try {
          data = await fs.promises.readFile(name, ‘utf-8);
     } catch (e) {
          console.log(‘Ошибка чтения файла:’, e);
     }
     data = JSON.parse(data);
     console.log(data);
})()
console.log(data);
```

Демонстрация

Давайте посмотрим, как можно использовать асинхронное программирование в JavaScript

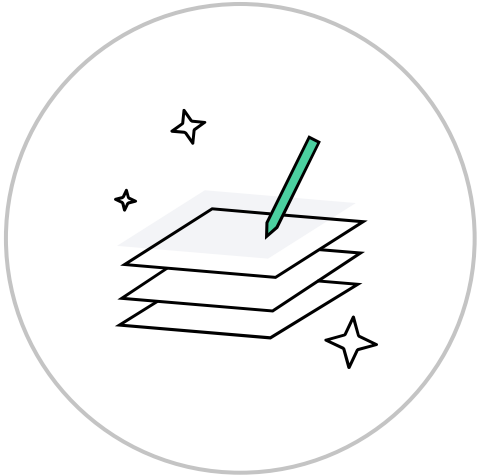
Итоги

1. Познакомились с асинхронным программированием в JavaScript
2. Поняли, как использовать колбэки
3. Научились использовать промисы
4. Изучили синтаксис async/await

Домашнее задание

Давайте посмотрим ваше домашнее задание

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- Обработка ошибок в учебнике по JavaScript
- Замыкания в учебнике по JavaScript
- Цикл событий в учебнике по JavaScript
- Цикл событий в браузере
- Цикл событий в Node.js
- Промисы в учебнике по JavaScript

Задавайте вопросы и пишите отзыв о лекции

Василий Петров
Разработчик Python, JavaScript