CS 4530: Fundamentals of Software Engineering

Module 06: Concurrency Patterns in Typescript

Adeel Bhutta, Mitch Wand

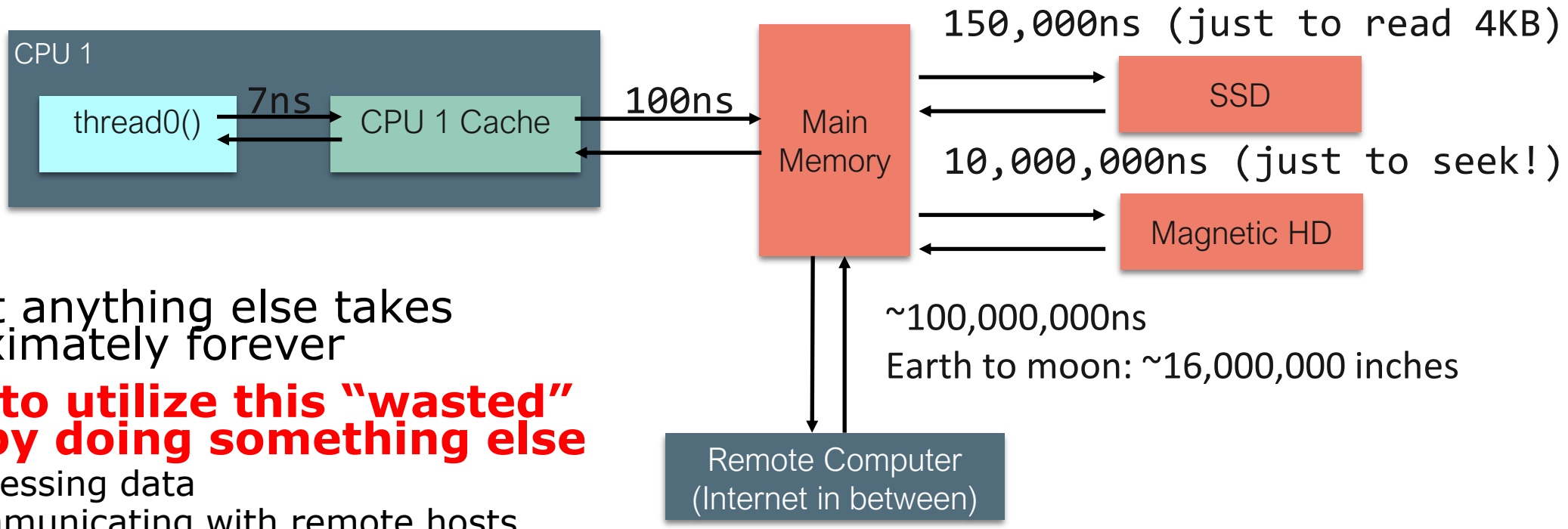
Khoury College of Computer Sciences

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**

- Typescript maintains a pool of processes, called **promises**.
- A promise always executes until it reaches its end.
- This is called "**run-to-completion** semantics".
- A promise can create other promises to be added to the pool.
- Promises interact mostly by passing values to one another; data races are minimized.

A promise can be in one of exactly 3 states

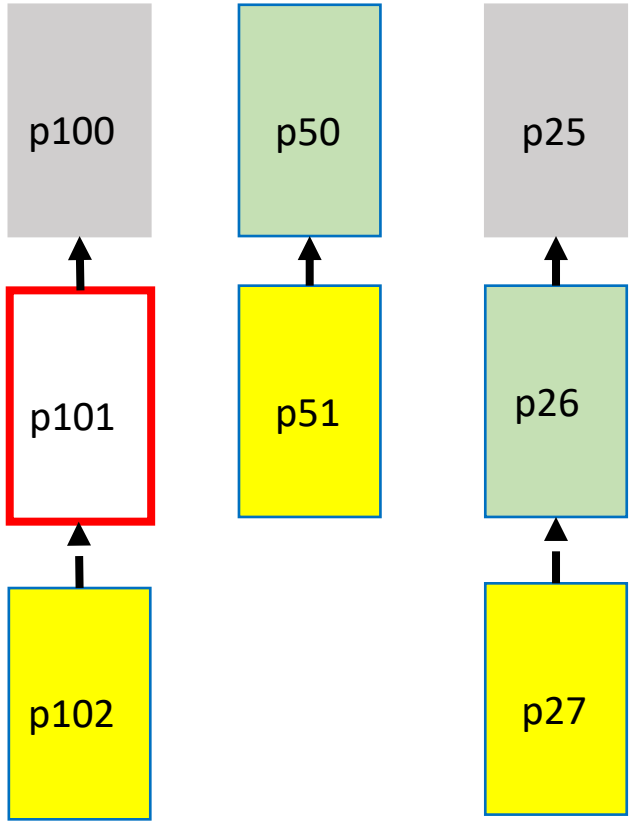
- A JavaScript promise can be in one of three states: pending, fulfilled, or rejected.
- Pending is the initial state where the promise is **waiting** for an operation to complete;
- Resolved: either fulfilled or rejected.
  - fulfilled means the operation was successful,
  - rejected indicates that the operation failed.

Subcategories of Pending Promises

- Waiting: pending, and some of the operations it was waiting for have not yet completed
- Ready: pending, but all the operations it was waiting for have completed
- Executing: pending (not resolved), but the code of the promise is currently being executed

- There can be at most **one** executing promise at any time

A snapshot of the promise pool



The grey promises are fulfilled

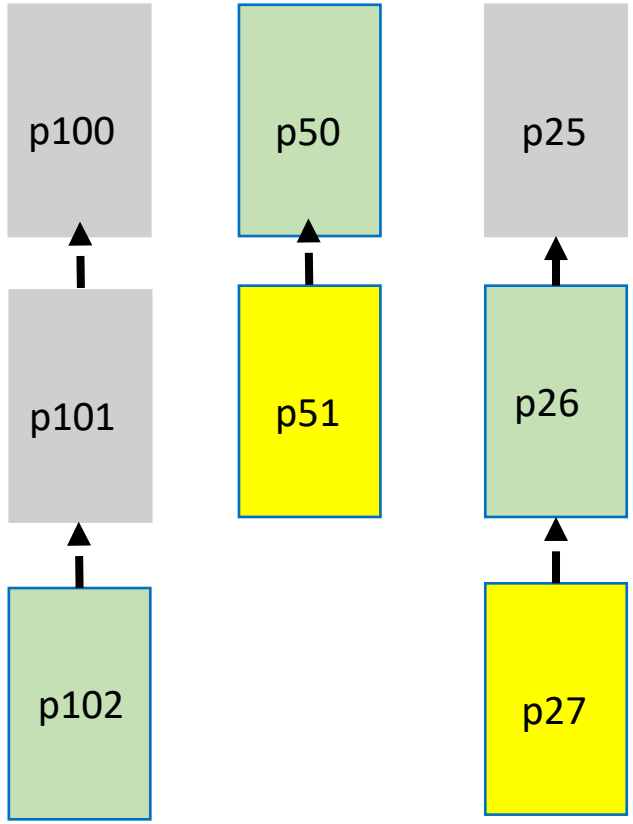
The green promises are pending and ready

The yellow promises are waiting.

The white promise is the currently executing promise

The arrows indicate that one promise is waiting for another

When the currently executing promise succeeds, the pool will look like this:



The grey promises are fulfilled

The green promises are pending and ready

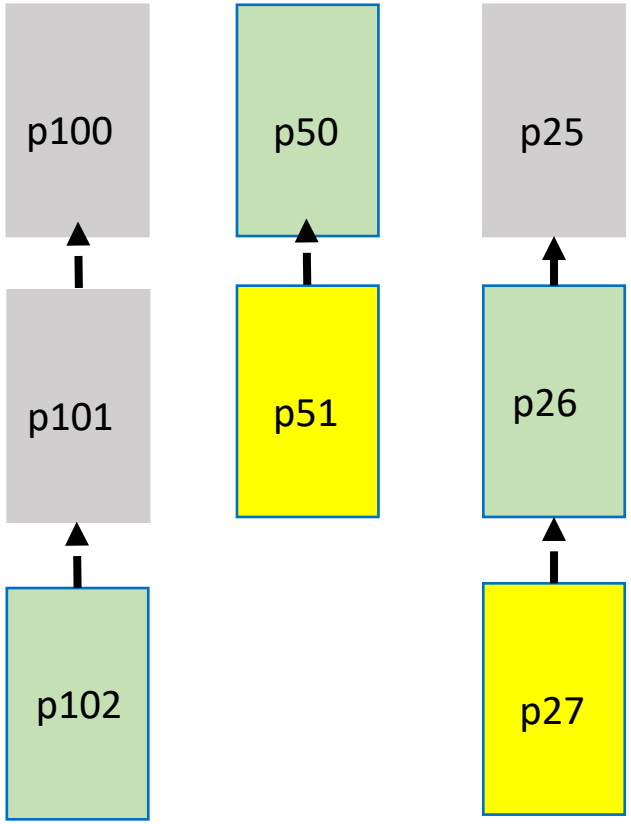
The yellow promises are waiting

The white promise is the currently executing promise

The currently executing promise may have created some new promises, not shown here. Some of them might be ready, too.

The arrows indicate that one promise is waiting for another

Any ready promise can be chosen as the next promise to be executed



The grey promises are fulfilled

The green promises are pending and ready

The yellow promises are waiting

The white promise is the currently executing promise

The arrows indicate that one promise is waiting for another

Computations always run until they are completed.

- Execution of a promise cannot be interrupted. That's what we mean by "run to completion".
- Along the way, it may create promises that can be run anytime after the current computation is completed (i.e. they will be in the "waiting" state).
  - We'll see that async/await provides an easy way to do that.
- A computation is completed when it returns from a procedure, but there are no procedures for it to return to (i.e. it returns to the "top level")
- When the current computation is completed, the operating system (e.g. node.js) chooses some "ready" promise to become the next current computation.

Programming with promises

- Typescript has primitives that create promises.
  - But you will never do this
- Some typescript libraries have API procedures that return promises
  - this is the usual way you'll get promises.
- Most of the time, you'll be building new promises out of the ones that are given to you.
- This is what async/await does…

Use async functions to create promises

- Typically, an async function gets a promise (from somewhere) and returns another promise.

Example:

```
/** given a string, returns a promise that prints a string
  * and then resolves.
**/
import promiseToPrint from "./promiseToPrint";

export async function example1(n: number): Promise<void> {
  console.log(`example1(${n}) starting`);
  const p1 = promiseToPrint(`example1(${n}) is printing`);
  await p1;
  console.log(`example1(${n}) finishing`);
}
```

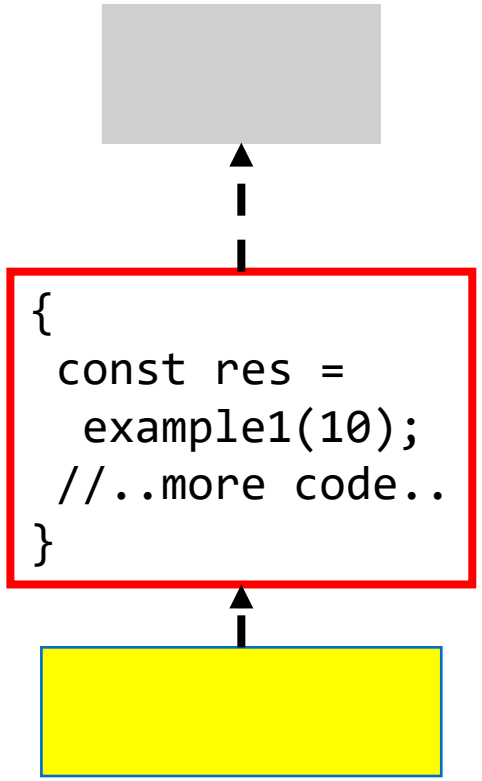
src/async-await/example1.ts

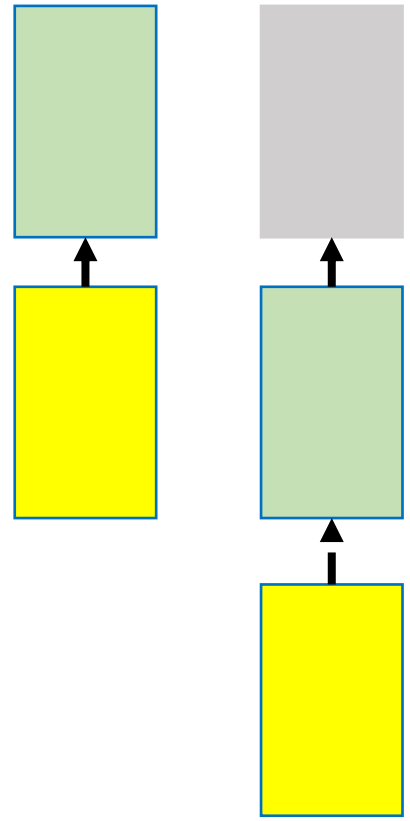
async/await: from the inside out

```
export async function example1(n: number): Promise<void> {
  console.log(`example1(${n}) starting`);
  const p1 = promiseToPrint(`p1 is printing`);
  await p1;
  console.log(`example1(${n}) finishing`);
}
```

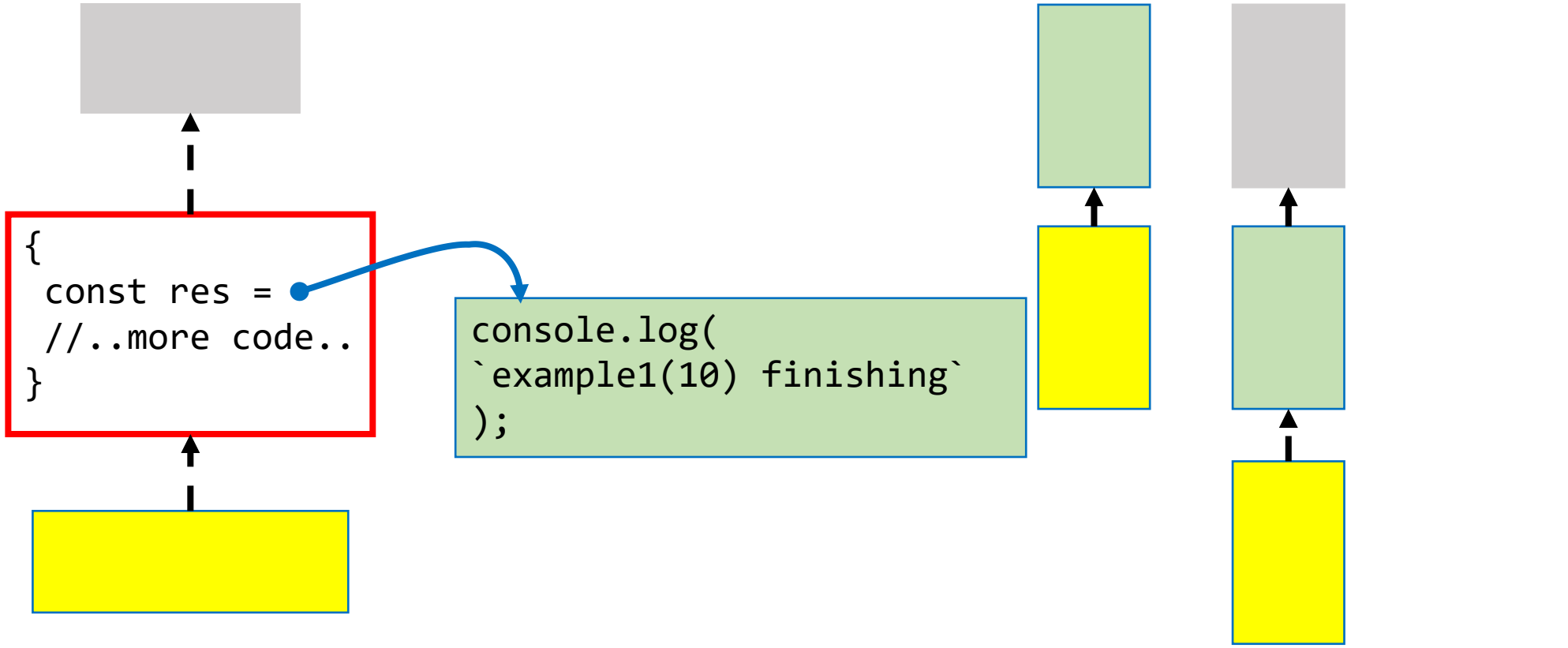
1. This function executes normally until it hits the **await**, printing out "example1(1) starting" and binding p1 to the value of `promiseToPrint('p1 is printing')`
2. When it hits the await, it takes all the code following the await and creates a new promise that can only be executed **after** p1 is completed.
3. The new promise becomes the value of example(n).
4. The caller of example(n) then continues its execution.
5. If example(n) has no caller, then the runtime system chooses some ready promise to execute.

The promise pool before before calling example1()





The promise pool after calling example1()



p1 is printing

Important:
The console.log can't run until after the 'more code' finishes

Async functions: from the outside in

- What can async functions do?
- What are the typical patterns for applying them?

Async functions return promises

```
export async function example1(n: number) {
  console.log(`example1(${n}) starting`);
  const p1 = promiseToPrint(`p1 is printing`);
  await p1;
  console.log(`example1(${n}) finishing`);
}

function main1() {
  console.log('starting main');
  const res = example1(10)
  console.log ('example1(10) returned', res)
  console.log('main finished');
}

main1();
```

```
$ npx ts-node AsyncReturnsPromise.ts
starting main
example1(10) starting
p1 is printing
example1(10) returned Promise { <pending> }
main finished
example1(10) finishing
```

src/async-await/AsyncReturnsPromise.ts

src/async-await/nestedAsyncs.ts

Asyncs can be nested

```
export async function example2(n: number):
Promise<void> {
  console.log(`example2(${n}) starting`);
  const p1 = example1(n);
  await p1;
  console.log(`example2(${n}) finishing`);
}

function main() {
  console.log('starting main');
  example2(10)
  console.log('main finished');
}

main();
```

```
$ npx ts-node nestedAsyncs.ts
starting main
example2(10) starting
example1(10) starting
p1 is printing
main finished
example1(10) finishing
example2(10) finishing
```

Running Multiple Promises Asynchronously

```
export async function example1(n: number) {
  console.log(`example1(${n}) starting`);
  const p1 = promiseToPrint(`p1 is printing`);
  await p1;
  console.log(`example1(${n}) finishing`);
}

function make3AsynchronousPromises() {
    console.log('starting make3AsynchronousPromises');
    example1(100);
    example1(200);
    example1(300);
    console.log('make3AsynchronousPromises finished');
  }

  make3AsynchronousPromises()
```

```
$ npx ts-node ThreeAsynchronousPromises.ts
starting make3AsynchronousPromises
example1(100) starting
p1 is printing
example1(200) starting
p1 is printing
example1(300) starting
p1 is printing
make3AsynchronousPromises finished
example1(100) finishing
example1(200) finishing
example1(300) finishing
```

src/async-await/ThreeAsynchronousPromises.ts

Running Multiple Promises Sequentially

```
export async function example1(n: number): {
  console.log(`example1(${n}) starting`);
  const p1 = promiseToPrint(`p1 is printing`);
  await p1;
  console.log(`example1(${n}) finishing`);
}

async function make3SequentialPromises() {
    console.log('starting make3SequentialPromises');
    await example1(100);
    await example1(200);
    await example1(300);
    console.log('make3SequentialPromises finished');
  }

  make3SequentialPromises()
```

```
$ npx ts-node ThreeSequentialPromises.ts
starting make3SequentialPromises
example1(100) starting
p1 is printing
example1(100) finishing
example1(200) starting
p1 is printing
example1(200) finishing
example1(300) starting
p1 is printing
example1(300) finishing
make3SequentialPromises finished
```

src/async-await/ThreeSequentialPromises.ts

src/async-await/PromisesPassingValues.ts

Promises can pass values to one another

```
export async function example1(n: number) {
  console.log(`example1(${n}) starting`);
  const p1 = promiseToPrint(`p1 is printing`);
  await p1;
  console.log(`example1(${n}) finishing`);
  // pass this to any waiting promises
  // this is NOT the value of the async function
  return n+10;
}

async function promisesPassingValues() {
    console.log('starting promisesPassingValues'
    const res1 = await example1(100);
    const res2 = await example1(res1);
    const res3 = await example1(res2);
    console.log(`res3 = ${res3}`);
    console.log('promisesPassingValues finished');
  }
```

```
$ npx ts-node PromisesPassingValues.ts
starting promisesPassingValues
example1(100) starting
p1 is printing
example1(100) finishing
example1(110) starting
p1 is printing
example1(110) finishing
example1(120) starting
p1 is printing
example1(120) finishing
res3 = 130
promisesPassingValues finished
```

Recover from failure with try/catch

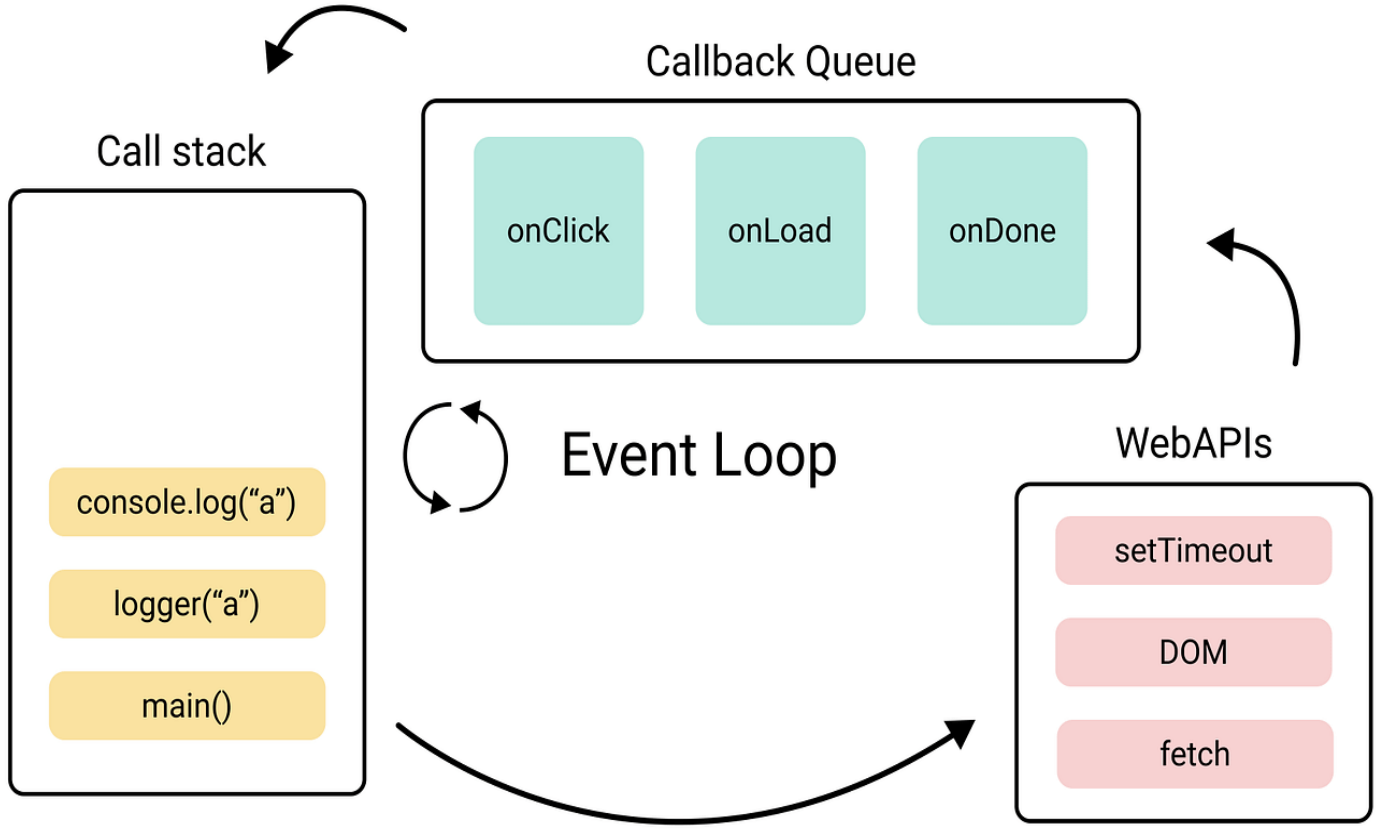
```
// promise to fail if shouldFail is true
import { promiseMaybeFail } from './promi

async function script(shouldFail:boolean)
    console.log('starting script with sho
    try {
        await promiseMaybeFail(shouldFail
        console.log('promise succeeded');
    }
    catch (e) { console.log('promise fail
    console.log('script finished successfully ');
}

async function main1() {
await script(false);
console.log('\n')
await script(true)
}

main1()
```

```
$ npx ts-node recoveringFromPromiseFailure.ts
starting script with shouldFail =  false
promise succeeded
script finished successfully


starting script with shouldFail =  true
promise failed, but error caught
script finished successfully
```

src/async-await/recoveringFromPromiseFailure.ts

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



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

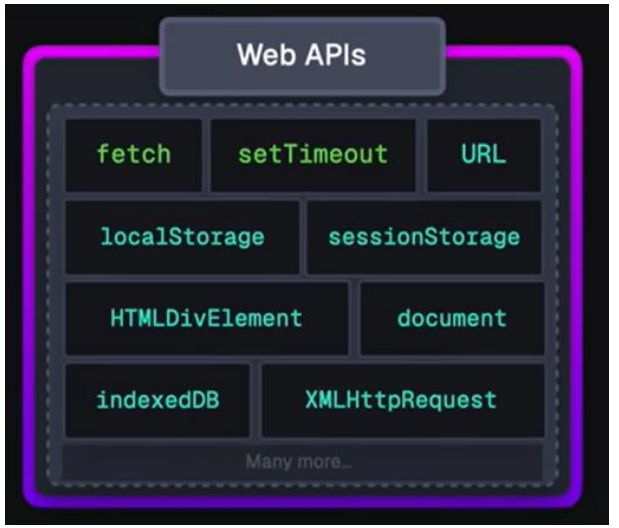
- These are things like http requests, I/O operations, or timers.
- Each of these returns a promise that you can **await**. The promise runs while it is pending, and produces the response from the http request, or the contents of the file, etc.
- You will hardly ever call one of these primitives yourself;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

src/async-await/makeThreeConcurrentRequests.ts

Running 3 concurrent requests

```
import axios from 'axios';

export async function makeRequest(
    console.log(`starting makeRequ
    const response = await axios.g
    console.log(`request:${request
}


  function make3ConcurrentRequests
    console.log('starting make3Con
    makeRequest(100);
    makeRequest(200);
    makeRequest(300);
    console.log('make3ConcurrentRequests finished');
  }

make3ConcurrentRequests()
```

```
$ npx ts-node makeThreeConcurrentRequests.ts
starting make3ConcurrentRequests
starting makeRequest(100)
starting makeRequest(200)
starting makeRequest(300)
make3ConcurrentRequests finished
request 300 returned
request 100 returned
request 200 returned
```

Promise.all takes a list of promises, runs them concurrently, and succeeds only when they have all succeeded.

```
export async function makeRequest(requestNumber:number) {
    console.log(`starting makeRequest(${requestNumber})`);
    await axios.get('https://rest-example.c
    console.log(`request ${requestNumber} r
    return requestNumber
}

async function manyConcurrentRequests(reque
    console.log('starting manyConcurrentRed
    const responses = await Promise.all(red
    console.log('responses:', responses);
    console.log('manyConcurrentRequests fin
  }

async function main() {
  manyConcurrentRequests([100,200,300,400])
}

main()
```

```
$ npx ts-node manyConurrentRequests.ts
starting manyConcurrentRequests
starting makeRequest(100)
starting makeRequest(200)
starting makeRequest(300)
starting makeRequest(400)
request 100 returned
request 300 returned
request 200 returned
request 400 returned
responses: [ 100, 200, 300, 400 ]
manyConcurrentRequests finished
```

src/async-await/manyConcurrentRequests.ts

If you add awaits, the requests will be processed sequentially

```
async function make3SequentialRequests() {
  console.log('starting make3SequentialRequests');
  await makeRequest(100);
  await makeRequest(200);
  await makeRequest(300);
  console.log('make3SequentialRequests finished');
}
```

```
$ npx ts-node
makeThreeSequentialRequests.ts
starting make3SequentialRequests
starting makeRequest(100)
request 100 returned
starting makeRequest(200)
request 200 returned
starting makeRequest(300)
request 300 returned
make3SequentialRequests finished
```

…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
```

Why is that?
Visualizing Promise.all

Sequential (await)

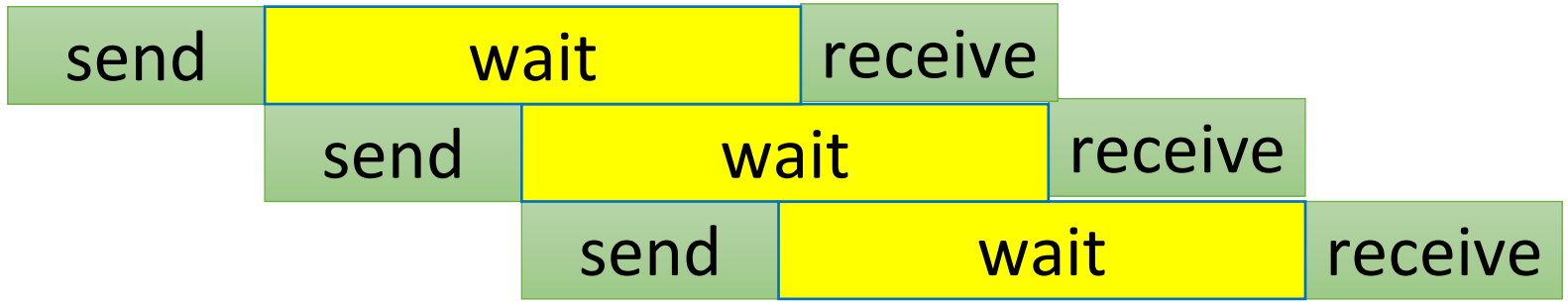
"Don't make another request until you got the last response back"

237 msec

Concurrent (Promise.all)

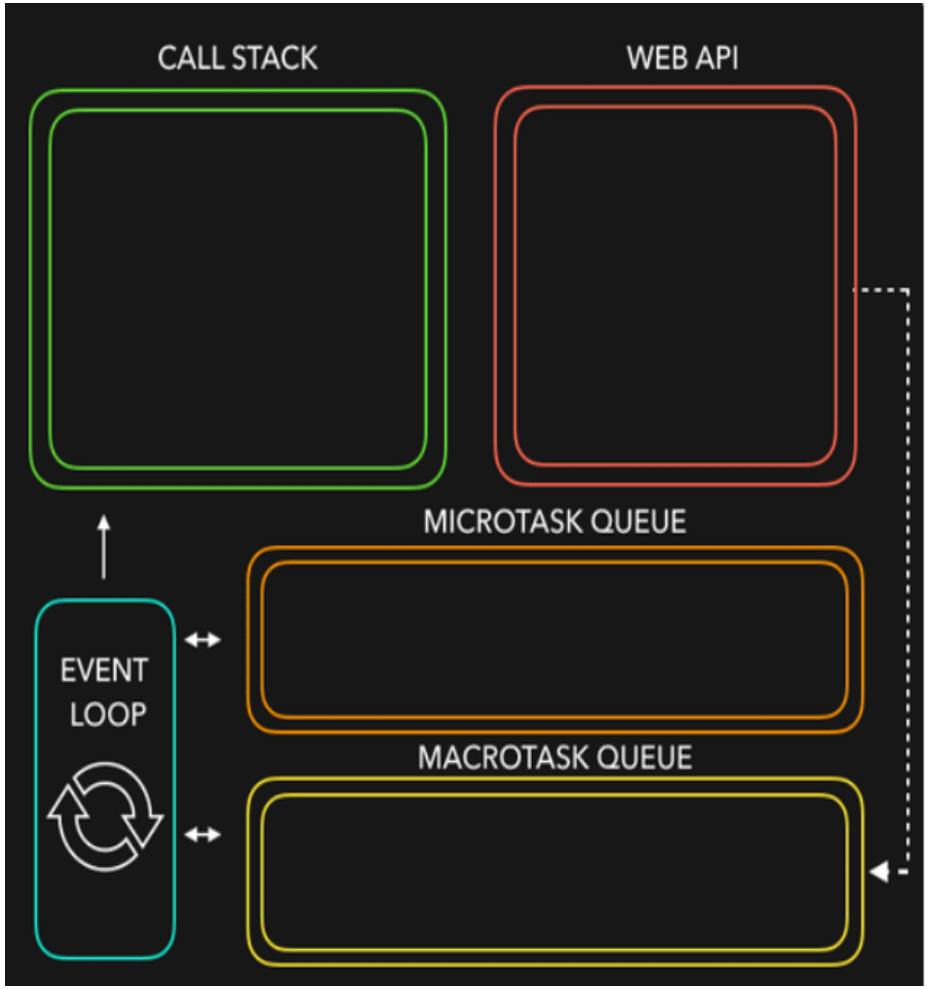
"Make all of the requests now, then wait for all of the responses"

34 msec



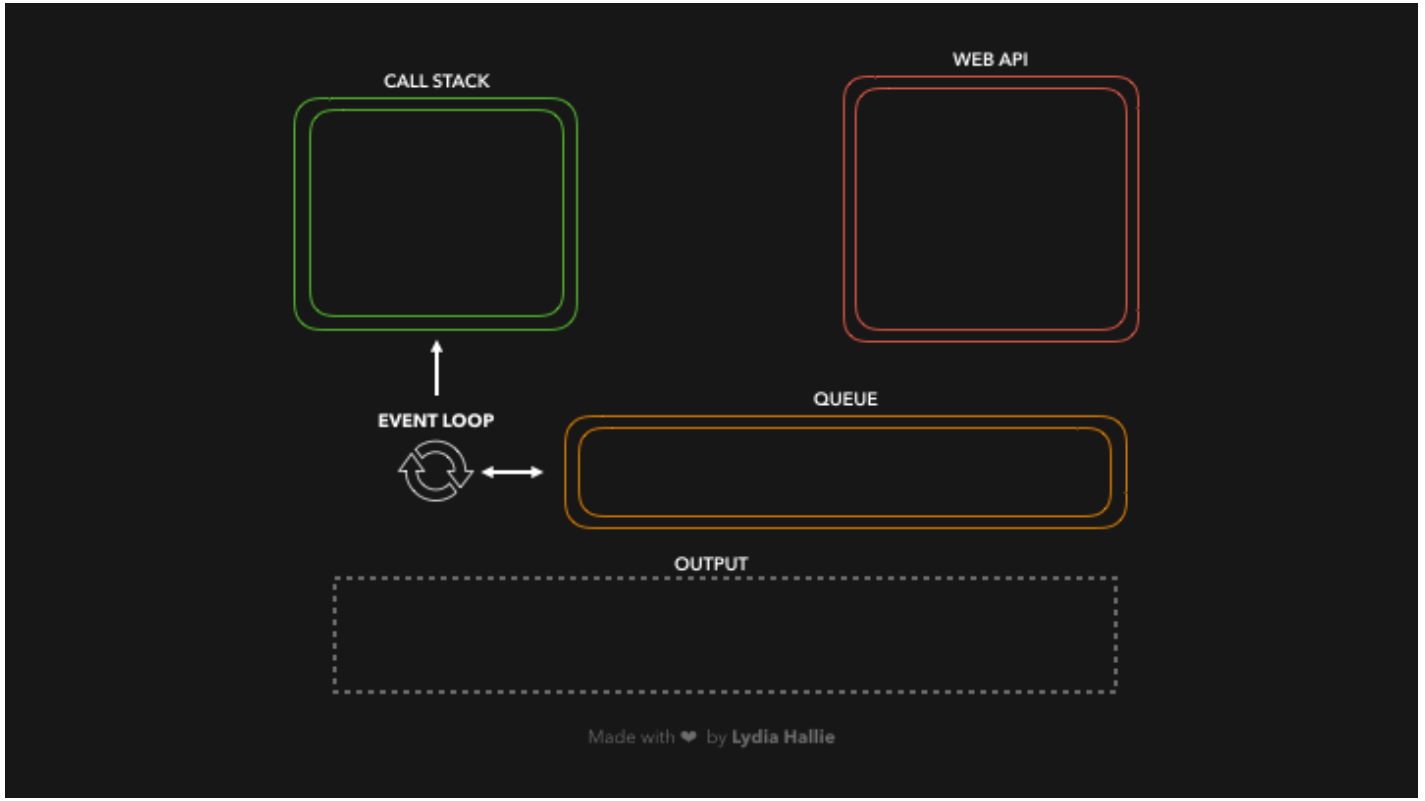
Let’s put it all together

- 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

Pattern for testing an async function

```
import axios from 'axios'

async function echo(str: string) : Promise<string> {
    const res =
        await axios.get(`https://httpbin.org/get?answer=${str}`)
    return res.data.args.answer
}

test('request should return its argument', async () => {
    expect.assertions(1)
    await expect(echo("33")).resolves.toEqual("33")
})
```

src/jest/jest-example.test.ts

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**

An Example Task Using the Transcript Server

- Given an array of StudentIDs:
  - Request each student's transcript, and save it to disk so that we have a copy, and calculate its size
  - Once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

Generating a promise for each student

```
async function asyncGetStudentData(studentID: number) {
    const returnValue =
     await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    return returnValue
}

async function asyncProcessStudent(studentID: number) : Promise<number> {
    // wait to get the student data
    const response = await asyncGetStudentData(studentID)
    // asynchronously write the file
    await fsPromises.writeFile(
        dataFileName(studentID),
        JSON.stringify(response.data))
    // last, extract its size
    const stats = await fsPromises.stat(dataFileName(studentID))
    const size : number = stats.size
    return size
}
```

Calling await also gives other processes a chance to run.

src/transcripts/simple.ts

Running the student processes concurrently

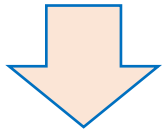
```
async function runClientAsync(studentIDs:number[]) {
    console.log(`Generating Promises for ${studentIDs}`);
    const studentPromises =
      studentIDs.map(studentID => asyncProcessStudent(studentID)) ;
    console.log('Promises Created!');
    console.log('Satisfying Promises Concurrently')
    const sizes = await Promise.all(studentPromises);
    console.log(sizes)
    const totalSize = sum(sizes)
    console.log(`Finished calculating size: ${totalSize}`);
    console.log('Done');
  }
```

Map-promises pattern: take a list of elements and generate a list of promises, one per element

src/transcripts/simple.ts

Output

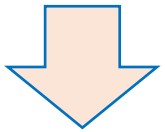
```
runClientAsync([411,412,423])
```



```
$ npx ts-node simple.ts
Generating Promises for 411,412,423
Promises Created!
Satisfying Promises Concurrently
[ 151, 92, 145 ]
Finished calculating size: 388
Done
```

But what if there's an error?

```
runClientAsync([411,412,87065,423,23044])
```



```
$ npx ts-node transcripts/simple.ts
Generating Promises for 411,412,87065,423,23044
Promises Created!
Satisfying Promises Concurrently

<blah blah
blah>\node_modules\axios\lib\core\createError.js
:16
  var error = new Error(message);
              ^
Error: Request failed with status code 404
```

Oops!

Need to catch the error

```
type StudentData = {isOK: boolean, id: number, payload?: any }

/** asynchronously retrieves student data,  */
async function asyncGetStudentData(studentID: number): Promise<StudentData> {
    try {
        const returnValue =
          await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
        return { isOK: true, id: studentID, payload: returnValue }
    } catch (e) {
        return { isOK: false, id: studentID }
    }
}
```

Catch the error and transmit it in a form the rest of the caller can handle.

src/transcripts/handle-errors.ts

And recover from the error…

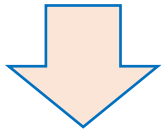
```
async function asyncProcessStudent(studentID: number): Promise<number> {
    // wait to get the student data
    const response = await asyncGetStudentData(studentID)
    if (!(response.isOK)) {
        console.error(`bad student ID ${studentID}`)
        return 0
    } else {
        await fsPromises.writeFile(
            dataFileName(studentID),
            JSON.stringify(response.payload.data))
        // last, extract its size
        const stats = await fsPromises.stat(dataFileName(studentID))
        const size: number = stats.size
        return size
    }
}
```

Design decision: if we have a bad student ID, we'll print out an error message, and count that as 0 towards the total.

src/transcripts/handle-errors.ts

New output

```
runClientAsync([411,32789,412,423,10202040])
```



```
$ npx ts-node transcripts/handle-errors.ts
Generating Promises for
411,32789,412,423,10202040
Promises Created!
Wait for all promises to be satisfied
bad student ID 32789
bad student ID 10202040
[ 151, 0, 92, 145, 0 ]
Finished calculating size: 388
Done
```

Odds and Ends You Should Know About

src/data-races/dataRace.ts

This is not Java!

```
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
    // nothing can happen between these two statements!!
    x = x + 1;   // statement 3
}

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

- In Java, you could get an interrupt between statement 2 and statement 3.
- In TS/JS statement 3 is guaranteed to be executed *immediately* after statement 2!
- No interrupt is possible.

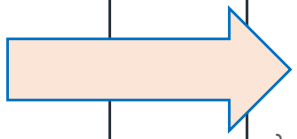
src/data-races/dataRace.ts

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)
}
```

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 run immediately 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()

The Self-Ticking Clock

- To make the clock self-ticking, add the following line to your clock:

```
constructor () {
  setInterval(() => {this.tick()},50)
}
```

Async/Await Programming Activity

Download the activity (includes instructions in README.md):
Linked from course webpage for Module 6

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.