

Nebenläufigkeit in JavaScript

Ziel dieses Workshops

- Verständnis für das Laufzeitmodell von JavaScript im Browser
- Verständnis zu synchron und asynchronen Funktionen
- Verständnis zu konkreten Modellen
 - Callbacks
 - Promises
 - Async / Await

Methodik

- Gemeinsames Erarbeiten der Grundlagen (Vortrag)
 - Fallstricke (Pitfalls) und Best Practices
- Zu jedem Themenblock Hands-On Übungen und Diskussion
 - Breakout Sessions?
- Wir werden viel Code sehen
 - `console.log`

Fast Facts

- JS wurde 1995 in 10 Tagen entwickelt
- Seit 1997 Standardisiert
 - ECMA-262 als ECMAScript
- Versionen: ES1, ES2, ES3, ES4, ES5, ES5.1, ES6, ES2016-2021
 - Mittlerweile jährlich im Juni
- Verantwortlich: TC39 Komitee
 - Mitglieder führender Tech-Companies
 - Tagung im 2-Monats-Takt
- Entwicklung nach Grundsatz „Don't break the web“

Fast Facts

- JS wurde für Netscape Navigator entwickelt
- Browser und Webseiten lange Zeit nicht standardisiert
 - Bspw.: Internet Explorer
- 2004 HTML Standard
 - WHATWG
- Aktuelle Version: HTML5

Definitionen – Nebenläufigkeit

“Concurrency is the composition of independently executing computations.

Concurrency is a way to structure software [...].

It is not parallelism.

[...] although it enables parallelism.

If you have only one processor, your program can still be concurrent but it cannot be parallel.

[...] a well-written concurrent program might run efficiently in parallel on a multiprocessor.”

Quelle: Go Concurrency Patterns <https://talks.golang.org/2012/concurrency.slide#6>

Definitionen – Synchron und Asynchron

- Synchron

- Auf einen Aufruf folgt immer eine Antwort
- Ohne Antwort geht's nicht weiter

Kellner: „Was darf's sein?“

Gast: „Ich gucke noch.“

Kellner: „Kein Problem, ich warte hier“

- Asynchron

- Auf einen Aufruf folgt nicht unmittelbar eine Antwort
- Die Antwort kann auch später entgegen genommen werden

Kellner: „Was darf's sein?“

Gast: „Ich gucke noch.“

Kellner: „Ok, dann komme ich später wieder“

Definitionen – (Nicht)-Blockieren

- Blockieren

- Das was „langsam“ ist
- Lange Untätigkeit durch warten

Kellner: „Was darf's sein?“

Gast: „Ich gucke noch.“

Kellner: „Kein Problem, ich warte hier“

Synchron ist immer Blockierend – also gleiches Beispiel

- Nicht-Blockieren

- Anstatt zu warten können andere Aufgaben übernommen werden

Kellner: „Was darf's sein?“

Gast: „Ich gucke noch.“

Kellner: „Ok, darf ich Ihnen in der Zwischenzeit etwas zu trinken bringen?“

Warum sollte mich das interessieren?

- W3Techs vermeldet einen Anteil von 97,9% für JS als Client-side-Language [1]
- Statisch gerenderte Webseiten ohne JS sind Rarität geworden [2]
- Alltagsarbeit in der Web-Entwicklung mit JS ist Komposition von
 - Nutzereingaben
 - Laden von Ressourcen
 - Rendern / DOM Manipulations

→ Alles Teil des Laufzeitmodells von JavaScript im Browser

[1] Quelle: https://w3techs.com/technologies/overview/client_side_language

[2] Eigene Einschätzung, keine belastbaren Statistiken

Laufzeitmodell im Browser


Basics

“An execution context is [...] used to track the runtime evaluation of code [...]. At any point in time, there is at most one execution context [...] that is actually executing code.”

Quelle: ECMA-262, 12th edition, June 2021 ECMAScript® 2021 Language Specification
<https://262.ecma-international.org/12.0/#sec-execution-contexts>

Call Stack

Stack



```
function throwError() {  
    throw new Error();  
}  
  
function callErrornousFunction() {  
    console.log("Calling errornous function");  
    throwError();  
}  
  
function printHelloWorld() {  
    console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```

Call Stack

```
function throwError() {  
    throw new Error();  
}
```

→

```
function callErrornousFunction() {  
    console.log("Calling errornous function");  
    throwError();  
}
```

```
function printHelloWorld() {  
    console.log("Hello, World!");  
}
```

```
printHelloWorld();  
callErrornousFunction();
```

Stack

Call Stack

```
function throwError() {  
    throw new Error();  
}  
  
function callErrornousFunction() {  
    console.log("Calling errornous function");  
    throwError();  
}  
  
→ function printHelloWorld() {  
    console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```

Stack

Call Stack

```
function throwError() {  
    throw new Error();  
}  
  
function callErrornousFunction() {  
    console.log("Calling errornous function");  
    throwError();  
}  
  
function printHelloWorld() {  
    console.log("Hello, World!");  
}  
  
→ printHelloWorld();  
   callErrornousFunction();
```

Stack

printHelloWorld

Call Stack

```
function throwError() {  
    throw new Error();  
}  
  
function callErrornousFunction() {  
    console.log("Calling errornous function");  
    throwError();  
}  
  
→ function printHelloWorld() {  
    console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```

Stack

console.log

printHelloWorld

Call Stack

```
function throwError() {  
  throw new Error();  
}  
  
function callErrornousFunction() {  
  console.log("Calling errornous function");  
  throwError();  
}  
  
function printHelloWorld() {  
  console.log("Hello, World!");  
}  
→ printHelloWorld();  
callErrornousFunction();
```

Hello, World!

[VM312:11](#)

Stack

printHelloWorld

Call Stack

```
function throwError() {  
  throw new Error();  
}  
  
function callErrornousFunction() {  
  console.log("Calling errornous function");  
  throwError();  
}  
  
function printHelloWorld() {  
  console.log("Hello, World!");  
}  
  
→ printHelloWorld();  
  callErrornousFunction();
```

Hello, World!

[VM312:11](#)

Stack

callErrornousFunction

Call Stack

```
function throwError() {  
  throw new Error();  
}  
  
→ function callErrornousFunction() {  
  console.log("Calling errornous function");  
  throwError();  
}  
  
function printHelloWorld() {  
  console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```

Hello, World!

[VM312:11](#)

Stack

console.log

callErrornousFunction

Call Stack

```
function throwError() {  
  throw new Error();  
}  
  
function callErrornousFunction() {  
  console.log("Calling errornous function");  
  throwError();  
}  
  
function printHelloWorld() {  
  console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```


Hello, World!	VM391:11
Calling errornous function	VM391:6

Stack

throwError

callErrornousFunction

Call Stack



```
function throwError() {  
    throw new Error();  
}  
  
function callErrornousFunction() {  
    console.log("Calling errornous function");  
    throwError();  
}  
  
function printHelloWorld() {  
    console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```

Hello, World!

[VM391:11](#)

Calling errornous function

[VM391:6](#)

Stack

new Error()

throwError

callErrornousFunction

Call Stack

→

```
function throwError() {  
  throw new Error();  
}  
  
function callErrornousFunction() {  
  console.log("Calling errornous function");  
  throwError();  
}  
  
function printHelloWorld() {  
  console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```

Hello, World! [VM477:11](#)

Calling errornous function [VM477:6](#)

✖ ▶ Uncaught Error [VM477:2](#)
 at throwError (<anonymous>:2:11)
 at callErrornousFunction (<anonymous>:7:5)
 at <anonymous>:15:1

Stack

throwError

callErrornousFunction

Call Stack

```
function throwError() {  
  throw new Error();  
}  
  
function callErrornousFunction() {  
  console.log("Calling errornous function");  
  throwError();  
}  
  
function printHelloWorld() {  
  console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```

Hello, World! [VM477:11](#)

Calling errornous function [VM477:6](#)

✖ ▶ Uncaught Error [VM477:2](#)
 at throwError (<anonymous>:2:11)
 at callErrornousFunction (<anonymous>:7:5)
 at <anonymous>:15:1

Stack

callErrornousFunction

Call Stack

```
function throwError() {  
    throw new Error();  
}  
  
function callErrornousFunction() {  
    console.log("Calling errornous function");  
    throwError();  
}  
  
function printHelloWorld() {  
    console.log("Hello, World!");  
}  
  
printHelloWorld();  
callErrornousFunction();
```

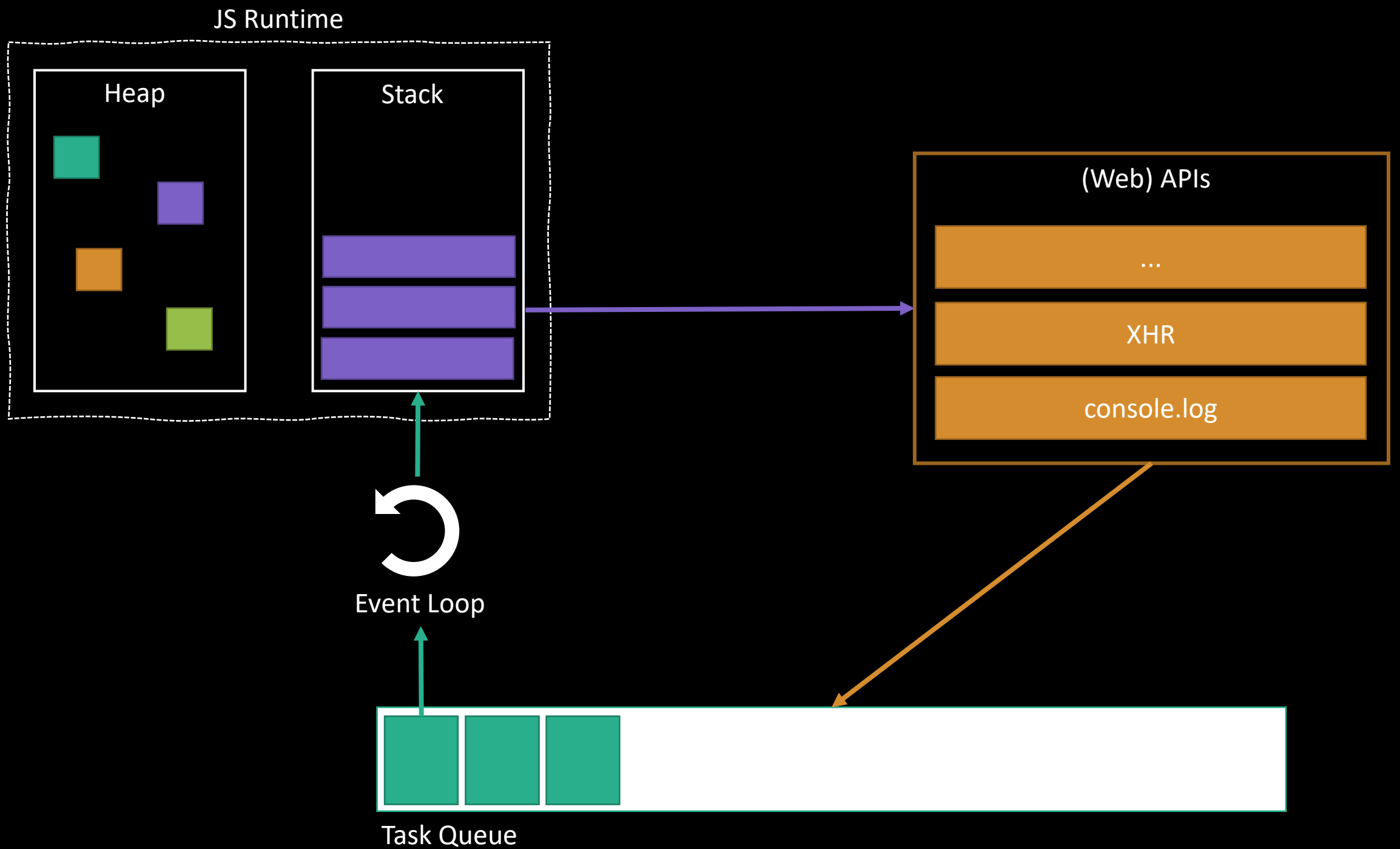


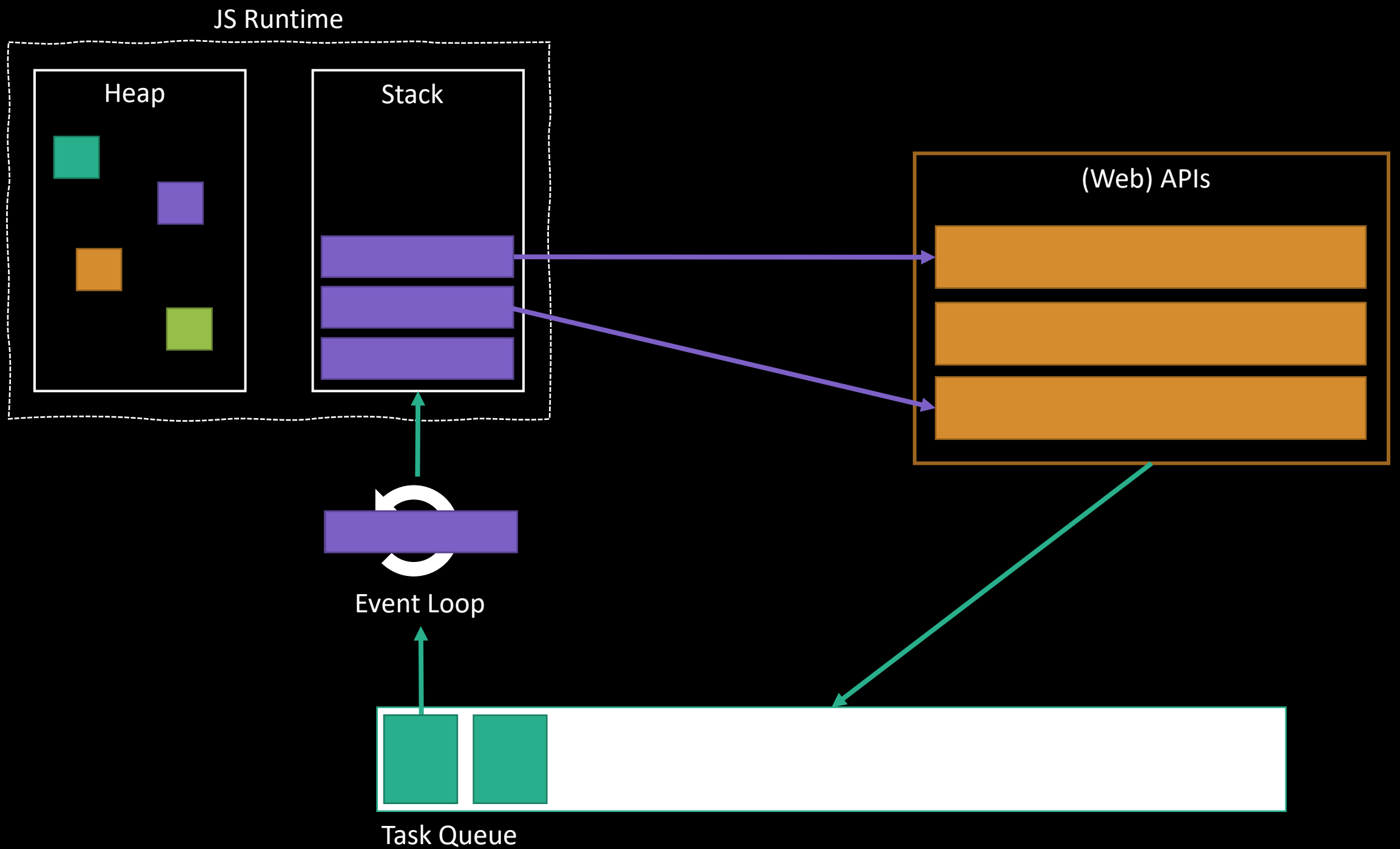
Stack

Basics

“An execution context is [...] used to track the runtime evaluation of code [...]. At any point in time, there is at most one execution context [...] that is actually executing code.”

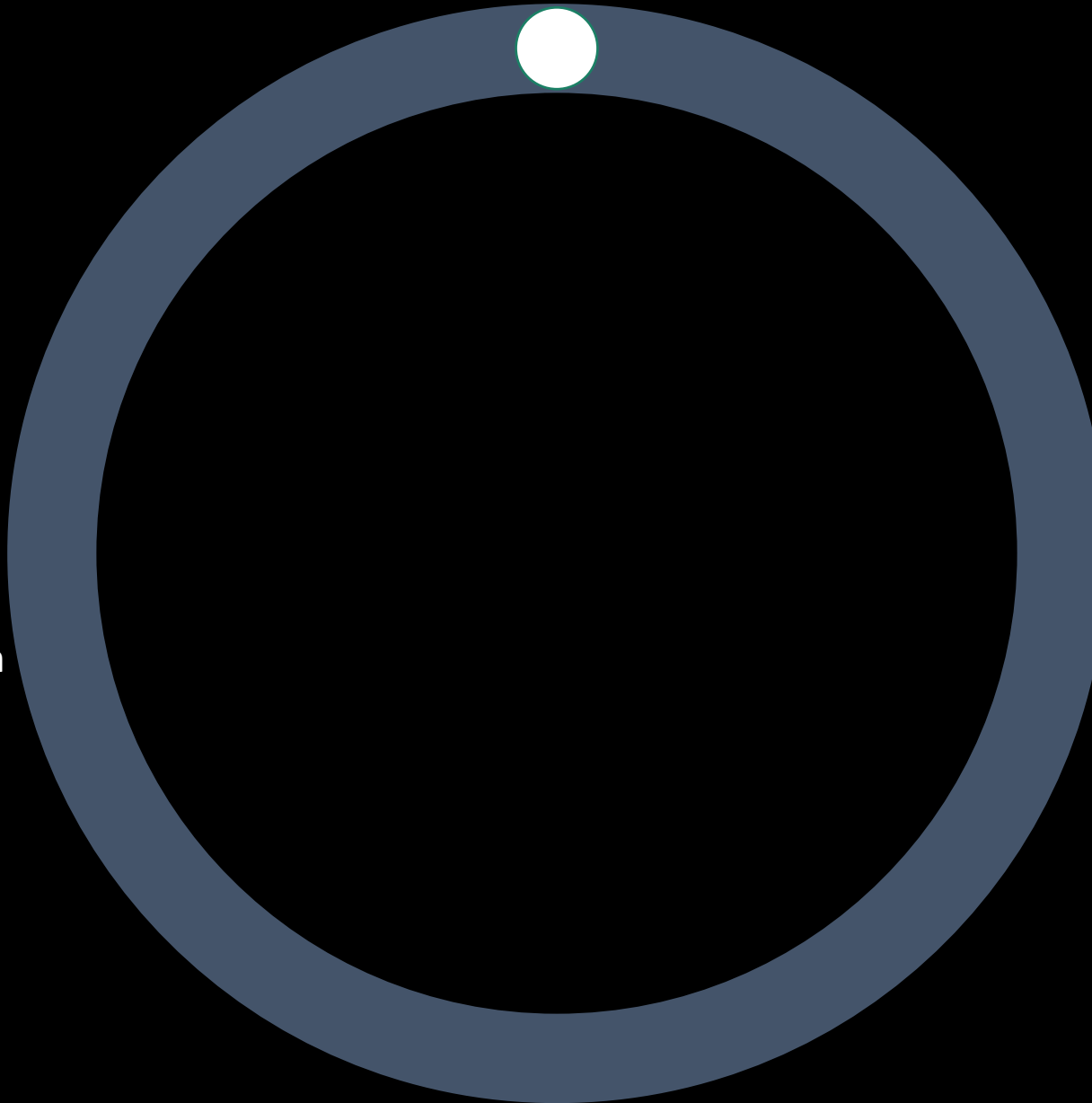
Quelle: ECMA-262, 12th edition, June 2021 ECMAScript® 2021 Language Specification
<https://262.ecma-international.org/12.0/#sec-execution-contexts>







1. Arbeit auf Task Queue?
-> Task bis zum Ende ausführen
2. Weiter im Loop



Update Rendering

Pseudocode Event Loop

```
while(true) {  
    if(taskQueue.hasTasks()) {  
        const task = taskQueue.dequeue();  
        task.execute();  
    }  
    updateRendering();  
}
```

Never block the Event Loop

„Run To Completion“-Semantics

- JavaScript Code wird immer „bis zum Ende“ ausgeführt
 - Egal ob aus einer Queue, Import oder im `<script>` Block
 - Erst danach wird das Rendering aktualisiert
- Gefahr: Blockieren
- Vorteil: Determinismus

```
element.innerText = "Mein Chef ist doof";  
element.hidden = false;  
element.innerText = "Mein Chef ist super";
```

Der Nutzer wird niemals sehen, dass der Chef doof war (oder ist?)

Zusammenfassung Ausführungsmodell

- Call Stack verfolgt Funktionsaufrufe
- Aus JS können Web APIs aufgerufen werden
 - Diese können Asynchron sein
 - Scheduling Antwort auf Task Queue
- Task Queue wird vom Event-Loop abgearbeitet
 - Abgearbeitet = Task wird auf Call Stack transportiert
 - Zwischen den Tasks wird ein Rendering update durchgeführt
- Event Loop arbeitet erst dann, wenn Call Stack leer ist
 - „Run-To-Completion semantics“

Callbacks

Wie kriege ich Dinge auf die Task Queue?

Callbacks

- Funktionen, die als Task registriert werden

```
setTimeout(() => console.log("Hello, World"), 100);
```

```
function printHelloWorld() {  
    console.log("Hello, World");  
}  
  
setTimeout(printHelloWorld, 100);
```

```
element.addEventListener('click', () =>  
    console.log("Hello, World!"));
```

Callbacks – Pitfalls

Callback Hell

```
setTimeout(() => {  
  console.log("Hello");  
  setTimeout(() => {  
    console.log("World");  
    setTimeout(() => {  
      console.log("From");  
      setTimeout(() => {  
        console.log("JavaScript");  
        setTimeout(() => {  
          console.log("!");  
        }, 100);  
      }, 100);  
    }, 100);  
  }, 100);  
}, 100);
```

Blocken im Callback

```
setTimeout(() => {  
  for(let i = 0; i < 1e15, ++i) {}  
}, 100);
```



Um in JS blockierenden Code zu schreiben, muss man sich schon stark bemühen. In der Realität ist das weniger ein Problem, aber wichtig sich im Hinterkopf zu behalten, dass ein Callback nicht magisch Blockierungen löst.

Callbacks – Good Practices

Shallow Code – KISS & DRY

```
function callC() {  
  console.log("C");  
}  
function callB() {  
  console.log("B");  
  setTimeout(callC, 100);  
}  
function callA() {  
  console.log("A");  
  setTimeout(callB, 100);  
}  
  
setTimeout(callA, 0);
```

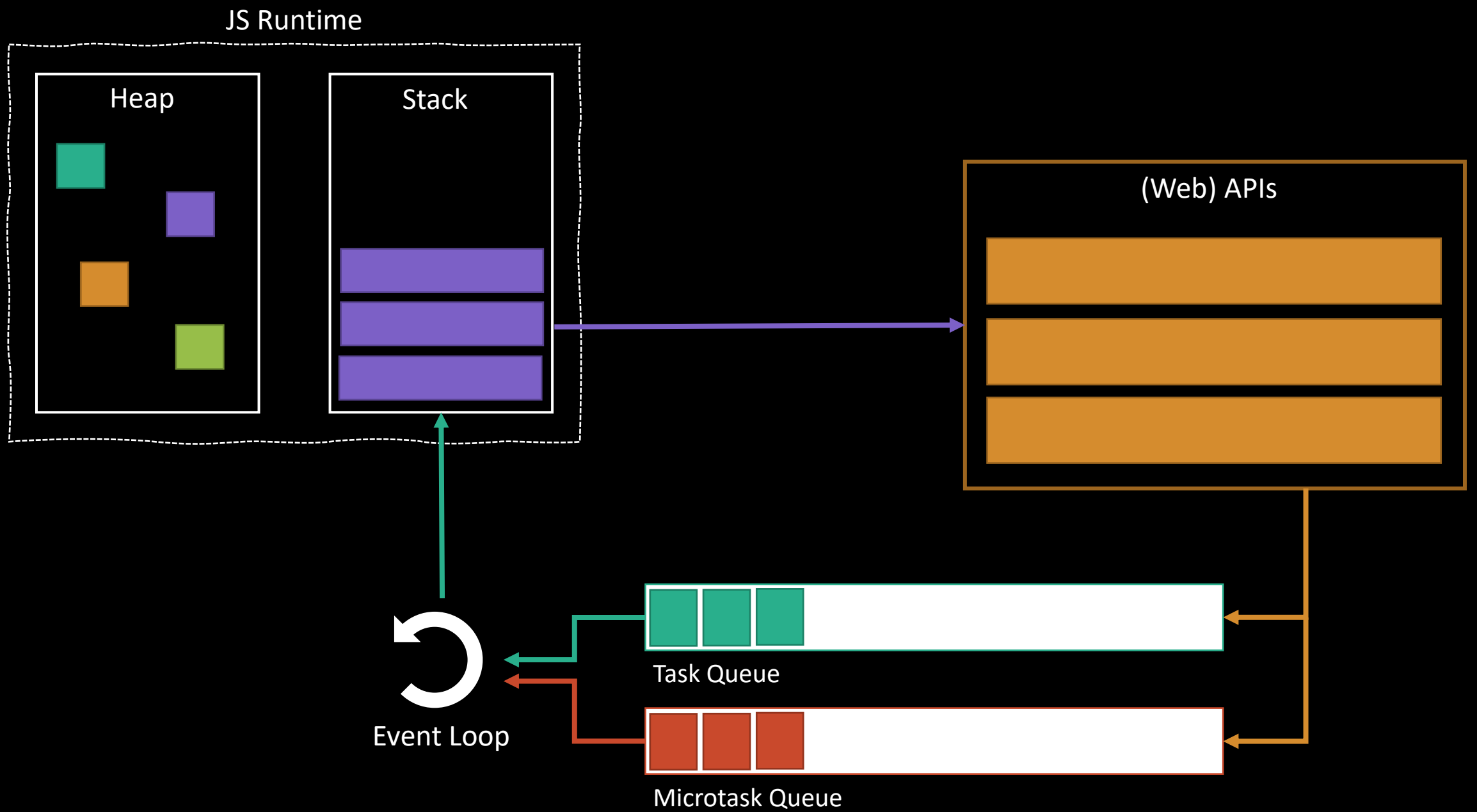
Error-First Callback-Style

```
function scheduleTask(date, callback) {  
  const now = Date.now();  
  if (date < now) {  
    callback(new Error());  
  }  
  
  setTimeout(() => {  
    callback(undefined, Date.now())  
  }, now - date);  
}  
  
scheduleTask(Date.now() - 100,  
  (err, date) => {  
    if(err) {  
      return console.error(err);  
    }  
    return console.log("It is time");  
  });
```

Zeit für eine Übung

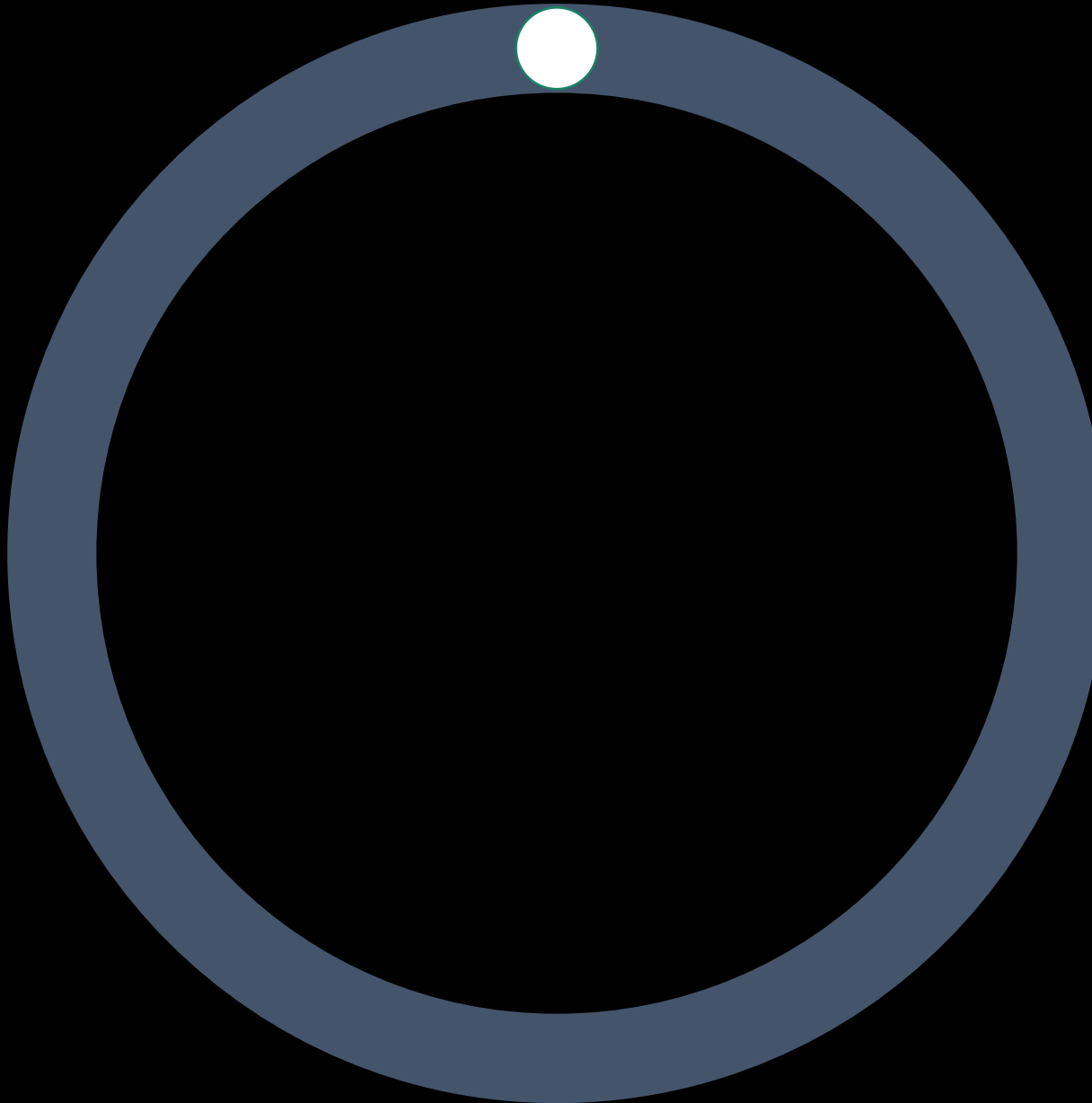
Promises

- In modernen, asynchronen APIs sind Callbacks selten anzutreffen
 - Callbacks dem Idiom nach für Event Handler
 - Auch hier modernere Alternativen (RxJS)
- Stattdessen werden Promises eingesetzt
 - ES6
 - Konzeptuell „Futures“ (auch in anderen Sprachen verbreitet)
 - Asynchroner Arbeitsablauf, der „irgendwann“ abgeschlossen wird
- Warum?
 - Einfache, Cleane Syntax, Erlaubt Shallow Code
 - Vereinfacht Fehlerbehandlung
 - Caching
 - Erlauben mit dem Async-Await-Pattern sequentielle Struktur (später mehr)





1. Arbeit auf Task Queue?
-> Task bis zum Ende ausführen
2. Arbeit auf Microtask Queue?
-> Microtasks ausführen bis Queue leer
3. Weiter im Loop



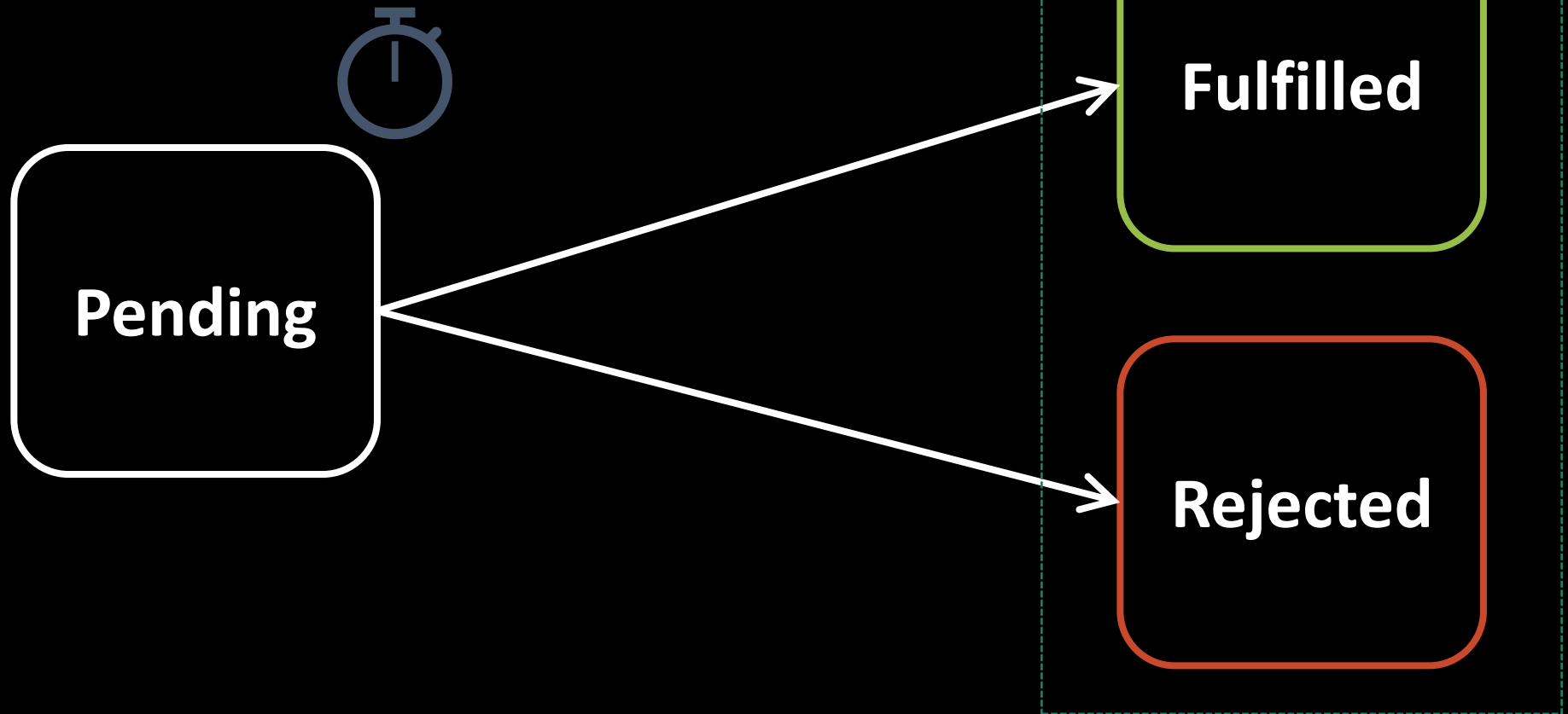
Update Rendering

Task Queue vs. Microtask Queue

- Task Queue
 - Dequeue Task
 - Task wird bis zum Ende ausgeführt
 - Nach Abschluss „läuft“ der Loop weiter
 - Rendering
- Microtask Queue
 - Dequeue Microtask
 - Microtask bis zum Ende ausführen
 - Weiterer Microtask? Repeat.
 - Wenn Queue leer „läuft“ der Loop weiter
 - Rendering

```
while(true) {  
  // Task Queue  
  if(taskQueue.hasTasks()) {  
    const task = taskQueue.dequeue();  
    task.execute();  
  }  
  // Microtask Queue  
  while(μTaskQueue.hasTasks()) {  
    const μTask = μTaskQueue.dequeue();  
    μTask.execute();  
  }  
  updateRendering();  
}
```


Promises



Promises

Konstruktor

```
new Promise((resolve, reject) => {  
  if(somethingWentWrong) {  
    reject(new Error());  
  }  
  resolve(value);  
});
```

- Der Konstruktor-Aufruf ist weiterhin synchron
- Erst mit resolve/reject startet die Asynchronität

Statische Methoden

```
Promise.resolve(value);  
Promise.reject(new Error());
```

- An **Promise.resolve** kann auch eine Promise übergeben werden.

```
Promise.resolve(  
  Promise.resolve(value)  
);  
Promise.resolve(  
  Promise.reject(new Error())  
);
```

Promise-Chaining

```
Promise.resolve(value)
```

```
.then(v => v * v)
```

Das Ergebnis der Promise wird in eine neue Promise transformiert.

```
.then(v => {
```

```
  if (v == 0) {
```

```
    throw new Error("Cannot divide by 0");
```

Eine Exception resultiert in eine rejected Promise

```
  }
```

```
  return v / 0;
```

```
})
```

```
.then(v => {
```

```
  return Promise.resolve(v - 1);
```

Keine Transformation. Diese Promise wird zurück gegeben.

```
})
```

```
.catch(err => console.error(err))
```

Rejections aus der Promise-Chain werden weitergeleitet

```
.finally(() => {
```

```
  connection.close();
```

Wird ausgeführt, wenn Promise settled ist

```
});
```

Promisify Callbacks

```
function promisedTimeout(ms) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve();  
    }, ms);  
  });  
}
```

```
new Promise((resolve, reject) => {  
  document.addEventListener("click", resolve);  
})  
  .then(event => console.log(event));
```

← Problem: Caching
(Fulfilled nur beim ersten click event)

Promise Caching

```
const remoteResource = fetch("https://httpbin.org/json")
  .then(result => {
    console.log("Fetched result");
    return result;
  });

const promiseA = remoteResource
  .then(result => {
    console.log("Result is in Promise A");
  });

const promiseB = remoteResource
  .then(result => {
    console.log("Result is in Promise B");
  });
```

Fetched result	VM39:3
Result is in Promise A	VM39:8
Result is in Promise B	VM39:12

>

Promise Pitfalls



Diese Pitfalls treten nur im Promise Konstruktor auf

- Never Settling Promise

```
new Promise(() => {});
```

```
new Promise(() => {  
  return 1;  
});
```

Eine Exception im Konstruktor
wird trotzdem zu einer rejected
Promise

- Never Fulfilled Promise

```
new Promise((resolve, reject) => {  
  if(someCondition) {  
    reject(new Error());  
  }  
  return true;  
});
```

```
new Promise((resolve, reject) => {  
  if(someCondition) {  
    throw new Error();  
  }  
  return true;  
});
```

Promise Pitfalls

- Exceptions mit Promises mischen

```
function funcThatThrows() {  
  throw new Error();  
}  
function funcThatReturnsAPromise() {  
  funcThatThrows();  
  return Promise.resolve(42);  
}  
  
// Funktionsaufruf wirft Exception  
funcThatReturnsAPromise()  
  .catch(err => console.error(err)); // Catch greift nicht
```

Promise Pitfalls

Besser:

```
function funcThatThrows() {
  throw new Error();
}
function funcThatReturnsAPromise() {
  return Promise.resolve()
    .then(() => funcThatThrows())
    .then(() => 42);
}

funcThatReturnsAPromise()
  // Catch greift
  .catch(err => console.error(err));
```

```
function funcThatThrows() {
  throw new Error();
}
function funcThatReturnsAPromise() {
  try {
    funcThatThrows();
    return Promise.resolve(42);
  }
  catch(err) {
    return Promise.reject(err);
  }
}

funcThatReturnsAPromise()
  // Catch greift
  .catch(err => console.error(err));
```


Promise Good Practices

- Kombinatoren verwenden

```
const promiseA = () => Promise.resolve("a");
const promiseB = () => Promise.resolve("b");

// Alle resolved
Promise.all([promiseA(), promiseB()]);
// Eine resolved
Promise.any([promiseA(), promiseB()]);
// Alle settled
Promise.allSettled([promiseA(), promiseB()]);
// Das erste Settling
Promise.race([promiseA(), promiseB()]);
```

Promise Good Practices

```
const promiseA = () => Promise.resolve("a");  
const promiseB = (a) => Promise.resolve(`B: ${a}`);
```

- Nested Chain (Zurück zur Callback Hell)

```
promiseA()  
  .then(res => {  
    return promiseB(res)  
      .then(res2 => {  
        console.log(res);  
      });  
  });
```

- Shallow Chaining

```
promiseA()  
  .then(res => {  
    return promiseB(res);  
  })  
  .then(res => {  
    console.log(res);  
  });
```

Zeit für eine Übung

Async / Await Pattern

- ES7 Feature
 - Erlaubt es asynchronen Code zu schreiben, der wie synchroner Code aussieht
 - Syntactic sugar
 - Gängiges Pattern auch in anderen Sprachen (C#, Kotlin, Rust, ...)
- Asynchrone Funktionen werden mit **async** gekennzeichnet
 - Returnen immer eine Promise
 - Exception -> Rejection
 - Return -> Fulfillment (auch implizites return)
 - **await** wartet auf eine Promise
 - Ausführung wird unterbrochen
 - Aber blockiert nicht

Async / Await Pitfalls

- Async Funktionen sind bis `await`, `throw` oder `return` synchron

```
async function testSynchronousCalls() {  
  console.log("Executed in sync");  
  return Promise.resolve("Executed async");  
}
```

```
console.log("Started in sync");  
testSynchronousCalls()  
  .then(res => console.log(res));  
console.log("Ended in sync");
```

Started in sync

[VM64:6](#)

Executed in sync

[VM64:2](#)

Ended in sync

[VM64:9](#)

Executed async

[VM64:8](#)

Async / Await Good Practices

- Kombinatoren verwenden

```
const promiseA = () => fetch("//foo");
const promiseB = () => fetch("//bar");

async function printResults() {
  // Ausführung wird unterbrochen und auf promiseA gewartet
  const resultA = await promiseA();
  const resultB = await promiseB();

  // Besser: Ausführung beginnt nebenläufig
  const [resultA, resultB] = await Promise.all([
    promiseA(),
    promiseB()
  ]);
}
```

Async / Await Good Practices

- `async` und `await` da einsetzen, wo sie gebraucht werden

Warum hier await'en?

```
async function getSomething() {  
  return await fetch("//foo");  
}  
  
async function printSomething() {  
  const something = await getSomething();  
  console.log(something);  
}
```

Hier wird Something doch erst benötigt.

Wir erinnern uns: Eine async function retutnt immer eine Promise. Warum also nicht direkt die „richtige“ Promise durchreichen und async sparen?

```
function getSomething() {  
  return fetch("//foo");  
}  
  
async function printSomething() {  
  const something = await getSomething();  
  console.log(something);  
}
```

Zeit für eine Übung

Ein letztes gemeinsames Beispiel (Real-World)

```
canActivate(route, state) {  
  return new Promise(async (resolve, reject) => {  
    try {  
      this.authenticated = await this.keycloakAngular.isLoggedIn();  
      this.roles = await this.keycloakAngular.getUserRoles(true);  
      const result = await this.isAccessAllowed(route, state);  
      resolve(result);  
    } catch (error) {  
      reject(new Error('An error happened. Details:' + error));  
    }  
  }  
}
```

Wie geht es von hier aus
weiter?

Asynchrone Generatoren

```
async function* asyncResponseGenerator() {  
  const codes = [200, 201, 400, 401, 403, 404]  
  for (const code of codes) {  
    yield fetch(`https://httpbin.org/${code}`);  
  }  
}
```

```
const iterable = asyncResponseGenerator();  
for await (const response of iterable) {  
  console.log(response);  
}
```

Ausblick: ES2022

Vorher:

```
import fetch from "node-fetch";
export default (async () => {
    const resp = await fetch('https://jsonplaceholder.typicode.com/users');
    users = resp.json();
})();
export { users };
```

ES2022 Top-Level-Await:

```
const resp = await fetch('https://jsonplaceholder.typicode.com/users');
const users = resp.json();
export { users };
```

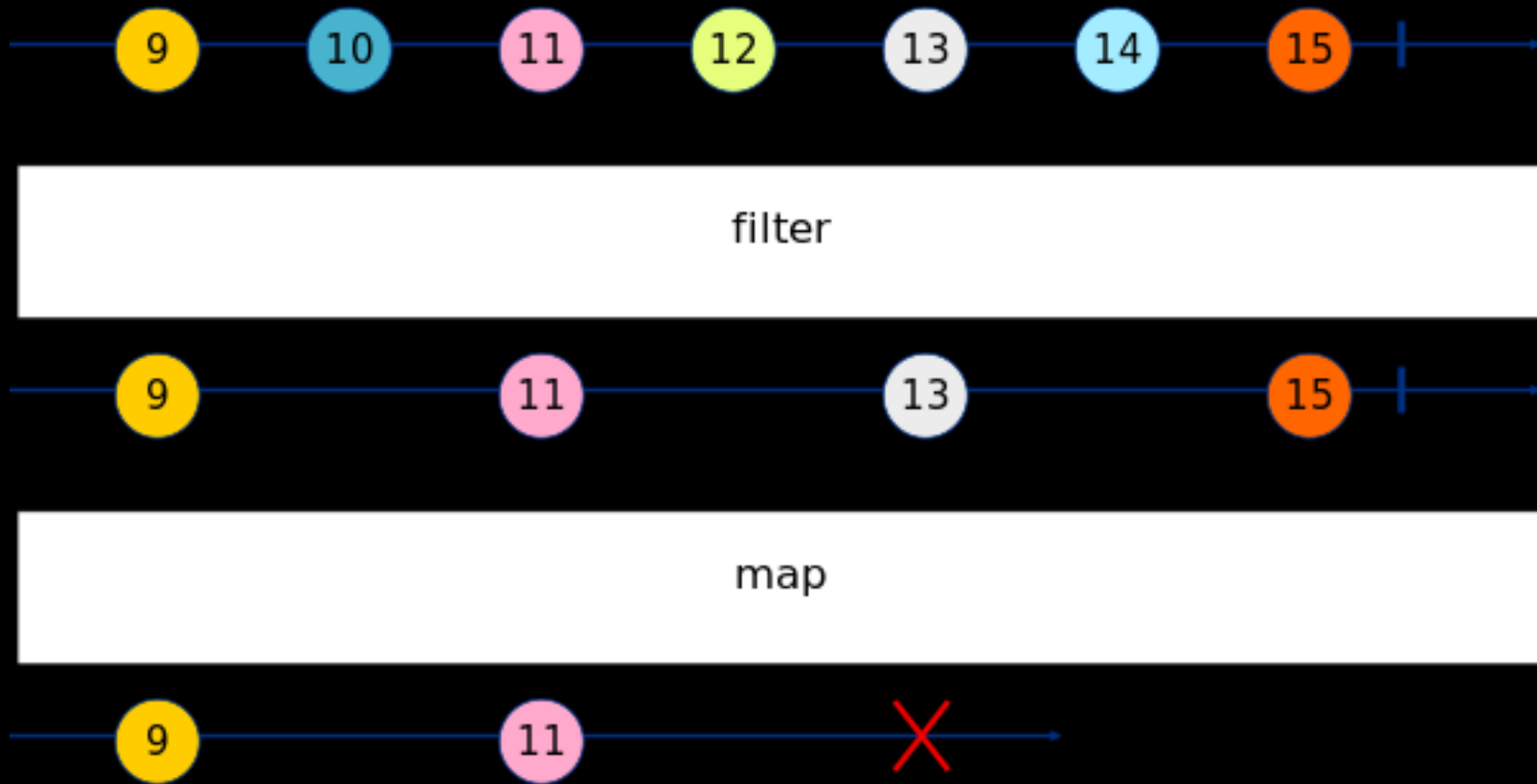
RxJS

```
const documentClick$ = fromEvent(document, "click")
  .pipe(
    debounceTime(1000),
    map(event => event.target),
    filter(target => !!target)
  );

documentClick$.subscribe(target => {
  target.innerHTML = "<h1>Hello, World!</h1>");
});

documentClick$.subscribe(() => {
  console.log("Document has been clicked");
});
```

RxJS




Ergänzendes Material


- Linter Rules:
 - [@typescript-eslint/no-floating-promises](#)
 - [@typescript-eslint/no-misused-promises](#)
 - [no-async-promise-executor](#)
 - [prefer-promise-reject-errors](#)
 - [eslint-plugin-promise](#)
 - [node/handle-callback-err](#)
 - [node/no-callback-literal](#)
 - [node/callback-return](#)

Ergänzendes Material

- Talks


- [What the heck is the event loop anyway? | Philip Roberts | JSConf EU – YouTube](#) 
- [Jake Archibald: In The Loop - JSConf.Asia – YouTube](#)
- [Asynchrony: Under the Hood - Shelley Vohr - JSConf EU - YouTube](#)

- Sonstiges

- Visualisierungstools für Task Queue, Microtask Queue, Callstack und Event Loop
 - [Loupe](#)
 - [JavaScript Visualizer 9000](#) 
 - [JELoop Visualizer](#)

Ergänzendes Material

- Literatur

- [JavaScript for impatient programmers \(ES2022 edition\) \(exploringjs.com\)](https://exploringjs.com) 
- [Asynchronous Programming :: Eloquent JavaScript](#)
- [Debugging Asynchronous JavaScript with Chrome DevTools - HTML5 Rocks](#)
- [JavaScript Callbacks are Pretty Okay - Andrew Kelley](#)
- [Callbacks are imperative, promises are functional: Node's biggest missed opportunity – The If Works \(jcoglan.com\)](#)

- Standards

- [HTML Standard \(whatwg.org\)](https://whatwg.org)
- [ECMA-262 - Ecma International \(ecma-international.org\)](https://ecma-international.org)

Ziel erreicht?

- Verständnis für das Laufzeitmodell von JavaScript im Browser
- Verständnis zu synchron und asynchronen Funktionen
- Verständnis zu konkreten Modellen
 - Callbacks
 - Promises
 - Async / Await