async programming in JS

Part2

callbacks, Promises and async/await

RECAP

History of async

- JS ako jazyk nemal žiadnu notáciu pre asynchronicitu
- Callback bol fajn mechanizmus ako to zapísať syntakticky a čiastočne aj semanticky (všetko čo je odtabované je “LATER“)
- Callback ako taký ale nič nevraví o asynchronicite

- promises (ako nezávislý štandard) potom ako súčasť ES
- generatory
- async await
- async iterators

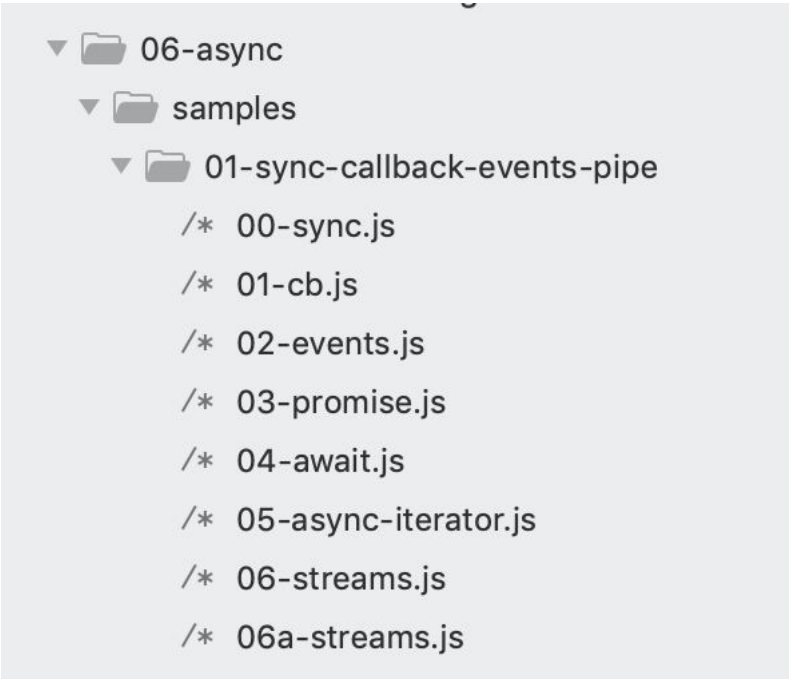
RECAP

Async styles

- sync

1. async **callback**
2. async **events**
3. async **promises**
4. async **async/await**
5. async **iterators/generators**
6. streaming, **pipe, pipeline**

- node.js core module **fs** is a good demo of all styles, it supports all of them, samples:



RECAP

Async function - definitions

- Pod **async funkciou** budeme zjednodušenie rozumieť jednu z konštrukcií, ktorá nám umožní zapísať, že niečo sa deje **NOW** a potom **LATER**. V zásade existujú 3 možnosti ako je async funkcia implementovaná:

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

```
function task1(params,callback) {
  //...
  callback(null, data);
  callback(err, data);
}

function task2(params) {
  //...
  return Promise.resolve(data);
  return Promise.reject(err);
}

async function task3(params) {
  //...
  return data; //impl. promise
  throw err;
}
```

RECAP

Async function - usage

- Podľa toho potom rôzne zapisujeme „**later**“

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

Niektoré knižnice sú napísané štýlom A) iné B) iné podporujú A/B/C ..... Niektoré runtimes podporujú len A, niektoré A,B,C

```
// how and where continuation
// is written

task1(args, (err, cb) => {
  if (err) {
    //...
  } else {
    //...
  }
});

task2(args)
  .then(data => {
    //...
  })
  .catch(err => {
    // ...
  });


try {
  let data = await task3(args);
  //...
} catch (err) {
  //...
}
```

1995-2020

2010
2015-2020

2017-2020

Callbacks

RECAP

Callbacks

- When your JS program makes async operation, you set up the "response" code in a function (commonly called a "callback"), and the JS engine tells the hosting environment, "Hey, I'm going to suspend execution for now, but whenever you finish with that request, and you have some data, please *call* this function *back*.“ (@getify)

Sync / async code side by side

sync.js

```
5  // function xyzSync(p1,p2){ return r, throw e }
6
7  function mkdtemp() {
8    const ret = fs.mkdtempSync(os.tmpdir());
9    return ret;
10
11 }
12
13 function writeFile(file, data) {
14   fs.writeFileSync(file, data);
15   return file;
16
17 }
18
19 function writeTempFile(file, data) {
20   const td = mkdtemp();
21
22   const tf = path.join(td, file);
23   const ret = writeFile(tf, data);
24   return ret;
25
26
27 }
28
29 const f = writeTempFile("x.txt", "test")
30 console.log(f);
```

async.js

```
5  // function xyzAync(p1,p2,cb){ cb(e,r) }
6
7  function mkdtemp(cb) {
8    fs.mkdtemp(os.tmpdir(), (err, ret) => {
9      cb(err, ret);
10   });
11 }
12
13 function writeFile(file, data, cb) {
14   fs.writeFile(file, data, (err) => {
15     cb(err, file)
16   });
17 }
18
19 function writeTempFile(file, data, cb) {
20   mkdtemp((err, td) => {
21     if (err) return cb(err);
22     const tf = path.join(td, file);
23     writeFile(tf, data, (err, ret) => {
24       cb(err, ret);
25     });
26   })
27 }
28
29 writeTempFile("x.txt", "test", (err, f) => {
30   console.log(f);
```

Callbacks (+ pozitíva)

- callbacks are by far the **most common way** that asynchronycity in JS programs is expressed and managed
- Indeed, the callback is the **most fundamental async pattern** in the language
- Countless JS programs, even very **sophisticated** and **complex** ones, have been written upon no other async foundation than the callback
- The callback function is the **async work horse for JavaScript**, and it does its job **respectably**

https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch2.md

Callbacks (- negatíva)

- It turns out that how we express asynchrony (with callbacks) in our code doesn't map very well at all to that **synchronous brain** planning behavior
- The reason it's difficult for us as developers to write async evented code, especially when all we have is the callback to do it, is that stream of consciousness thinking/planning is **unnatural for most of us.**
- We think in step-by-step terms, but the tools (callbacks) available to us in code are not expressed in a step-by-step fashion once we move from synchronous to asynchronous.
- And that is why it's so **hard to accurately author and reason about async JS code with callbacks**: because it's not how our brain planning works.

https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch2.md

Callbacks – odpoveď na tému sekvencneho kodu

- **Getify**: We think in step-by-step terms, but the tools (callbacks) available to us in code are not expressed in a step-by-step fashion once we move from synchronous to asynchronous.

- **Marcus**: tak prestan premyslat step-by-step a zacni rozmyslat deklarativne, v zmysle co na com zavisi a mas po probleme, vid async patterns co sme si ukazovali, ale jasne ze nie na kazdy problem je toto vhodne riesenie

Callback hell (čo to znamená **naozaj**)

Toto sú podstatné problémy callbackov:

1. order of code is not sequential
2. with callback **we *trust* the 3rd party** (the one who calls our function) to
   A. call us **only once**
      A. but it can call us **multiple times,**
      B. or **not at all**
   B. call us **in the async or sync** fashion, but it can call us sync one time and async second time
3. hardcoded continuations (names of next functions)
4. error handling paths complicate the code
5. error handling of sync parts of async. functions (writeFile and encoding)
6. cluttered code *hard to reason* about
7. … indentation … is just side effect not helping previous real problems
8. …..

(1) Poradie kódu a poradie vykonávania

```
a(function() {
  b();

  c(function() {
    d();
  })

  e();
});
f();
```

Zo syntaxe zápisu usudzujeme, že

- **a, c** sú **async** funkcie
- **b,d,e** sú **sync** funkcie

Poradie vykonania bude potom:

a, f, b, c, d, e

Čo nezodpovedá poradiu zápisu v kóde, čo komplikuje čítanie a analýzu kódu.

(1) Poradie kódu a poradie vykonávania

```
a(function() {
  b();

  c(function() {
    d();
  })

  e();
});
f();
```

Zo synaxe zápisu len usudzujeme, že

- a, c sú async funkcie
- b,d,e sú sync funkcie

Čo ak to tak **nie je ?** čo aj je jedna z nich synchrónna , alebo obe ? dostaneme iné poradie:

```
// order-async-ac.js          a,f,b,c,e,d
// order-sync-a.js            a,b,c,e,f,d
// order-sync-c.js            a,f,b,c,d,e

// order-sync-depends.js
//                            a,f,b,c,e,d
//                            a,b,c,d,e,f
//                            a,b,c,e,f,d
//                            a,f,b,c,d,e
```

prednasky/06-async/samples/02-callbacks/order-*.js

(2A) Počet zavolaní callback-u

```
a(function() {
  b();

  c(function() {
    d();
  })

  e();
});
f();
```

Pri callbackoch sa spoliehame, na autora knižnice, že náš cb zavolá práve raz, ale to nemusí byť pravda, môže nás zavolať viackrát (by design alebo “lebo bug“),

a() - je implementovaná „zle“ a niekedy zavolá callback raz, inokedy dva razy

b() – zase niekedy nezavolá cb vôbec

```
// order:
// a,f,b,c,e,d
// a,f,b,c,e,b,c,e
// a,f,b,c,e,b,c,e,d,d
```

prednasky/06-async/samples/02-callbacks/order-times.js

(2B) niekedy async, niekedy sync

```
a(function() {
  b();

  c(function() {
    d();
  })

  e();
});
f();
```

Zo syntaxe zápisu len usudzujeme, že a(), c() sú async funkcie

Ale čo ak autor kódu implementoval volanie cb raz async a raz sync spôsobom ?

Prečo by to robil ?

napr: caching v nasledovnom príklade:

```
function a(arg, cb) {
  if (cache[arg])    //sync
      cb(null, cache[arg]);
  else                //async
      doSomeIO(arg, cb);
}
```

prednasky/06-async/samples/02-callbacks/order-times.js

(2B) niekedy async, niekedy sync

```
function a(arg, cb) {
    if (cache[arg])     //sync
        cb(null, cache[arg]);
    else                //async
        doSomeIO(arg, cb);
}
```

> If you have an API which takes a callback,
> and *sometimes* that callback is called immediately,
> and *other times* that callback is called at some point in the future,
> then you will render any code using this API impossible to reason about, and cause the release of Zalgo.

prednasky/06-async/samples/02-callbacks/sometimes-async.js

prečo je “niekedy sync/async“ funkcia problém (ozaj v skratke):

- nemôžte sa spoľahnúť na poradie (viď predošlý príklad)
  - b,c,e .... e vs.
  - b,c,d,e
- Maximum call stack size exceeded

http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony

https://blog.ometer.com/2011/07/24/callbacks-synchronous-and-asynchronous/

Riešenia:

(2AA) cb called multiple times

- As authors: it is bug, do all callback paths coverage with your **tests**
- As consumers: TODO: naštudovať, vymyslieť

(2AB) cb never called

- As authors: it is bug, do all callback paths coverage with your **tests**
- As consumers: wrap your functions with **async.timeout(fn)**

(3) cb called sometimes in sync sometimes in async fashion:

- As authors: Do not produce sometimes sync, async functions
- As consumers: When consuming sometimes async function wrap it with **async.ensureAsync(fn)**

solutions

async.**ensureAsync** and
async.**timeout**
once: **TODO**

```
function a(cb) {
  // sometime sync
  console.log("a");
  sometimes() ? setTimeout(cb, 0) : cb();
}
function c(cb) {
  console.log("c");
  // sometimes never
  if (sometimes()) setTimeout(cb, 0);
}
```

```
 3  a(function() {
 4    b();
 5    c(function(err) {
 6      if (err) throw err;
 7      d();
 8    })
 9
10    e();
11  });
12  f();
```

```
// order:
// a;b;c;e;f;d  - sync a
// a;b;c;e;f;   - never d
// a;f;b;c;e,d  - async a
// a;f;b;c;e    - never d
```

```
 3  ensureAsync(a)(function() {
 4    b();
 5    timeout(c, 100)(function(err) {
 6      if (err) throw err;
 7      d();
 8    })
 9
10    e();
11  });
12  f();
```

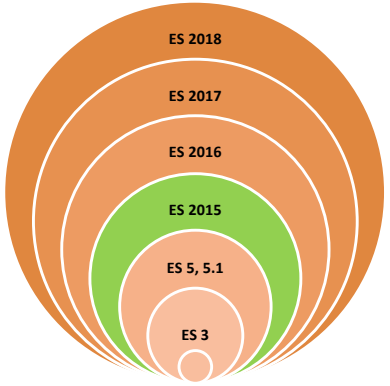
```
// order:
// a;f;b;c;e;d          - async a
// a;f;b;c;e; error     - never d
```

Promises

Promises history

- Implemented before ES 2015
  - koncept z 1976
  - **jQuery** Deferred 2011
  - dojo toolkit 2012
  - $q ????
  - bluebird 2013
  - ...
- *„standardized“* before ES 2015
  - http://wiki.**commonjs**.org/wiki/Promises

ES 2015 (*6th*)



- Promise, Generators
- arrow functions
- destructuring assignment, spread operator
- rest and default parameters
- Template Strings
- block scope variables (let, const)
- for..of loop and iterators on build ins
- Object.* - setPrototypeOf, assign, is() SameValue equality,
- Symbols
- Classes
- Number.* non coercing functions
- String codePoints
- Map, Set
- Proxy, Reflect

CommonJS:

- Promises/A by Kris Zyp — "Thenables"
- Promises/B by Kris Kowal — Opaque Q API
- Promises/KISS by AJ ONeal
- Promises/C has been redacted
- Promises/D by Kris Kowal — "Promise-sendables", for interoperable instances of Promises/B.

Promises - motivation

- **RECAP** - ***callbacks,*** in fact, ***are not*** specialized ***async API***, callbacks can be used to express async APIs (continuation) because JS has functions as first class object
- **Motivation**: create ***Async API*** for JS - promises try to solve *the "problems" of callbacks*
  A. lack of sequentiality
  B. lack of trustability
  C. manage basic concurrency patterns

Promises - features

- A Promise is an object representing the **eventual** completion or failure of an asynchronous operation
- Instead of „*returning from async function*“ by calling callback, we return Promise object, instead of immediately returning the final value, **the asynchronous method returns a *promise* to *supply the value at some point in the future*.**
- Promise is a returned **object to which you attach callbacks, instead of passing callbacks into a function**
- Instead of handing the continuation of our program to another party, we have now capability to know when its task finishes, and then **our code could decide what to do next**

Promises - features

- Callbacks will never be called before the completion of the current run of the JavaScript event loop or in other words: functions passed to then() **will never be called synchronously**
- Callbacks added with then() **even after the success or failure** of the asynchronous operation, will be called.
- **Multiple callbacks** may be added by calling then() several times. Each callback is executed one after another, in the order in which they were inserted
- Promises are **chainable**
- Promises support **Error propagation**
- Support basic **async patterns** – waterfall, all, race
- ...

RECAP

Async function - definitions

- Pod **async funkciou** budeme zjednodušenie rozumieť jednu z konštrukcií, ktorá nám umožní zapísať, že niečo sa deje **NOW** a potom **LATER**. V zásade existujú 3 možnosti ako je async funkcia implementovaná:

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

```
function task1(params,callback) {
  //...
  callback(null, data);
  callback(err, data);
}

function task2(params) {
  //...
  return Promise.resolve(data);
  return Promise.reject(err);
}

async function task3(params) {
  //...
  return data; //impl. promise
  throw err;
}
```

RECAP

Async function - usage

- Podľa toho potom rôzne zapisujeme „**later**“

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

Niektoré knižnice sú napísané štýlom A) iné B) iné podporujú A/B/C ..... Niektoré runtimes podporujú len A, niektoré A,B,C

```
// how and where continuation
// is written

task1(args, (err, cb) => {
  if (err) {
    //...
  } else {
    //...
  }
});

task2(args)
  .then(data => {
    //...
  })
  .catch(err => {
    // ...
  });


try {
  let data = await task3(args);
  //...
} catch (err) {
  //...
}
```

1995-2020

2010
2015-2020

2017-2020

Working with promises

How do we get promise:

1. **using API that already returns** promises (most of newer APIs already do)
2. **Converting** *cb-async-function* to ***promise-async-function***
3. *Using **promisified** npm module*
4. Manually creating new Promise
   - new Promise()
   - Promise.resolve()
   - Promise.reject()
5. Result of ***„combining“*** *promise async functions (.then) and promises (.all, .race)*

Working with promises – promise API

How do we use promises:

- then()
- catch()
- finally()

- promise.all()
- promise.race()

Non standard

- .done()
- .cancel()
- Promise.any()
- …

node.js APIs - How do we get promise ?

- node core modules already producing promises (10+, experimental APIs)
  - DNS Promises API
  - FS promises API
  - ...
- converting callback APIs to promises
  - util.promisify
  - ...
- Promisified modules (on npm)
  - request-promise
  - fs-extras
  - ...
- Hybrid modules: (returns a Promise if no callback is passed to it)
  - node-mongodb-native
  - ...

fs.promises example

- [1] – require *promisified* version of fs module
- [8] – mktempdir now returns promise, when folder is created then [9] call another function, which creates file [11], which once created, return name of the tempFile [12].

```
1  const fs = require("fs").promises;
2  const os = require("os");
3  const path = require("path");
4
5  function writeTempFile(fileName, ...args) {
6
7    let tempDir = path.join(os.tmpdir(), `${process.pid}-`);
8    return fs.mkdtemp(tempDir)
9      .then((folder) => {
10       let tempFile = path.join(folder, fileName);
11       return fs.writeFile(tempFile, ...args)
12         .then(() => tempFile);
13     })
14 }
15
16 writeTempFile("test.txt", "test")
17   .then((path) => console.log(path));
```

07-prednaska/samples/03-promises/01-fs.promises

See compared with cb style on next slide

Promises vs. callbacks, writeTempFile samples

```
function writeTempFile(fileName, ...args) {

  let tempDir = path.join(os.tmpdir(), `${process.pid}-`);
  return fs.mkdtemp(tempDir)
    .then((folder) => {
      let tempFile = path.join(folder, fileName);
      return fs.writeFile(tempFile, ...args)
        .then(() => tempFile);
    })
}
```

- clean signatures, **no cb**
- **return** statement instead of CB
- **error propagation**
- sync error handled inside then() as well
- less cluttered

```
function writeTempFile(fileName, ...args) {
  let cb = args.pop();

  let tempDir = path.join(os.tmpdir(), `${process.pid}-`);
  fs.mkdtemp(tempDir, (err, folder) => {
    if (err) cb(err);
    else {
      try {
        let tempFile = path.join(folder, fileName);
        fs.writeFile(tempFile, ...args, (err) => {
          if (err) cb(err);
          else {
            cb(null, tempFile);
          }
        });
      } catch (e) {
        cb(e);
      }
    }
  })
}
```

07-prednaska/samples/03-promises/01-fs.promises

01-callbacks-writeTempFile.js

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

function writeTempFile(fileName, ...args) {
  const cb = args.pop();

  const tempDir = path.join(os.tmpdir(),
    `${process.pid}-`);

  fs.mkdtemp(tempDir, (err, folder) => {
    if (err) cb(err);
    else {
      try {
        const tempFile = path.join(folder, fileName);
        fs.writeFile(tempFile, ...args, (err) => {
          if (err) cb(err);
          else {
            cb(null, tempFile);
          }
        });
      } catch (e) {
        cb(e);
      }
    }
  })
}

writeTempFile("test1.txt", "test", (err, path) => {
  if (err) throw err;
  console.log(path);
});

writeTempFile("test2.txt", "test", "FOO", (err, path) => {
  if (err) console.error("err:", err)
  else console.log(path);
});
```

02-promises-writeTempFile.js

```
const fs = require("fs").promises;
const os = require("os");
const path = require("path");

function writeTempFile(fileName, ...args) {

  const tempDir = path.join(os.tmpdir(),
    `${process.pid}-`);

  return fs.mkdtemp(tempDir)
    .then((folder) => {
      const tempFile = path.join(folder, fileName);
      return fs.writeFile(tempFile, ...args)
        .then(() => tempFile);
    })
}



writeTempFile("test1.txt", "test")
  .then((path) => console.log(path));


writeTempFile("test2.txt", "test", "FOO")
  .then((path) => console.log(path))
  .catch((err) => console.error("err:", err));
```

RECAP

Promises

Remember:

- promises have nicer syntax and more “linear code”
- but this is not (only) about syntax
- its about promises features eliminating “real callback hell problems”
  - called once
  - called async
  - async and also sync errors propagation
  - …..

Promises API

Promises states

- **pending**
- **settled**
  - **fulfilled**
  - **rejected**

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

const promise = fs.writeFile(tempFile);
```

Pending – operácia prebieha

Fulfilled – operácia zbehla úspešne

Rehected – operácia zlyhala

*promise.**then**(onFulfilled)*; - return values

- A function called if the Promise i**s fulfilled**.
- This function has one argument, the **fulfillment value**
- returns a Promise in the **pending status**
- gets called **asynchronously, (***when the stack is empty*)
- **onFulfilled môže byť sync aj async funkcia**

- **sync can return**
  - value
  - does not return
  - throws
- **async** can return Promise
  - pending
  - fulfilled
  - rejected

```
fs.readdir(`${__dirname}/..`,)
  .then((files) => {
    return //...
  }) // -> the promise returned by then

// a) returns a value,
// gets resolved with the returned value as its value;

// b) doesn't return anything,
// gets resolved with an undefined value

// c) throws an error,
// gets rejected with the thrown error as its value;

// d) returns an already resolved promise,
// gets resolved with that promise's value as its value;

// e) returns an already rejected promise,
// gets rejected with that promise's value as its value;

// f) returns another pending promise object,
// the resolution/rejection of the promise returned by then
// will be subsequent to the resolution/rejection
// of the promise returned by the handler.
// Also, the value of the promise returned by then
// will be the same as the value
// of the promise returned by the handler.
```

*promise.**then(onFulfilled)***; - chaining

- returns a Promise in the **pending status**
- so it can be **chained**
- you can have chain of **sync** and **async** functions mixed
- as *arrows* or as *named functions*

```
1  const fs = require("fs").promises;
2
3  ls()                                  // async -> Promise
4    .then(dirsOnly)                     // sync named
5    .then(dirs => dirs[0].name)         // sync arrow
6    .then(ls)                           // async -> Promise
7    .then(filesOnly)                    // sync
8    .then((files) =>                    // sync map
9      files.map(({ name }) => name)
10   )
11   .then(print)                        // sync undefined
12
13 function ls(d) {
14   return fs.readdir(`${__dirname}/..`, {
15     withFileTypes: true
16   });
17 }
18 function dirsOnly(files) {
19   return files.filter((f) => f.isDirectory());
20 }
21 function filesOnly(files) {
22   return files.filter((f) => f.isFile());
23 }
24 function print(data) {
25   console.log(data);
26 }
```

07-prednaska/samples/03-promises/02-promises-api/02-then-chaining.js

*promise.**then(onFulfilled)***; - multiple

- ***not chaining*** syntax, ***naming promises***
- **Multiple callbacks** may be added by calling then() several times.
- Each callback is executed one after another, **in the order** in which they were inserted

can have more declarative code

- instead of when.then.then (procedural), we can have
- when, then, when, then style – remember async.js and auto sample

07-prednaska/samples/03-promises/02-promises-api/03-then-multiple.js

```
1  const fs = require("fs").promises;
2
3  const whenFiles = fs.readdir(`${__dirname}/..`, {
4    withFileTypes: true
5  });
6
7  whenFiles.then(dirsOnly).then(processDirs);
8  whenFiles.then(filesOnly).then(processFiles);
9
10 function dirsOnly(files) {
11   return files.filter((f) => f.isDirectory());
12 }
13
14 function filesOnly(files) {
15   return files.filter((f) => f.isFile());
16 }
17
18 function processDirs(data) {
19   console.log(data);
20 }
21
22 function processFiles(data) {
23   console.log(data);
24 }
```

array of promises Promise.all

- [].map(async) -> [] of Promises
  - ls is async function
- how to wait for all promises to be resolved: Promise.all
- Then we have [] of values
- The **Promise.all(*iterable*)** method returns a single Promise that resolves when all of the promises in the *iterable* argument have resolved or when the *iterable* argument contains no promises. It rejects with the reason of the first promise that rejects.

```
1  const fs = require("fs").promises;
2
3  ls()
4    .then(dirsOnly)
5    .then(dirs => dirs.map(ls))          // [] of Promises of []s
6    .then(files => Promise.all(files))   // Promise of [] of []s
7    .then(files => [].concat(...files)) // [[],[],...]-> [.,.,.]
8    .then(filesOnly)
9    .then((files) =>
10     files.map(({ name }) => name)
11   )
12   .then(print) // sync undefined
13
14 function ls(d) {
15   return fs.readdir(`${__dirname}/..`, {
16     withFileTypes: true
17   });
18 }
19
20 function dirsOnly(files) {
21   return files.filter((f) => f.isDirectory());
22 }
23
24 function filesOnly(files) {
25   return files.filter((f) => f.isFile());
26 }
27
28 function print(data) {
29   console.log(data);
30 }
```

07-prednaska/samples/03-promises/02-promises-api/04-array-of-promises.js

*promise.**then**(onFulfilled, onRejected);*

- onRejected – 2nd parameter of then
- a function called if the Promise is **rejected**. This function has one argument, the **rejection reason**.
- If it is not a function, it is internally replaced with a “thrower" function (it throws an error it received as argument).

```
1  const fs = require("fs").promises;
2
3  fs.readdir('foo')
4    .then(
5      (files) => console.log("resolved:", files),
6      (err) => console.error("rejected:", err)
7    )
8
9  // rejected: Error: ENOENT: no such file or directory
10
11 // When a value is simply returned from within
12 // a then handler,
13 // it will effectively return
14
15 //   Promise.resolve(<
16 //      value returned
17 //      by whichever handler was called
18 //   >).
19
20 // A then call will return a rejected promise
21 // if the function throws an error or returns
22 // a rejected Promise.
```

*promise.**then**(onFulfilled, onRejected);*

- on fulfilled on rejected, same rules for return values apply for both
- We need to understand this to understand chaining and error handling

```
a()                                 // -> Promise

.then((d)=>{ },(e)=>{ })            // -> Promise

.then((d)=>{ },(e)=>{ })            // -> Promise

.then((d)=>{ },(e)=>{ })            // -> Promise

.catch((e)=>{ })                    // -> Promise

.finally(()={ })                    // -> Promise
```

```
// When a value is simply returned from within
// a then handler,
// it will effectively return

//   Promise.resolve(<
//      value returned
//      by whichever handler was called
//   >).

// a) returns a value,
// gets resolved with the returned value as its value;

// b) doesn't return anything,
// gets resolved with an undefined value

// c) throws an error,
// gets rejected with the thrown error as its value;

// d) returns an already resolved promise,
// gets resolved with that promise's value as its value;

// e) returns an already rejected promise,
// gets rejected with that promise's value as its value;

// f) returns another pending promise object,
// the resolution/rejection of the promise returned by then
// will be subsequent to the resolution/rejection
// of the promise returned by the handler.
// Also, the value of the promise returned by then
// will be the same as the value
// of the promise returned by the handler.
```

then() returns a Promise

A) Zápis ako chain

```
let pF =
  asyncFunction()
  .then((v)=>{},(e)=>{})
  .then((v)=>{},(e)=>{})
  .then((v)=>{},(e)=>{})
  .then((v)=>{},(e)=>{})
```

Pending

Pending

Pending

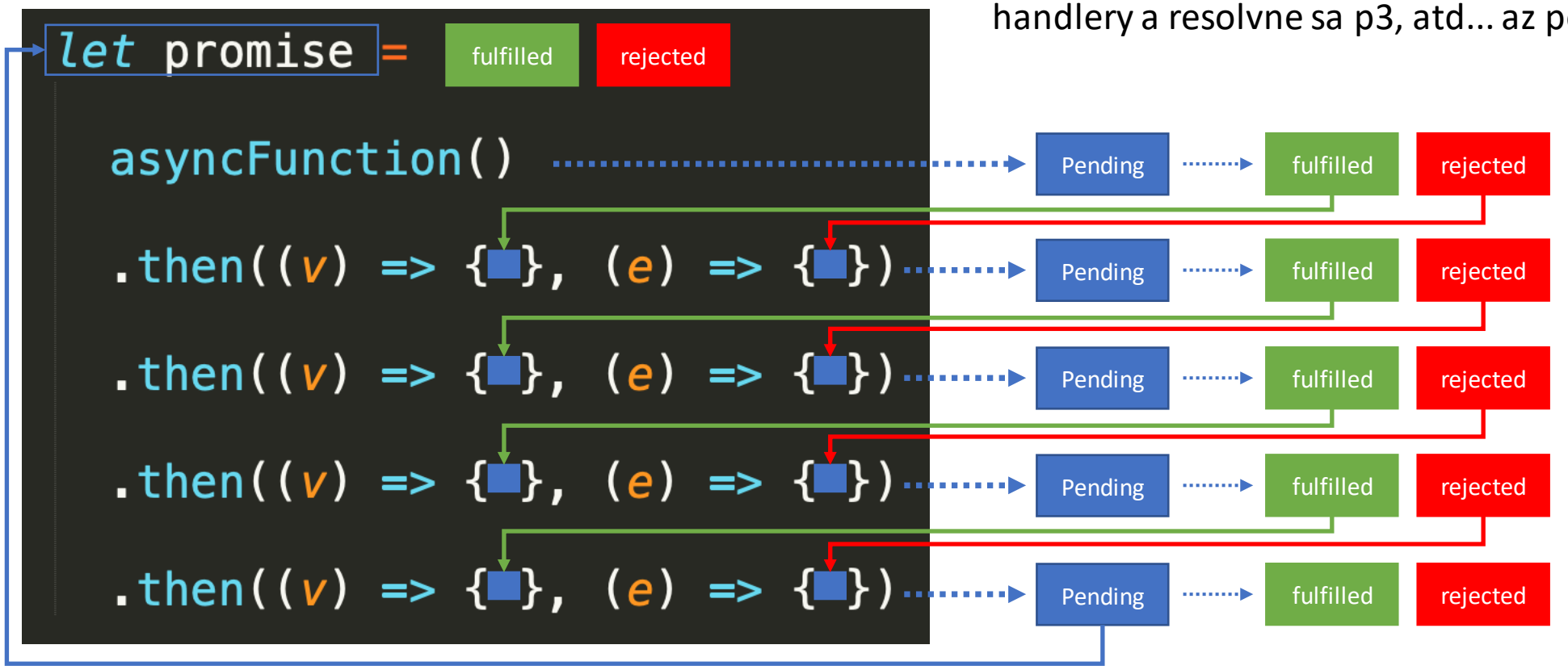
Pending

B) Zápis ako sekvencia

```
let p1 = asyncFunction()
let p2 = p1.then(v=>{},e=>{})
let p3 = p2.then(v=>{},e=>{})
let p4 = p3.then(v=>{},e=>{})
let pF = p4.then(v=>{},e=>{})
```

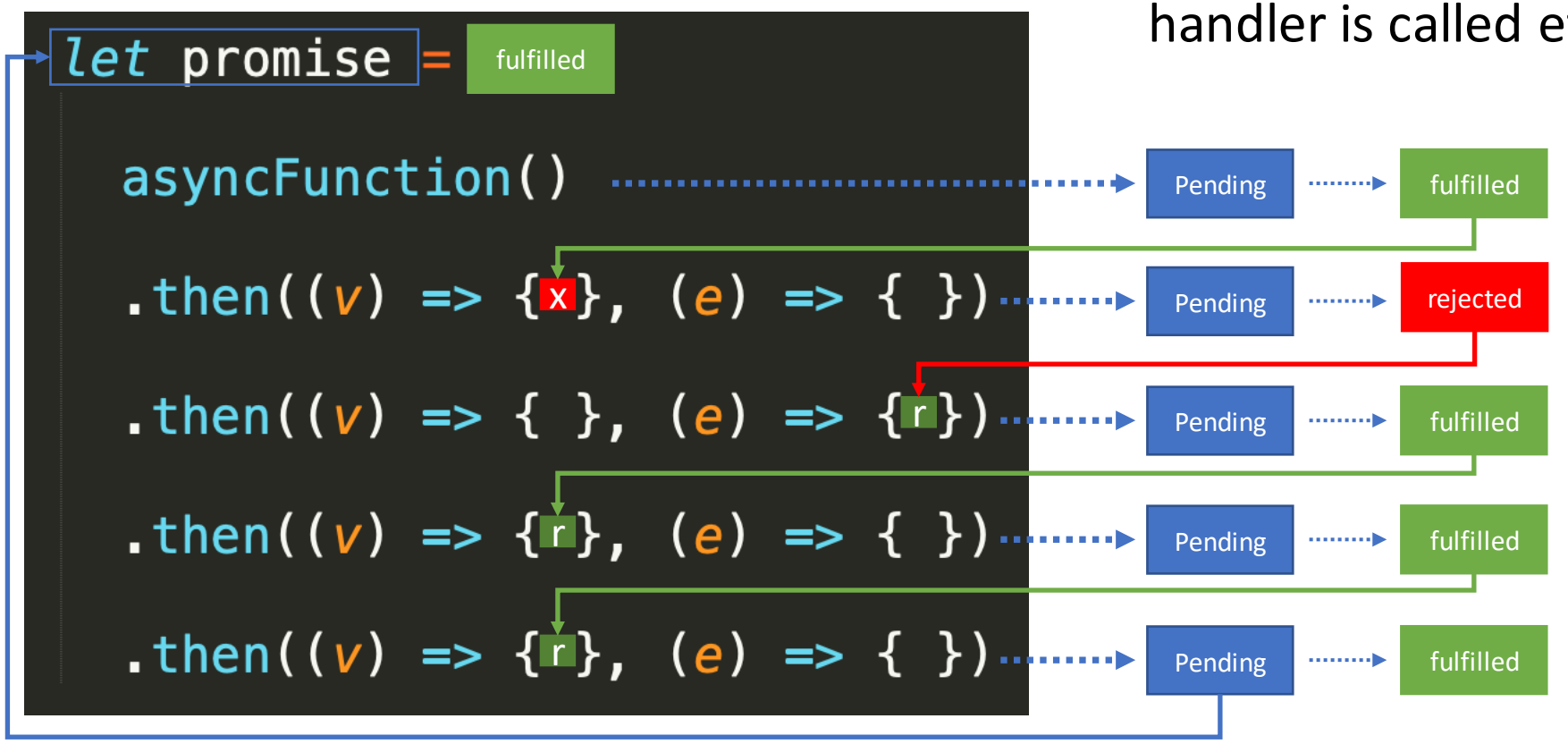
Chaining scenarios

Vznikne nam 5 pending promisov, vsetky su pending dobehne async, prva promise je fulfilled alebo rejected zavolaju sa prislusne then handlery a podla toho sa resolvne p2, podla toho ako sa resolvla p2 sa zase zavolaju prislusne handlery a resolvne sa p3, atd... az po finalny resolve pF



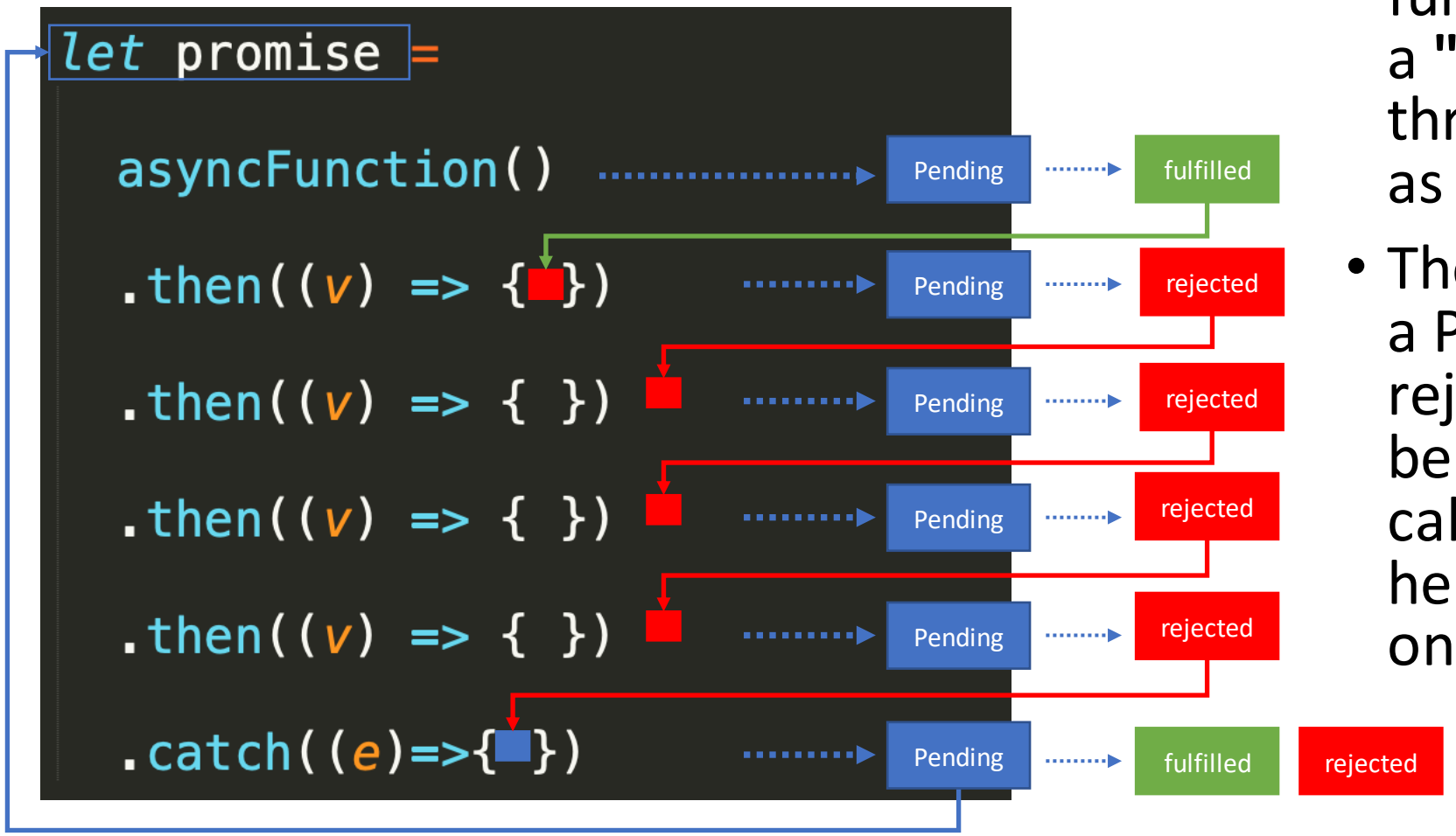
Chaining scenarios

Appropriate handler is called, once the promise is ***settled*** *(fullifiled or rejected).* Based on the result value of handler, next handler is called etc...



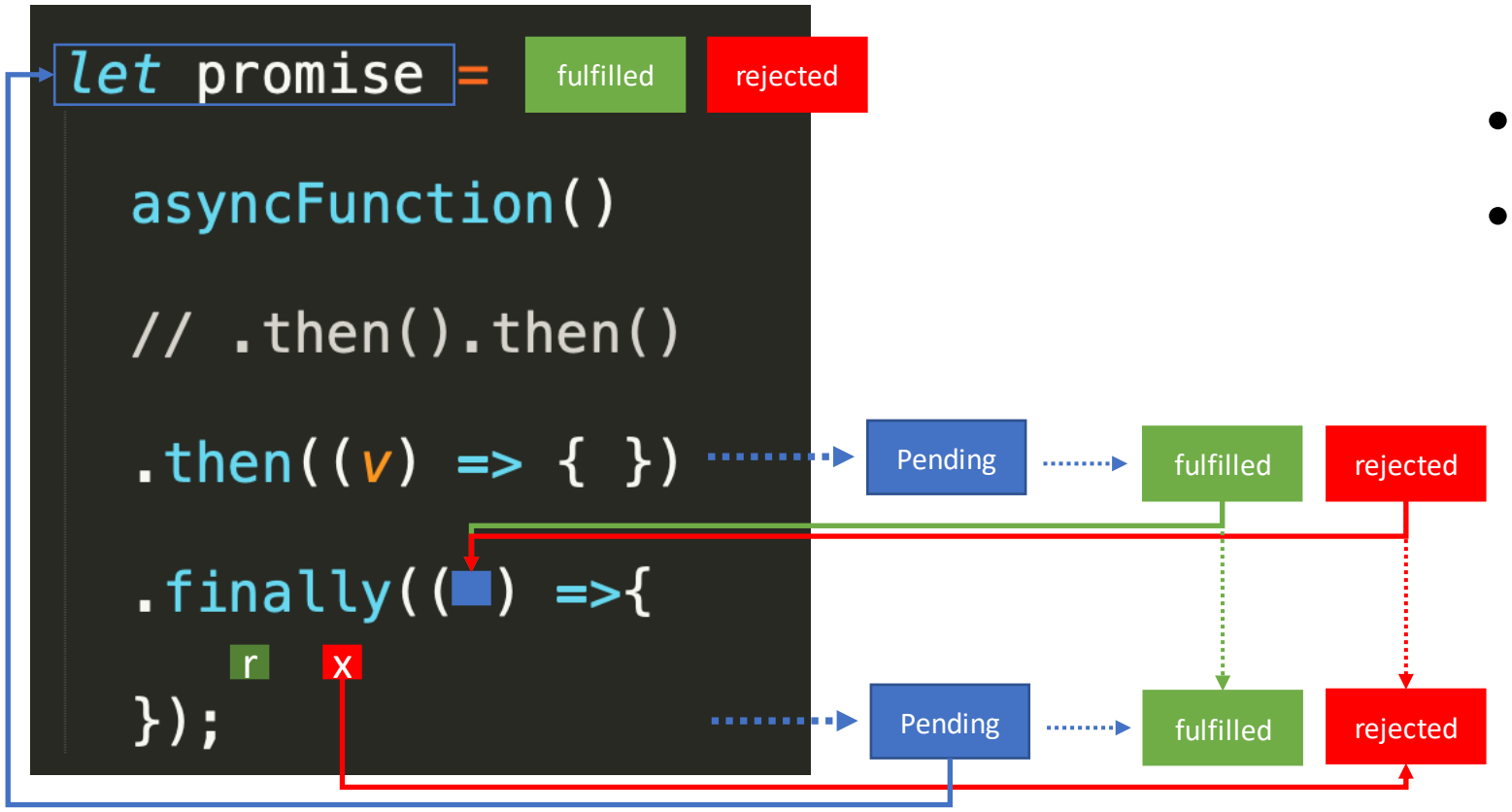
- x If the handler throws or returns rejected promise, onRejected handler is called,
- r If the handler returns value or nothing, or fulfilled promise, the onFulfilled handler is called

then and catch



- then(onRejected) - If not a function, it is replaced with a **"Thrower" function** (it throws an error it received as argument).
- The catch() method returns a Promise and deals with rejected cases only. It behaves the same as calling Promise.prototype.then(undefined, onRejected)

then, and, finally



- Handling ***settled*** promise (rejected or fulfilled)
- It receives ***no argument***
- It returns Promise, which is resolved or rejected *„based on previous promise“*
  - unless finally throws, then the promise is rejected with thrown error
  - return from finally is ignored

Then, catch, finally

```
let promise = a() r x

  .then((d) => {r x}, (e) => {r x})

  .catch((err) => {r x})

  .finally(() => {r x});

promise.then(
  (v) => console.log("v", v),
  (e) => console.error("e", e)
);
```

?

- Mali by ste rutinne dokázať riešiť takéto matice situácií: čo ak popadá a(), čo ak onFulfilled vráti rejected promise, ak onRejected vráti undefined, ak catch nerethrowne exception atď....

Mixing sync and async code

```
1  const fs = require("fs").promises;
2  const path = require("path");
3
4  ls(".") X                        // async -> Promise
5    .then(dirsOnly)                // sync named
6    .then(dirs => dirs[0].name)    // sync arrow
7    .then(ls) X                    // async -> Promise
8    .then(filesOnly)               // sync
9    .then((files) =>               // sync map
10     files.map(({ name }) => name)
11   )
12   .then(print) // sync undefined
13
14 function ls(d) {
15   let dir = path.join(__dirname, "..", d); // sync X
16   return fs.readdir(dir, {                 // async
17     withFileTypes: true
18   });
19 }
```

- Async function contains mix of sync and async code, problem:
  - What if line 15 (sync) fails ?
  - For line 7 when ls is callback this is ok and will be wrapped as Rejected promise
  - For line 4 it is a problem and program fill fail with unexpected exception and not with rejected promise

- One of the solutions:
  - Enclose the sync code in callback internally as well
  - To do this, enclose mix [18,19,20] in handler [17]
  - To be able to do .then() you need a promise, created by Promise.resolve [15] with undefined value

```
14 function ls(d) {
15   return Promise.resolve()
16     // enclose sync code in handler
17     .then(() => {
18       let dir = path.join(__dirname, "..", d);
19       return fs.readdir(dir, { // async
20         withFileTypes: true
21       })
22     });
23 }
```

Mixing sync and async code

- keď kódujete async funkcie, nemali by ste miešať sync a async kód, najmä ak async kód môže hodiť sync exception. Takáto funkcia je problém, ak nie je použitá v rámci then.
- Buď budete kódovať vašu funkciu ako:
  A. **Promise.resolve().then({sync/async mix})**
  B. alebo try{sync} catch{ Promise.reject(ex)}, async
  C. ....
- ... alebo budete každý flow začínať **Promise.resolve()**.then().then().... aby sa problémová funkcia vyskytla „obalená“ v then

Mixing sync and async code

01-bad.js

```
10 // lsr(".");     //-> prints
11 // lsr("FOO");   //-> prints fail
12 lsr(999);        //-> throws TypeError //!!!
13
14 function lsr(dir) {
15   ls(dir) // !!!
16     .then(dirsOnly)
17     .then(dirs => dirs[0].name)
18     .then(ls)
19     .then(filesOnly)
20     .then((files) =>
21       files.map(({ name }) => name)
22     )
23     .then(print)
24     .catch((err) =>
25       console.error("fail", err)
26     );
27 }
28
29 function ls(d) { //!!!
30   const dir = path.join(__dirname, "..", d);  //sync
31   return fs.readdir(dir, opts)                //async
32 }
33
34
35
```

01-good.js

```
10 // lsr(".");     //-> prints
11 // lsr("FOO");   //-> prints fail
12 lsr(999);        //-> prints fail TypeError
13
14 function lsr(dir) {
15   ls(dir) // !!!
16     .then(dirsOnly)
17     .then(dirs => dirs[0].name)
18     .then(ls)
19     .then(filesOnly)
20     .then((files) =>
21       files.map(({ name }) => name)
22     )
23     .then(print)
24     .catch((err) =>
25       console.error("fail", err)
26     );
27 }
28
29 function ls(d) {
30   return Promise.resolve() //!!!
31     .then(() => {
32       const dir = path.join(__dirname, "..", d);
33       return fs.readdir(dir, opts);
34     })
35 }
```

Mixing sync and async code –alebo sync výnimky z async kódu

- Otázka znie kedy je sync výnimka by design (???) a kedy nie, alebo prečo niekto kódol writeFile tak, že hádže sync výnimku, keď je zlý encoding a async error keď file neexistuje ? Čo je ozaj výnimka (async error) a čo je bug (zlý encoding)

07-prednaska/samples/03-promises/04-mixing-sync-async/02-writeFile.js

```
1  // ako sa sprava promisifikovane API fs
2  // bude hadzat sync exceptions ako fs callback api ?
3  // alebo civilizovane vracat rejected promise ?
4
5  const data = "test";
6  const { promisify } = require("util");
7
8  const fsC = require("fs");
9  const fsP = require("fs").promises;
10
11
12 // fsC hadze exception pri zlom encodingu
13 try {
14   fsC.writeFile("./test.txt", data, "FOOBAR", () => {})
15 } catch (ex) {
16   console.error("throws");
17 }
18
19 // ako sa sprava fs.promise.writeFile ?
20 // hadze vynimku alebo rejected promise ?
21 fsP.writeFile("./test.txt", data, "FOOBAR")
22   .catch((err) => console.error("rejects"));
23
24 // ako sa sprava promisifikovana verzia
25 // callback funkcie ?
26 promisify(fsC.writeFile)("./test.txt", data, "FOOBAR")
27     .catch((err) => console.error("rejects"));
```

Patterns with promises

Promise.all

- The Promise.all (iterable) method **returns a single Promise**
- **That resolves** when all of the promises in the iterable argument have resolved or when the iterable argument contains no promises.
- **It rejects** with the reason of the first promise that rejects.

```
const a = (cb) => new Promise(res => { setTimeout(res, 3000, "a") });
const b = (cb) => new Promise(res => { setTimeout(res, 2000, "b") });
const c = (cb) => new Promise(res => { setTimeout(res, 8000, "c") });

Promise.all([
    a(),
    b(),
    c()
  ])
  .then((data) => {
    // [ 'a', 'b', 'c' ]
    console.log(data);
  })
  .catch((err) => {
    console.error(err);
  });
```

Promise.race

- The **Promise.race(iterable)** method returns a promise
- that **resolves** or **rejects** as soon as **one of the promises** in the iterable **resolves or rejects**, with the value or reason from that promise.

```
const a = (cb) => new Promise(res => { setTimeout(res, 3000, "a") });
const b = (cb) => new Promise(res => { setTimeout(res, 2000, "b") });
const c = (cb) => new Promise(res => { setTimeout(res, 8000, "c") });

Promise.race([
    a(),
    b(),
    c()
  ])
  .then((data) => {
    // b
    console.log(data);
  })
  .catch((err) => {
    console.error(err);
  });
```

waterfall

- syntakticky zapis pomocoy then.then.then je vlastne waterfall
- ale co ked je zoznam taskov dynamicky ? v nejakej premennej ?

waterfall

- This pattern is not part of the promises API, it can be implemented by chaining with reduce() or by other techniques

```
1  // let's have async functions
2  const a = () => new Promise(res => { setTimeout(res, 3000, "a") });
3  const b = (d) => new Promise(res => { setTimeout(res, 2000, d + "b") });
4  const c = (d) => new Promise(res => { setTimeout(res, 8000, d + "c") });
5
6  // waterfall must be implemented
7  const waterfall = (promises) =>
8    // TODO: review
9    promises.reduce((p, f) => p.then(f), Promise.resolve());
10
11 waterfall([
12     a,
13     b,
14     c
15   ])
16   .then((data) => {
17     // b
18     console.log(data);
19   })
20   .catch((err) => {
21     console.error(err);
22   });
```

Other patterns and custom libraries

A. **async.js** + asyncify
B. **bluebird.js** - Promises in Node.js 10 are significantly faster than before. Bluebird still includes a lot of features like
   - cancellation**, iteration methods** and warnings that native promises don‘t.
   - If you are using Bluebird for performance rather than for those - please consider giving native promises a shot and running the benchmarks yourself.
   - https://github.com/petkaantonov/bluebird/issues/1434
C. ....
D. ....
E. ES2017 **async Functions, await and generators**

RECAP

Prínos a výhody promise API

Na prvý pohľad:

- Krajšia syntax a lineárny kód
- Error handling kód “separovaný mimo” a nie pomiešaný v kóde

Podstatné:

- funkcia v then() bude zavolaná vždy asynchrónne
- nebude v then() nebude nikdy zavolaná viackrát
- môžeme mať niekoľko then() nad jedným now, s garantovaným poradím vykonania
- ľahšie kombinovanie async a sync funkcií

Anti patterns

- http://bluebirdjs.com/docs/anti-patterns.html
- http://2ality.com/2016/03/promise-rejections-vs-exceptions.html
- …

ES2017 Async Functions

Simplified introduction to new ES 2017 syntax

RECAP: *async-function definition*

"async-function": Minule sme si zadefinovali async funkciu pre naše potreby takto:

Async function

- Pod **async funkciou** budeme zjednodušenie rozumieť jednu z konštrukcií, ktorá nám umožní zapísať, že niečo sa deje NOW a potom LATER. V zásade existujú 3 možnosti:

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

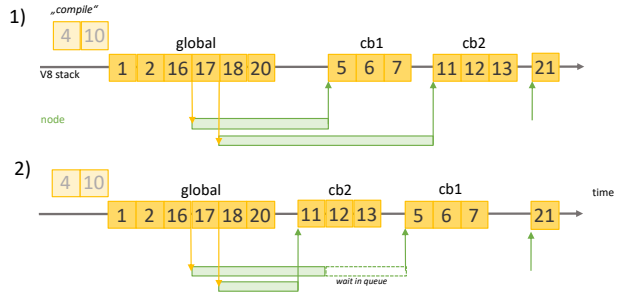
```
function a(params,callback) {
  //...
  callback(null, data);
  callback(err, data);
}

function b(params) {
  //...
  return Promise.resolve(data);
  return Promise.reject(err);
}

async function c(params) {
  //...
  return data; //impl. promise
  throw err;
}
```

run-to-completion - a zadefinovali funkciu ako zaklad atomicity

Async and Concurrency

```
1  let a = 1;
2  let b = 2;
3
4  function cb1() {
5    a++;
6    b = b * a;
7    a = b + 3;
8  }
9
10 function cb2() {
11   b--;
12   a = 8 + b;
13   b = a * 2;
14 }
15
16 const https = require("https");
17 https.get("https://nodejs.org/en/", cb1);
18 https.get("https://nodejs.org/en/", cb2);
19
20 process.on("beforeExit", () => {
21   console.log(a, b);
22 });
```

/samples/concurrency.js



- **async**: some code happends ***now*** [1,2,...] and some ***later*** [5,5,6], [11,12,13],[21]
- **concurrent**: callbacks happen in certain timeframe but not in predictable and *controllable* order
- **run-to-completion**: function is *„atomic"* unit and must execute as whole, there is no chance for (11,5,12,... to happen)

```
$ node ./concurrency.js
11 22
$ node ./concurrency.js
11 22
$ node ./concurrency.js
183 180
$ node ./concurrency.js
11 22
```

What are ES AsyncFunctions

- The **async function declaration** defines an asynchronous function, which returns an AsyncFunction object.
- An asynchronous function is a function which operates asynchronously via the event loop, using an **implicit Promise to return its result.**
- But the **syntax and structure of your code using async functions is much more like using standard synchronous functions.**
- Async/await are built on the concept of promises so **you must understand promises well**

```
// Async function declaration
async function af() {

}
// Async function expression
const af = async function () {

};
// Async arrow function
const af = async () => {

};

// Async method definition
let obj = {
    async af() {

    }
}
```

always returns promise

- af() call - returns a Promise which will be resolved with the value returned by the async function, or rejected with an uncaught exception thrown from within the async function.
- **Returning a non-Promise** value fulfills p with that value. [a,b]
- **Returning a Promise** means that p now mirrors the state of that Promise. [c,d]

```
1  // A returning value
2  async function a() {
3    return 10;
4  }
5  a() // promise resolved with value
6    .then(console.log, console.error);
7
8  // B throwing
9  async function b() {
10   throw new Error("boom");
11 }
12 b() // promise rejected with thrown
13   .then(console.log, console.error);
14
15 // C returning promise that will be resolved
16 async function c() {
17   return Promise.resolve(10);
18 }
19 c() // promise resolved with ret promise value
20   .then(console.log, console.error);
21
22 // D returning promise that will be rejected
23 async function d() {
24   return Promise.reject(new Error("boom"));
25 }
26 d() // promise rehected with ret promise err
27   .then(console.log, console.error);
28
```

await

- Načo nám je AsyncFunction keď aj tak vracia promise ?
- **syntax and structure of your code using async functions is much more like using standard synchronous functions**
- **Aby sme miesto tohto, mohli písať toto:**

```
a()
  .then((r) => console.log(r));
```

```
let r = await a();
console.log(r);
```

await

- The await operator is used **to wait for a Promise**.
- The await expression causes
  - **async function execution to pause** until a Promise is ***settled***, (fulfilled or rejected),
  - and to **resume execution of the async function** after fulfillment.
- When resumed, the **value of the await expression** is that of the fulfilled Promise.
- If the Promise is **rejected, the await expression throws the rejected value**.
- It can only be used inside an async function.
- *Paused and resumed* vs. run-to-completion

Using async await (writeTempFile sample)

writeTempFile.js — promises

```
5  function writeTempFile(fileName, ...args) {
6
7    let tempDir = path.join(os.tmpdir(),
8      `${process.pid}-`);
9    return fs.mkdtemp(tempDir)
10     .then((folder) => {
11       let tempFile = path.join(folder, fileName);
12       return fs.writeFile(tempFile, ...args)
13         .then(() => tempFile);
14     })
15 }
```

writeTempFile.js — es-async

```
5  async function writeTempFile(fileName, ...args) {
6
7    let tempDir = path.join(os.tmpdir(),
8      `${process.pid}-`);
9    let folder = await fs.mkdtemp(tempDir);
10   let tempFile = path.join(folder, fileName);
11   await fs.writeFile(tempFile, ...args)
12   return tempFile;
13
14 }
15
```

```
17 writeTempFile("test.txt", "test")
18   .then((path) => console.log(path));
19
20
21 writeTempFile("test.txt", "test", "FOO")
22   .then((path) => console.log(path))
23   .catch((err) => console.error("err:", err));
```

Client code can be the same, async function can be used as any normal function returning promise

- Sequential code
- Call of async function has own syntax (await a())
- Return plain value, implicit promises

Await can be used only in async functions and basic error handling

writeTempFile.js — es-async — FIIT

writeTempFile.js — promises

```
}


writeTempFile("test.txt", "test")
  .then((path) => console.log(path));


writeTempFile("test.txt", "test", "FOO")
  .then((path) => console.log(path))
  .catch((err) => console.error("err:", err));

```

writeTempFile.js — es-async

```


(async () => {


  let path = await writeTempFile("test.txt", "test");
  console.log(path);

  try {
    let path2 = await writeTempFile("test.txt", "test", "FOO");
    console.log(path2)
  } catch (err) {
    console.error("err:", err);
  }

})();
```

- Errors from await are thrown so you can use try catch
- Writing async code in global block is problematic, must enclose in async function

Async functions – details and inner working

- http://exploringjs.com/es2016-es2017/ch_async-functions.html
- https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch4.md

Mixing sync and async code

- https://www.youtube.com/watch?v=bfxglBVSNDI

JS quiz: async function execution order

```
let x = 0;

async function test() {
  x += await 2;
  console.log(x);
}

test();
x += 1;
console.log(x);
```

new asyc vs. Promises – subjektívne úvahy na zaver

- async await štýl sa vracia „*nazad*“ od funkcionálneho štýlu k imperatívnemu, štruktúrovanému programovaniu
- Čitateľnosť sa zvyšuje nie na úrovni promise vs. await, ale hlavne na úrovni funkcionálne vs. štrukturované programovanie
- Inak povedané ak ste vedeli čítať a písať z použitím FP konštrukcií , stačili vám promises
- Ak ste to nedokázali a všetko píšete “*cez premenne a cykly*“ tak potrebujete nutne **async/await** a **async iterators**
- **……**
- Is this JavaScript for „more dummies“ ?

ES2017 Async Functions

Simplified introduction to new ES 2017 syntax

Async function - definitions

- Pod **async funkciou** budeme zjednodušenie rozumieť jednu z konštrukcií, ktorá nám umožní zapísať, že niečo sa deje **NOW** a potom **LATER**. V zásade existujú 3 možnosti ako je async funkcia implementovaná:

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

```
function task1(params,callback) {
  //...
  callback(null, data);
  callback(err, data);
}

function task2(params) {
  //...
  return Promise.resolve(data);
  return Promise.reject(err);
}

async function task3(params) {
  //...
  return data; //impl. promise
  throw err;
}
```

Async function - usage

- Podľa toho potom rôzne zapisujeme „**later**“

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

Niektoré knižnice sú napísané štýlom A) iné B) iné podporujú A/B/C ..... Niektoré runtimes podporujú len A, niektoré A,B,C

```
// how and where continuation
// is written

task1(args, (err, cb) => {
  if (err) {
    //...
  } else {
    //...
  }
});

task2(args)
  .then(data => {
    //...
  })
  .catch(err => {
    // ...
  });


try {
  let data = await task3(args);
  //...
} catch (err) {
  //...
}
```

1995-2020

2010
2015-2020

2017-2020

What are ES AsyncFunctions

- The **async function declaration** defines an asynchronous function, which returns an AsyncFunction object.
- An asynchronous function is a function which operates asynchronously via the event loop, using an **implicit Promise to return its result.**
- But the **syntax and structure of your code using async functions is much more like using standard synchronous functions.**
- Async/await are built on the concept of promises so **you must understand promises well**

```
// Async function declaration
async function af() {

}
// Async function expression
const af = async function () {

};
// Async arrow function
const af = async () => {

};

// Async method definition
let obj = {
    async af() {

    }
}
```

always returns promise

- af() call - returns a Promise which will be resolved with the value returned by the async function, or rejected with an uncaught exception thrown from within the async function.
- **Returning a non-Promise value** - fulfills p with that value. [a,b]
- **Returning a Promise** means that p now mirrors the state of that Promise. [c,d]

```
1  // A returning value
2  async function a() {
3    return 10;
4  }
5  a() // promise resolved with value
6    .then(console.log, console.error);
7
8  // B throwing
9  async function b() {
10   throw new Error("boom");
11 }
12 b() // promise rejected with thrown
13   .then(console.log, console.error);
14
15 // C returning promise that will be resolved
16 async function c() {
17   return Promise.resolve(10);
18 }
19 c() // promise resolved with ret promise value
20   .then(console.log, console.error);
21
22 // D returning promise that will be rejected
23 async function d() {
24   return Promise.reject(new Error("boom"));
25 }
26 d() // promise rehected with ret promise err
27   .then(console.log, console.error);
28
```

await

- Načo nám je AsyncFunction keď aj tak vracia promise ?
- **syntax and structure of your code using async functions is much more like using standard synchronous functions**
- **Aby sme miesto tohto, mohli písať toto:**

```
a()
  .then((r) => console.log(r));
```

```
let r = await a();
console.log(r);
```

await

- The await operator is used **to wait for a Promise**.
- The await expression causes
  - **async function execution to pause** until a Promise is ***settled***, (fulfilled or rejected),
  - and to **resume execution of the async function** after fulfillment.
- When resumed, the **value of the await expression** is that of the fulfilled Promise.
- If the Promise is **rejected, the await expression throws the rejected value**.
- It can only be used inside an async function.
- *Paused and resumed* vs. run-to-completion

Promises compared to async await (impl)

```
5  function writeTempFile(fileName, ...args) {
6
7    let tempDir = path.join(os.tmpdir(),
8      `${process.pid}-`);
9    return fs.mkdtemp(tempDir)
10     .then((folder) => {
11       let tempFile = path.join(folder, fileName);
12       return fs.writeFile(tempFile, ...args)
13         .then(() => tempFile);
14     })
15 }
```

```
5  async function writeTempFile(fileName, ...args) {
6
7    let tempDir = path.join(os.tmpdir(),
8      `${process.pid}-`);
9    let folder = await fs.mkdtemp(tempDir);
10   let tempFile = path.join(folder, fileName);
11   await fs.writeFile(tempFile, ...args)
12   return tempFile;
13
14 }
15
```

- Sequential code
- Call of async function has own syntax (await a())
- Return plain value, implicit promises

Promises compared to async await (usage)

```
writeTempFile("test.txt", "test")
  .then((path) => console.log(path));


writeTempFile("test.txt", "test", "FOO")
  .then((path) => console.log(path))
  .catch((err) => console.error("err:", err));
```

```
(async () => {


  const path = await writeTempFile("test.txt", "test");
  console.log(path);

  try {
    const path2 = await writeTempFile("test.txt", "test", "FOO");
    console.log(path2)
  } catch (err) {
    console.error("err:", err);
  }

})();
```

- Errors from await are thrown so you can use try catch
- Writing async code in global block is problematic, must enclose in async function
- ;-(A sme nazad v imperativnom programovani, premenne a bloky miesto funkcii

- Nezavisle do toho ako je napisane API (promises, async) mozeme klienta napisat a) alebo b) sposobom

Datails and inner working

Toto by sme nestihli za celé dve prednášky, takže samoštúdium:

Ďalšie detaily k async funkciám ktoré na prednáške boli/neboli

- https://exploringjs.com/es2016-es2017/ch_async-functions.html

Ako vznikli async / await (2017) z generatorov (2015) a promises (2015) a ako vlastne funguju na ioch základoch:

- https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/sync-async/ch4.md

asyc/await – bad parts
subjektívne úvahy na zaver

- async await štýl sa vracia „*nazad*“ od funkcionálneho štýlu k imperatívnemu, štruktúrovanému programovaniu
- Čitateľnosť sa zvyšuje nie na úrovni promise vs. await, ale hlavne na úrovni funkcionálne vs. štrukturované programovanie
- Inak povedané ak ste vedeli čítať a písať z použitím FP konštrukcií , stačili vám promises
- Ak ste to nedokázali a všetko píšete “*cez premenne a cykly*“ tak potrebujete nutne **async/await** a **async iterators**

- Is this JavaScript for „more dummies“ ?

Anti príklady použitia await

```
// bezny koder je schopny
// toto kludne napisat
// lebo mu dal niekto "await"
// co je zle na tom kode ?

async function calc() {

  let r1 = await data1();
  let r2 = await data2();
  let r3 = await data3();

  return r1 + r2 + r3;
}

(async () => {

  const r = await calc();
  console.log(r);

})();
```

Anti príklady použitia await

- Nejde o vymyslený scenár, takýchto a podobných „pseodo synchrónnych“ a serializovaných kódov je veľa, lebo sa dajú ľahko napísať
- aj niekým, kto z async programingu videl akurát await a niekto múdry mu povedal „že má dať pred každé async volanie await a „thread“ sa mu suspendne a bude mat pekný kód“
- [Viď: https://eslint.org/docs/rules/no-await-in-loop](https://eslint.org/docs/rules/no-await-in-loop)

```
// bezny koder je schopny
// toto kludne napisat
// lebo mu dal niekto "await"
// co je zle na tom kode ?

async function calc(nums) {

  const out = [];
  for (let i = 0; i < nums.length; i++) {
    const c = await asyncMath(nums[i]) //!!!
    out.push(c);
  }
  const sum = out.reduce((sum, n) => sum += n);
  return sum;
}

(async () => {

  const sum = await calc([1, 2, 3])
  console.log(sum);

})();
```

Calling async function as sync

- https://www.youtube.com/watch?v=bfxgIBVSNDI
- **5.2.3 Async functions are started synchronously, settled asynchronously** https://exploringjs.com/es2016-es2017/ch_async-functions.html

03a-async-execution-order.js

```
1
2
3  let x = 0;
4
5  async function test() {
6    x = x + await 2;
7    console.log(x);
8  }
9
10 test();
11 x = x + 1;
12 console.log(x);
13
14
```

03b-async-execution-order.js

```
1  (async () => {
2
3    let x = 0;
4
5    async function test() {
6      x = x + await 2;
7      console.log(x);
8    }
9
10   await test();
11   x = x + 1;
12   console.log(x);
13
14 })();
```

07-prednaska/samples/04-es2017-async/03a-async-execution-order.js
07-prednaska/samples/04-es2017-async/03b-async-execution-order.js

asyc/await – good parts

- Už zo signatúry je jasné, že je async, nemusíte lúštiť kód a doksy
- Vždy vráti promise, nestane sa Vám, že raz vracia promise raz value, ako je pomerne bežné u veľa starších promise base APIčiek a knižníc
- ….

asyc/await – good parts

- Synchrónny throw je zabalený ako reject promisu. Spomente si na príklady z ls()
- Prvý je problém
- Druhý je riešenie s Promise API
- Tretie je ako async funkcia

```
function ls(d) { //!!!
  const dir = path.join(__dirname, "..", d);  //sync
  return fs.readdir(dir, opts)                //async
}
```

```
function ls(d) {
  return Promise.resolve() //!!!
    .then(() => {
      const dir = path.join(__dirname, "..", d);
      return fs.readdir(dir, opts);
    })
}
```

```
async function ls(d) { //!!! async solution
  const dir = path.join(__dirname, "..", d);
  return fs.readdir(dir, opts);
}
```

07-prednaska/samples/03-promises/04-mixing-sync-async