

Promise

- Промисы
- Цепочки промисов
- Обработка ошибок
- Микро и макро задачи
- Async/await
- Сетевые запросы

Урок

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18

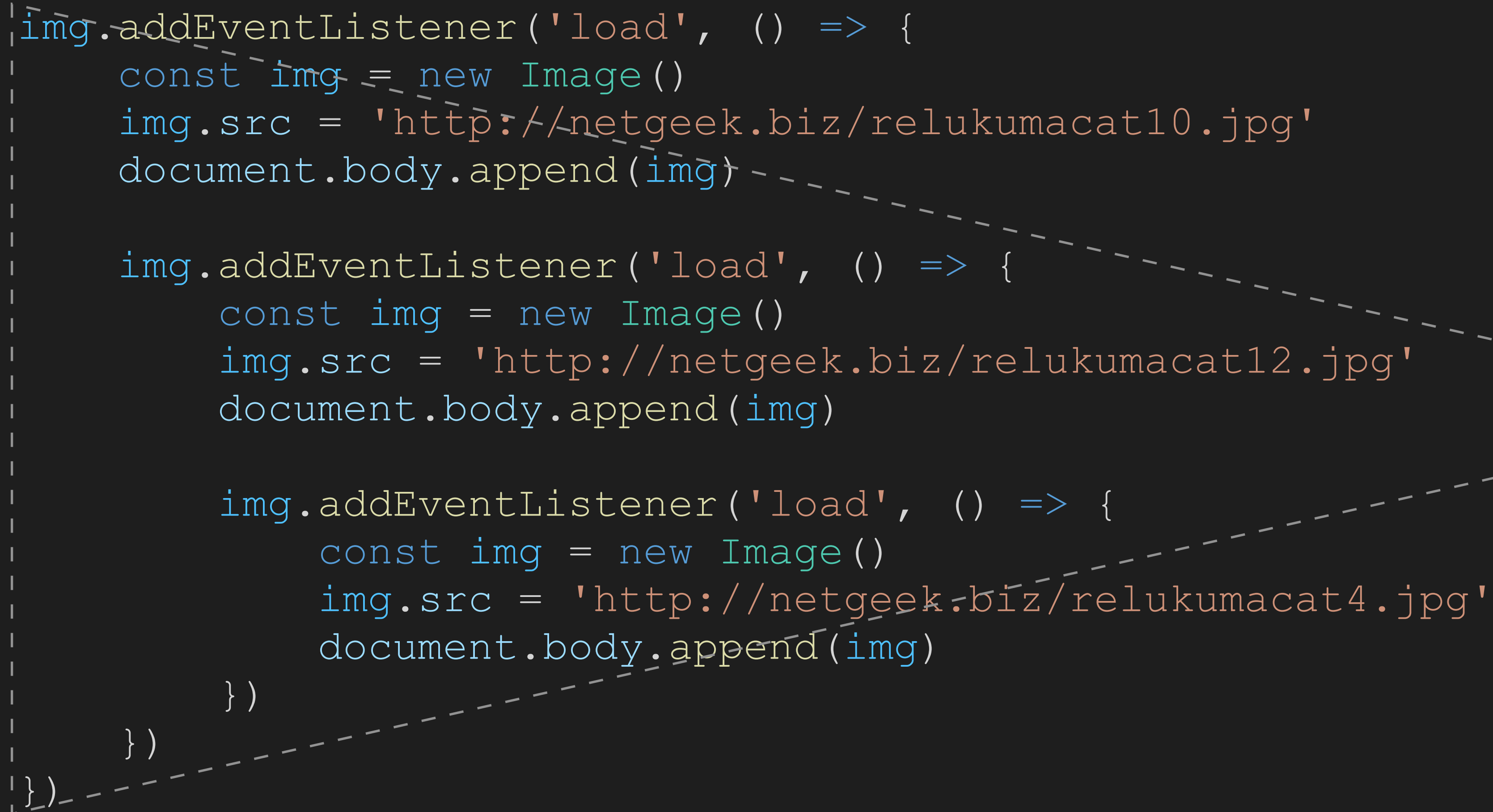
Call-back hell

Callback Hell - это паттерн для управления конкурирующими (асинхронными) запросами, который обеспечивает последовательность их выполнения.

Паттерн "Callback Hell" не удобен для восприятия, из-за этого он и получил свое название. Чем больше уровней вложенности коллбеков, тем труднее понять что происходит. Для того, чтобы предложить более удобную форму записи и облегчить жизнь программистов в ES6 был реализован специальный объект Promise.

Call-back hell

```
img.addEventListener('load', () => {  
  const img = new Image()  
  img.src = 'http://netgeek.biz/relukumacat10.jpg'  
  document.body.append(img)  
  
  img.addEventListener('load', () => {  
    const img = new Image()  
    img.src = 'http://netgeek.biz/relukumacat12.jpg'  
    document.body.append(img)  
  
    img.addEventListener('load', () => {  
      const img = new Image()  
      img.src = 'http://netgeek.biz/relukumacat4.jpg'  
      document.body.append(img)  
    })  
  })  
})
```

A diagram illustrating the concept of 'call-back hell'. It features three nested, dashed-line arrows pointing to the right. The first arrow starts at the first 'img.addEventListener' call and points to the second. The second arrow starts at the second 'img.addEventListener' call and points to the third. The third arrow starts at the third 'img.addEventListener' call and points to the right, ending in a solid grey arrowhead. This visualizes how each callback must wait for the previous one to complete before it can execute, leading to a long chain of dependencies.

Promise

Объект Promise служит удобной оберткой для обслуживания асинхронного функционала.

Основное назначение Promise создавать цепочки вызовов асинхронных функций.

Promise

```
function loadImage(url) {  
  return new Promise((resolve) => {  
    const img = new Image()  
    img.src = url  
    document.body.append(img)  
  
    img.addEventListener('load', () => {  
      resolve()  
    })  
  })  
}  
  
loadImage('http://netgeek.biz/relukumacat7.jpg')  
  .then(() => loadImage('http://netgeek.biz/relukumacat10.jpg'))  
  .then(() => loadImage('http://netgeek.biz/relukumacat12.jpg'))  
  .then(() => loadImage('http://netgeek.biz/relukumacat4.jpg'))
```

Синтаксис создания Promise

Создание объекта Promise

```
let examplePromise = new Promise ( ( resolve, reject ) => {} )
```

Promise принимает в себя функцию executor с двумя аргументами

Executor описывает выполнение какой-то асинхронной работы, по завершении которой необходимо вызвать функцию `resolve` или `reject`. Обратите внимание, что возвращаемое значение функции `executor` игнорируется

Синтаксис создания Promise

Первый аргумент (resolve) вызывает успешное исполнение promise, второй (reject) отклоняет его.

Возвращаем результат успешной работы

Возвращаем результат ошибки

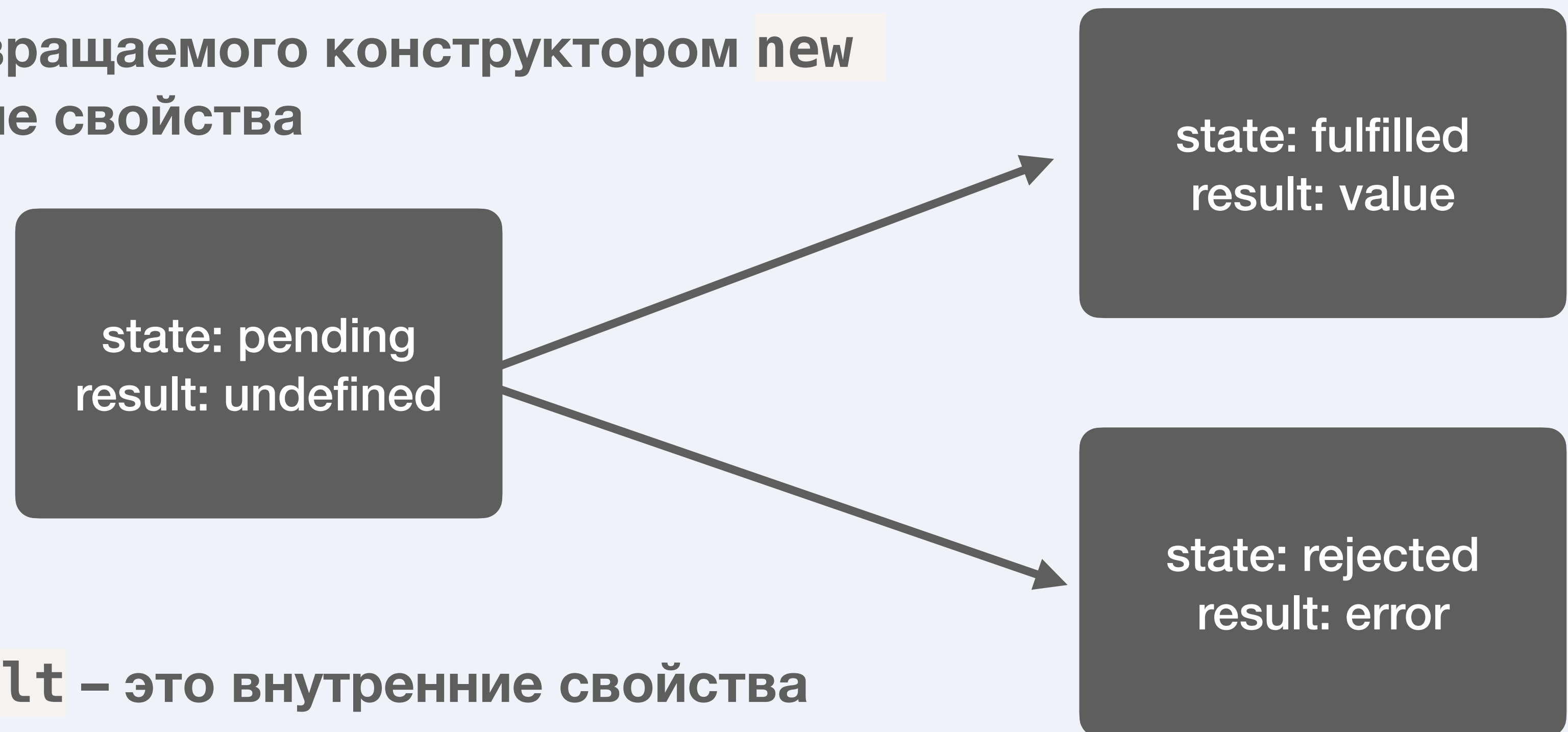
```
let examplePromise = new Promise(( resolve, reject ) => {  
  
    // ... Эxecutor запускается автоматически, он должен выполнить работу, а  
    // затем вызвать resolve или reject  
  
    resolve('Success')  
  
    reject(new Error('error'))  
  
})
```


Важно знать !

- Executor function должен вызвать что-то одно: `resolve` или `reject`. Состояние `promise` может быть изменено только один раз.
- Чтобы снабдить функцию функционалом `promise`, нужно просто вернуть в ней объект `Promise`
- Вызывайте `reject` с объектом `Error`
- Обычно `executor function` делает что-то асинхронное и после этого вызывает `resolve/reject`, то есть через какое-то время. Но это не обязательно, `resolve` или `reject` могут быть вызваны сразу

Promise state

У объекта `promise`, возвращаемого конструктором `new Promise`, есть внутренние свойства



- Свойства `state` и `result` – это внутренние свойства объекта `Promise` и мы не имеем к ним прямого доступа. Для обработки результата следует использовать методы `.then/.catch/.finally`

Методы Promise

`.then()`



Выполняет функцию в зависимости от успешного или неуспешного выполнения promise

`.catch()`



Полезен для обработки ошибок в вашей структуре обещаний.

`.finally()`



Если необходимо произвести какие-либо вычисления или очистку, как только Promise завершен, вне зависимости от результата.

.then ()

- Возвращает `Promise`, который мы можем объединить в цепочку с другим `then`.
- Метод может принимать два аргумента: колбэк-функции для случаев выполнения и отклонения промиса.
- Значения возвращаемые из `onFulfilled` или `onRejected` колбэков будут автоматически обёрнуты в обещание.

.then ()

onFulfilled

onRejected

```
let examplePromise = new Promise((resolve, reject) => {  
  setTimeout(() => resolve('Ready'), 500)  
})  
  
examplePromise.then(  
  result => alert(result),  
  error => alert(new Error('Error'))  
)
```

.catch ()

Сокращённый, «укороченный» вариант `.then(null, f)`.

```
let examplePromise = new Promise((resolve, reject) => {  
    setTimeout(() => reject(new Error('error')), 500)  
})  
  
examplePromise.catch(error => console.log(error))
```

.finally ()

Метод `finally()` возвращает Promise. Когда Promise (обещание) был выполнен, в не зависимости успешно или с ошибкой, указанная функция будет выполнена. Это даёт возможность запустить один раз определённый участок кода, который должен выполняться вне зависимости от того, с каким результатом выполнялся Promise.

`finally` не получает аргументов, так как не существует способа определить, будет ли обещание выполнено успешно или с ошибкой. Данный метод необходимо использовать, если не важна причина ошибки или результат успешного выполнения и, следовательно, нет необходимости её/его передавать.

.finally ()

```
let examplePromise = new Promise((resolve, reject) => {  
    setTimeout(() => reject(new Error('error')), 500)  
})  
  
examplePromise.finally(console.log('THE END'))
```


Цепочка Promise

Обычно требуется выполнить две или более асинхронных операций подряд, при этом каждая последующая операция начинается, когда предыдущая операция завершается успешно, с результатом с предыдущего шага. Мы достигаем этого, создавая цепочку обещаний.

```
function timeCounter(sec) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      console.log(`Прошло ${sec} секунд!`);  
      sec++  
      resolve(sec)  
    }, 1000)  
  })  
}
```

```
timeCounter(1)  
  .then(sec => timeCounter(sec))  
  .then(sec => timeCounter(sec))  
  .then(timeCounter)  
  .then(timeCounter) // сокращенный синтаксис
```

Прошло 1 секунд!

Прошло 2 секунд!

Прошло 3 секунд!

Прошло 4 секунд!

Практика

Воспользуйтесь функцией `setTimeout()` и через секунду после загрузки страницы запросите у пользователя:

- имя
- возраст
- работу
- любимый фильм

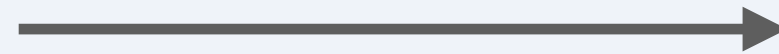
Каждый вопрос для пользователя начинайте спустя 1 секунду как он успешно ответил на предыдущий вопрос.

Реализуйте эту задачу с помощью цепочки из `Promise` которые последовательно запрашивают данные у пользователя.

Если пользователь успешно ввел все данные, то по окончании цепочки выведите сообщение с его данными. А если в каком то поле он не ввел данные то отследите эту ошибку.

Для самостоятельного чтения на learn.javascript.ru

Промисы,
async / await



Главы 11.1 - 11.8