KINGSLAND UNIVERSITY

Asynchronous Programming and Promise

Promises. Async / Await.

Table of Contents

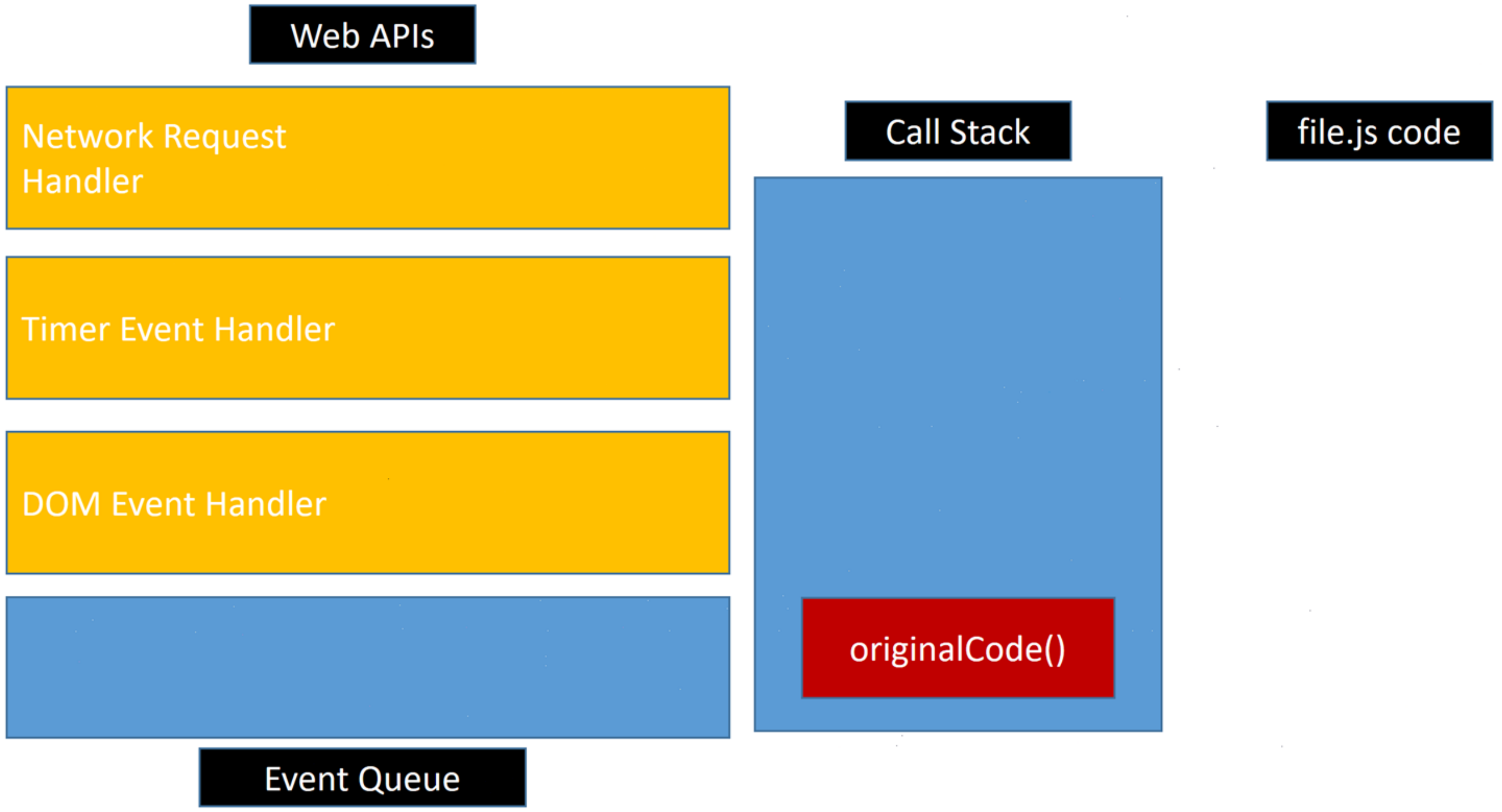
Have a Question?

#js-advanced

Synchronous vs Asynchronous

Asynchronous Programming

Event Loop

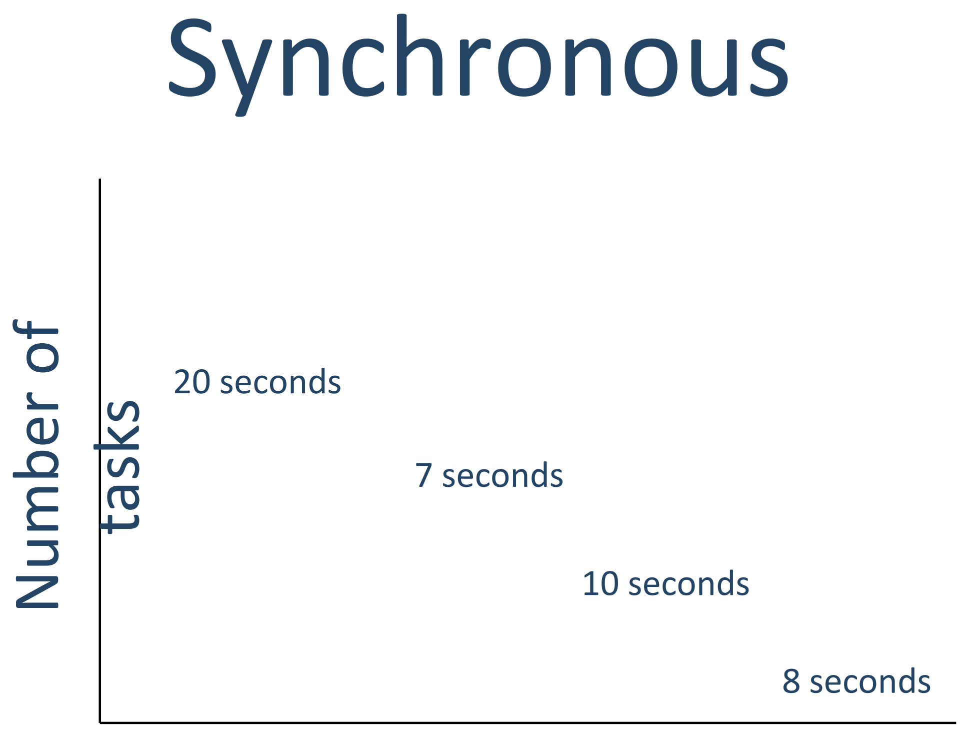

Web APIs
Network Request Handler
Timer Event Handler
DOM Event Handler
Event Queue
Call Stack
originalCode()
file.js code

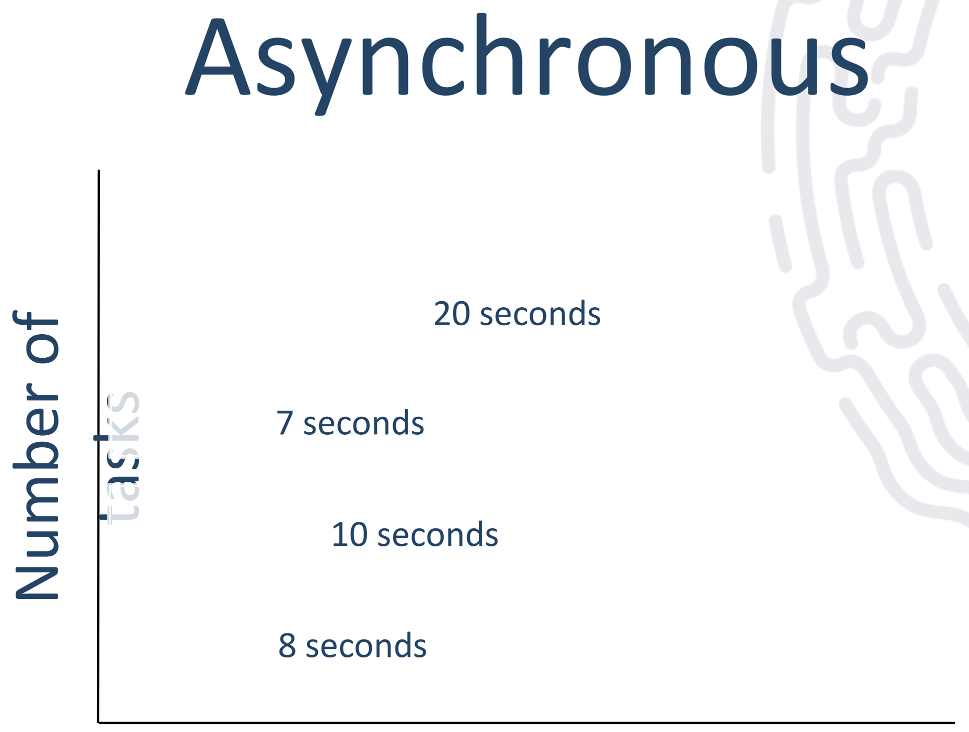
Asynchronous Programming in JS

- Not the same thing as **concurrent** or **multi-threaded**
- There can be **asynchronous code**, but it is **generally single-threaded**
- Structured using **callback functions**
- In current versions of JS there are:
  - **Callbacks**
  - **Promises**
  - **Async Functions**

Asynchronous Programming

- Runs several tasks (pieces of code) in parallel, **at the same time**





Asynchronous Programming – Example

- The following commands will be executed as follows:

```
console.log("Hello.");
```

```
setTimeout(function() {
  console.log("Goodbye!");
}, 2000);
```

```
console.log("Hello again!");
```

```
// Hello.
```

```
// Hello again!
```

```
// Goodbye!
```

Objects Holding Asynchronous Operations

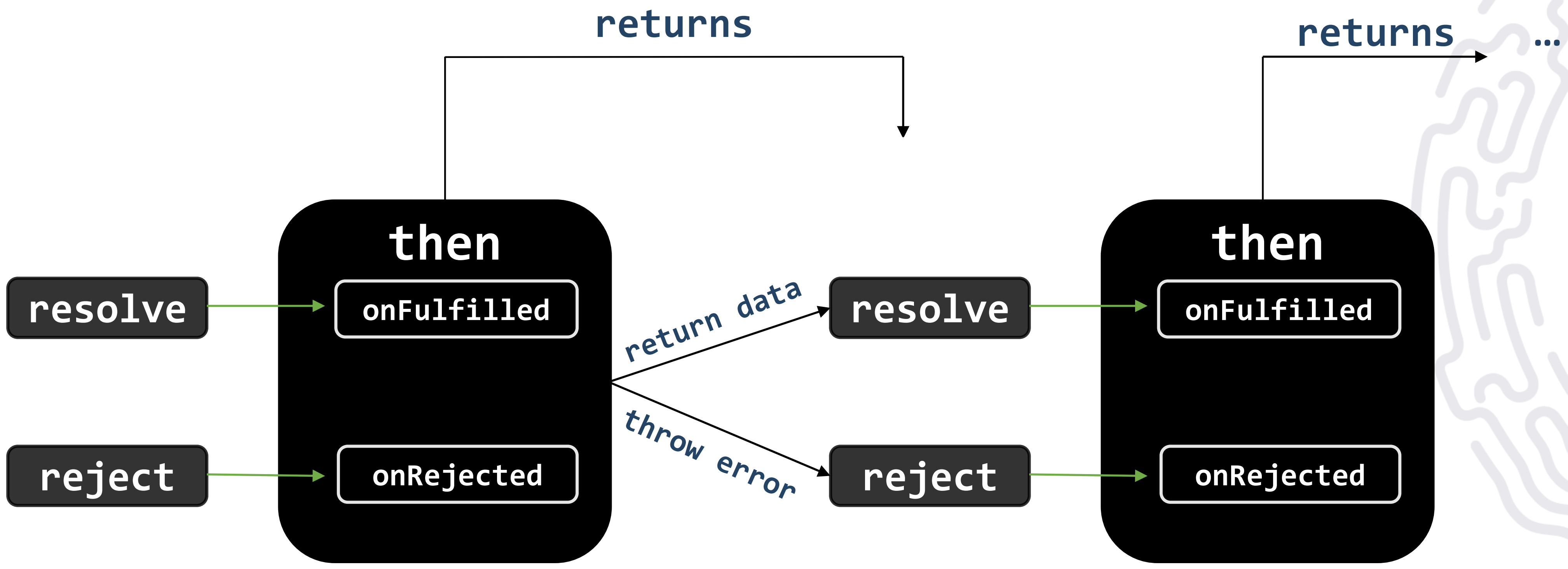
Promises

What is a Promise?

- A promise is an **asynchronous action** that **may complete** at some point and **produce a value**
- States:
  - **Pending** - operation still running (unfinished)
  - **Fulfilled** - operation finished (the result is available)
  - **Failed** - operation failed (an error is present)
- Promises use the **Promise** object

```
new Promise(executor);
```

What is a Promise?



Promise Methods

- **`Promise.reject`**(reason)
  - Returns an **object** that is **rejected** with the given **reason**
- **`Promise.resolve`**(value)
  - Returns an object that is **resolved** with the given **value**
- **`Promise.all`**(iterable)
  - Returns a **promise**
    - Fulfills when **all** of the promises **have fulfilled**
    - Rejects as soon as **one** of them **rejects**

Promise Methods

- **`Promise.allSettled`**(iterable)
  - Wait until all promises have settled
- **`Promise.race`**(iterable)
  - Returns a promise that fulfills or rejects as soon as one of the promises in an iterable is settled
- **`Promise.finally`**()
  - The handler is called when the promise is settled

Promise.then() – Example

```
console.log('Before promise');
```

```
new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve('done');
  }, 500);
})
.then(function(res) {
  console.log('Then returned: ' + res);
});
```

Resolved after 500 ms

```
console.log('After promise');
```

```
// Before promise
```

```
// After promise
```

```
// Then returned: done
```

Promise.catch() – Example

```
console.log('Before promise');
```

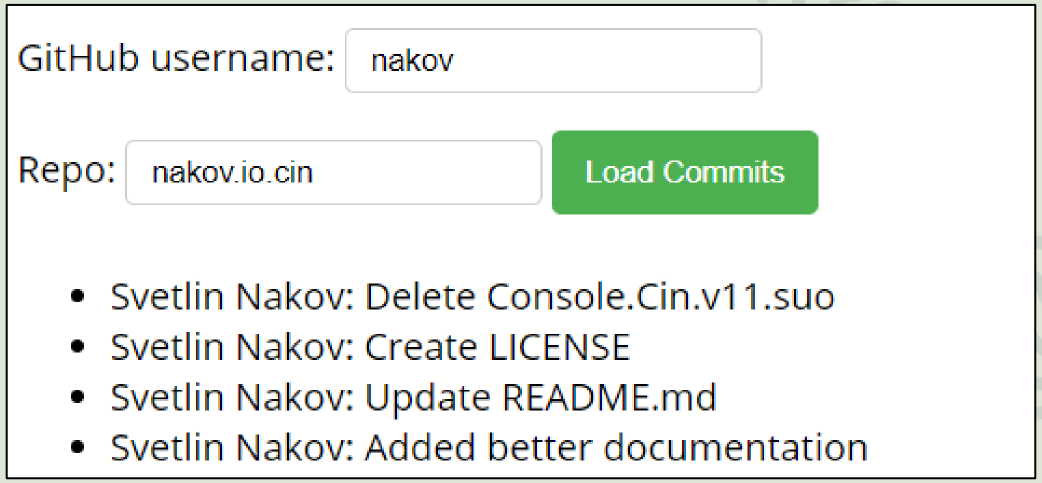
```
new Promise(function (resolve, reject) {
  setTimeout(function () {
    reject('fail');
  }, 500);
})
  .then (function (result) { console.log(result); })
  .catch (function(error) { console.log(error); });
```

Rejected after 500 ms

```
console.log('After promise');
```

Problem: Load GitHub Commits

```
GitHub username:
<input type="text" id="username" value="nakov" /> <br>
Repo: <input type="text" id="repo" value="nakov.io.cin" />
<button onclick="loadCommits()">Load Commits</button>
<ul id="commits"></ul>
<script>
  function loadCommits() {
     // Use Fetch API
  }
</script>
```


GitHub username: nakov

Repo: nakov.io.cin

- Svetlin Nakov: Delete Console.Cin.v11.suo
- Svetlin Nakov: Create LICENSE
- Svetlin Nakov: Update README.md
- Svetlin Nakov: Added better documentation

Simplified Promises

Async / Await

Async Functions

- Returns a **promise**, that can await other promises in a way that **looks synchronous**
- Operate **asynchronously** via the event loop
- Contains an **await** expression that:
  - Is **only valid** inside **async functions**
  - **Pauses** the execution of that function
  - Waits for the Promise's **resolution**

Async Functions (2)

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

```
Expected output:
// calling
// resolved
```

```
async function asyncCall() {
  console.log('calling');
  let result = await resolveAfter2Seconds();
  console.log(result);
}
```

Async Functions (3)

- Do not confuse **`await`** with **`Promise.then()`**
  - **`await`** is always used for a **single promise**
  - To **await two or more** promises in **parallel**, use **`Promise.then()`**
- If a promise resolves normally, then **`await`** promise **returns the result**
- In case of a rejection, it **throws an error**

Async/Await vs Promise.then

Promise.then

```
function logFetch(url) {
  return fetch(url)
    .then(response => {
      return response.text()
    })
    .then(text => {
      console.log(text);
    })
    .catch(err => {
      console.error(err);
    });
}
```

Async/Await

```
async function logFetch(url) {
  try {
    const response =
       await fetch(url);
    console.log(
      await response.text()
    );
  }
  catch (err) {
    console.log(err);
  }
}
```

Error Handling

```
async function f() {
  try {
    let response = await fetch();
    let user = await response.json();
  } catch (err) {
    // catches errors both in fetch andresponse.json
    alert(err);
  }}
```

```
async function f() {
  let response = await fetch();
}
// f() becomes a rejected promise
f().catch(alert);
```

Sequential Execution

- To execute different promise methods **one by one**, use **Async /Await**

```
function execute(x,sec) {
  return new Promise(resolve => {
  console.log('Start: ' + x);
    setTimeout(() => {
      console.log('End: ' + x);
      resolve(x);
  }, sec *1000); }); }
```

```
async function serialFlow() {
  let result1 = await execute(1, 1);
  let result2 = await execute(2, 2);
  let result3 = await execute(3, 3);
  let finalResult = result1 + result2 + result3;
  console.log(finalResult);
}
```

```
// Start: 1
// End: 1
// Start: 2
// End: 2
// Start: 3
// End: 3
// 6
```

Concurrent Execution

```
async function parallelFlow() {
  let result1 = execute(1,1);
  let result2 = execute(2,2);
  let result3 = execute(3,3);
  let finalResult = await result1 +
                    await result2 +
                    await result3;
  console.log(finalResult);
}
```

```
// Expected output:
// Start: 1
// Start: 2
// Start: 3
// End: 1
// End: 2
// End: 3
// 6
```

Live Exercises

Summary

- Asynchronous programming
  - Runs **several tasks** in **parallel**, at the **same time**
- Promises hold **operations**
  - Can be **`resolved`** or **`rejected`**
- **`Async`** functions contain an **`await`** expression
  - **Pauses** the **execution**
  - **Waits** for the **Promise's resolution**

Questions?

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © Kingsland University – https://kingslanduniversity.com

KINGSLAND UNIVERSITY

THANK YOU