PHASE 1

WEEK 2

DAY 4

ELBRUS/BOOTCAMP

План

1. Асинхронность (asynchrony)
1.1. Callback Hell
2. Promises
3. Промисификация
4. fs.promises API
5. Async / await

Асинхронность

Асинхронность в JavaScript

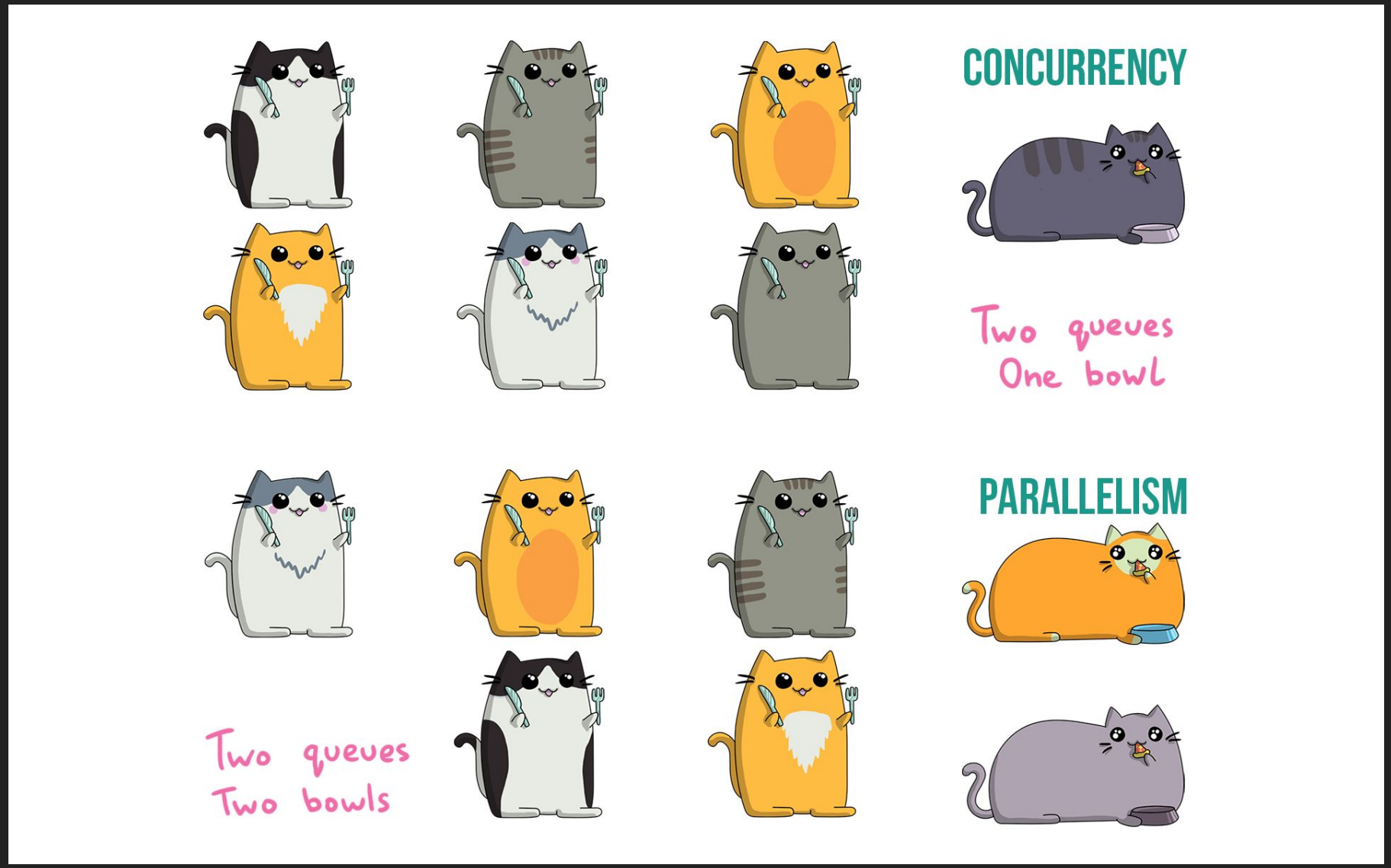
JavaScript — это…

- конкуррентный (concurrent)
  в одно и то же время может происходить несколько явлений.
- асинхронный
  эти явления могут происходить независимо и не дожидаться завершения друг друга
- однопоточный язык
  может обрабатывать лишь одну инструкцию за раз

Асинхронность в JavaScript

Параллелизм в JavaScript достигается за счёт **среды выполнения**, а не языка.

Асинхронность в JavaScript



Callback Hell

Пример callback hell

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

// добро пожаловать в ад callback-функций :)
fs.writeFile(`file.txt`, `1\n`, () => {
 fs.appendFile(`file.txt`, `2\n`, () => {
   fs.appendFile(`file.txt`, `3\n`, () => {
     fs.appendFile(`file.txt`, `4\n`, () => {
     });
   });
 });
});
```

Недостатки callback hell

- Код растянут в ширину, его неудобно читать и понимать
- Больше вероятность совершить ошибку
- Результат выполнения доступен только на следующем уровне вложенности

Promise

обещание

Promise

`Promise` – объект, который предоставляет доступ к значению, которое будет получено когда-то в будущем.

Promise позволяет установить значение только один раз.

Состояние объекта Promise

`Promise` может находиться в одном из трёх состояний:

- Pending / ожидание
- Fulfilled / выполнено
- Rejected / отклонено

Пример промиса

```
const promise = new Promise((resolve, reject) => {
  if (Math.random() > 0.5) {
    resolve('Success');
  } else {
    reject('Error');
  }
});

promise
  .then(console.log)
  .catch(console.log);

// что будет в консоли и в каком порядке?
console.log(promise);
```

Promise и параллелизм

`Promise` позволяет подождать завершение асинхронной операции.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    if (Math.random() > 0.5) {
      resolve('Success');
    } else {
      reject('Error');
    }
  }, 2000);
});

promise
  .then(console.log)
  .catch(console.log);
```

Цепочка промисов / promise chain

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

fs.readFile('./tmp/1.txt', 'utf8')
  .then((fileData) => fs.writeFile('./tmp/1-copy.txt', fileData))
  .then(() => fs.readFile('./tmp/2.txt', 'utf8'))
  .then((fileData) => fs.writeFile('./tmp/2-copy.txt', fileData))
  .then(() => console.info('Файлы скопированы успешно'))
  .catch(console.error);
```

Методы объекта Promise

- `Promise.prototype.then()`
  - Принимает обработчик успешного завершения.
  - Опционально принимает обработчик ошибки.

- `Promise.prototype.catch()`
  - Принимает только обработчик ошибки.

- `Promise.prototype.finally()`
  - Принимает callback-функцию, которая сработает при завершении работы промиса, независимо от полученного результата.

Методы класса Promise

- `Promise.all()`
  - Принимает итерируемый объект (например, массив) с несколькими промисами.
  - Возвращает единый промис, который выполнится в массив результатов всех промисов.
  - Если какой-нибудь из входных промисов отклонится, возвращаемый промис также сразу отклонится.
  - Использовать для выполнения связанных асинхронных операций, все из которых должны завершиться успешно.
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Promise.all()

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 5000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 3000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(data => console.log(data)); // когда все промисы
выполнятся, формируется массив результатов каждого промиса
```

Методы класса Promise

- `Promise.allSettled()`
  - Возвращает единый промис, который выполнится в массив результатов всех промисов.
  - Все входные промисы выполняются до конца.
  - Использовать для выполнения несвязанных асинхронных операций, либо когда важно отследить статус выполнения каждого промиса.
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled

Promise.allSettled()

```
const promise1 = Promise.resolve(2021);
const promise2 = new Promise((resolve, reject) =>
setTimeout(reject, 1000, 'No!'));

Promise.allSettled([promise1, promise2])
  .then((results) => results.forEach((result) =>
console.log(result.value))) // 2021, undefined
```

Методы класса Promise

- `Promise.race()`
  - Возвращает единый промис, который выполнится или отклонится с результатом первого выполненного промиса.
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race

Методы класса Promise

- `Promise.resolve()`
  - Возвращает промис, который выполнится с указанным результатом.
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve

- `Promise.reject()`
  - Возвращает промис, который будет отклонён с указанной причиной.
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject

Промисификация

Промисификация

Преобразование функции, которая принимает `callback`, в функцию, которая возвращает объект `Promise`.

Промисификация

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

function myReadFile(file, encoding) {
  return new Promise((resolve, reject) => {
    fs.readFile(file, encoding, (error, data) => {
      if (error) {
        return reject(error);
      }
      resolve(data);
    });
  });
}

myReadFile('./files/1.txt', 'utf8')
  .then(console.log)
  .catch(console.error);
```

fs.promises API

fs.promises API

`fs.promises API` обеспечивает альтернативный набор асинхронных методов файловой системы, которые в свою очередь возвращают объект `Promise`, а не классические `callback`.

Пример подключения модуля:

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

fs.readFile('./files/1.txt', 'utf8')
  .then((data) => console.log(data));
```

async / await

синтаксическая обёртка для работы с Promise

Async / await

`Async` – ключевое слово, которое ставится перед функцией. После указания `async` функция всегда возвращает промис.

`Await` тоже ключевое слово, которое заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от `await` не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

async !== асинхронная функция

Эта функция **не** асинхронная.

```
async function foo() {
  console.log('Foooo');
}

console.log('Start');
foo();
console.log('Finish');
```

А эта?

```
async function foo() {
 return new Promise((resolve) => {
   console.log('Foooo');
   resolve();
 });
}

console.log('Start');
foo();
console.log('Finish');
```

Асинхронное выполнение и await

```
// Пример 1
async function foo() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Foooo');
      resolve();
    }, 1000);
  });
}
console.log('Start');
foo();
console.log('Finish');
```

Асинхронное выполнение и await

```
// Пример 2
async function foo() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Foooo');
      resolve();
    }, 1000);
  });
}
console.log('Start');
await foo(); // запускать необходимо в окружении async
console.log('Finish');
```

А где catch?

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

// так делать не надо (неудобно читать, непонятен инициатор ошибки)
try {
 let fileContent = await fs.readFile('./tmp/1.txt', 'utf-8');
 await fs.writeFile('./tmp/1-copy.txt', fileContent);
 fileContent = await fs.readFile('./tmp/2.txt', 'utf-8');
 await fs.writeFile('./tmp/2-copy.txt', fileContent);
} catch (err){
 console.error('Ошибка', err);
}
console.info('Файлы скопированы успешно');
```