

# CS 4530: Fundamentals of Software Engineering

## Module 6: Concurrency Patterns in Typescript

---

Jon Bell, Adeel Bhutta, Mitch Wand  
Khoury College of Computer Sciences

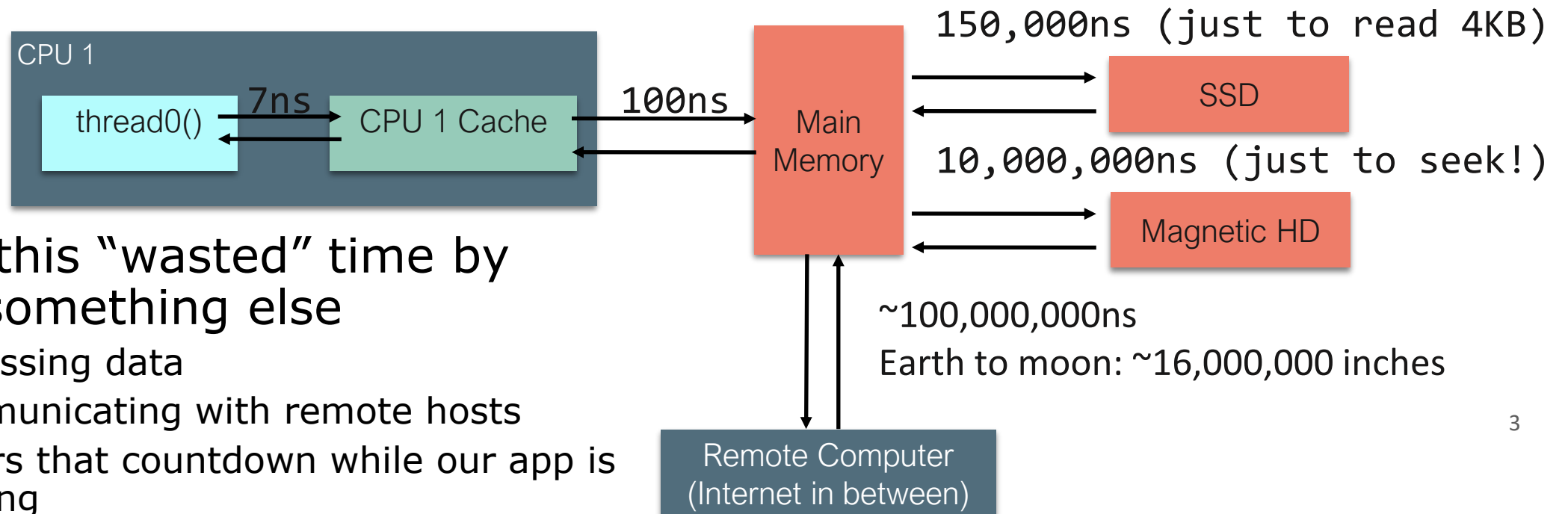
# Learning Goals for this Lesson

---

- 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.

# Our goal is to mask latency with concurrency

- Consider: a 1Ghz CPU executes an instruction every 1 ns
- Almost anything else takes forever (approximately)



- Utilize this “wasted” time by doing something else
  - Processing data
  - Communicating with remote hosts
  - Timers that countdown while our app is running
  - Waiting for users to provide 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.)

# Cooperative multiprocessing maintains a pool of **promises**.

---

- Typescript maintains a pool of processes, called promises.
- A promise always executes until it is completed
  - 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 4 states

---

- **Executing**
  - there is only one of these; we call it the "current promise" or the "current computation"
- **Ready** for execution
- **Waiting** for some event
  - Typically for some other promise to complete
- **Terminated**
  - The technical term is "resolved".

# Computations always run until they are completed.

---


- Along the way, it may create promises that can be run anytime after the current computation is completed (i.e. they are in the "ready" state)
- It may also create promises that are in the "waiting" state-- waiting for some event, at which time they become "ready".
- When the current computation is completed, the operating system (e.g. node.js) chooses some "ready" process to become the next current computation.



# Create promises with `async`

---

- Async functions create and return promises
- Here, `f`
  1. creates a promise to produce a "1".
  2. Marks it as ready and puts it in the process pool.
  3. Returns that promise as its value.

```
src > promises > TS asyncNoAwait.ts >  f
1  async function f(n:number) {return n}
2
3  const k: Promise<number>
4  const k = f(1)
```

# Example0.ts

---

```
async function f (n) {  
  console.log("entering f")  
  return n  
}
```

```
function main() {  
  const p = f(1);  
  console.log('p =', p)  
}
```

```
main()
```

```
$ npx ts-node example0.ts  
entering f  
p = Promise { 1 }
```

# Use async/await to create a promise that waits for some other promise to complete

---

```
async function example() {  
  doThisNow();  
  const p1 = somePromise();  
  const response = await p1  
  doThisLater();  
}
```

1. Executes doThisNow()
2. Evaluates somePromise() {which should return a promise}. Puts that promise in the pool, and marks it as **ready**.
3. Puts in the pool a promise to doThisLater(), and marks it as **waiting** for the completion of the promise **p1**.
4. Last, it returns the promise p1 to its caller.

# A running example

---

```
export async function example(n:number) {  
  doThisNow(n);  
  const p1 = somePromise(n);  
  console.log('p1 =', p1)  
  const response = await p1  
  doThisLater(n);  
}
```

```
function doThisNow (n) {console.log("doThisNow", n)}  
async function somePromise (n) {return `somePromise ${n}`}  
function doThisLater (n) {console.log("doThisLater", n)}
```

# Simplest example

```
import { example } from "./asyncExample"

function main () {
  console.log("calling example(1)")
  example(1)
  console.log("returning to main")
  console.log("main finished\n")
}

main()
```

```
export async function
example(n:number) {
  doThisNow(n);
  const p1 = somePromise(n);
  console.log('p1 =', p1)
  const response = await p1
  doThisLater(n);
}
```

```
$ npx ts-node example1.ts
calling example(1)
doThisNow 1
p1 = Promise { 'somePromise 1' }
returning to main
main finished
```

src/async-await/example1.ts

# You can start multiple threads

src/async-await/example2.ts

```
import { example }
  from "./asyncExample";

async function main() {
  example(1)
  example(2)
  example(3)
  console.log("main finished\n")
}

main()
```

```
$ npx ts-node example2.ts
doThisNow 1
p1 = Promise { 'somePromise 1' }
doThisNow 2
p1 = Promise { 'somePromise 2' }
doThisNow 3
p1 = Promise { 'somePromise 3' }
main finished

doThisLater 1
doThisLater 2
doThisLater 3
```

# Use await to make promises execute sequentially

src/async-await/example3.ts

```
import { example }
  from "../asyncExample";

async function main() {
  await example(1)
  await example(2)
  await example(3)
  console.log("main finished\n")
}

main()
console.log("example3 finished\n")
```

```
$ npx ts-node example3.ts
doThisNow 1
p1 = Promise { 'somePromise 1' }
example3 finished

doThisLater 1
doThisNow 2
p1 = Promise { 'somePromise 2' }
doThisLater 2
doThisNow 3
p1 = Promise { 'somePromise 3' }
doThisLater 3
main finished
```

# Use Promise.all to synchronize on the completion of several promises

```
async function forkJoin() {
  console.log("forkJoin started")
  const promises
    = [example(1), example(2), example(3)]
  console.log(promises)
  await Promise.all(promises)
  console.log("forkJoin finished\n")
}

async function main() {
  forkJoin()
  console.log("main finished\n")
}
```

src/async-await/example4.ts

```
$ npx ts-node example4.ts
forkJoin started
doThisNow 1
p1 = Promise { 'somePromise 1' }
doThisNow 2
p1 = Promise { 'somePromise 2' }
doThisNow 3
p1 = Promise { 'somePromise 3' }
[ Promise { <pending> }, Promise {
  <pending> }, Promise { <pending> } ]
main finished

doThisLater 1
doThisLater 2
doThisLater 3
forkJoin finished
```



# 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

---

```
async function makeRequest(requestNumber:number) {  
    // some code (to be executed now)  
    const response =  
        await axios.get('https://rest-example.covey.town')  
    // more code (to be executed after the .get() returns.)  
}
```

- The http request is sent immediately.
- A promise is created to run the **more code** *after* the http call returns (i.e., the code after “awaits” is blocked)
- The caller of **makeRequest** resumes immediately.

# The pattern in action

```
export async function makeRequest(requestNumber:number) {
  console.log(`makeRequest is about to start request ${requestNumber}`);
  const response = await axios.get('https://rest-exa
  console.log(`makeRequest resumes request ${request
  console.log(`makeRequest reports that for request
  response.data);
}
```

```
console.log("main thread is about to call makeRequest"
makeRequest(1000);
console.log("main thread continues after makeRequest r
console.log("end of main thread")
```

1. Axios.get starts the http request in the background, and
2. Creates a promise to do the code after the await.
3. The call to makeRequest returns.

```
$ npx ts-node example1
main thread is about to call makeRequest
makeRequest is about to start request 1000
main thread continues after makeRequest returns
```

end of main thread

```
makeRequest resumes request 1000
```

```
makeRequest reports that for request '1000', server replied: This is GET number 200 on the current
server
```

4. The main thread finishes.
5. The computation resumes the promise

# Running several requests concurrently

```
import makeRequest from './makeRequest';
import timeIt from './timeIt'

async function makeThreeSimpleRequests() {
  makeRequest(1);
  makeRequest(2);
  makeRequest(3);
  console.log("Three requests made; main thread finishes")
}

timeIt("main thread", makeThreeSimpleRequests)
```

src/requests/example2.ts

Requests are made in order

But the response for request 3 arrived at the server before request 1.

```
$ npx ts-node example2
makeRequest is about to start request 1
makeRequest is about to start request 2
makeRequest is about to start request 3
Three requests made; main thread finishes
Elapsed time for main thread: 41.064 milliseconds
makeRequest reports that for request '3', server replied: This is GET number 223
on the current server
makeRequest reports that for request '1', server replied: This is GET number 224
on the current server
makeRequest reports that for request '2', server replied: This is GET number 225
on the current server
```

```
import makeRequest from './makeRequest';
import timeIt from './timeIt'

async function makeThreeSerialRequests() {
  await makeRequest(1);
  await makeRequest(2);
  await makeRequest(3);
  console.log("Three requests made; main thread finishes")
}

timeIt("main thread", makeThreeSerialRequests)
```

**await**  
makes your  
code more  
sequential

```
$ npx ts-node example3
makeRequest is about to start request 1
makeRequest reports that for request '1', server rep
number 232 on the current server
makeRequest is about to start request 2
makeRequest reports that for request '2', server replied: This is GET
number 233 on the current server
makeRequest is about to start request 3
makeRequest reports that for request '3', server replied: This is GET
number 234 on the current server
Three requests made; main thread finishes
Elapsed time for main thread: 800.270 milliseconds
```

Second request doesn't start  
until to first request returns

# Promise.all waits for all of the promises in a list to finish

```
async function makeThreeConcurrentRequests() {  
  const p1 : Promise<void> = makeRequest(1);  
  const p2 : Promise<void> = makeRequest(2);  
  const p3 : Promise<void> = makeRequest(3);  
  const thePromises = [p1,p2,p3]  
  await Promise.all(thePromises)  
  console.log(`main thread reports: thePromises = [${thePromises}]`)  
  console.log(`main thread finishes`)  
}
```

src/requests/example4.ts

```
timeIt("main thread", makeThreeConcurrentRequests)
```

Main thread doesn't resume until ALL of the promises are satisfied

```
$ npx ts-node example5
```

```
makeRequest is about to start request 1
```

```
makeRequest is about to start request 2
```

```
makeRequest is about to start request 3
```

```
makeRequest reports that for request '2', server replied: This is GET number 259 on the current server
```

```
makeRequest reports that for request '1', server replied: This is GET number 260 on the current server
```

```
makeRequest reports that for request '3', server replied: This is GET number 261 on the current server
```

```
main thread reports: thePromises = [[object Promise],[object Promise],[object Promise]]
```

```
main thread finishes
```

```
Elapsed time for main thread: 256.518 milliseconds
```

# Visualizing Promise.all (1)

---

**Sequential version: ~206 msec**

```
async function makeThreeSerialRequests():  
Promise<void> {  
    await makeOneGetRequest(1);  
    await makeOneGetRequest(2);  
    await makeOneGetRequest(3);  
    console.log('Heard back from all of the  
requests')  
}
```

“Don’t make another request  
until you got the last response  
back”

**Concurrent version: ~80 msec**

```
async function makeThreeConcurrentRequests():  
Promise<void> {  
    await Promise.all([  
        makeOneGetRequest(1),  
        makeOneGetRequest(2),  
        makeOneGetRequest(3)  
    ])  
    console.log('Heard back from all of the requests')  
}
```

“Make all of the requests now,  
then wait for all of the  
responses”



# Visualizing Promise.all (2)

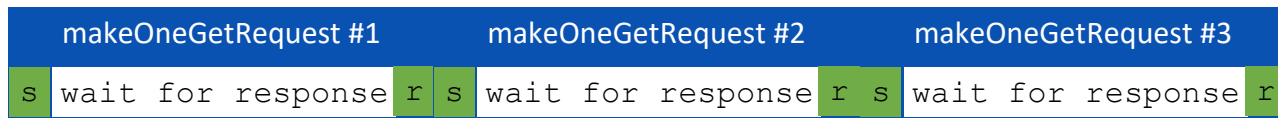
Sequential version: ~206 msec

```
async function makeThreeSerialRequests():  
Promise<void> {  
  await makeOneGetRequest(1);  
  await makeOneGetRequest(2);  
  await makeOneGetRequest(3);  
  console.log('Heard back from all of the  
requests')  
}
```

Concurrent version: ~80 msec

```
async function makeThreeConcurrentRequests():  
Promise<void> {  
  await Promise.all([  
    makeOneGetRequest(1),  
    makeOneGetRequest(2),  
    makeOneGetRequest(3)  
  ])  
  console.log('Heard back from all of the requests')  
}
```

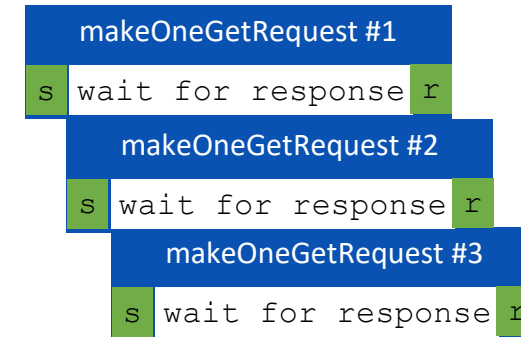
Time →



s send

r receive

Time →



# 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

src/transcripts/simple.ts

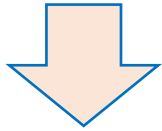
```
async function runClientAsync(studentIDs:number[]) {
  console.log(`Generating Promises for ${studentIDs}`);
  const studentPromises =
    studentIDs.map(studentID => asyncProcessStudent(studentID)) ;
  console.log('Promises Created!');
  console.log('Satisfying Promises Concurrently')
  const sizes = 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

# Output

---

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

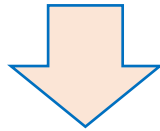


```
$ npx ts-node simple.ts  
Generating Promises for 411,412,423  
Promises Created!  
Satisfying Promises Concurrently  
[ 151, 92, 145 ]  
Finished calculating size: 388  
Done
```

# But what if there's an error?

---

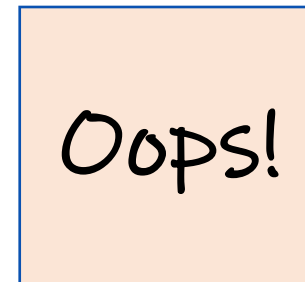
```
runClientAsync([411,412,87065,423,23044])
```



```
$ 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
```



# 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.

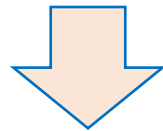
src/transcripts/handle-errors.ts



# 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
```

# 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**

# Odds and Ends You Should Know About

---

# 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.

# 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  
}
```

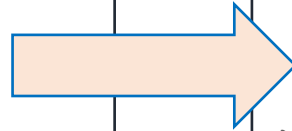
```
async function asyncIncrementTwice() {  
    // start an asynchronous computation and wait for the result  
    await makeOneGetRequest(2);  
    x = x + 1; // statement 2  
    x = x + 1; // statement 3  
}
```

```
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

---

- Your task is to write a new `async` function, `importGrades`, which takes in input of the type `ImportTranscript[]`.
- `importGrades` should create a student record for each `ImportTranscript`, and then post the grades for each of those students.
- After posting the grades, it should fetch the transcripts for each student and return an array of transcripts.

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.