CS 4530: Fundamentals of Software Engineering

Module 06: Concurrency Patterns in Typescript

Adeel Bhutta, Joydeep Mitra and Mitch Wand

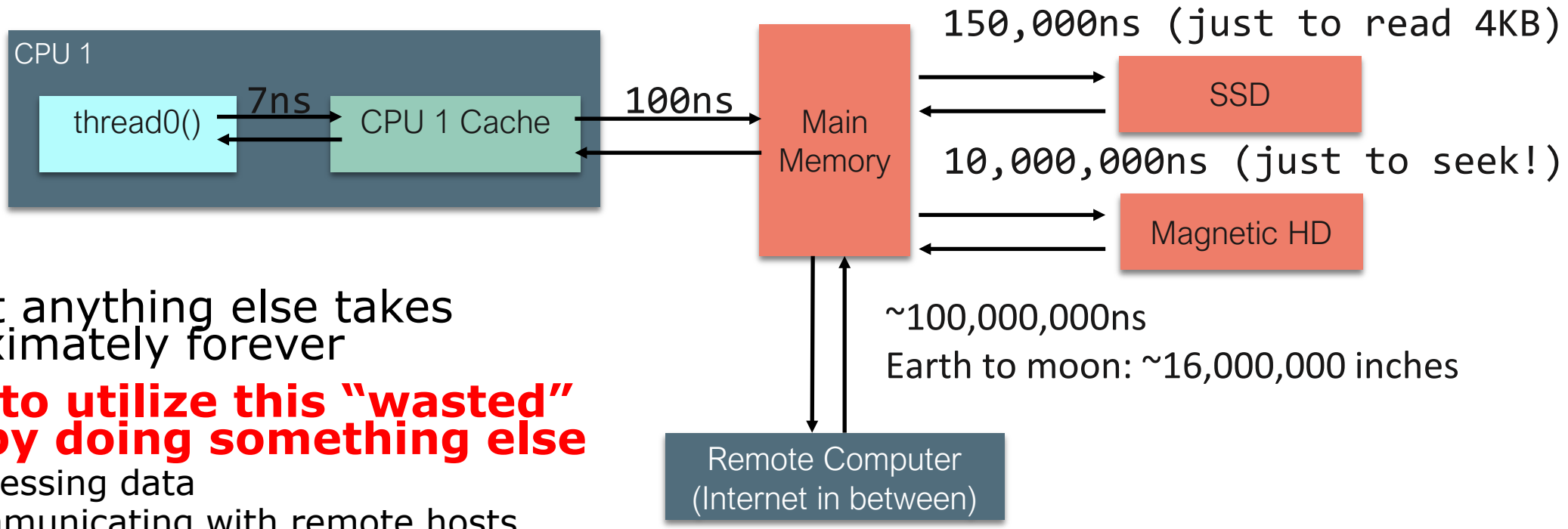
Khoury College of Computer Sciences

Learning Goals for this Lesson

- At the end of this lesson, you should be prepared to:
  - Explain the difference between JS run-to-completion semantics and interrupt-based semantics.
  - Given a simple program using async/await, work out the order in which the statements in the program will run.
  - Write simple programs that create and manage promises using async/await
  - Write simple programs to mask latency with concurrency by using non-blocking IO and Promise.all in TypeScript.

Your app probably spends most of its time waiting

- Consider: a 1Ghz CPU executes an instruction every 1 ns



- Almost anything else takes approximately forever
- **Want to utilize this "wasted" time by doing something else**
  - Processing data
  - Communicating with remote hosts
  - Timers that countdown while our app is running
  - Echoing user input

We achieve this goal using two techniques:

1. cooperative multiprocessing

2. non-blocking IO

Most OS's use **pre-emptive multiprocessing**

- OS manages multiprocessing with multiple threads of execution
- Processes may be ***interrupted*** at unpredictable times
- Inter-process communication by shared memory
- Data races abound
- Really, really hard to get right: need critical sections, semaphores, monitors (all that stuff you learned about in op. sys.)

Javascript/Typescript uses **cooperative multiprocessing**

- In cooperative multiprocessing, only one process is executed at a time.
- Each process pauses when it is convenient to allow other processes to make progress.
- To make this practical (and avoid cheating!), we need a programming model that encourages this behavior

async/await: a programming model for cooperative multiprocessing

- In async/await, the program is organized into a set of "async functions".
- An async function is like an ordinary function, except that it will pause at two well-defined points in its execution. We will show you those with an example.
- When one program pauses, the runtime can choose to resume executing any process that is ready to run.

A typical async function

```
async function someFunction(i: number) {
    const j = i + 1;
    // ...
    const k = await someOtherAsyncFunction(j);
    // ...
    const m = k + 100
    return m;
}
```

An async function can only pause in two places

```
async function someFunction(i: number) {
    const j = i + 1;
    // ...
    const k = await someOtherAsyncFunction(j);
    // ...
    const m = k + 100;
    return m;
}
```

It will never pause in here

Or in here

src/data-races/dataRace2.ts

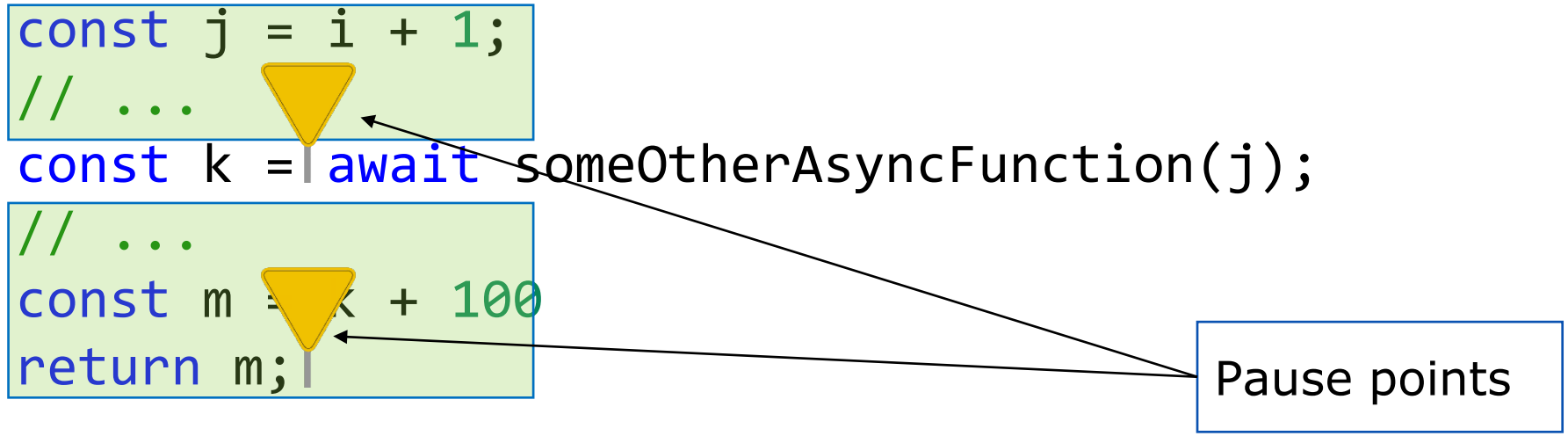
This is not Java!

```
async function asyncPrintA() {
    await waitRandom(1);
    console.log('A');
}


async function asyncPrintBC() {
    await waitRandom(2);
    console.log('B');
    console.log('C');
}


async function run() {
    await Promise.all([asyncPrintA(), asyncPrintBC()])
}
```

- In Java, you could get an interrupt between the print B and the print C.
- So the output would be BAC.
- In TS/JS, the print B and print C are in the same critical section, so BAC is impossible!

What happens at those pause points?

```
async function someFunction(i: number) {
    const j = i + 1;
    // ...
    const k = await someOtherAsyncFunction(j);
    // ...
    const m = k + 100
    return m;
}
```



Terminology: promises and run-to-completion

- Each uninterruptible unit of work is called a "promise"
- The pattern we've just talked about is called "run-to-completion" semantics, because a pause point corresponds exactly to the end of one of these units of work
- You can do lots of different things with promises.
- Let's look some typical patterns.

Example:

src/XXXXX/oneRequest.ts

```
// fakeRequest(n) is an async that waits for 1 second and then
// resolves with the number n+10
import { fakeRequest } from "./fakeRequest";
import { timeIt } from "./timeIt";

async function main() {
    console.log('main started');
    const request = 32
    const res = await fakeRequest(request);
    console.log(`fakeRequest(${request}) returned: ${res}`);
    console.log('main done');
}

timeIt(main)
```

```
$ npx ts-node oneRequest.ts
main started
fakeRequest received request: 32
time passes....
fakeRequest(32) returned: 42
main done
1015.98 msec
```

Use Promise.all to execute several requests concurrently

```
async function main() {
    console.log('starting main');
    const promises = [fakeRequest(1),
                      fakeRequest(2),
                      fakeRequest(3)];
    const results = await Promise.all(
    console.log('results:', results);
    console.log('main done');
}

timeIt(main)
```

```
$ npx ts-node
threeRequestsConcurrently.ts
starting main
fakeRequest received request: 1
time passes....
fakeRequest received request: 2
time passes....
fakeRequest received request: 3
time passes....
results: [ 11, 12, 13 ]
main done
1018.81 msec
```

If you add awaits, the requests will be processed sequentially

```
async function main() {
    console.log('starting main');
    const res1 = await fakeRequest(1);
    console.log(`fakeRequest(1) returned: ${res1}`);
    const res2 = await fakeRequest(2);
    console.log(`fakeRequest(2) returned: ${res2}`);
    const res3 = await fakeRequest(2);
    console.log(`fakeRequest(2) returned: ${res3}`);
    console.log('main done');


}

timeIt(main)
```

```
$ npx ts-node threeRequestsSequentially.ts
starting main
fakeRequest received request: 1
time passes....
fakeRequest(1) returned: 11
fakeRequest received request: 2
time passes....
fakeRequest(2) returned: 12
fakeRequest received request: 3
time passes....
fakeRequest(3) returned: 13
main done
3024.03 msec
```

src/XXXXX/threeRequestsSequentially.ts

…but it would be much slower

```
$ npx ts-node timeComparison.ts
After 100 runs of length 10
makeRequestsConcurrently: min = 23  avg = 34 max = 190 milliseconds
makeRequestsSerially    : min = 210  avg = 237 max = 812 milliseconds
```

Need to redo with this semester's examples

Why is that?
Visualizing Promise.all

Sequential (await)

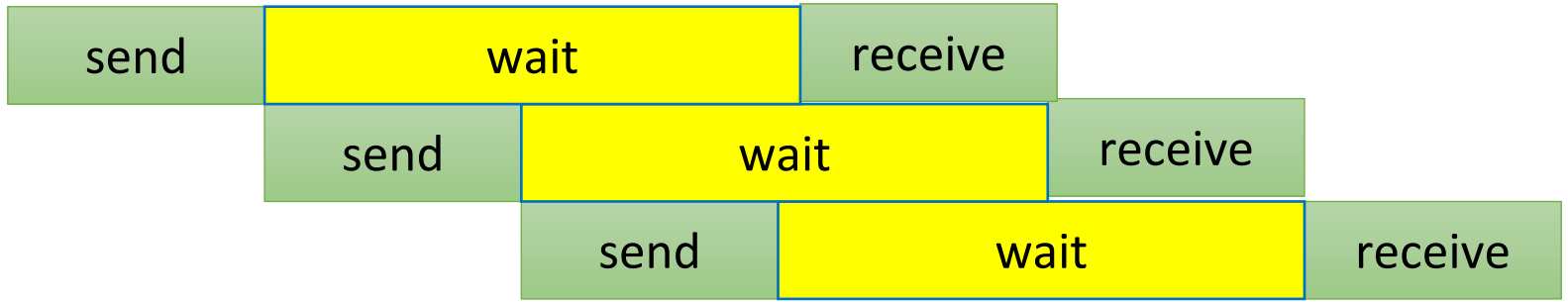
"Don't make another request until you got the last response back"

237 msec

Concurrent (Promise.all)

"Make all of the requests now, then wait for all of the responses"

34 msec



Requests can also be chained (if they are serial)

```
async function main() {
    console.log('main started');
    const request1 = 32
    const res1 = await fakeRequest(request1);
    console.log(`fakeRequest(${request1}) returned: ${res
    const res2 = await fakeRequest(res1);
    console.log(`fakeRequest(${res1})returned: ${res2}`);
    const res3 = await fakeRequest(res2);
    console.log(`fakeRequest(${res2})returned: ${res3}`);
    console.log([request1, res1, res2, res3]);
    console.log('main done');
}
```

```
$ npx ts-node
threeRequestsChained.ts
main started
fakeRequest received request: 32
time passes....
fakeRequest(32) returned: 42
fakeRequest received request: 42
time passes....
fakeRequest(42)returned: 52
fakeRequest received request: 52
time passes....
fakeRequest(52)returned: 62
[ 32, 42, 52, 62 ]
main done
3080.99 msec
```

Recover from errors with try/catch

```
// a request that may fail
async function maybeFailingRequest(req: number): Promise<number> {
    const res = await fakeRequest(req);
    if (res < 0) {
        throw new Error(`Request ${req} failed because response ${res} < 0`);
    } else {
        return res;
    }
}
```

try/catch, continued

```
async function main() {
    console.log('main started');
    const req1 = -32
    let res: number;
    try {
        res = await maybeFailingRequest(req1);
        console.log(`fakeRequest(${req1}) returned: ${res}`);
    } catch (err) {
        console.error(`Error occurred for request ${req1}`);
        res = 0
    }
    console.log('main done with res =', res);
}

timeIt(main);
```

```
$ npx ts-node tryCatchExample.ts
main started
fakeRequest received request: -32
time passes....
Error occurred for request -32
main done with res = 0
1007.52 msec
```

Pattern for testing an async function

```
test('fakeRequest should return its argument + 10', async () => {
    expect.assertions(1)
    await expect(fakeRequest(33)).resolves.toEqual(43)
})

// this will succeed, because it does not await the promise
test('bogus test', async () => {
    // expect.assertions(1)
    expect(fakeRequest(33)).resolves.toEqual(99)
})
```

src/XXX/jest-example.test.ts

src/XXXXX/oneRequestNoAwait.ts

AntiPattern 1: un**await**ed promise

```
// fakeRequest(n) is an async that waits for 1 second and then
// resolves with the number n+10
import { fakeRequest } from "./fakeRequest";
import { timeIt } from "./timeIt";

async function main() {
    console.log('main started');
    const request = 32
    const res = fakeRequest(request);
    console.log(`fakeRequest(${reques
    console.log('main done');
}

timeIt(main)
```

```
$ npx ts-node oneRequestNoAwait.ts
main started
fakeRequest(32) returned: [object Promise]
main done
2.64 msec
fakeRequest received request: 32
time passes....
```

What just happened?

```
$ npx ts-node oneRequestNoAwait.ts
main started
fakeRequest(32) returned: [object Promise]
main done
2.64 msec
fakeRequest received request: 32
time passes....
```

1. main() called fakeRequest(32).
2. fakeRequest(32) created a unit of work (a Promise), and told the runtime to run it sometime or other.
3. Normally, we wouldn't see the actual value returned by fakeRequest(32), because we'd just wait for the unit of work to run before proceeding.
4. But here, we didn't wait-- we just took the value returned by fakeRequest(32)-- the Promise-- and printed it.
5. We finished our current unit of work, printing "main done", which informed the runtime that we were done.
6. The runtime then looked around for another unit of work to do. In this case, it found the unit of work created by fakeRequest(32), and ran it, printing the last two lines

Wow! That was complicated!

- We try to make our code easy to understand.
- That's why it's an antipattern.
- Luckily, in real code we don't need to do this very often

AntiPattern1a: async with no await

```
async function main() {
    console.log('main started');
    const request = 32
    const res = fakeRequest(request);
    console.log(`fakeRequest(${request}) returned: ${res}`);
    console.log('main done');
}
```

AntiPattern 2: Side-effect before **await**

```
async function f() {
    console.log('f started');
    await g();
    console.log('f done');
}

async function g() {
    console.log('g started');
    const res = await fakeRequest(32);
    console.log(`fakeRequest(32) returned:
${res}`);
}
```

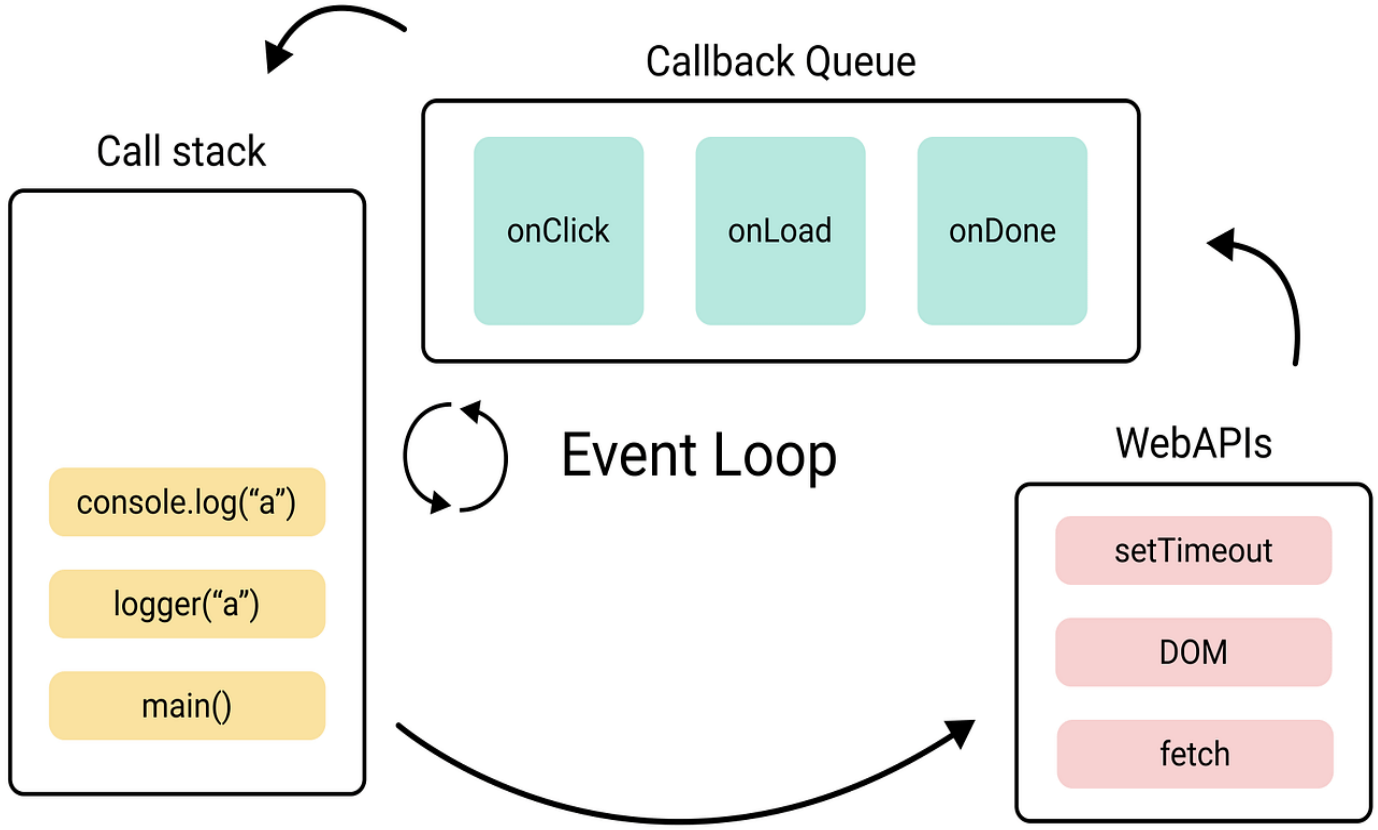
Critical Section #1: no interruption between 'f started' and 'g started'

Critical Section #2

Critical Section #3

How does JS Engine make this happen?

- One Event Loop means that we have single thread of execution
- WebAPI are used for asynchronous tasks
- Queues are used for “await”-ing tasks
- When call stack gets empty, event loop picks up tasks from Callback Queue



But where does the non-blocking IO come from?

We achieve this goal using two techniques:

1. cooperative multiprocessing

2. non-blocking IO 

Answer: JS/TS has some primitives for starting a non-blocking computation

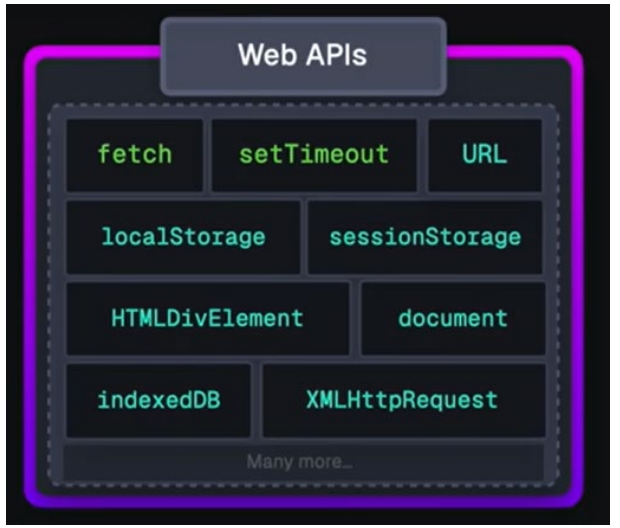
- These are things like http requests, I/O operations, or timers.
- Each of these returns a promise that you can **await**. The promise runs while it is pending, and produces the response from the http request, or the contents of the file, etc.
- You will hardly ever call one of these primitives yourself; usually they are wrapped in a convenient procedure, e.g., we write

```
axios.get('https://rest-example.covey.town’)
```

to make an http request, or

```
fs.readFile(filename)
```

to read the contents of a file.



Pattern for starting a concurrent computation using non-blocking I/O

```
export async function makeRequest(requestNumber:number) {
    console.log(`starting makeRequest(${requestNumber})`);
    const response = await axios.get('https://rest-example.covey.town');
    console.log('request:', requestNumber, '\nresponse:', response.data);
}
```

1. The first console.log is printed
2. The http request is sent, using non-blocking i/o
3. A promise is created to run the second console.log *after* the axios.get returns
4. The makeRequest() returns to its caller.

Pattern for starting a concurrent computation using non-blocking I/O

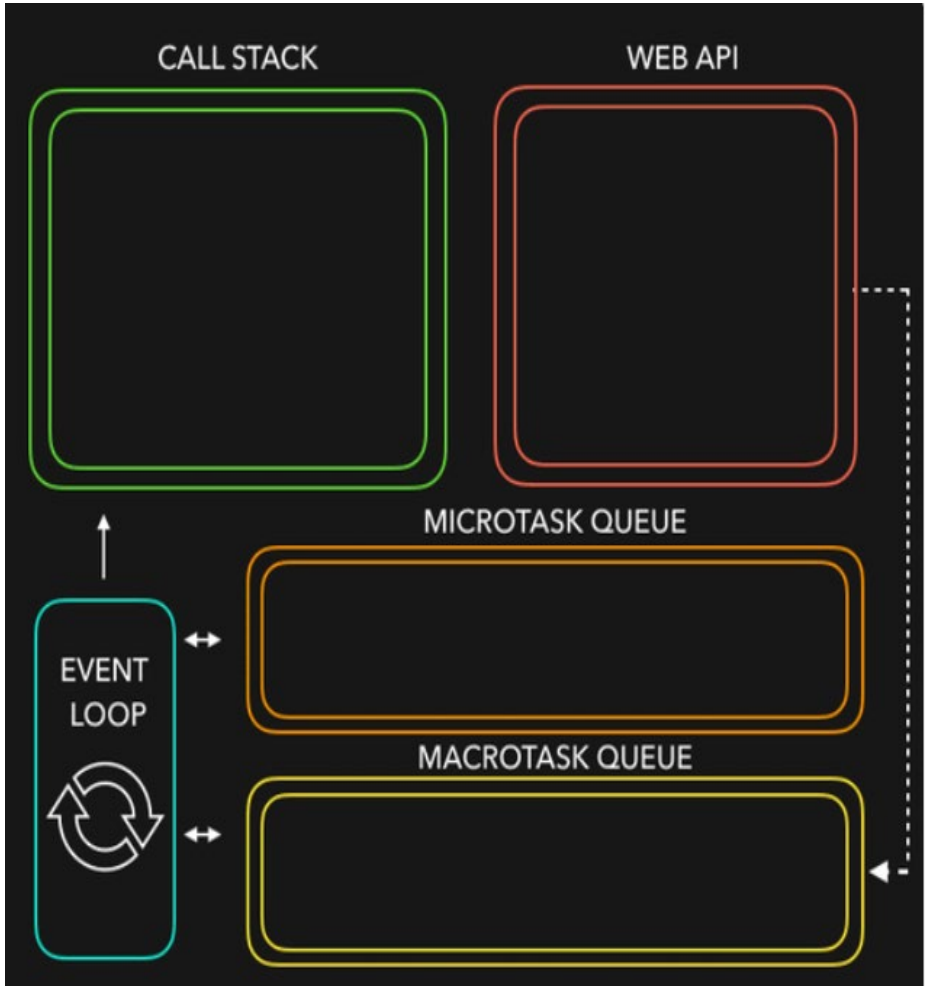
Maybe better?

```
export async function makeRequest(requestNumber:number) {
    console.log(`starting makeRequest(${requestNumber})`);
    const response = await axios.get('https://rest-example.covey.town');
    console.log('request:', requestNumber, '\nresponse:', response.data);
}
```

1. The first console.log is printed
2. The http request is sent, using non-blocking i/o
3. The browser goes about its business
4. Eventually, the axios.get returns.
5. Some time after that, the console.log is printed and the makeRequest concludes.
6. Any promises that are waiting for the result of makeRequest become eligible for execution.

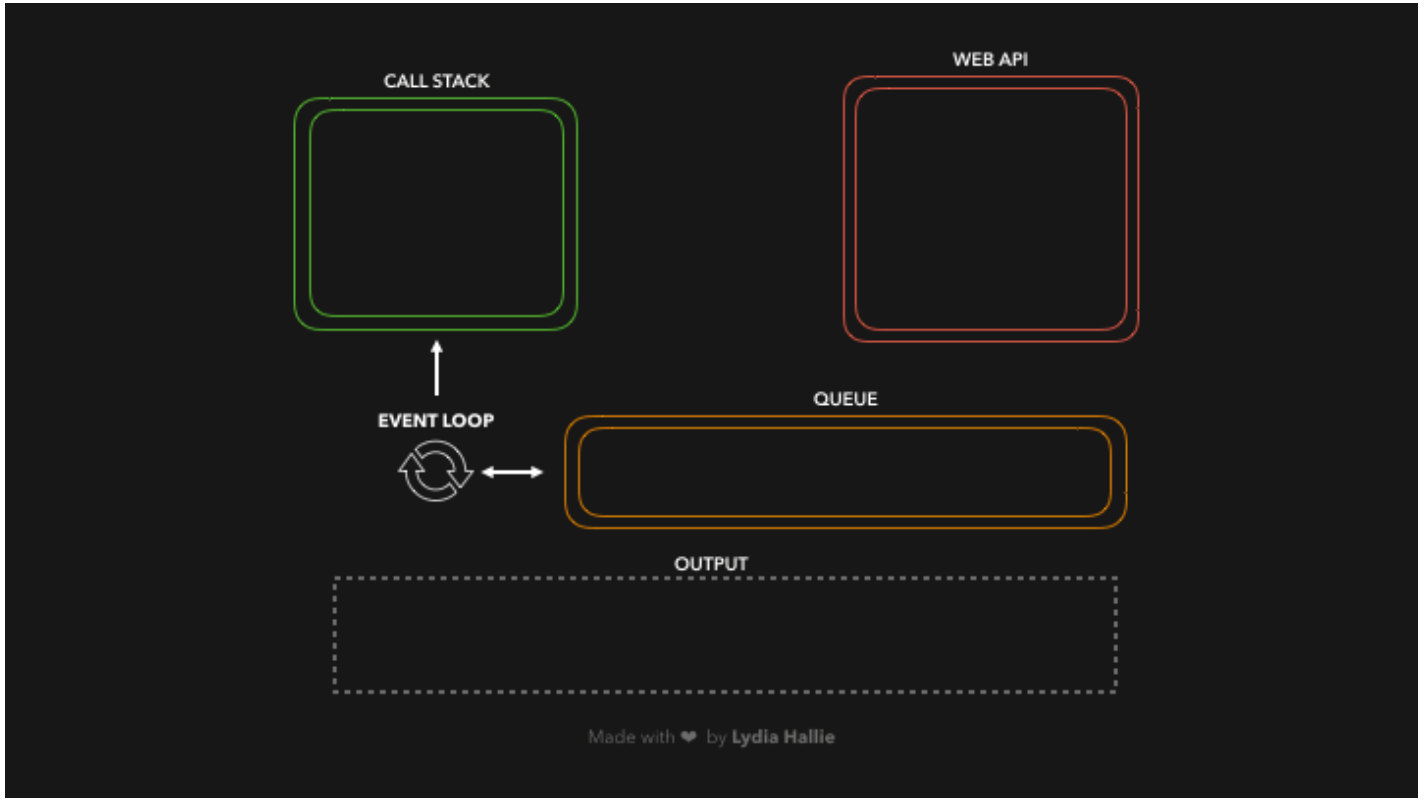
Let’s put it all together

- JS/TS has single event loop
- We outsource most of the non-blocking IO work (to WebAPIs) for asynchronous work
- Upon completion, they are placed in queues (Microtask queue has priority over Macrotask queue)
- Event loop picks them up from queue when call stack is empty!



Here is a quick demo for you

```
const foo = () => console.log("First");
const bar = () => setTimeout(() => console.log("Second"), 500);
const baz = () => console.log("Third");

bar();
foo();
baz();
```



Courtesy of https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif

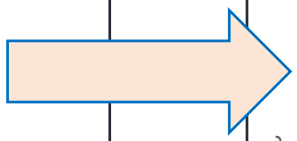
General Rules for Writing Asynchronous Code

- You can’t return a value from a promise to an ordinary procedure.
  - You can only send the value to another promise that is awaiting it.
- Call async procedures only from other async functions or from the top level.
- Break up any long-running computation into **async/await** segments so other processes will have a chance to run.
- Leverage concurrency when possible
  - Use **promise.all** if you need to wait for multiple promises to return.
- Check for errors with **try/catch**

Odds and Ends You Should Know

Async/await code is compiled into promise/then code

```
async function
makeThreeSerialRequests(){
1.   console.log('Making first
request');
2.   await makeOneGetRequest();
3.   console.log('Making second
request');
4.   await makeOneGetRequest();
5.   console.log('Making third
request');
6.   await makeOneGetRequest();
7.   console.log('All done!');
}
makeThreeSerialRequests();
```



```
console.log('Making first request');
makeOneGetRequest().then( () =>{
  console.log('Making second request');
  return makeOneGetRequest();
}).then(() => {
  console.log('Making third request');
  return makeOneGetRequest();
}).then(()=>{
  console.log('All done!');
});
```

Promises Enforce Ordering Through "Then"

```
1. console.log('Making requests');
2. axios.get('https://rest-example.covey.town/')
    .then((response) =>{
        console.log('Heard back from server');
        console.log(response.data);
   });
3. axios.get('https://www.google.com/')
      .then((response) =>{
       console.log('Heard back from Google');
      });
4. axios.get('https://www.facebook.com/')
      .then((response) =>{
        console.log('Heard back from Facebook');
      });
5. console.log('Requests sent!');
```

- **axios.get** returns a promise.
- **p.then** mutates that promise so that the then block is not run until after the original promise returns.
- The resulting promise isn't completed until the then block finishes.
- You can chain .**then**'s, to get things that look like p.then().then().then()
- Each **then** is a pause point.

src/data-races/dataRace.ts

But you can still have a data race

```
let x : number = 10

async function asyncDouble() {
    // start an asynchronous computation and wait for the result
    await makeOneGetRequest(1);
    x = x * 2  // statement 1
}


async function asyncIncrementTwice() {
    // start an asynchronous computation and wait for the result
    await makeOneGetRequest(2);
    x = x + 1;   // statement 2
    x = x + 1;   // statement 3
}

async function run() {
    await Promise.all([asyncDouble(), asyncIncrementTwice()])
    console.log(x)
}
```

Review

- You should now be prepared to:
  - Explain the difference between JS run-to-completion semantics and interrupt-based semantics.
  - Given a simple program using async/await, work out the order in which the statements in the program will run.
  - Write simple programs that create and manage promises using async/await
  - Write simple programs to mask latency with concurrency by using non-blocking IO and Promise.all in TypeScript.