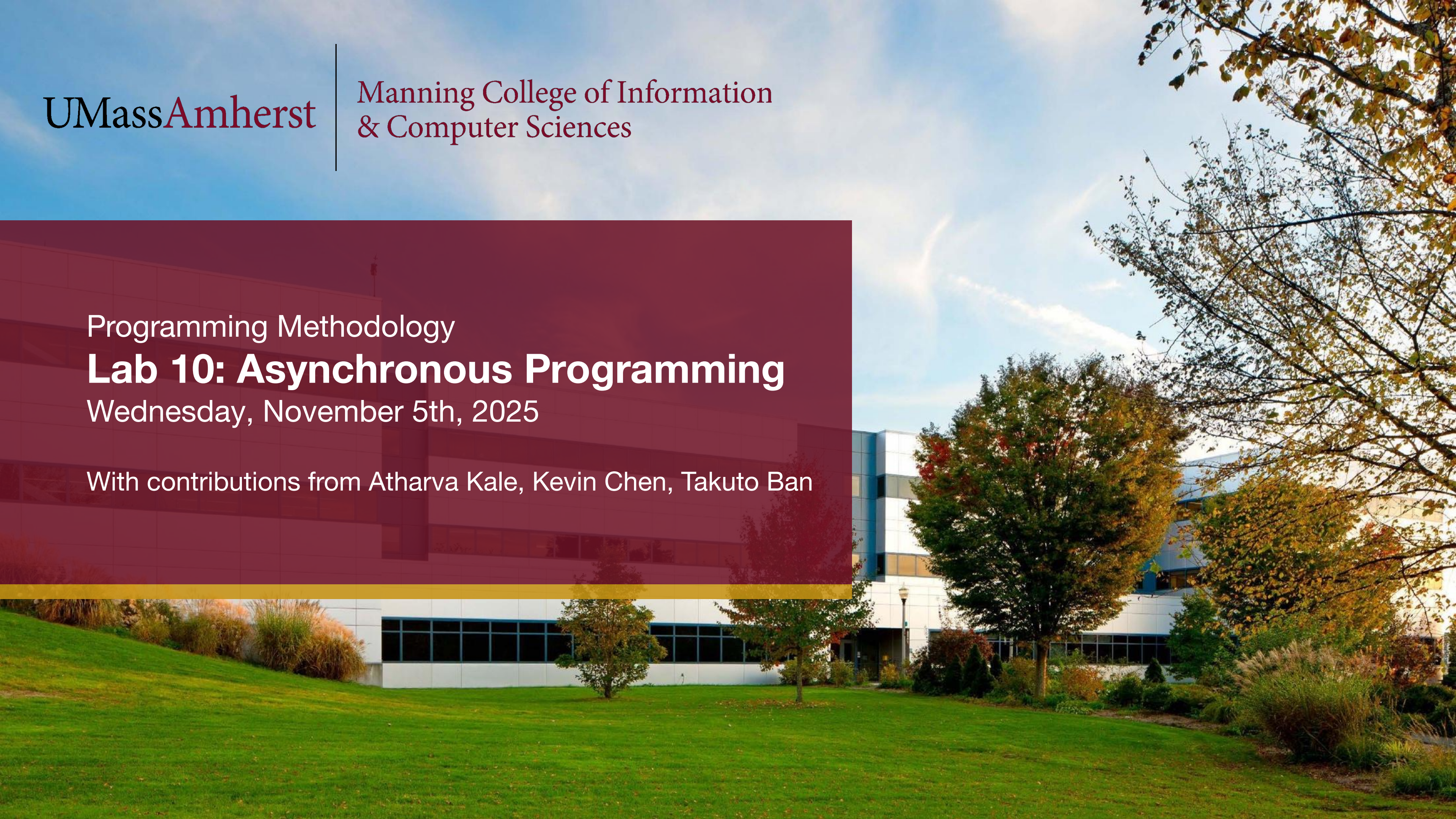
UMassAmherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 10: Asynchronous Programming

Wednesday, November 5th, 2025

With contributions from Atharva Kale, Kevin Chen, Takuto Ban



Agenda

Weekly Reminders

Motivate Async Programming

then

Exercise 1
Get hands-on with Promise API

Exercise 2
Scavenger Hunt

Use Gradescope

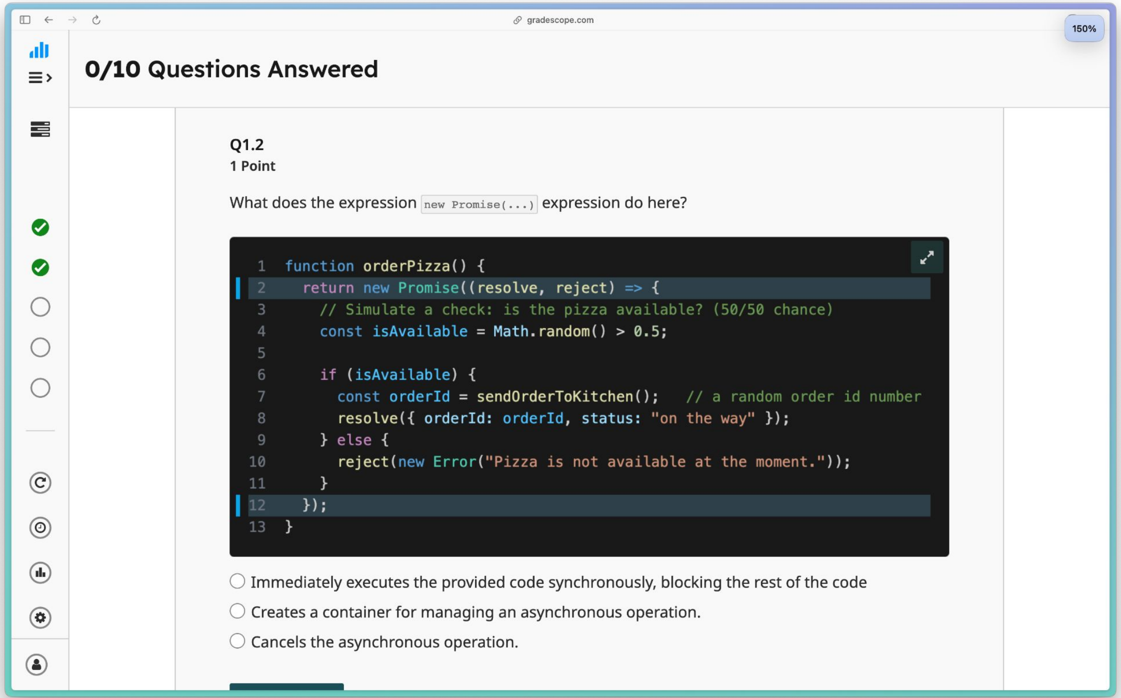
Reminders

- Midterm 2 is being graded; we are aiming to have grades ready Friday!
- Next homework will be released by the end of the week
  - Will be due Sunday, November 16th
  - No homework due this Sunday (11/9)
  - Come to office hours for help!
- If you need to miss lab and have a valid reason according to the syllabus (medical, other personal) please fill out the questionnaire on Canvas before the start time of your lab.
  - Waking up late, bus was late are NOT valid reasons to miss lab.

UMassAmherst | Manning College of Information & Computer Sciences

New Lab Format for Today

We have some questions on Gradescope today :)


gradescope.com

150%

0/10 Questions Answered

Q1.2
1 Point

What does the expression `new Promise(...)` expression do here?

```
function orderPizza() {
  return new Promise((resolve, reject) => {
    // Simulate a check: is the pizza available? (50/50 chance)
    const isAvailable = Math.random() > 0.5;

    if (isAvailable) {
      const orderId = sendOrderToKitchen();   // a random order id number
      resolve({ orderId: orderId, status: "on the way" });
    } else {
      reject(new Error("Pizza is not available at the moment."));
    }
  });
}
```

- ○ Immediately executes the provided code synchronously, blocking the rest of the code
- ○ Creates a container for managing an asynchronous operation.
- ○ Cancels the asynchronous operation.

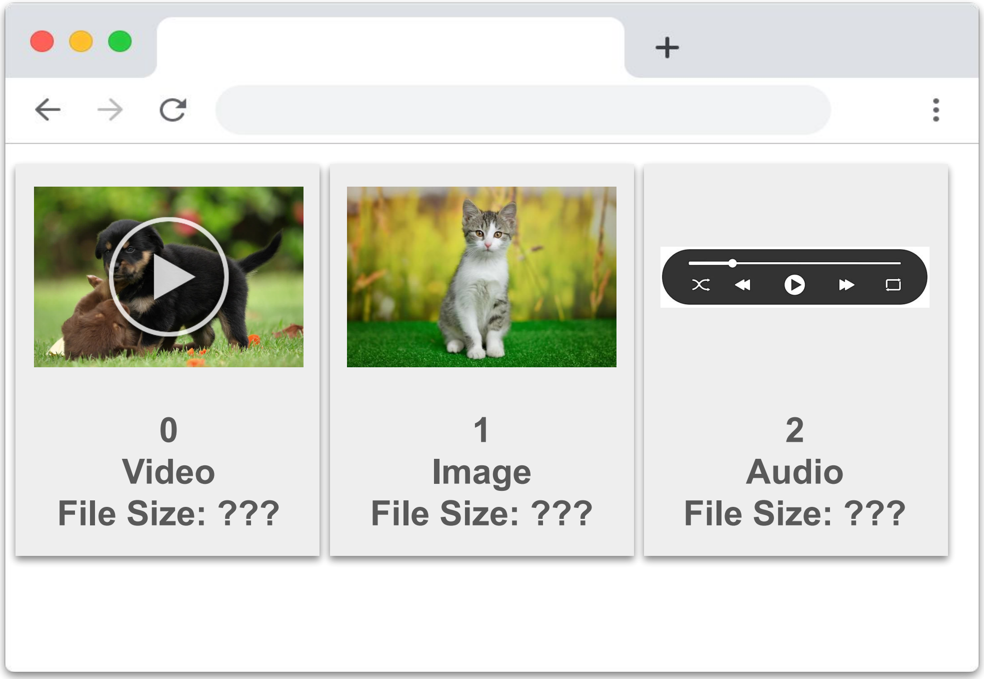
UMassAmherst | Manning College of Information & Computer Sciences

Motivation - Asynchronous Programming

Let's say you are developing a browser responsible for displaying three items:

1) A video
2) An image
3) Some audio

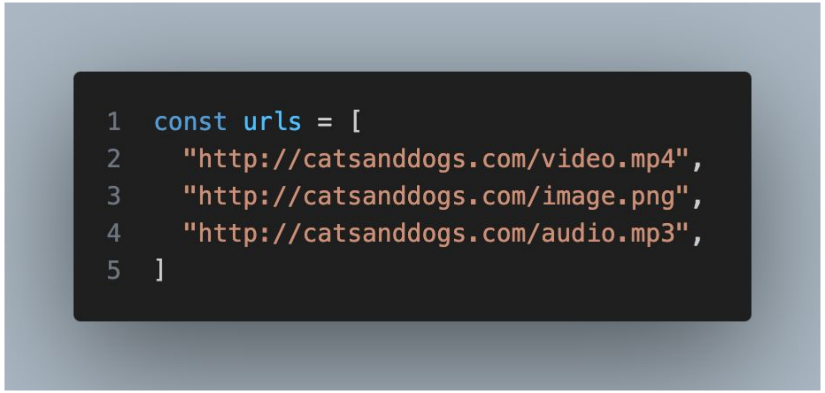
Goal: Give user best user experience



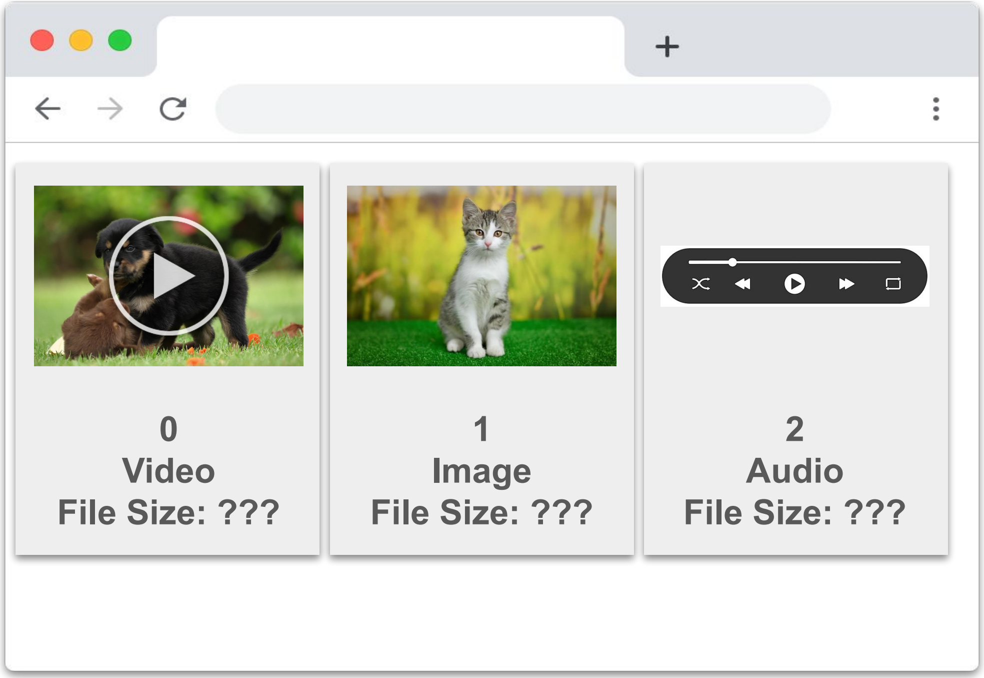
UMassAmherst | Manning College of Information & Computer Sciences

Motivation - Asynchronous Programming

We have three URLs:



```
const urls = [
  "http://catsanddogs.com/video.mp4",
  "http://catsanddogs.com/image.png",
  "http://catsanddogs.com/audio.mp3",
]
```



UMassAmherst | Manning College of Information & Computer Sciences

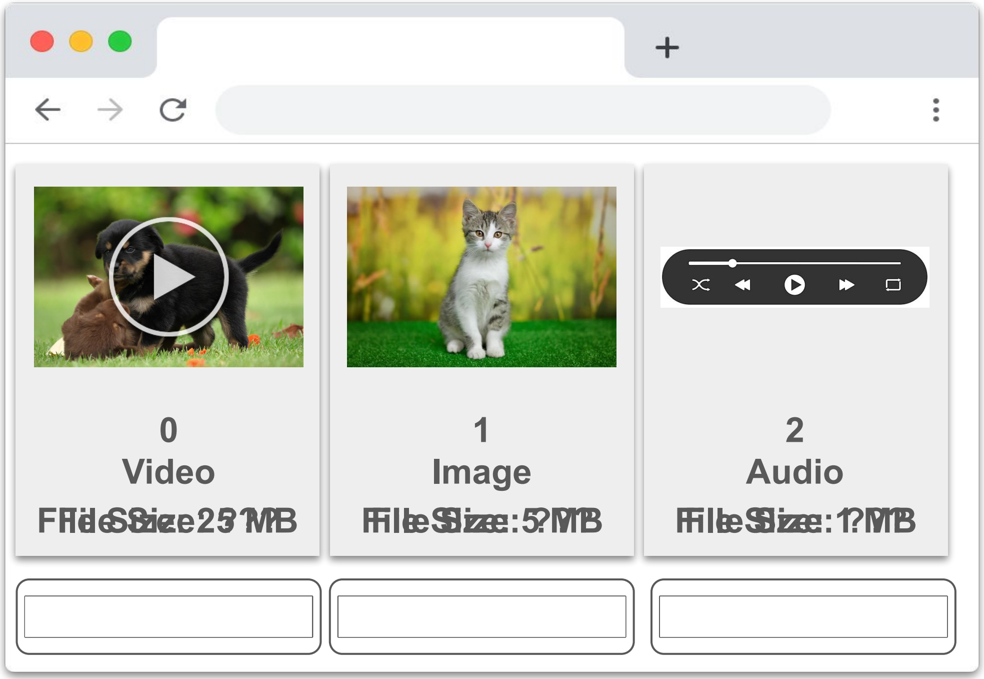
Motivation - Asynchronous Programming

With a basic implementation, we could:

1) Download all assets sequentially and then display them all at once to the user

or....

1) Download an asset and display it sequentially



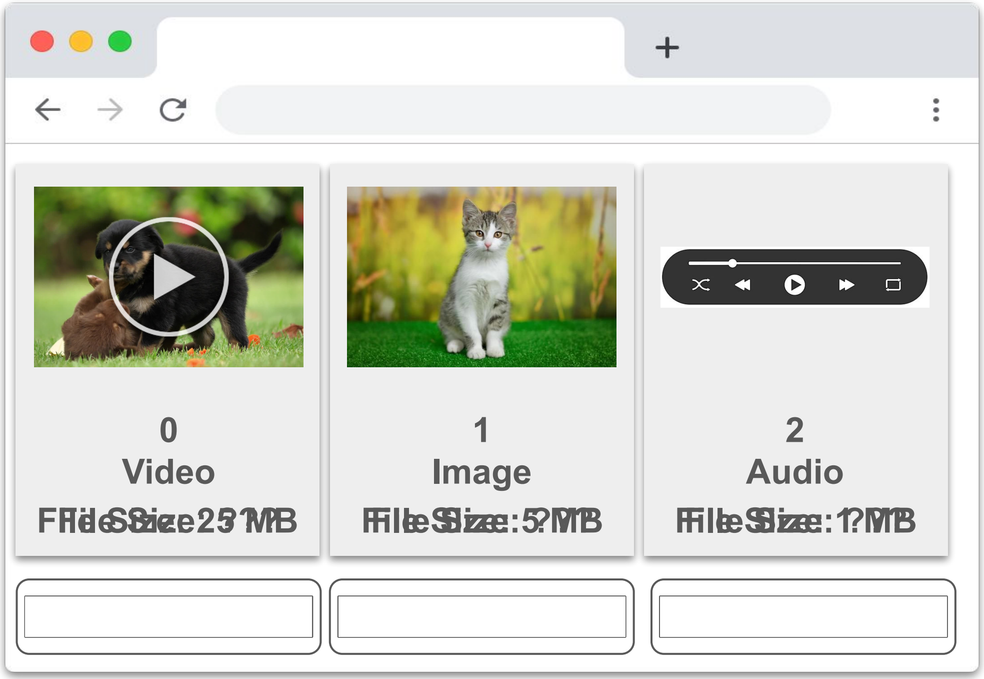
Motivation - Asynchronous Programming

But what if we could start downloading them all at once

AND

Display each right after they finish downloading?

The Promise API lets us do that!



Review: Asynchronous Programming

So far, we have mainly dealt with **synchronous** programming in this class. In general, this means our code runs line by line, and each line will **only** run once the previous line has finished running.

However, in JavaScript/TypeScript (and many other languages), we can also have asynchronous code, which we don't need to wait for before we execute other code.

We can see how asynchronous programming fits in with the example from the previous slides, where we want to download several different files for a website. Instead of waiting for each download to complete, we can begin downloading multiple files all at once!

UMassAmherst | Manning College of Information & Computer Sciences

Review: Promises

The **Promise** class is very useful for asynchronous programming in JS/TS. A Promise object typically represents the results of some asynchronous task.

At any point, a Promise object will be in one of three states:

- **Pending**: no value or result is available <u>yet</u> from the Promise
- **Fulfilled**: the task has <u>completed</u> and the Promise has resolved to a result
- **Rejected**: the task <u>failed</u> and the Promise rejects (usually with an error)

Note: once the Promise objects fulfills or rejects, the object is still a Promise. It does not turn directly into the result of the asynchronous task or the resulting error.

Review: Promises

An example of an asynchronous function in JS/TS is `setTimeout`. If we start a stopwatch when we run the following code, the code will print “`two`” after 1 second, “`three`” after 2 seconds, and then “`one`” after 4 seconds.

```
setTimeout(() => console.log(“one”), 4000);
setTimeout(() => console.log(“two”), 1000);
setTimeout(() => console.log(“three”), 2000);
```

Outputs →

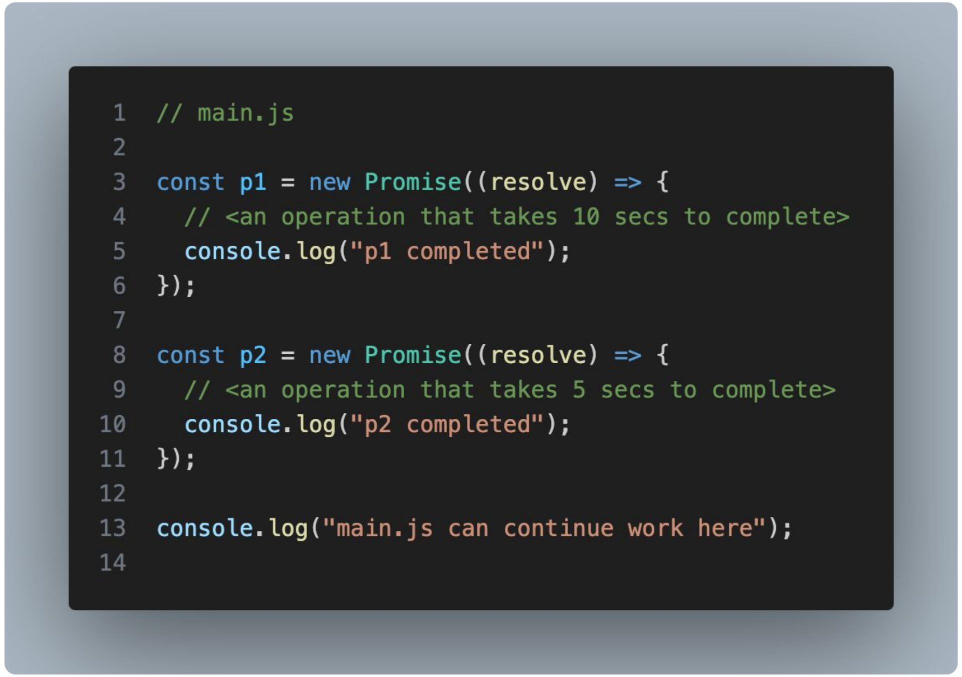
```
‘two’
‘three’
‘one’
```

Now, suppose we have an asynchronous function `timerPromise(n)`, which returns a Promise that resolves to “`hi!`” after n seconds. Then, in the first 5 seconds, `myPromise` is **pending**. After those 5 seconds, `myPromise` **fulfills**.

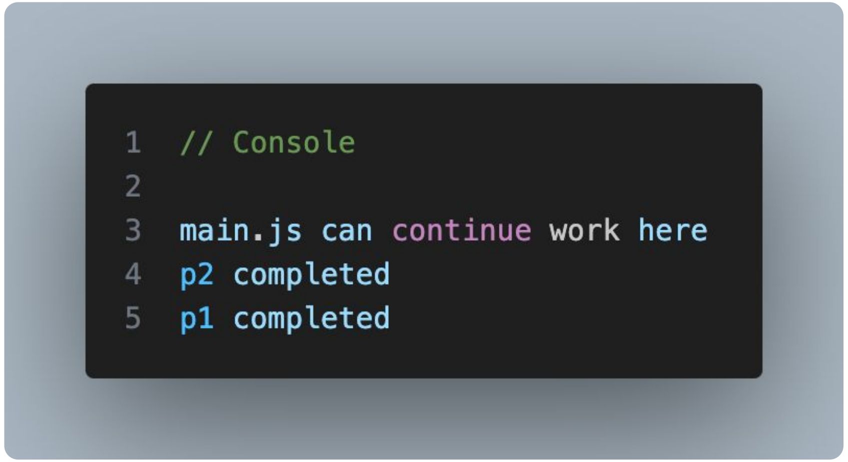
```
const myPromise = timerPromise(5); // after fulfilling, myPromise is
                                   still a Promise, just resolved to “hi!”
```

UMassAmherst | Manning College of Information & Computer Sciences

Creating Promises



```
// main.js

const p1 = new Promise((resolve) => {
  // <an operation that takes 10 secs to complete>
  console.log("p1 completed");
});

const p2 = new Promise((resolve) => {
  // <an operation that takes 5 secs to complete>
  console.log("p2 completed");
});

console.log("main.js can continue work here");

```

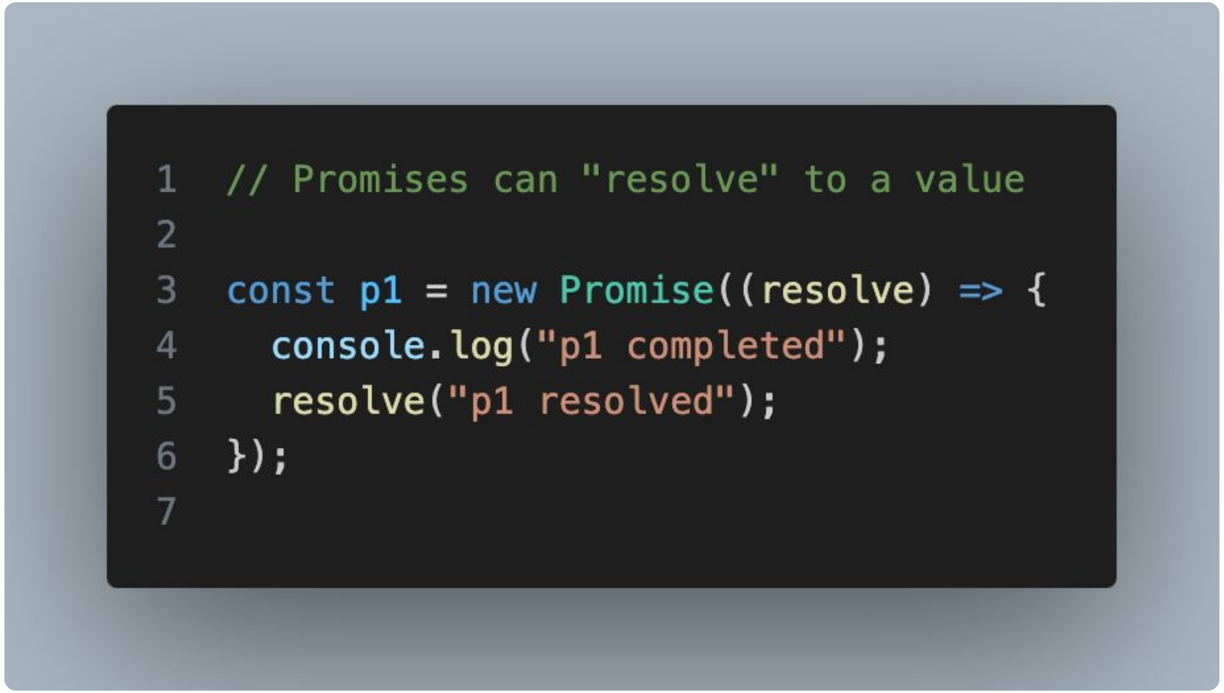


```
// Console

main.js can continue work here
p2 completed
p1 completed
```

Promise API

UMassAmherst | Manning College of Information & Computer Sciences

Resolving Promises



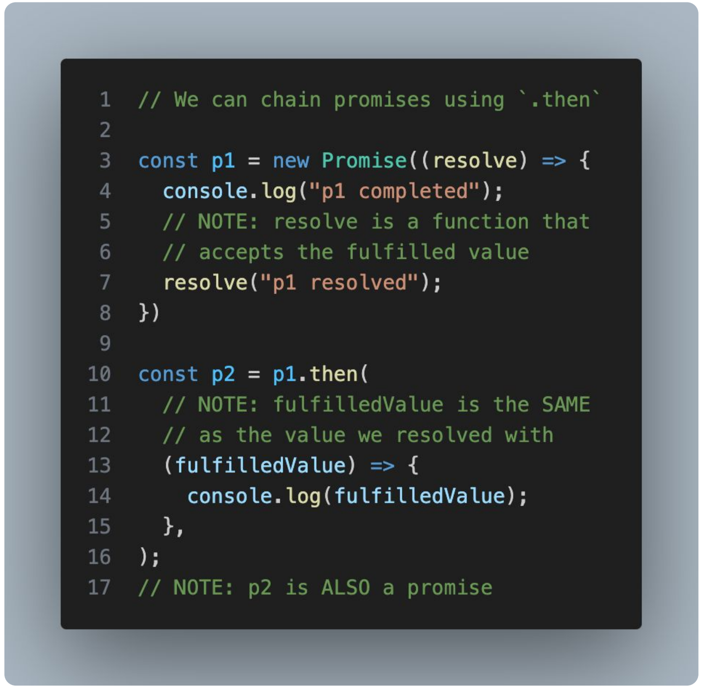
```javascript
// Promises can "resolve" to a value

const p1 = new Promise((resolve) => {
  console.log("p1 completed");
  resolve("p1 resolved");
});

```

What can we do with the fulfillment value?

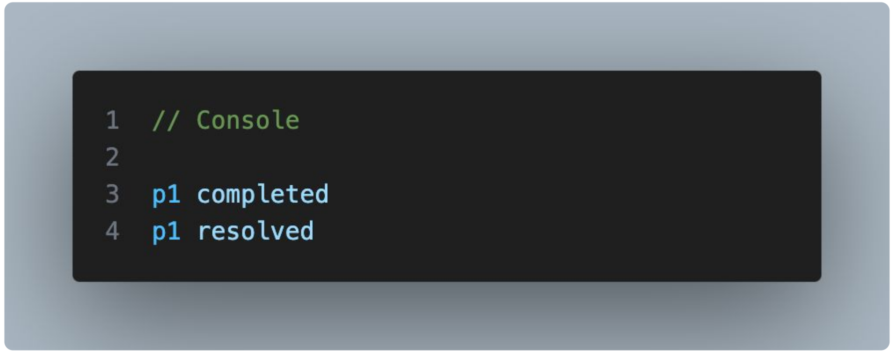
Promise API

UMassAmherst | Manning College of Information & Computer Sciences

Resolving Promises



```javascript
// We can chain promises using `.then`

const p1 = new Promise((resolve) => {
  console.log("p1 completed");
  // NOTE: resolve is a function that
  // accepts the fulfilled value
  resolve("p1 resolved");
})

const p2 = p1.then(
  // NOTE: fulfilledValue is the SAME
  // as the value we resolved with
  (fulfilledValue) => {
    console.log(fulfilledValue);
  },
);
// NOTE: p2 is ALSO a promise
```

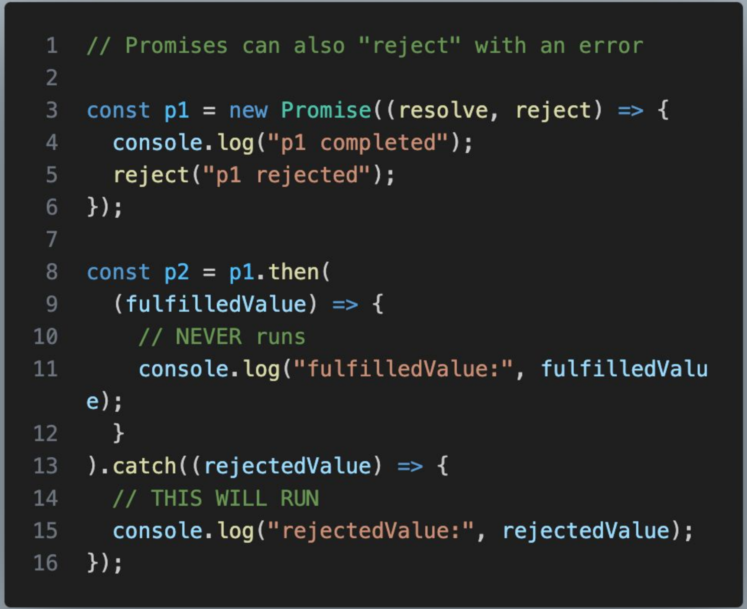


```
// Console

p1 completed
p1 resolved
```

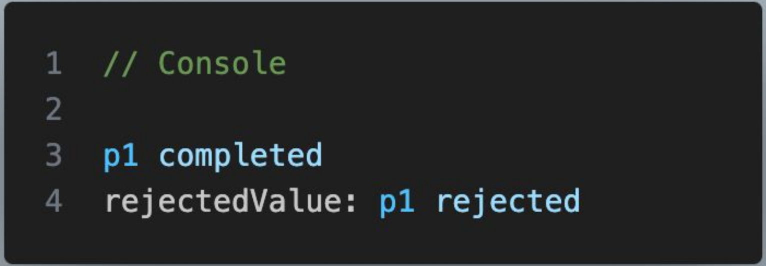
Read the MDN docs for the full specification of `Promise.prototype.then`

UMassAmherst | Manning College of Information & Computer Sciences

Promises that “reject”



```
// Promises can also "reject" with an error

const p1 = new Promise((resolve, reject) => {
  console.log("p1 completed");
  reject("p1 rejected");
});

const p2 = p1.then(
  (fulfilledValue) => {
    // NEVER runs
    console.log("fulfilledValue:", fulfilledValu
e);
  }
).catch((rejectedValue) => {
  // THIS WILL RUN
  console.log("rejectedValue:", rejectedValue);
});
```



```
// Console

p1 completed
rejectedValue: p1 rejected
```

Read the MDN docs for the full specification of `Promise.prototype.catch`

Review: Promise API

In summary, we can access the result of a Promise using **.then()** or **.catch()**. Suppose we have a variable `myPromise` storing a Promise.

- If `myPromise` **rejects**, then `myPromise.catch(myFunc)` will call the function `myFunc` on the value that `myPromise` rejected with.
- If `myPromise` **fulfills**, then `myPromise.then(myFunc)` will call the function `myFunc` on the value that `myPromise` fulfilled with.

.then() can accept 2 functions, where `myPromise.then(myFunc1, myFunc2)` will call `myFunc1` if `myPromise` **fulfills**, and `myFunc2` if `myPromise` **rejects**.

Also, `.then()` and `.catch()` return a Promise as well, allowing you to chain these methods together.

Exercise 1: Promises

Analyze the behavior of asynchronous code:

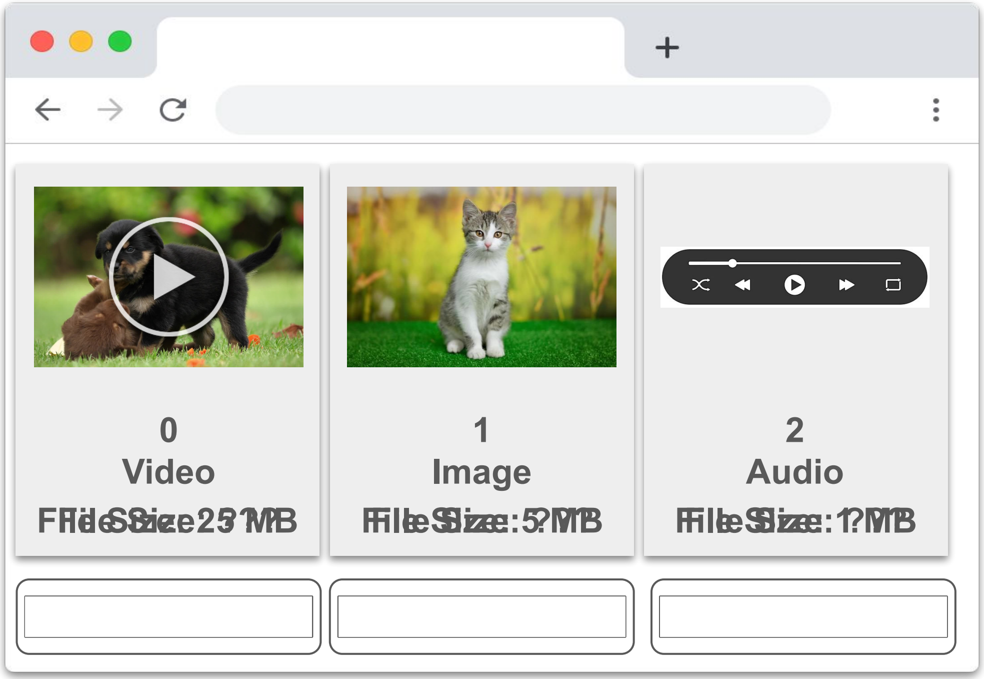
- **Get the starter code** from the website
- Open Exercise 1 in ex1.js
- In your groups, discuss the behavior of the code
- Answer the questions (1.1 - 1.6) on **Gradescope**

TAs will walk around to help ensure you are on the right track!

Basic Promise API

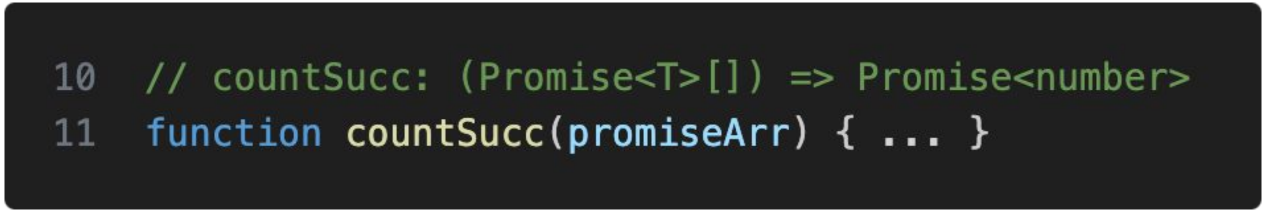
We know how to:

- ✓ Create Promises
- ✓ Chain Promises
- ✓ Handle rejected Promises
- ? But how do we **start several asynchronous tasks** at once and then **combine the result**?



Combining Promises

Task: Write a function `countSucc` that **accepts an array of promises** and returns a Promise that fulfills with the number of successful promises from the input array



```
// countSucc: (Promise<T>[]) => Promise<number>
function countSucc(promiseArr) { ... }
```

Definition from the docs:
`Promise.allSettled(promiseArr):` returns a Promise that always fulfills with an array of objects describing the outcome of each input promise:
`{ status: "fulfilled" | "rejected", value?: someType, reason?: errType }`

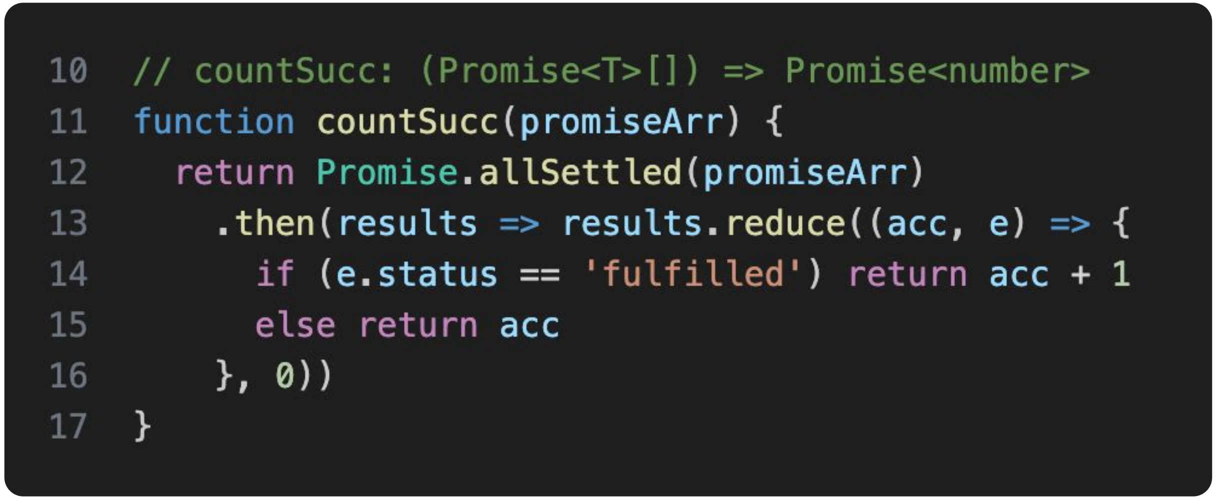
Discuss: Using `Promise.allSettled`, how would you implement `countSucc`? Briefly discuss with your group for 3-5 minutes. No need to submit any code!

Promise API

UMassAmherst | Manning College of Information & Computer Sciences

Combining Promises

Task: Write a function `countSucc` that **accepts an array of promises** and returns a Promise that fulfills with the number of successful promises from the input array



```
// countSucc: (Promise<T>[]) => Promise<number>
function countSucc(promiseArr) {
  return Promise.allSettled(promiseArr)
    .then(results => results.reduce((acc, e) => {
      if (e.status == 'fulfilled') return acc + 1
      else return acc
    }, 0))
}
```

Definition from the docs:
`Promise.allSettled(promiseArr)`:
returns a Promise that always fulfills with an array of objects describing the outcome of each input promise:
`{ status: "fulfilled" | "rejected", value?: someType, reason?: errType }`

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2: More Promises

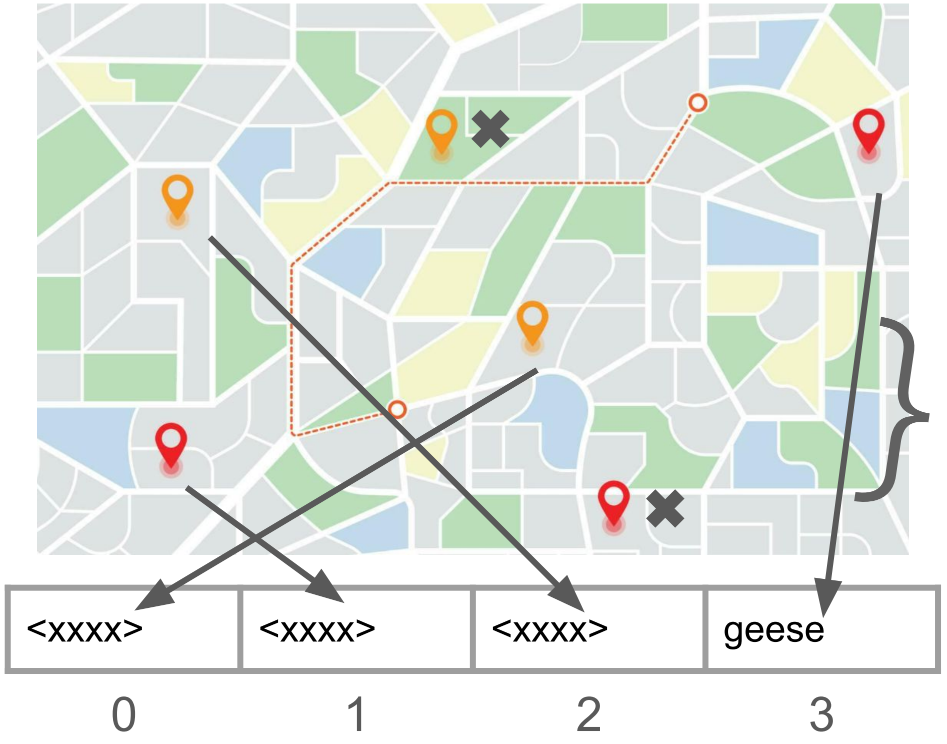
Scavenger Hunt!

- You are tasked with **finding a password** hidden across campus. Time is limited!
- Use Promise API to search for password pieces in a number of locations asynchronously.
- Assemble the password to *save the day!*

Exercise 2: More Promises

UMassAmherst | Manning College of Information & Computer Sciences

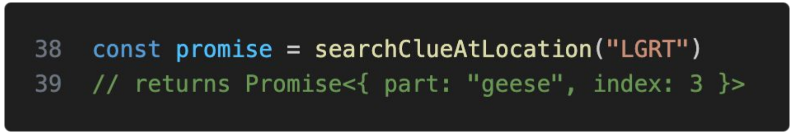
Scavenger Hunt!



You are given the function `searchClueAtLocation`.

Calling this function on certain locations on campus will produce a Promise. This Promise fulfills to a password part and its index in the entire password.

You will be given an array of locations to search for password parts at.



```
const promise = searchClueAtLocation("LGRT")
// returns Promise<{ part: "geese", index: 3 }>
```

Exercise 2: More Promises

Scavenger Hunt!

- **Get the starter code** from the website
- Open Exercise 2 in ex2.js
- In your groups, **find the password**
- Answer the questions on Gradescope

TAs will walk around to help ensure you are on the right track

UMassAmherst | Manning College of Information & Computer Sciences

Solution: Scavenger Hunt

```
const passwordParts = Array(locations.length).fill("");
const searchPromises = locations.map(searchClueAtLocation);
// ...
```

We begin by initializing an array to store password parts in, such that we can store the password part from each location. We then use the given **searchClueAtLocation** function, calling it on each location using **map**. We store the Promises returned in the array **searchPromises**.

UMassAmherst | Manning College of Information & Computer Sciences

Solution: Scavenger Hunt

```
const passwordParts = Array(locations.length).fill("");
const searchPromises = locations.map(searchClueAtLocation);
Promise.allSettled(searchPromises).then(results => {
    // ...
});
```

Next, we use **Promise.allSettled** to combine all the Promises we obtained into one Promise. This combined Promise will resolve only once all Promises in **searchPromises** are no longer pending.

The combined Promise is also guaranteed to fulfill with an array of objects:
`{ status: "fulfilled" | "rejected", value?: someType, reason?: errType }`

These objects describe the results of each Promise from the array **searchPromises**. Using **.then()**, the array of objects from **Promise.allSettled** is passed in as the argument **results**.

Solution: Scavenger Hunt

```
const passwordParts = Array(locations.length).fill("");
const searchPromises = locations.map(searchClueAtLocation);
Promise.allSettled(searchPromises).then(results => {
    results.forEach(result => {
        if (result.status === "fulfilled") {
            // ...
        }
    });
    // ...
});
```

For each result object in **results**, we only want to examine the ones that correspond to a Promise that fulfilled. We therefore check this by looking at the **status** property of each result object.

Solution: Scavenger Hunt

```
const passwordParts = Array(locations.length).fill("");
const searchPromises = locations.map(searchClueAtLocation);
Promise.allSettled(searchPromises).then(results => {
    results.forEach(result => {
        if (result.status === "fulfilled") {
            const { part, index } = result.value;
            passwordParts[index] = part;
        }
    });
    // ...
});
```

If the result is for a fulfilled Promise, then we know that **result.value** will contain a password part and its index. We therefore extract these from **result.value** and store the password part in our **passwordParts** array.

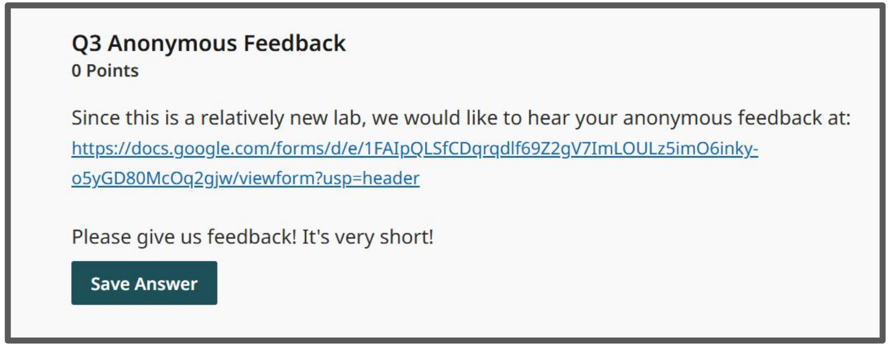
UMassAmherst | Manning College of Information & Computer Sciences

Solution: Scavenger Hunt

```
const passwordParts = Array(locations.length).fill("")
const searchPromises = locations.map(searchClueAtLocation);
Promise.allSettled(searchPromises).then(results => {
    results.forEach(result => {
        if (result.status === "fulfilled") {
            const { part, index } = result.value
            passwordParts[index] = part
        }
    });
    const password = passwordParts.join("");
    console.log(`Password: ${password}`);
    completeHunt(password);
});
```

Lastly, after going through all the results we obtained from the different locations, we combine the password parts and complete our scavenger hunt!

Lab Feedback



Q3 Anonymous Feedback
0 Points

Since this is a relatively new lab, we would like to hear your anonymous feedback at: https://docs.google.com/forms/d/e/1FAIpQLSfCDqrqdlf69Z2gV7ImLOULz5imO6inky-o5yGD80McOq2gjw/viewform?usp=header

Please give us feedback! It's very short!

[Anonymous Feedback](https://docs.google.com/forms/d/e/1FAIpQLSfCDqrqdlf69Z2gV7ImLOULz5imO6inky-o5yGD80McOq2gjw/viewform?usp=header)