



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**



Exercise 1
Get hands-on with
Promise API



Exercise 2
Scavenger Hunt

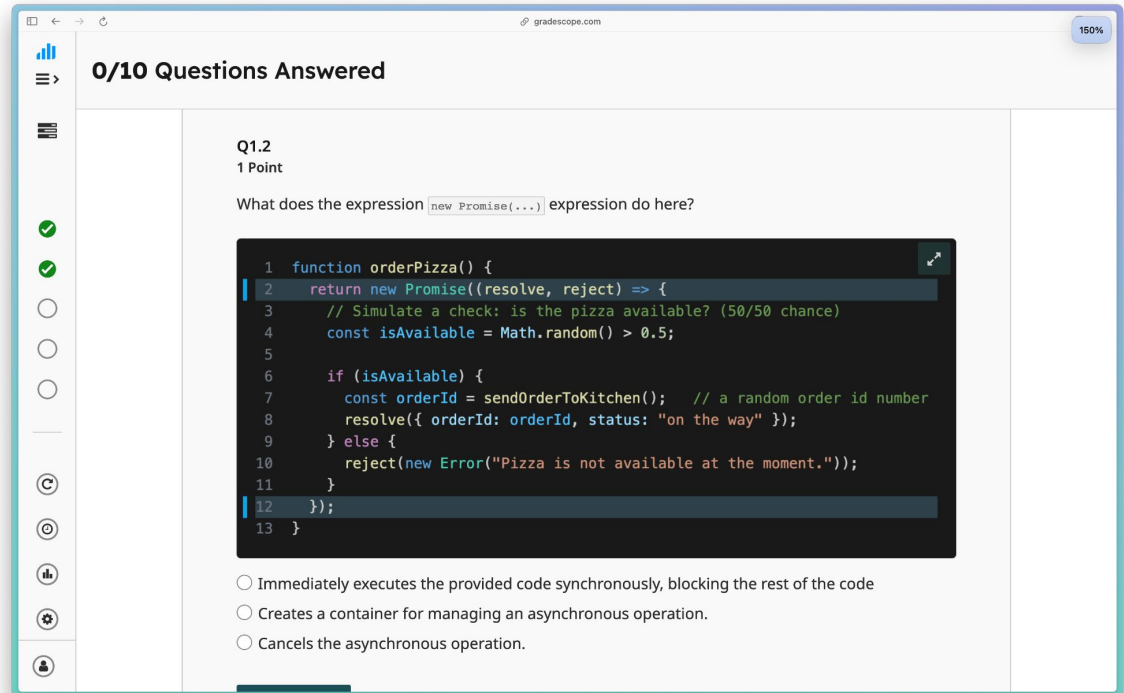
**Use
Gradescope**

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.

New Lab Format for Today

We have some questions on
Gradescope today :)



The screenshot shows a Gradescope question titled "Q1.2" worth "1 Point". The question asks: "What does the expression `new Promise(...)` expression do here?". A code editor displays the following JavaScript code:

```
1 function orderPizza() {  
2   return new Promise((resolve, reject) => {  
3     // Simulate a check: is the pizza available? (50/50 chance)  
4     const isAvailable = Math.random() > 0.5;  
5  
6     if (isAvailable) {  
7       const orderId = sendOrderToKitchen(); // a random order id number  
8       resolve({ orderId: orderId, status: "on the way" });  
9     } else {  
10      reject(new Error("Pizza is not available at the moment."));  
11    }  
12  });  
13 }
```

Below the code, there are three radio button options:

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

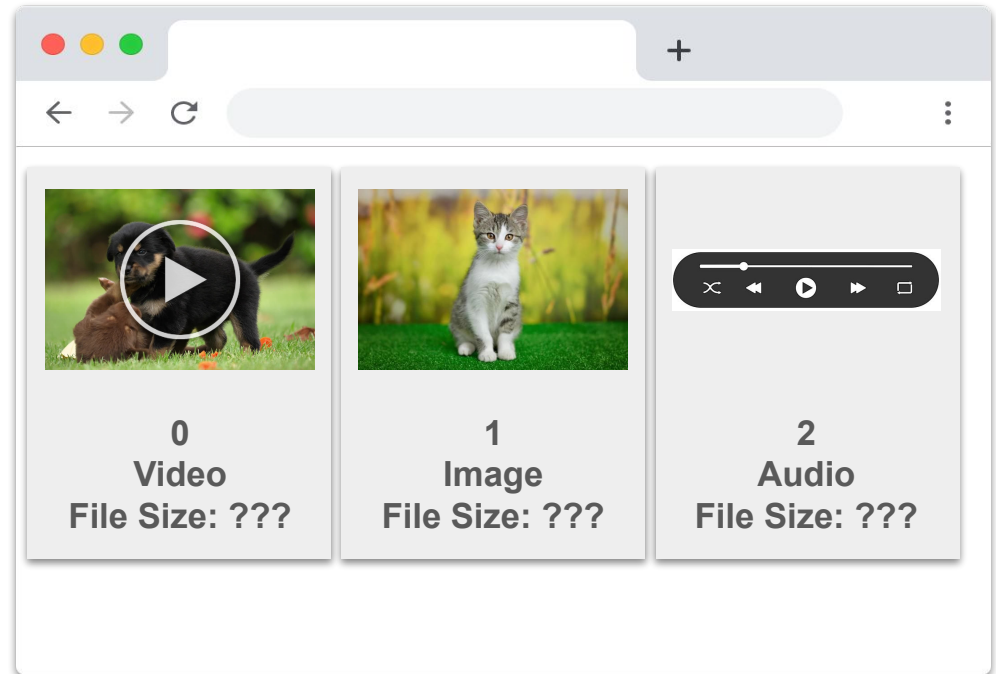
The interface also shows a sidebar with navigation icons and a top bar indicating "0/10 Questions Answered" and a "150%" zoom level.

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

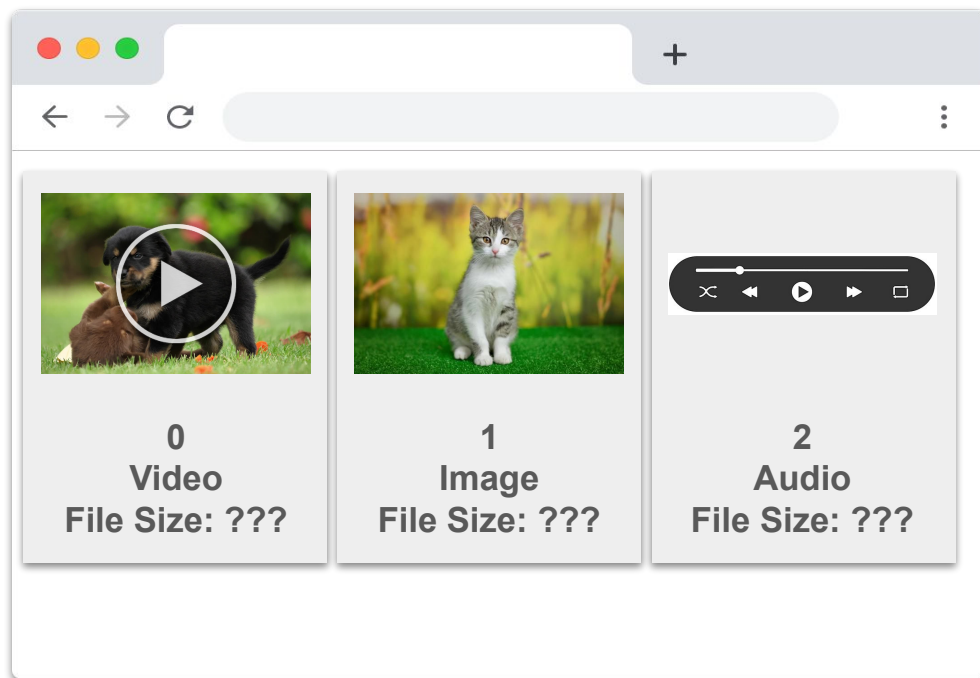
Goal: Give user best user experience



Motivation - Asynchronous Programming

We have three URLs:

```
1 const urls = [  
2   "http://catsanddogs.com/video.mp4",  
3   "http://catsanddogs.com/image.png",  
4   "http://catsanddogs.com/audio.mp3",  
5 ]
```



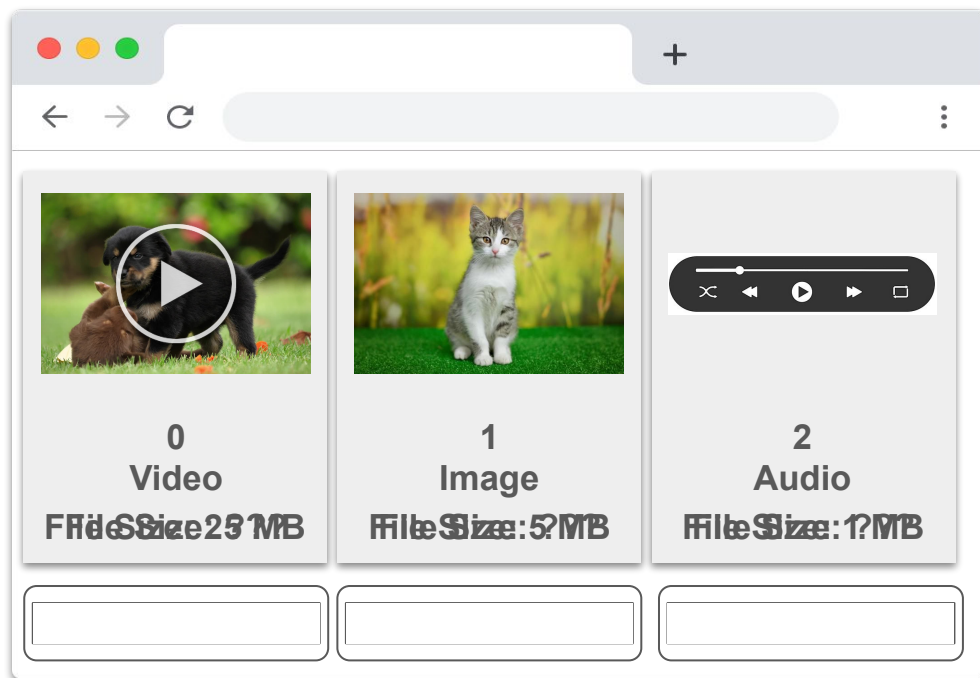
Motivation - Asynchronous Programming

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



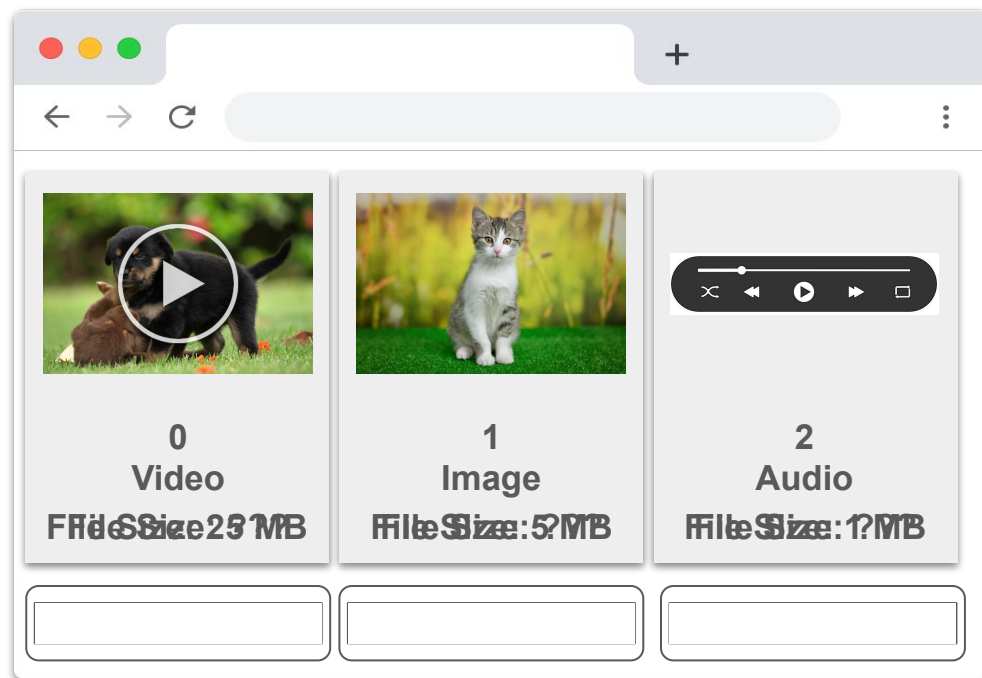
Motivation - Asynchronous Programming

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!

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 yet from the Promise
- **Fulfilled:** the task has completed and the Promise has resolved to a result
- **Rejected:** the task failed 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.

Creating Promises

```
1 // main.js
2
3 const p1 = new Promise((resolve) => {
4   // <an operation that takes 10 secs to complete>
5   console.log("p1 completed");
6 });
7
8 const p2 = new Promise((resolve) => {
9   // <an operation that takes 5 secs to complete>
10  console.log("p2 completed");
11 });
12
13 console.log("main.js can continue work here");
14
```

```
1 // Console
2
3 main.js can continue work here
4 p2 completed
5 p1 completed
```

Resolving Promises

```
1 // Promises can "resolve" to a value
2
3 const p1 = new Promise((resolve) => {
4   console.log("p1 completed");
5   resolve("p1 resolved");
6 });
7
```

**What can we do with
the fulfillment value?**

Resolving Promises

```
1 // We can chain promises using `.then`
2
3 const p1 = new Promise((resolve) => {
4   console.log("p1 completed");
5   // NOTE: resolve is a function that
6   // accepts the fulfilled value
7   resolve("p1 resolved");
8 })
9
10 const p2 = p1.then(
11   // NOTE: fulfilledValue is the SAME
12   // as the value we resolved with
13   (fulfilledValue) => {
14     console.log(fulfilledValue);
15   },
16 );
17 // NOTE: p2 is ALSO a promise
```

```
1 // Console
2
3 p1 completed
4 p1 resolved
```

Read the [MDN docs](#) for the full specification
of `Promise.prototype.then`

Promises that “reject”

```
1 // Promises can also "reject" with an error
2
3 const p1 = new Promise((resolve, reject) => {
4   console.log("p1 completed");
5   reject("p1 rejected");
6 });
7
8 const p2 = p1.then(
9   (fulfilledValue) => {
10     // NEVER runs
11     console.log("fulfilledValue:", fulfilledValue);
12   }
13 ).catch((rejectedValue) => {
14   // THIS WILL RUN
15   console.log("rejectedValue:", rejectedValue);
16 });
```

```
1 // Console
2
3 p1 completed
4 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:

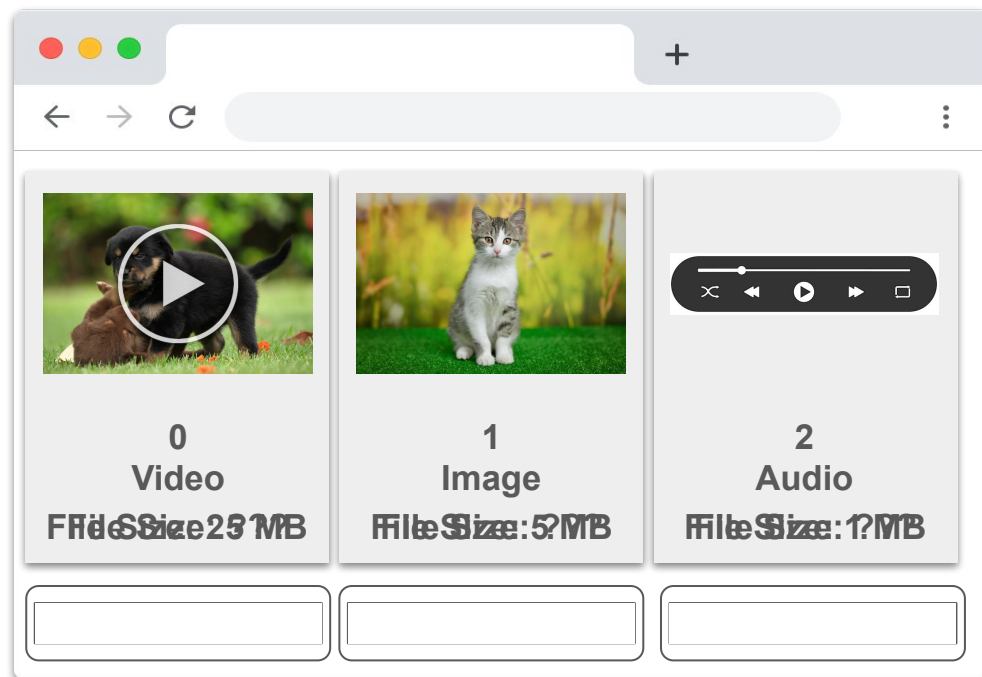
- **Get the starter code** from the website
- Open Exercise 1 in ex1.js
- In your groups, discuss the behavior of the code
- Answer the questions (1.1 - 1.6) on **Gradescope**

**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

```
10 // countSucc: (Promise<T>[]) => Promise<number>
11 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`? Discuss with your group for 3-5 minutes. No need to submit any code for this!

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

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.

```
38 const promise = searchClueAtLocation("LGRT")
39 // 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

