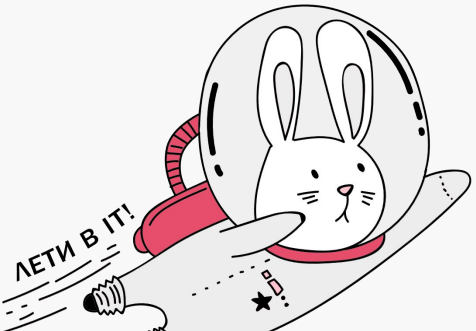
Synchronous/Asynchronous programming
Promise
Async/await
Handling errors in JS



dev {education}

Synchronous programming

В традиционной практике программирования большинство операций ввода-вывода происходит синхронно. Если посмотреть как, например, Java читает файл, получим что-то вроде этого:

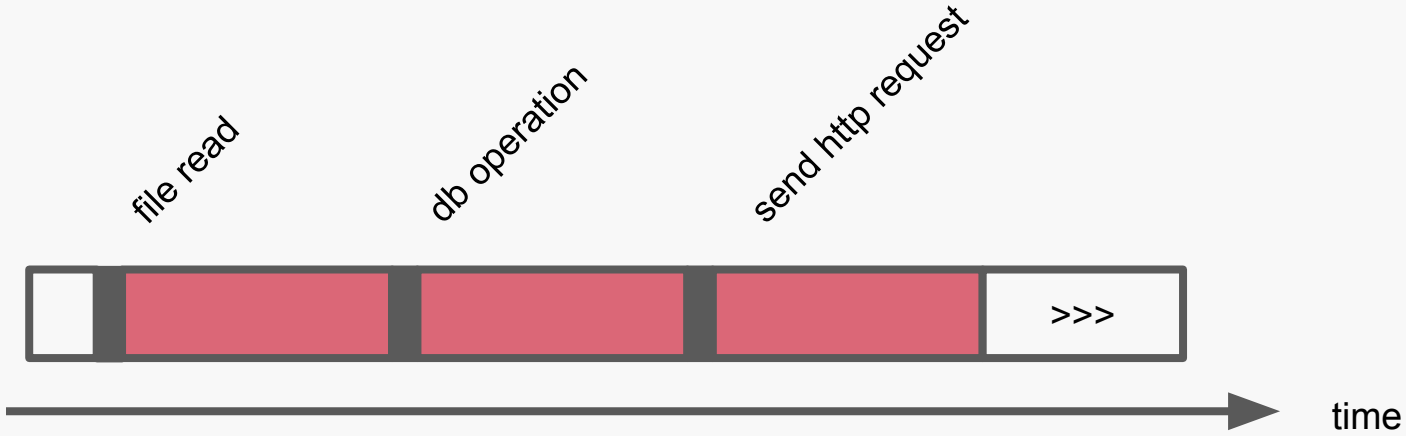
```
try (FileInputStream inputStream = new FileInputStream("foo.txt")) {
   Session IOUtils;
   String  fileContent = IOUtils.toString(inputStream);
}
```

Synchronous programming

Что происходит в фоновом режиме? Основной поток будет заблокирован до тех пор, пока файл не будет прочитан, а это означает, что за это время ничего другого не может быть сделано. Чтобы решить эту проблему и лучше использовать CPU, нам придется управлять потоками вручную.

Если у нас больше блокирующих операций, очередь событий становится ещё хуже.

Synchronous programming



Красные полосы отображают промежутки времени, в которые процесс ожидает ответа от внешнего ресурса и блокируется, чёрные полосы показывают, когда ваш код работает, белые полосы отображают остальную часть приложения

Asynchronous programming

Для решения этой проблемы Node.js предлагает модель асинхронного программирования.

Асинхронный ввод-вывод — это форма обработки ввода/вывода, позволяющая продолжить обработку других задач, не ожидая завершения передачи.

Synchronous programming

Простой пример синхронного чтения файла с использованием Node.js:

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

let data;

try {
   data = fs.readFileSync('file.md', 'utf-8');
} catch (exception) {
   console.log(exception);
}

console.log(data);
```

Synchronous programming

Мы читаем файл, используя синхронный интерфейс модуля fs. Он работает ожидаемым образом: в переменную content сохраняется содержимое file.md. Проблема с этим подходом заключается в том, что Node.js будет заблокирована до завершения операции, то есть, пока читается файл, она не может сделать ничего полезного.

Посмотрим, как мы можем это исправить.

Asynchronous programming

Асинхронное программирование, в том виде, в каком мы знаем его в JavaScript, может быть реализовано только при условии, что функции являются объектами первого класса: они могут передаваться как любые другие переменные другим функциям. Функции, которые могут принимать другие функции в качестве аргументов, называются функциями высшего порядка.

Асинхронное программирование

Один из самых простых примеров функций высшего порядка:

```
const numbers = [2, 4, 1, 5, 4];

function isBiggerThanTwo(num) {
   return num > 2;
}

numbers.filter(isBiggerThanTwo);
```

Асинхронное программирование

В приведенном выше примере мы передаем функцию isBiggerThanTwo в функцию filter. Таким образом, мы можем определить логику фильтрации.

Так появились функции обратного вызова (колбеки): если вы передаете функцию другой функции в качестве параметра, вы можете вызвать её внутри функции, когда она закончит свою работу. Нет необходимости возвращать значения, нужно только вызывать другую функцию с этими значениями.

Asynchronous programming

В основе Node.js лежит принцип «первым аргументом в колбеке должна быть ошибка». Его придерживаются базовые модули, а также большинство модулей, найденных в NPM.

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

fs.readFile('file.md', 'utf-8', function (error, data) {
    if (error) {
        console.log(error);
    }
    console.log(data);
});
```

Asynchronous programming

Что следует здесь выделить:

- обработка ошибок: вместо блока try-catch вы проверяете ошибку в колбеке
- отсутствует возвращаемое значение: асинхронные функции не возвращают значения, но значения будут переданы в колбеки

Немного изменим код выше, чтобы увидеть, как это работает на практике:

Asynchronous programming

```
const fs = require('fs');
console.log('start reading a file...');

fs.readFile('file.md', 'utf-8', function (error, data) {
   if (error) {
       throw new Error("error happened during reading the file", error);
   }
   console.log(data);
});

console.log('end of the file');
```

Asynchronous programming

Результатом выполнения этого кода будет:

```
start reading a file...
end of the file
ErrorMessage = error happened during reading the file
```

Как только мы начали читать наш файл, выполнение кода продолжилось, а приложение вывело end of the file. Наш колбек вызвался только после завершения чтения файла. Как такое возможно?
Благодаря циклу событий (event loop).

Event Loop

Цикл событий лежит в основе Node.js и JavaScript и отвечает за планирование асинхронных операций.

Программирование с управлением по событиям представляет собой парадигму программирования, в которой поток выполнения программы определяется событиями, такими как действия пользователя (щелчки мышью, нажатия клавиш), выходы датчиков или сообщения из других программ/потоков. (Событийная модель).

На практике это означает, что приложения реагируют на события.

Кроме того, как мы уже узнали, с точки зрения разработчика Node.js является однопоточным. Это означает, что вам не нужно иметь дело с потоками и синхронизировать их, Node.js скрывает эту сложность за абстракцией. Всё, кроме кода, выполняется параллельно.

Promise

Объект Promise используется для отложенных и асинхронных вычислений. Промис представляет собой операцию, которая еще не завершена, но ожидается в будущем.

Promise

Синтаксис:

```
new Promise(executor);
new Promise(function(resolve, reject) { ... });
```

executor

Объект функции с двумя аргументами resolve и reject. Первый аргумент вызывает успешное выполнение обещания, второй отклоняет его. Мы можем вызывать эти функции по завершении нашей операции.

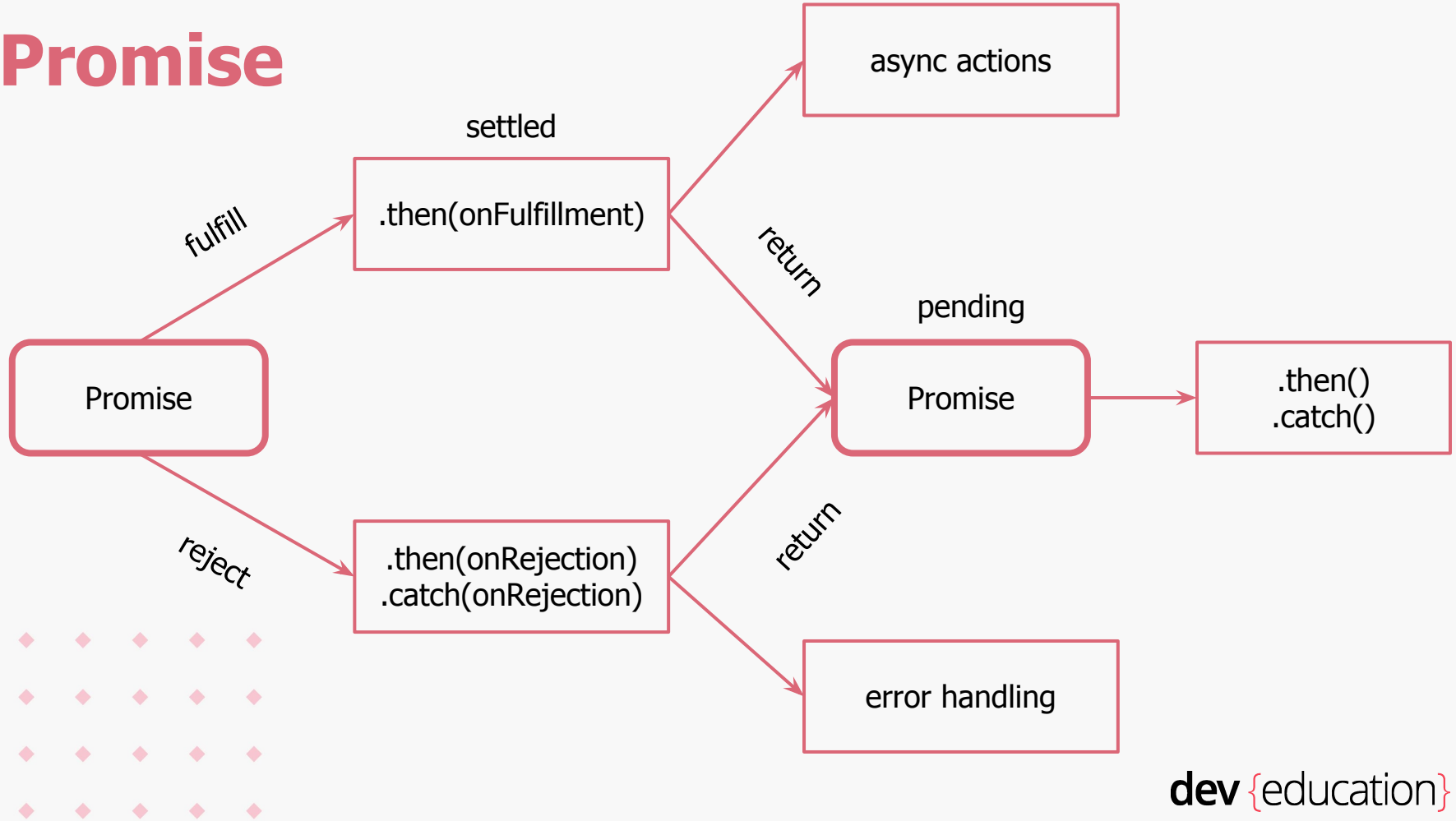
Promise

Интерфейс Promise (обещание) представляет собой обертку для значения, неизвестного на момент создания обещания. Он позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными: вместо конечного результата асинхронного метода возвращается обещание получить результат в некоторый момент в будущем.

При создании обещание находится в ожидании (pending), а затем может стать выполнено (fulfilled), вернув полученный результат (значение), или отклонено (rejected), вернув причину отказа. В любом из этих случаев вызывается обработчик, прикрепленный к обещанию методом then. Если в момент прикрепления обработчика обещание уже сдержано или нарушено, он все равно будет выполнен, т.е. между выполнением обещания и прикреплением обработчика нет «состояния гонки», как, например, в случае с событиями в DOM.

Так как методы Promise.prototype.then и Promise.prototype.catch сами возвращают обещания, их можно вызывать цепочкой, создавая соединения.

Promise



dev {education}

Promise

Методы:

Promise.all(iterable)

Возвращает обещание, которое выполнится после выполнения всех обещаний в передаваемом итерируемом аргументе.

Promise.race(iterable)

Возвращает обещание, которое будет выполнено или отклонено с результатом исполнения первого выполненного или отклонённого итерируемого обещания.

Promise.reject(reason)

Возвращает объект Promise, который отклонен с указанной причиной.

Promise.resolve(value)

Возвращает объект Promise, который выполнен с указанным значением. Если значение может быть продолжено (имеется метод then), то возвращаемое обещание будет "следовать" продолжению, выступая адаптером его состояния; в противном случае будет возвращено ожидание в выполненном состоянии.

Promise

На практике пример чтения файла можно переписать следующим образом:

```
function stats (file) {
    return new Promise((resolve, reject) => {
        fs.stat(file, (err, data) => {
            if (err) {
                reject(err);
            }
            resolve(data);
        })
    })
}
```

```
Promise.all([
  stats('file1'),
  stats('file2'),
  stats('file3')
])
  .then((data) => console.log(data))
  .catch((err) => console.log(err));
```

Async/await

Объявление async function определяет асинхронную функцию, которая возвращает объект AsyncFunction.

```
async function name([param[, param[, ... param]]]) {
   statements
}
```

Async/await

После вызова функция async возвращает Promise. Когда результат был получен, Promise завершается, возвращая полученное значение. Когда функция async выбрасывает исключение, Promise ответит отказом с выброшенным (throws) значением.

Функция async может содержать выражение await, которое приостанавливает выполнение функции async и ожидает ответа от переданного Promise, затем возобновляя выполнение функции async и возвращая полученное значение.
Ключевое слово await допустимо только в асинхронных функциях. В другом контексте вы получите ошибку SyntaxError.

Цель функций async/await упростить использование promises синхронно и воспроизвести некоторое действие над группой Promises. Точно так же как Promises подобны структурированным callback-ам, async/await подобна комбинации генераторов и promises.

Async/await

```
function resolveAfter2Seconds(x) {
      return new Promise(resolve => {
            setTimeout(() => {
                  resolve(x);
            }, 2000);
      });
}

async function add1(x) {
      const a = await resolveAfter2Seconds(20);
      const b = await resolveAfter2Seconds(30);
      return x + a + b;
}

add1(10).then(v => {
      console.log(v);  // prints 60 after 4 seconds.
});
```

```
async function add2(x) {
   const a = resolveAfter2Seconds(20);
   const b = resolveAfter2Seconds(30);
   return x + await a + await b;
}

add2(10).then(v => {
   console.log(v);  // prints 60 after 2 seconds.
});
```

Async/await

Функция add1 приостанавливается на 2 секунды для первого await и еще на 2 для второго. Второй таймер создается только после срабатывания первого. В функции add2 создаются оба и оба же переходят в состояние await. В результате функция add2 завершится скорее через две, чем через четыре секунды, поскольку таймеры работают одновременно. Однако запускаются они все же не параллельно, а друг за другом - такая конструкция не означает автоматического использования Promise.all. Если два или более Promise должны разрешаться параллельно, следует использовать Promise.all.

Handling errors in JS

Инструкция throw позволяет генерировать исключения, определяемые пользователем. При этом выполнение текущей функции будет остановлено (инструкции после throw не будут выполнены), и управление будет передано в первый блок catch в стеке вызовов. Если catch блоков среди вызванных функций нет, выполнение программы будет остановлено.

```
throw new Error(args);
```

Handling errors in JS

Конструкция try...catch помечает блок инструкций как try, и в зависимости от того, произошла ошибка или нет, вызывает дополнительный блок инструкций catch.

Синтаксис:

```
try {

} catch (error) {

}
```

Handling errors in JS

Конструкция try содержит блок try, в котором находится одна или несколько инструкций ({} должно быть всегда использовано, даже для одиночных инструкций), и как минимум один блок catch, или один блок finally, или оба. Здесь приведены три возможных варианта использования конструкции try:

try...catch
try...finally
try...catch...finally

Handling errors in JS

Блок catch содержит инструкции, которые будут выполнены, если в блоке try произошла ошибка. Это сделано для того, чтобы была возможность обработать ошибку в блоке catch, при её возникновении. Если какая-либо инструкция вызывает ошибку в try блоке, то управление незамедлительно переходит в блок catch. Если в try блоке не будет никакой ошибки, то блок catch пропускается.

Handling errors in JS

Блок finally выполнится после выполнения блоков try и catch, но перед инструкциями, следующими за конструкцией try...catch. Этот блок всегда выполняется независимо от того, была ошибка или нет.

Вы можете размещать один или более try оператор. Если внутренний try оператор не имеет catch блок, будет использован catch внешнего оператора try.

Вы также можете использовать оператор try для обработки JavaScript исключений.

Handling errors in JS

Конструктор Error создаёт объект ошибки. Экземпляры объекта Error выбрасываются при возникновении ошибок во время выполнения. Объект Error также может использоваться в качестве базового для пользовательских исключений.

new Error([message[, fileName[, lineNumber]]])

message необязательный - Человеко-читаемое описание ошибки.

fileName необязательный - Значение свойства fileName созданного объекта Error. Значением по умолчанию является имя файла, содержащего код, вызвавший конструктор Error().

lineNumber необязательный - Значение свойства lineNumber созданного объекта Error. Значением по умолчанию является номер строки, содержащей вызов конструктора Error().