WBE: JAVASCRIPT

ASYNCHRONES PROGRAMMIEREN

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events”
- Promises, Async/Await

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events”
- Promises, Async/Await

SYNCHRONES LESEN AUS DATEI

```
1  const fs = require('fs')
2  let data = fs.readFileSync('/etc/hosts')
3  console.log(data)
4  /* → <Buffer 23 23 0a 23 20 48 6f 73 74 20 44 61 74 61 62 61 ...> */
5
6  data = fs.readFileSync('/etc/hosts', 'utf8')
7  console.log(data)
8  /* →
9  ##
10 # Host Database
11 #
12 ...
13 */
```

Problem?

Speaker notes

Wenn keine Zeichencodierung angegeben wird, nimmt Node.js an, dass eine Binärdatei gelesen werden soll. Zurückgegeben wird in diesem Fall ein Buffer-Objekt, ein Array-ähnliches Objekt, welches die Bytes der Datei in Form von 8-Bit-Zahlen enthält.

```
const fs = require('fs')
let data = fs.readFileSync('/etc/hosts')
console.log(Array.from(data))
// [ 35,  35,  10,  35,  32, ... ]
```

Anstatt das ganze Paket zu binden, kann auch nur die benötigte Funktion ausgewählt werden:

```
let {readFileSync} = require("fs")
let data = readFileSync('/etc/hosts')
```

EIN-/AUSGABE

Access	Cycles
L1	3 cycles
L2	14 cycles
RAM	250 cycles
DISK	41’000’000 cycles
NETWORK	240’000’000 cycles

Zahlen nicht mehr aktuell (ca. 2010), aber die Grössenordnung dürfte in etwa noch stimmen

Speaker notes

Tatsächlich ist es so: die Prozessoren werden schneller schneller als der Speicher. Aktuelle SSDs sind sicher schneller als traditionelle "Festplatten". Ausserdem kommt es stark drauf an, wie und ob gelesen oder geschrieben wird, aufeinanderfolgende Bytes oder random access etc.

SYNCHRONES LESEN AUS DATEI

```
1  data = fs.readFileSync('/etc/hosts', 'utf8')
2  /*
3    wait...
4    wait...
5    wait...
6    wait...
7    wait...
8    wait...
9    wait...
10 */
11 console.log(data)
```

FAIL!

ASYNCHRONES LESEN AUS DATEI

```
1 const fs = require('fs')
2 fs.readFile('/etc/hosts', "utf8", (err, data) => {
3   if (err) throw err
4   console.log(data)
5 })
6
7 doSomethingElse()
```

WIN √

CALLBACKS

- Ein Callback ist eine Funktion, welche als Argument einer anderen Funktion übergeben wird und erst aufgerufen wird, wenn das Ereignis eingetreten ist
- Beispiel: `fs.readFile` mit Callback
- Ursprünglich in JS: Reaktion auf Webseiten-Ereignisse

```
document.getElementById('button').addEventListener('click', () => {
  //item clicked
})
```

Mehr zum Browser-DOM in einer späteren Lektion

FILE-API

- Datei-Operationen sind in der Regel langsam
- Sie sollten praktisch immer asynchron ausgeführt werden
- Erstes Argument statt Pfad auch: File Descriptor
- Methode `open` liefert einen File Descriptor

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

fs.open('test.txt', 'r', (err, fd) => {
  // fd is our file descriptor
})
```

Speaker notes

Wie bei `fs.readFile` gibt es auch bei `fs.open` eine synchrone Variante, die seltener sinnvoll ist:

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

try {
  const fd = fs.openSync('test.txt', 'r')
} catch (err) {
  console.error(err)
}
```

Das zweite Argument liefert (ähnlich wie in C) den Modus, wie die Datei geöffnet werden soll:

mode	meaning
`r`	open the file for reading
`r+`	open the file for reading and writing
`w+`	open the file for reading and writing, positioning the stream at the
	beginning of the file; the file is created if not existing
`a`	open the file for writing, positioning the stream at the end of the
	file; the file is created if not existing
`a+`	open the file for reading and writing, positioning the stream at the
	end of the file; the file is created if not existing

Grössere Dateien werden besser mit *Streams* gelesen.

DATEI-INFORMATIONEN

```
1  const fs = require('fs')
2  fs.stat('test.txt', (err, stats) => {
3    if (err) {
4      console.error(err)
5      return
6    }
7
8    stats.isFile()             /* true  */
9    stats.isDirectory()        /* false */
10   stats.isSymbolicLink()     /* false */
11   stats.size                 /* 1024000 = ca 1MB */
12 })
```

https://nodejs.org/api/fs.html#fs_class_fs_stats

DATEIEN UND PFADE

```
1 const path = require('path')
2 const notes = '/users/bkrt/notes.txt'
3
4 path.dirname(notes)                                /* /users/bkrt */
5 path.basename(notes)                               /* notes.txt   */
6 path.extname(notes)                                /* .txt        */
7 path.basename(notes, path.extname(notes))          /* notes       */
```

- Diverse weitere Methoden
- https://nodejs.org/api/path.html

DATEIEN SCHREIBEN

```
1  const fs = require('fs')
2  const content = 'Node was here!'
3
4  fs.writeFile('/Users/bkrt/test.txt', content, (err) => {
5    if (err) {
6      console.error(`Failed to write file: ${err}`)
7      return
8    }
9    /* file written successfully */
10 })
```

Speaker notes

Als drittes Argument kann auch noch der Modus, hier in Form eines Objekts, übergeben werden:

```
fs.writeFile('/Users/flavio/test.txt', content, { flag: 'a+' }, (err) => {})
```

Zum Anhängen an eine bestehende Datei kann auch `fs.appendFile` verwendet werden.

STREAMS

- Grössere Dateien eher mit Streams lesen und schreiben
- Daten werden nach und nach geliefert oder geschrieben
- Lesen: `data`- und `end`-Events

Mehr zum Thema *Streams* in einer späteren Lektion

VERZEICHNISSE

- Im `fs`-Modul: auch Funktionen zur Arbeit mit Ordnern
- Die meisten davon gibt es auch in einer synchronen Variante

Funktion	Bedeutung
`fs.access`	Zugriff auf Datei oder Ordner prüfen
`fs.mkdir`	Verzeichnis anlegen
`fs.readdir`	Verzeichnis lesen, liefert Array von Einträgen
`fs.rename`	Verzeichnis umbenennen
`fs.rmdir`	Verzeichnis löschen

Speaker notes

Das Modul `fs-extra` bietet einige Erweiterungen zu `fs` an. Installation:

```
npm install fs-extra
```

Beispiel:

```
const fs = require('fs-extra')

async function removeFolder(folder) {
  try {
    await fs.remove(folder)
    //done
  } catch (err) {
    console.error(err)
  }
}
```

MEHR ZUM FS-MODUL

Funktion	Bedeutung
`fs.chmod`	Berechtigungen ändern
`fs.chown`	Besitzer und Gruppe ändern
`fs.copyFile`	Datei kopieren
`fs.link`	Hardlink anlegen
`fs.symlink`	Symbolic Link anlegen
`fs.watchFile`	Datei auf Änderungen überwachen

Speaker notes

Weiteres Beispiel:

```
const fs = require('fs')
fs.watchFile('target.txt', function() {
  console.log("File 'target.txt' just changed!")
})
console.log("Now watching target.txt for changes...")
```

Script starten und target.txt ändern:

```
touch target.txt
```

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events”
- Promises, Async/Await

EIN THREAD

- JavaScript-Code wird in einem Thread abgearbeitet
- Probleme vermieden, die bei paralleler Ausführung auftreten können (gemeinsame Ressourcen, mögliche Blockaden)
- Vorsicht: Thread darf nicht blockiert werden
- Browser: Tabs normalerweise mit unabhängigen Event Loops

Speaker notes

Genauer: JavaScript-Code wird *normalerweise* in *einem* Thread abgearbeitet. Es gibt schon APIs, welche parallele Verarbeitung von JavaScript-Code in eigenen Threads erlauben. Web Workers erlauben es, ein Script in einem Hintergrund-Thread abzuarbeiten. Es hat jedoch keinen Zugriff auf das DOM (Document Object Model).

https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API

BEISPIEL

```
// script.js
setTimeout(() => {
  console.log("fertig :)")
}, 5000)

console.log("starting...")
```

Aufruf:

```
$ node script.js
starting...
fertig :)
$
```

- Mehr zu `setTimeout()` gleich
- Callback nach Ablauf des Timers aufgerufen
- Ausgabe fertig :) erscheint 5 Sekunden nach starting...

ABLAUF

- Script wird ausgeführt
- Funktionsaufrufe → Call Stack
- Callbacks asynchroner Operationen in Event Queue(s) gelegt
- Wenn Call Stack leer, d.h. (synchrone) Aufrufe abgearbeitet:
  - Übergang in eine so genannte Event Loop
  - Callbacks aus Event Queue abgearbeitet
  - Event Queue leer: Programm beendet

Speaker notes

Dies ist ein vereinfachtes Modell. Tatsächlich sind in der Event Loop mehrere Queues eingebunden.

Der Punkt "Callbacks aus Event Queue abgearbeitet" bedeutet für den Timer: Überprüfen, ob der Timer abgelaufen, dann Callback abarbeiten. Ansonsten: weiter in der Event Loop.

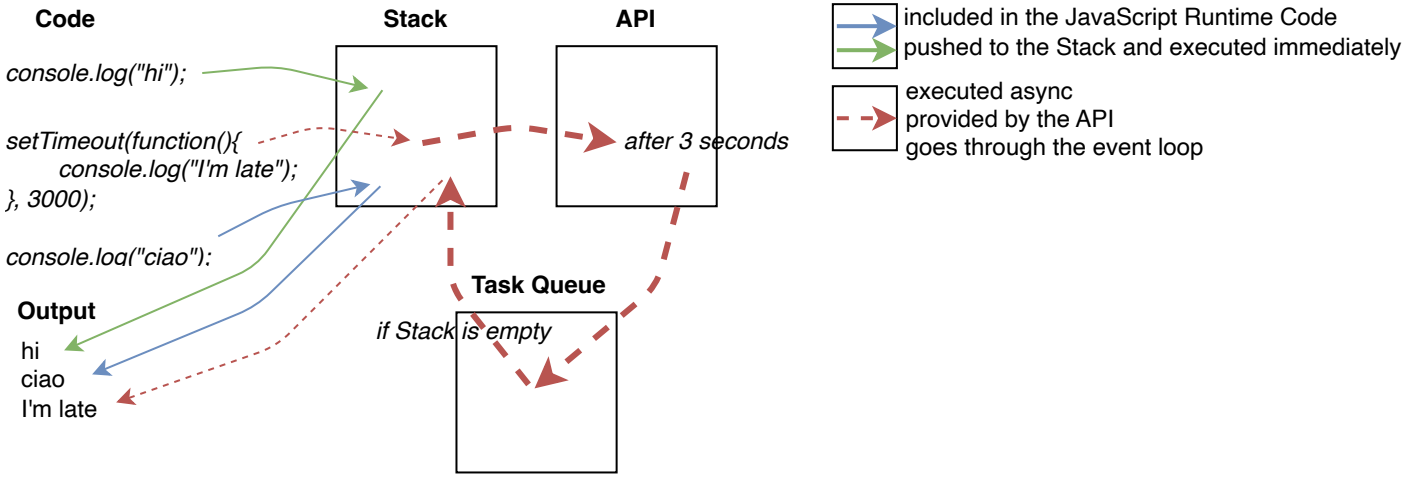
EVENT LOOP

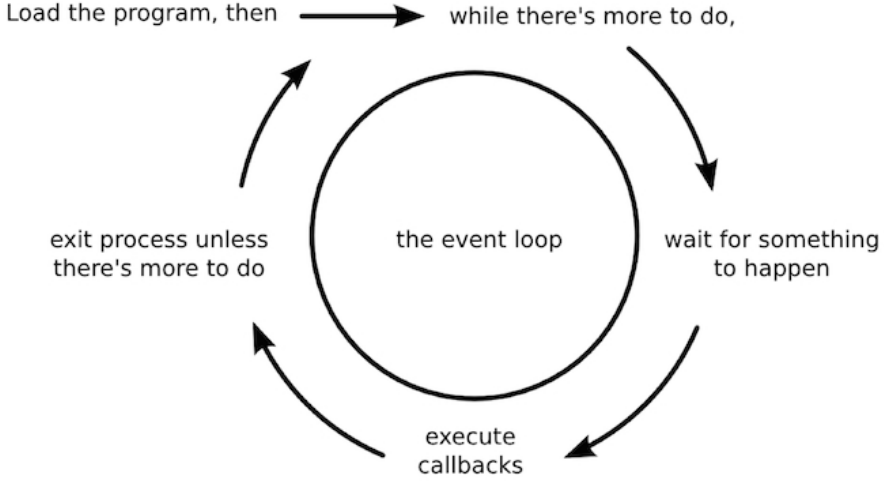
Ein vereinfachtes Modell kann das Verhalten asynchroner Programme in vielen Situationen ganz gut erklären. Es basiert auf diesen Annahmen:

- Es gibt *eine* Event Queue
- Ablage der Callbacks in der Event Queue basiert auf OS-APIs

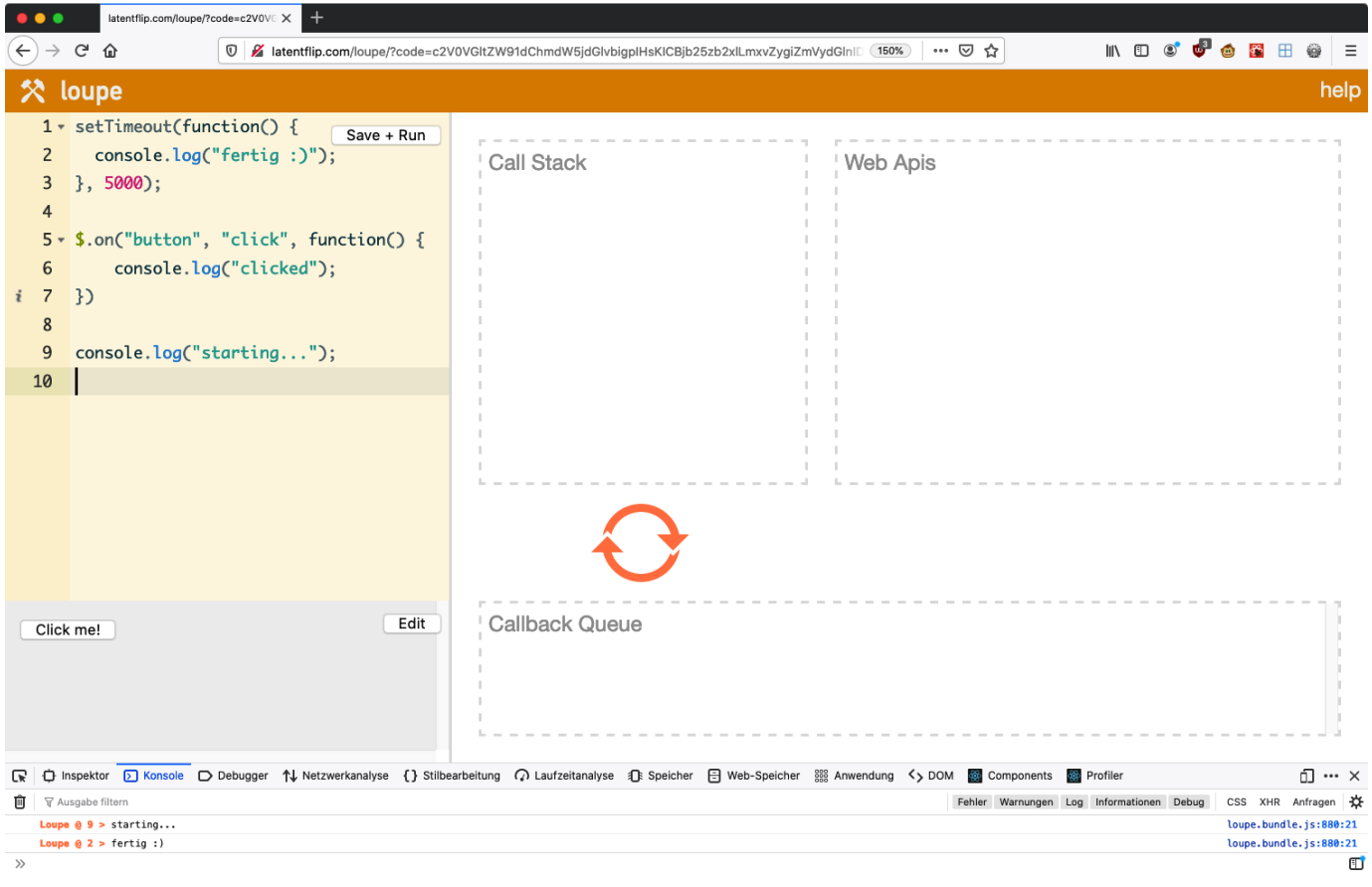
Speaker notes

Event Loop: vereinfacht





EVENT LOOP: SIMULATOR



http://latentflip.com/loupe/

Speaker notes

Dieser Simulator basiert ebenfalls auf einem vereinfachten Modell der Event Loop.

Beispielcode:

```
setTimeout(function() {
  console.log("fertig :)")
}, 5000)

$.on("button", "click", function() {
    console.log("clicked")
})

console.log("starting...")
```

Zunächst wird "starting..." ausgegeben. Nach fünf Sekunden "fertig :)". Wenn während oder nach diesen fünf Sekunden Button-Events auftreten, werden diese ebenfalls in die Callback Queue eingereiht. Tatsächlich existieren sogar mehrere solche Queues.

WICHTIG

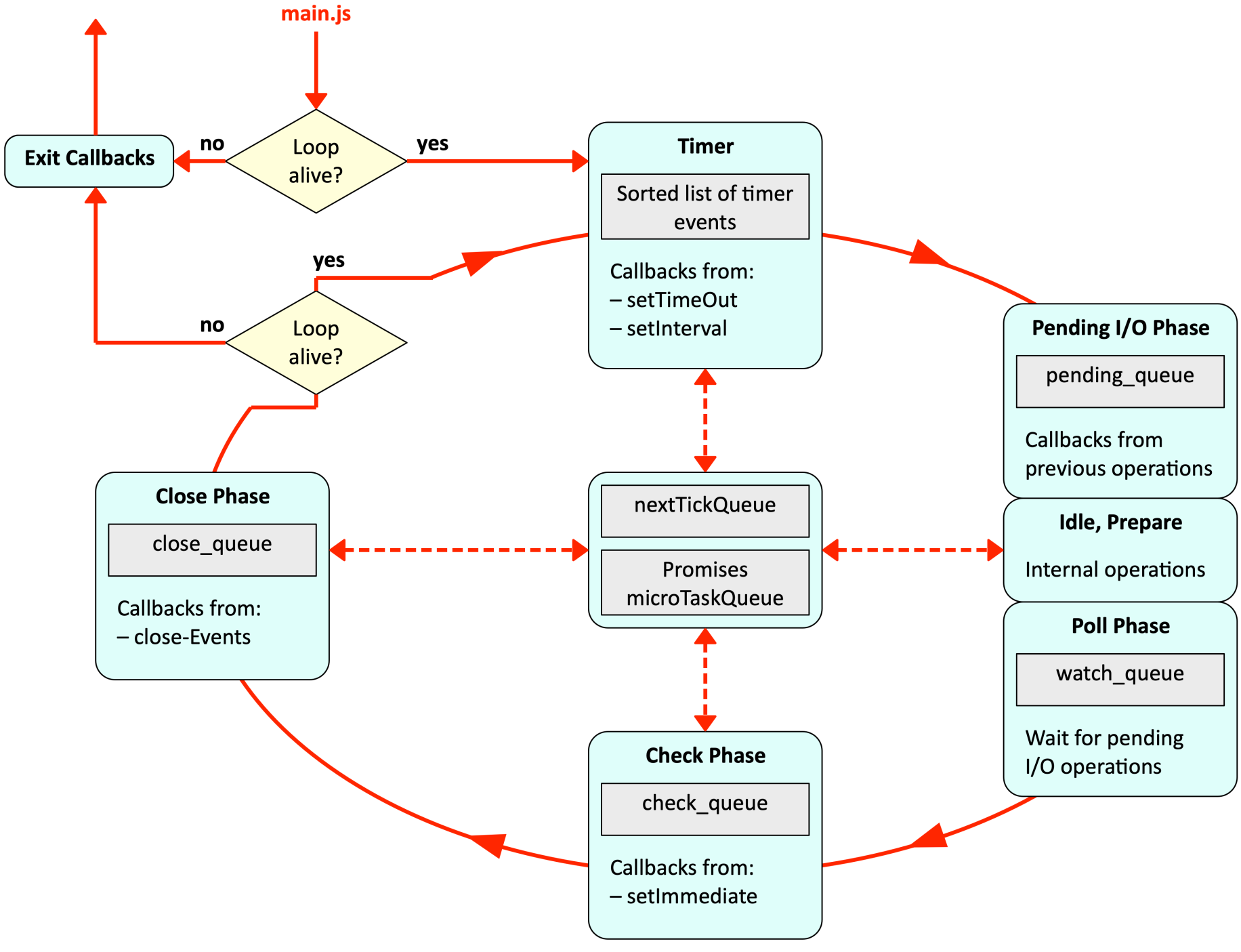
- Event Loop nicht blockieren!
- Grund: blockiert die gesamte Applikation
- Im Browser: blockiert den Browser
- Zu vermeiden also:
  - synchrone Operationen (etwa für Datei- oder Netzwerkzugriff)
  - aufwändige Berechnungen ohne Unterbrechung
  - Endlosschleifen

Speaker notes

In modernen Browsern wird immerhin nur ein Tab blockiert.

EVENT LOOP: MEHR DETAILS

- Einfaches Modell der Event Loop entspricht nicht der Realität
- Es genügt, um viele, aber nicht alle Situationen zu erklären
- Ein paar Richtigstellungen
  - Event Loop ist nicht Teil der JS-Engine sondern steuert diese
  - Es gibt mehrere Queues
  - Die Event Loop läuft nicht in einem separaten Thread
  - Mit `setTimeout` wird keine OS-API beauftragt


main.js
Exit Callbacks
no
Loop alive?
yes
Timer
Sorted list of timer events
Callbacks from:
– setTimeOut
– setInterval
yes
no
Loop alive?
Pending I/O Phase
pending_queue
Callbacks from previous operations
Close Phase
close_queue
Callbacks from:
– close-Events
nextTickQueue
Promises microTaskQueue
Idle, Prepare
Internal operations
Poll Phase
watch_queue
Wait for pending I/O operations
Check Phase
check_queue
Callbacks from:
– setImmediate

Speaker notes

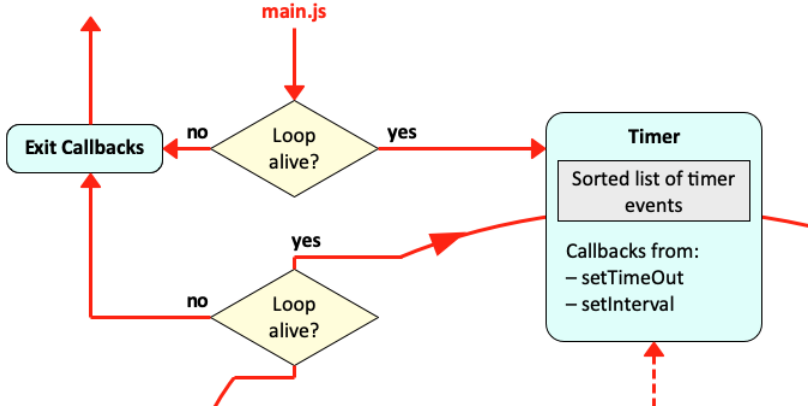
Dies ist ein realistischeres Modell der Event Loop. Die Details werden in den folgenden Slides weiter ausgeführt.

https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

EVENT LOOP: ABLAUF

- Script-Aufruf: Event Loop und Datenstrukturen angelegt
- Script mit synchronen Operationen ausgeführt (Call Stack)
- Dabei werden die Listen und Queues ggf. mit Callbacks gefüllt
- Nach Abschluss des Scripts (Call Stack leer):
  Eintritt in die Event Loop
- Schleife bis alle Callbacks abgearbeitet

EVENT LOOP: TIMER



- Für Callbacks des Zeitgebers (`setTimeout`, `setInterval`)
- Sortierte Liste (keine Queue) nach Zeitstempel der Fälligkeit

- Callbacks für bereits verstrichenen Zeitpunkte abgearbeitet
- Abbruch auch, wenn systemspezifisches Limit erreicht

Speaker notes

Der letzte Punkt bedeutet, dass es möglich ist, dass in einem Durchgang nicht alle eigentlich erfüllten Timer-Events abgearbeitet werden, manche also in den nächsten Schleifendurchgang verschoben werden.

SETTIMEOUT

- Mit `setTimeout` kann Code definiert werden, der zu einem späteren Zeitpunkt ausgeführt werden soll
- Eintrag in die Timer-Liste, auch wenn Zeit auf 0 gesetzt wird
- Kann mit `clearTimeout` entfernt werden

```
setTimeout(() => {
  /* runs after 50 milliseconds */
}, 50)
```

Speaker notes

Parameter für die Callback-Funktion können mitgegeben werden:

```
const myFunction = (firstParam, secondParam) => {
  // do something
}

// runs after 2 seconds
setTimeout(myFunction, 2000, firstParam, secondParam)
```

`setTimeout` gibt eine Timer-Id zurück, über welche der Timer mit `clearTimeout` wieder gelöscht werden kann, bevor das Callback aufgerufen wurde.

Auch wenn die Zeit auf 0 gesetzt wird, erfolgt ein Eintrag in die Timer-Liste, tatsächlich sogar mit einer minimalen Verzögerung (1-4ms) versehen. Meist wird das Callback dann bei der nächsten Konsultation der Timer-Liste ausgeführt, ggf. aber auch erst im nächsten Durchgang.

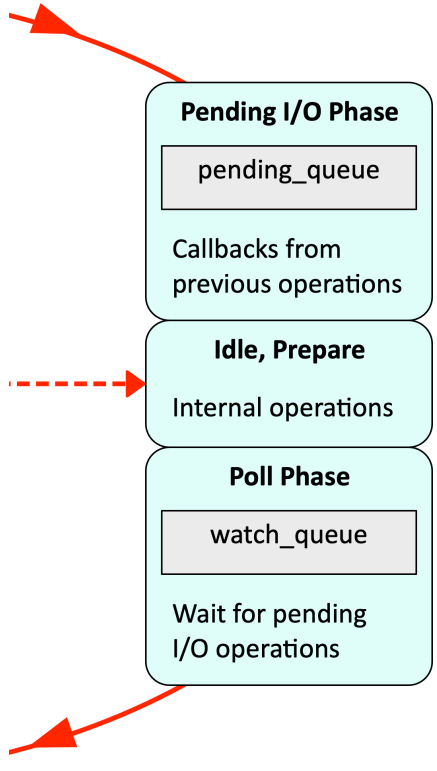
SETINTERVAL

- Callback alle n Millisekunden in die Callback Queue eingefügt
- Kann mit `clearInterval` beendet werden

```
const id = setInterval(() => {
  // runs every 2 seconds
}, 2000)

clearInterval(id)
```

EVENT LOOP: PENDING I/O, ...

- Von vorhergehenden Durchgängen aufgeschobene Callbacks
- Beispiel: Fehlermeldungen bestimmter TCP-Aufrufe
- Idle, Prepare: interne Aufgaben



Speaker notes

Aus der Node.js-Dokumentation:

pending callbacks

This phase executes callbacks for some system operations such as types of TCP errors. For example if a TCP socket receives ECONNREFUSED when attempting to connect, some *nix systems want to wait to report the error. This will be queued to execute in the pending callbacks phase.

EVENT LOOP: POLL PHASE

- Abarbeiten der `watch_queue` (auch: `poll_queue`)
- Auf I/O (Verbindungsanfragen etc.) warten
- Wartezeit abhängig von Füllstand der Timer-Liste und der `check_queue` (nächster Punkt in der Loop)
- Abbruch auch, wenn systemspezifisches Limit erreicht

Speaker notes

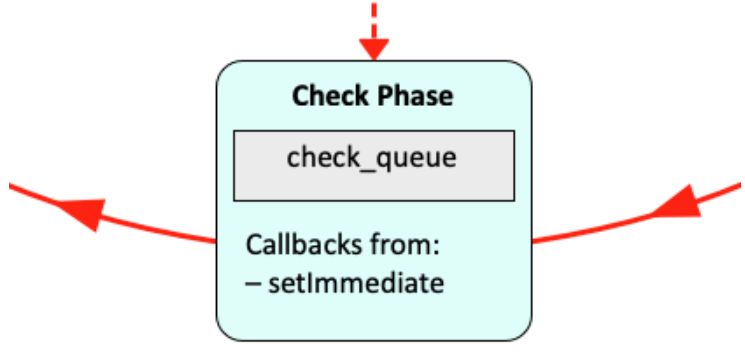
Beispiel aus der Node.js-Dokumentation:

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

function someAsyncOperation(callback) {
  // Assume this takes 95ms to complete
  fs.readFile('/path/to/file', callback);
}

const timeoutScheduled = Date.now();

setTimeout(() => {
  const delay = Date.now() - timeoutScheduled;
  console.log(`${delay}ms have passed since I was scheduled`);
}, 100);

// do someAsyncOperation which takes 95 ms to complete
someAsyncOperation(() => {
  const startCallback = Date.now();
  // do something that will take 10ms...
  while (Date.now() - startCallback < 10) {
    // do nothing
  }
});
```

Ein Timeout-Callback wird mit 100ms in die Timer-Liste eingetragen und eine asynchrone Dateioperation wird gestartet. Da der Timer nicht parat ist, wird diese Phase übersprungen. In der Poll Phase wird auf das Ergebnis der Dateioperation gewartet. Annahme: diese ist nach 95ms erledigt, dann wird das Callback abgearbeitet, das eine 10ms Verzögerung enthält. Das Timer Callback wird dann nach 105ms ausgeführt.

EVENT LOOP: CHECK PHASE

- Abarbeiten der `check_queue`
- Callbacks von `setImmediate`
- Abbruch auch, wenn systemspezifisches Limit erreicht



Speaker notes

Aus der Node.js-Dokumentation:

Generally, as the code is executed, the event loop will eventually hit the poll phase where it will wait for an incoming connection, request, etc. However, if a callback has been scheduled with setImmediate() and the poll phase becomes idle, it will end and continue to the check phase rather than waiting for poll events.

SETIMMEDIATE

- Node.js API (im Browser nicht unterstützt)
- Callbacks, die direkt nach der Poll Phase ausgeführt werden
- Damit: Unterschied zwischen `setImmediate(...)` und `setTimeout(..., 0)`

```
setImmediate(() => {
  console.log('immediate')
})
```

Speaker notes

`setImmediate` ist eine Node.js API. Es wird (mit Ausnahme von einigen Versionen des Internet Explorers und Edge) von keinem Browser unterstützt.

`setImmediate(...)` und `setTimeout(..., 0)` verhalten sich unterschiedlich, je nachdem, in welcher Phase die Funktion aufgerufen wird.

```
// timeout_vs_immediate.js
setTimeout(() => {
  console.log('timeout')
}, 0)

setImmediate(() => {
  console.log('immediate')
})
```

Werden die Funktionen im Haupt-Script ausgeführt, ist die Reihenfolge laut Node.js-Dokumentation nicht definiert. Eigentlich sollte man annehmen, dass entsprechend dem Aufbau der Event Loop immer zunächst das Timeout bearbeitet wird. In diversen Versuchen war das auch immer der Fall:

```
$ node immediate.js
timeout
immediate
```

Warum ist die Reihenfolge trotzdem undefiniert? Der Grund ist, dass die JavaScript-Umgebung `setTimeout`- und `setInterval`- Callbacks mit einer minimalen Verzögerung versehen. Diese beträgt im Browser 4ms und in Node.js 1ms. Konsequenz: Die erste Timer-Phase kann beim Übergang in die Event Loop übersprungen werden, auch wenn 0 als Zeitintervall spezifiziert wurde.

Info:
https://ajahne.github.io/blog/javascript/2018/05/10/javascript-timers-minimum-delay.html

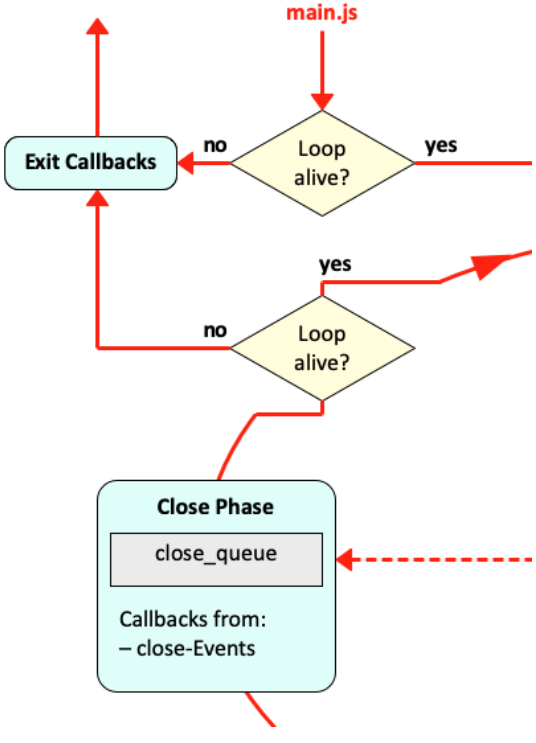
Wenn die beiden Aufrufe in den I/O-Zyklus verschoben werden, wird immer zuerst das Immediate-Event bearbeitet.

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

fs.readFile("immediate.js", () => {
  setTimeout(() => {
    console.log('timeout')
  }, 0)
  setImmediate(() => {
    console.log('immediate')
  })
})
```

```
$ node immediate2.js
immediate
timeout
```

https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

EVENT LOOP: CLOSE PHASE



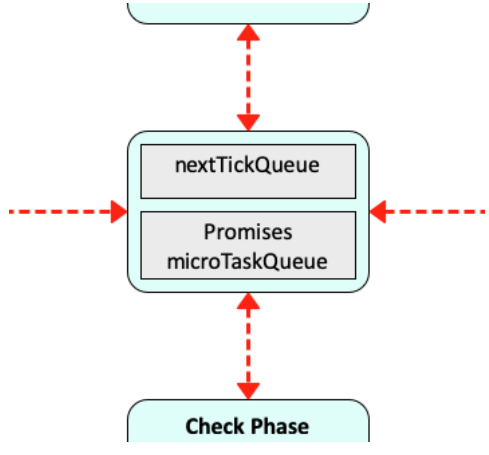
- Verarbeiten bestimmter close-Events
- Zum Beispiel:

```
socket.on('close', ...)
```

- Wenn dann alle Queues und Listen leer sind, wird die Event Loop beendet

NEXTTICKQUEUE UND PROMISES

- So früh wie möglich abgearbeitet
- Nicht Teil der Event Loop



- Von Node.js nach jeder Operation eingefügt
- Operation hier: JavaScript-Aufruf von C/C++ aus

Speaker notes

Beides sind Micro Tasks: `nextTick`-Callbacks und Callbacks von erfüllten oder abgewiesenen Promises. Sie werden zwischen den Phasen der Event Loop und ab Node.js 11 sogar zwischen den Callbacks der anderen Phasen eingefügt (s.u.).

NEXTTICKQUEUE

- Durch die API `process.nextTick` angelegte Callbacks
- `process.nextTick` daher vor `setImmediate` bearbeitet

```
fs.readFile("nexttick.js", () => {
  setTimeout(()        => { console.log('timeout'); }, 0)
  setImmediate(()      => { console.log('immediate');  })
  process.nextTick(() => { console.log('nexttick');   })
})

// Output:
// nexttick
// immediate
// timeout
```

Speaker notes

Die Namensgebung ist hier etwas unglücklich. Auch wenn der Name etwas anderes nahe legt, werden `setImmediate`-Callbacks nicht nach jeder Operation aufgerufen, sondern nur in der Check Phase. Dagegen werden `process.nextTick`-Callbacks nach jeder Operation ausgeführt.

PROMISES MICROTASKQUEUE

- Callbacks von erfüllten/abgewiesenen Promises
- Das betrifft die native Promise-API von JavaScript
- Nach den `nextTick`-Callbacks abgearbeitet

```
Promise.resolve().then(() => console.log('promise resolved'))
setImmediate(() => console.log('set immediate'))
process.nextTick(() => console.log('next tick'))
setTimeout(() => console.log('set timeout'), 0)

// next tick
// promise resolved
// set timeout
// set immediate
```

Speaker notes

Erfüllte/abgewiesene Promises werden also mit hoher Priorität bearbeitet. Das gilt für die native Promise-API von JavaScript. Promise-Bibliotheken verwenden zum Beispiel `process.nextTick` (Q) oder `setImmediate` (Bluebird).

Info und ausführlicheres Beispiel:
https://blog.insiderattack.net/promises-next-ticks-and-immediates-nodejs-event-loop-part-3-9226cbe7a6aa

ÄNDERUNG SEIT NODE.JS 11

- Micro Tasks werden neu auch zwischen den Callbacks der anderen Phasen ausgeführt
- Entspricht dem Verhalten in Browsern

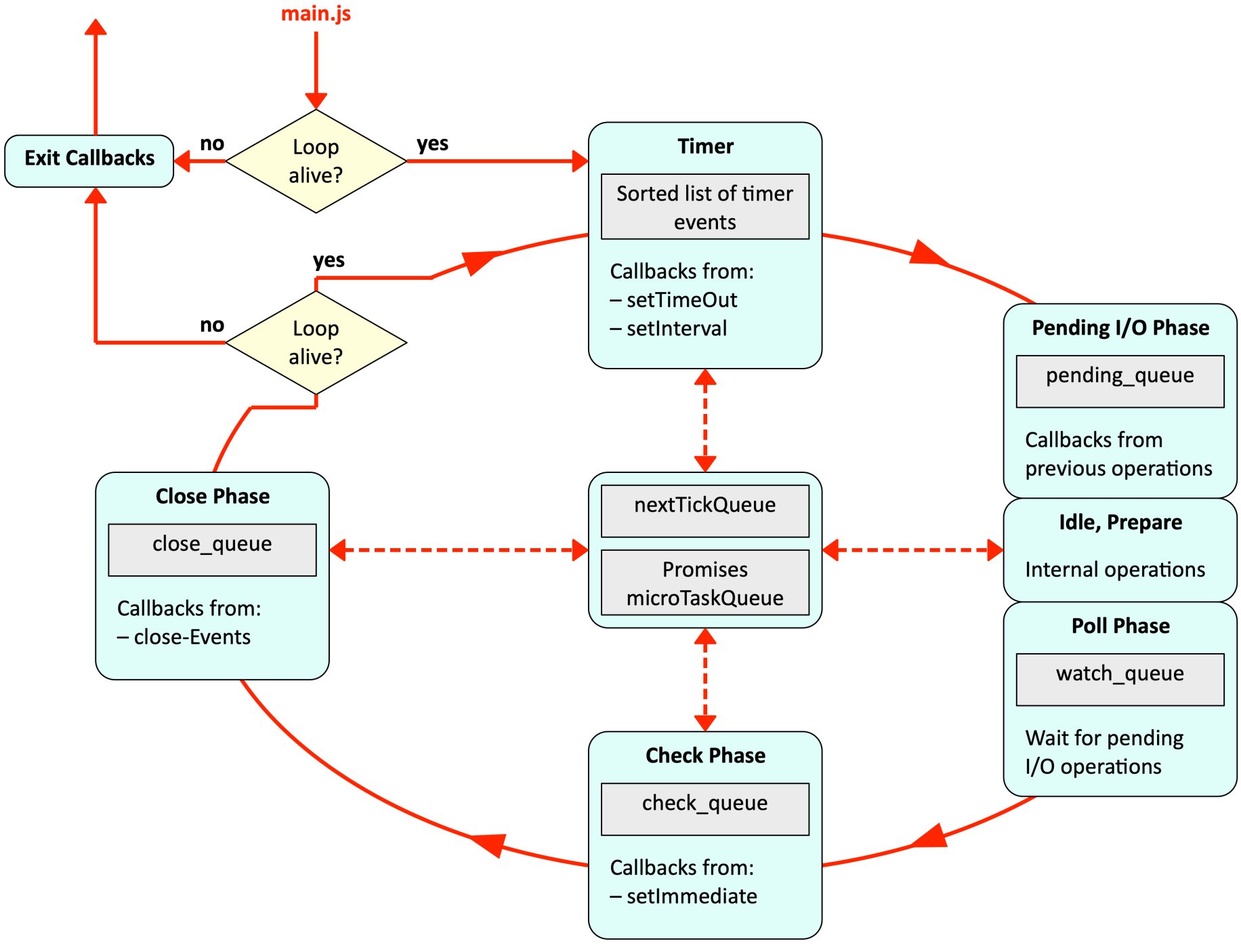
```
setTimeout(() => {
  console.log('timeout1')
  Promise.resolve().then(() => console.log('promise resolve'))
})
setTimeout(() => console.log('timeout2'))
```

```
Node.js < 11:

timeout1
timeout2
promise resolve
```

```
Node.js >= 11 und Browser:

timeout1
promise resolve
timeout2
```

Speaker notes

Info und weitere Beispiele:
https://blog.insiderattack.net/new-changes-to-timers-and-microtasks-from-node-v11-0-0-and-above-68d112743eb3


main.js
Exit Callbacks
no
Loop alive?
yes
Timer
Sorted list of timer events
Callbacks from:
– setTimeOut
– setInterval
yes
no
Loop alive?
Pending I/O Phase
pending_queue
Callbacks from previous operations
Close Phase
close_queue
Callbacks from:
– close-Events
nextTickQueue
Promises microTaskQueue
Idle, Prepare
Internal operations
Poll Phase
watch_queue
Wait for pending I/O operations
Check Phase
check_queue
Callbacks from:
– setImmediate

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events”
- Promises, Async/Await

EVENT EMITTER

```
const EventEmitter = require('events')
const door = new EventEmitter()
```

- Verwaltet Liste von Listeners zu bestimmten Events
- Listener für das Event können hinzugefügt oder entfernt werden
- Event kann ausgelöst werden → Listener werden informiert

LISTENER HINZUFÜGEN

```
const EventEmitter = require('events')
const door = new EventEmitter()

door.on('open', () => {
  console.log('Door was opened')
})
```

- Fügt Event Listener hinzu
- Alias: `emitter.addListener`

EVENT AUSLÖSEN

```
door.on('open', (speed) => {
  console.log(`Door was opened, speed: ${speed || 'unknown'}`)
})

door.emit('open')
door.emit('open', 'slow')
```

- Methode `emit`
- Ruft *synchron* alle Listener auf
- Und zwar in der Reihenfolge wie sie definiert wurden
- Es können Argumente übergeben werden

Speaker notes

`EventEmitter`-Instanzen stellen einen Kanal für Ereignisse zur Verfügung, in den interessierte Komponenten sich einklinken können: *Event Bus*.

THIS

```
const myEmitter = new EventEmitter()
myEmitter.on('event', function (a, b) {
  console.log(a, b, this, this === myEmitter)
  // Prints:   a b EventEmitter { domain: null, ... } true
})
myEmitter.emit('event', 'a', 'b')
```

- Normale Listener-Funktion: `this` referenziert die EventEmitter-Instanz, an welche der Listener angehängt ist
- Achtung: Dies gilt nicht für Arrow Functions

Speaker notes

In Arrow Functions wird `this` aus der Umgebung übernommen. Im globalen Kontext, zum Beispiel in der REPL, ist dies das globale Objekt:

```
const myEmitter = new EventEmitter()
myEmitter.on('event', (a, b) => {
  console.log(a, b, this)
  // Prints: a b <ref *1> Object [global] {
  //         global: [Circular *1],
  //         clearInterval: [Function: clearInterval],
  //         ... }
})
myEmitter.emit('event', 'a', 'b')
```

EVENTS ASYNCHRON

- Nach Ereignisauslösung (`emit`) werden die Listener ausgeführt
- Listener werden synchron aufgerufen
- Und zwar in der Reihenfolge der Registrierung
- Listener können selbst auf asynchronen Modus wechseln

```
myEmitter.on('event', (a, b) => {
  setImmediate(() => {
    console.log('this happens asynchronously')
  })
})
```

Speaker notes

Weiteres Beispiel:

```
const EventEmitter = require('events')
const door = new EventEmitter()

door.on('open', (arg) => {
  console.log('Door was opened ' + arg)
})

process.nextTick(() => {
  console.log('next tick')
  door.emit('open', 'async')
})

door.emit('open', 'sync')

// Door was opened sync
// next tick
// Door was opened async
```

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events”
- Promises, Async/Await

PROMISES

- Eingeführt mit ES6 (ES2015)
- Vermeiden von verschachtelten Callbacks
- Vereinfacht Fehlerbehandlung

Promise

Platzhalter für einen Wert, der erst später voraussichtlich verfügbar sein wird

FUNKTION MIT CALLBACKS

- Bekanntes Beispiel
- Asynchrones Lesen mit `fs`-Modul
- Diesmal in Funktion verpackt

```
1  const fs = require('fs')
2
3  function readFileAsync (file, success, error) {
4    fs.readFile(file, "utf8", (err, data) => {
5      if (err) error(err)
6      else success(data)
7    })
8  }
9
10 /* Aufruf: */
11 readFileAsync(file, okCallback, failCallback)
```

Speaker notes

Konkreter Aufruf:

```
readFileAsync('/etc/hosts',
  console.log,
  () => {
    console.log("Error reading file")
  })
```

Oder das Callback im Erfolgsfall etwas ausführlicher:

```
readFileAsync('/etc/hosts',
  (data) => {
    console.log(data)
  },
  () => {
    console.log("Error reading file")
  })
```

Wenn die Callbacks selbst auch wieder asynchrone Funktionen aufrufen, wird es schnell unübersichtlich.

FUNKTION MIT PROMISE

```
1  function readFilePromise (file) {
2    let promise = new Promise(
3      function resolver (resolve, reject) {
4        fs.readFile(file, "utf8", (err, data) => {
5          if (err) reject(err)
6          else resolve(data)
7        })
8      })
9    return promise
10 }
```

- Gibt nun ein Promise-Objekt zurück
- Promise-Konstruktor erhält *resolver*-Funktion

Speaker notes

Die Funktion `resolver` muss natürlich keinen Namen haben. Dieser dient hier nur zum besseren Verständnis.

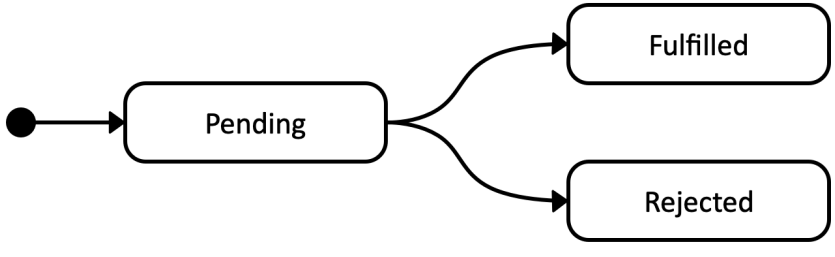
FUNKTION MIT PROMISE

- Rückgabe einer Promise: potentieller Wert
- Kann später erfüllt oder zurückgewiesen werden
- Aufruf neu:

```
readFilePromise('/etc/hosts')
  .then(console.log)
  .catch(() => {
    console.log("Error reading file")
  })
```

PROMISE-ZUSTÄNDE

- `pending`: Ausgangszustand
- `fulfilled`: erfolgreich abgeschlossen
- `rejected`: ohne Erfolg abgeschlossen



- Nur ein Zustandsübergang möglich
- Zustand in Promise-Objekt gekapselt

ÜBUNG: AUSGABE?

```
1  var promise = new Promise((resolve, reject) => {
2      setTimeout(resolve, 500, 'done')
3      setTimeout(reject, 300, 'failed')
4      /* throw new Error('So goes it not :)') */
5  })
6
7  promise.then(function (data) {
8    console.log('success: ' + data)
9  })
10 .catch(function (data) {
11   console.log('fail: ' + data)
12 })
```

Speaker notes

Wenn im Resolver im Zustand `pending` eine Exception geworfen wird, geht die Promise in den Zustand `rejected`. Sobald die resolve- oder reject-Funktion aufgerufen wurde, haben weitere Aufrufe dieser Funktionen keine Wirkung mehr. Auch geworfene Exceptions haben dann keine Wirkung mehr.

Nebenbei bemerkt: Die Aufrufsequenz im Beispiel zeigt, dass das Setzen von Semikolons am Ende von Anweisungen potenziell fehleranfällig ist:

```
promise.then(function(data) {
  console.log('success: ' + data)
})                                  // <== hier darf kein Semikolon stehen
.catch(function(data) {
  console.log('fail: ' + data)
})                                  // <== hier würde ein Semikolon stehen
```

PROMISES

- `then`-Aufruf gibt selbst Promise zurück
- `catch`-Aufruf ebenfalls, per Default erfüllt
- So können diese Aufrufe verkettet werden
- Promise, welche unmittelbar resolved wird:
  `Promise.resolve(...)`
- Promise, welche unmittelbar rejected wird:
  `Promise.reject(...)`

Speaker notes

Die Rückgabe von `then` wird in eine Promise verpackt, wenn es noch keine ist. Auf diese Weise wird erreicht, dass die Aufrufe verkettet werden können.

```
// calls to then return a new promise

let p1 = Promise.resolve('initial resolved')

let p2 = p1.then((data) => {
  console.log("1: " + data)
  return "[" + data + "]"
})

let p3 = p2.then((data) => {
  console.log("2: " + data)
})

// 1: initial resolved
// 2: [initial resolved]
```

Es kann natürlich auch eine Promise zurückgegeben werden:

```
const wait = (ms) => {
  return new Promise(resolve => setTimeout(resolve, ms))
}

wait(2000).then(() => {
  console.log("first")
  return wait(2000)
})
.then(() => {
  console.log("second")
});
```

ÜBUNG: AUSGABE?

```
1  var promise = new Promise((resolve, reject) => {
2    throw new Error('fail')
3    resolve()
4  })
5
6  promise
7    .then (() => console.log('step1'))
8    .then (() => { throw Error('fail') })
9    .then (() => console.log('step2'))
10   .catch(() => console.log('catch1'))
11   .then (() => console.log('step3'))
12   .catch(() => console.log('catch2'))
13   .then (() => console.log('step4'))
```

Speaker notes

```
catch1
step3
step4
```

PROMISES VERKNÜPFEN

- `Promise.all()`
  - Erhält Array von Promises
  - Erfüllt mit Array der Resultate, wenn alle erfüllt sind
  - Zurückgewiesen sobald eine Promise zurückgewiesen wird
- `Promise.race()`
  - Erhält Array von Promises
  - Erfüllt sobald eine davon erfüllt ist
  - Zurückgewiesen sobald eine davon zurückgewiesen wird

Speaker notes

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'first')
})

var p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 4000, 'second')
})

Promise.all([p1, p2])
  .then((data) => console.log(data))
  .catch(console.log)

Promise.race([p1, p2])
  .then((data) => console.log(data))
  .catch(console.log)

// first
// [ 'first', 'second' ]
```

Mit ES2021 wurde noch `Promise.any()` eingeführt. Während `Promise.race()` sich auf die erste erfüllte oder zurückgewiesene Promise bezieht, wartet `Promise.any()` auf die erste erfüllte Promise, zurückgewiesene Promises werden ignoriert. Wenn alle zurückgewiesen werden, wird ein AggregateError ausgelöst.

ASYNC / AWAIT

- Asynchrone Funktionen
- Grundlage: Promise API
- Eingeführt mit ES8 (ES2017)
- Grund: Einsatz von Promises immer noch kompliziert
- Nun: asynchroner Code ähnlich synchronem Code aufgebaut

ASYNC/AWAIT: BEISPIEL 1

```
/* Bekanntes Beispiel */
const readHosts = () => {
  readFilePromise('/etc/hosts')
    .then(console.log)
    .catch(() => {
      console.log("Error reading file")
    })
}
```

```
/* Mit async/await */
const readHosts = async () => {
  try {
    console.log(await readFilePromise('/etc/hosts'))
  }
  catch (err) {
    console.log("Error reading file")
  }
}
```

Speaker notes

Oder direkt aufgerufen:

```
(async () => {
  try {
    console.log(await readFilePromise('/etc/hosts'))
  }
  catch (err) {
    console.log("Error reading file")
  }
})()
```

ASYNC/AWAIT: BEISPIEL 2

```
1  function resolveAfter2Seconds (x) {
2    return new Promise(resolve => {
3      setTimeout(() => {
4        resolve(x)
5      }, 2000)
6    })
7  }
8
9  async function add1(x) {
10   var a = resolveAfter2Seconds(20)
11   var b = resolveAfter2Seconds(30)
12   return x + await a + await b
13 }
14
15 add1(10).then(console.log)
```

PROMISE API VON FS

- Ab Node.js 10
- Bisher: Callback oder Promise selber bauen
- Nun: viele fs-Methoden mit Promise-Rückgabe

```
1 const {readFile} = require("fs/promises")
2
3 readFile("file.txt", "utf8")
4   .then(text => console.log("The file contains:", text))
```

Speaker notes

Weiteres Beispiel: Datei schreiben

```
const fsp = require("fs/promises")

try {
    await fsp.writeFile("/tmp/test.txt", "Hi!");
    console.info("File created successfully!");
} catch (error){
    console.error(error);
}
```

Zum Vergleich – Beispiel mit Callback:

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

fs.writeFile("/tmp/test.txt", "Hi!", error => {
    if (error) console.error(error);
    else console.log("File created successfully!")
});
```

Oder – Promise selbst hergestellt:

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

const writeFilePromise = (file, data) => {
    return new Promise((resolve, reject) => {
        fs.writeFile(file, data, error => {
            if (error) reject(error);
            resolve("File created successfully!")
        });
    });
};

writeFilePromise("/tmp/test.txt", "Hi!")
    .then(result => console.log(result))
    .catch(error => console.log(error))
```

Quelle:
https://dev.to/mrm8488/from-callbacks-to-fspromises-to-handle-the-file-system-in-nodejs-56p2

QUELLEN

- Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/
- Ältere Slides aus WEB2 und WEB3
- Dokumentationen, u.a. zu Node.js

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Einzelne Abschnitte in Kapitel 11 von:
  Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/