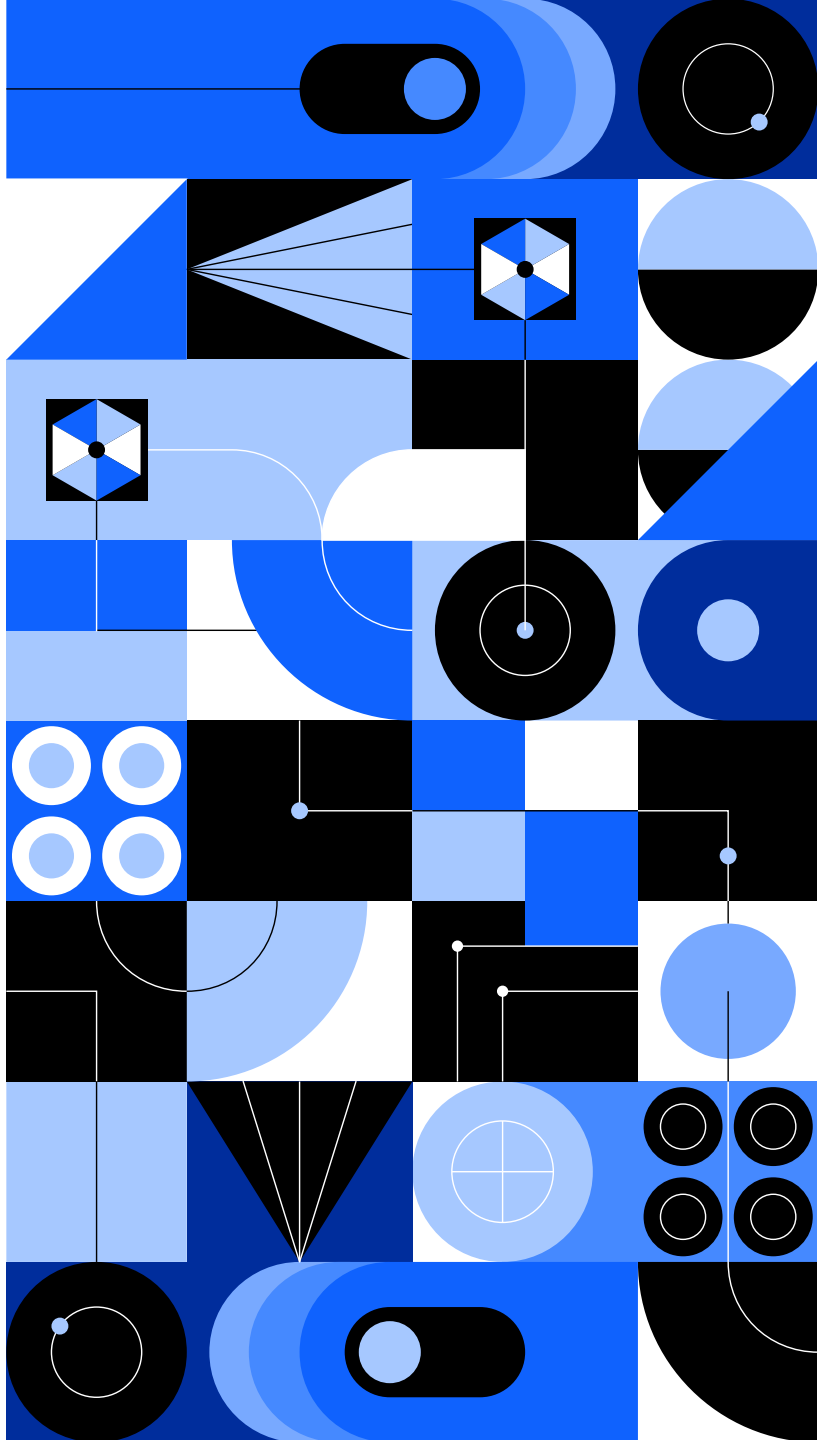
IBM Accelerate Software Developer Track

Wednesdays June 5th – July 25th
6:00pm – 8:00pm Eastern Time



• Optional practice test

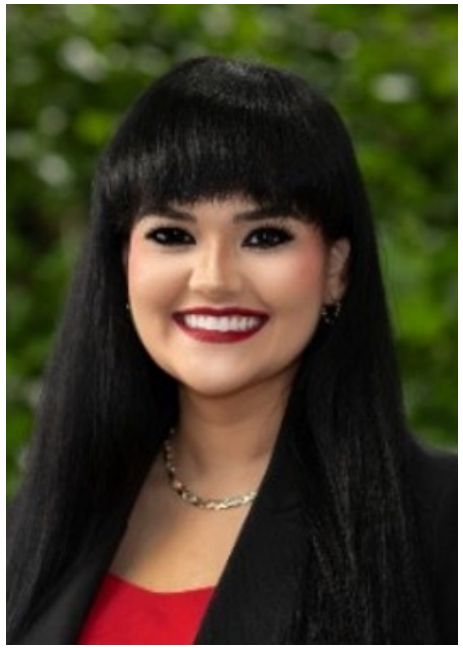
Wednesday, June 19, 2024

JavaScript Synchronicity, Software Testing

Today's speakers:

Austin Eberle
Sr. Firmware Developer,
Flash Systems Storage,
IBM Infrastructure

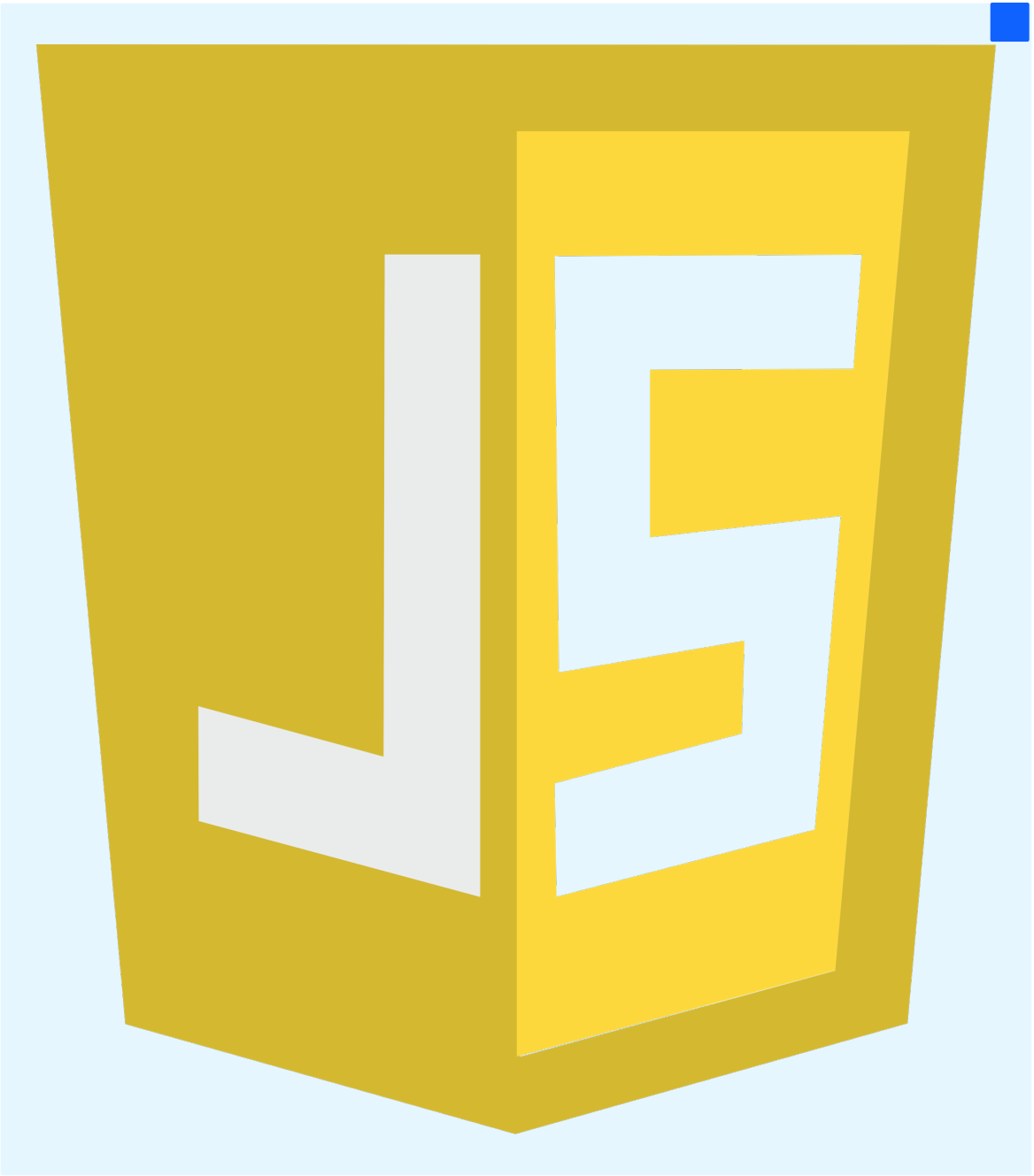
Sujeily Fonseca-Gonzalez
Software Engineering Manager,
Technical Leader,
IBM Software,
SW Track Leader



Session Agenda

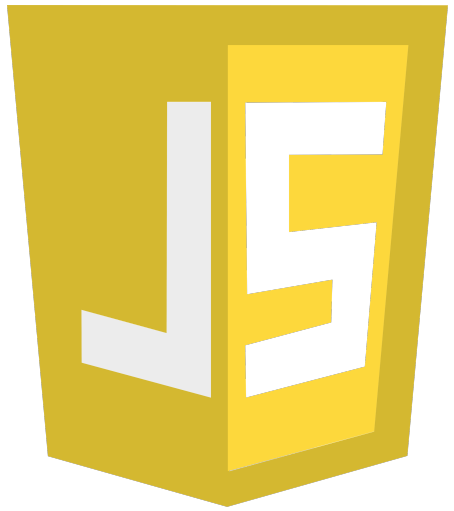
1. JavaScript Synchronicity
   Asynchronous vs synchronous events, Callbacks, Promises
2. Software Testing
   Why we test, types of testing, testing JavaScript and React
3. Q&A
4. Lab / Project and GitHub Classroom
5. Required HackerRank test
6. Next Steps...

JavaScript Synchronicity



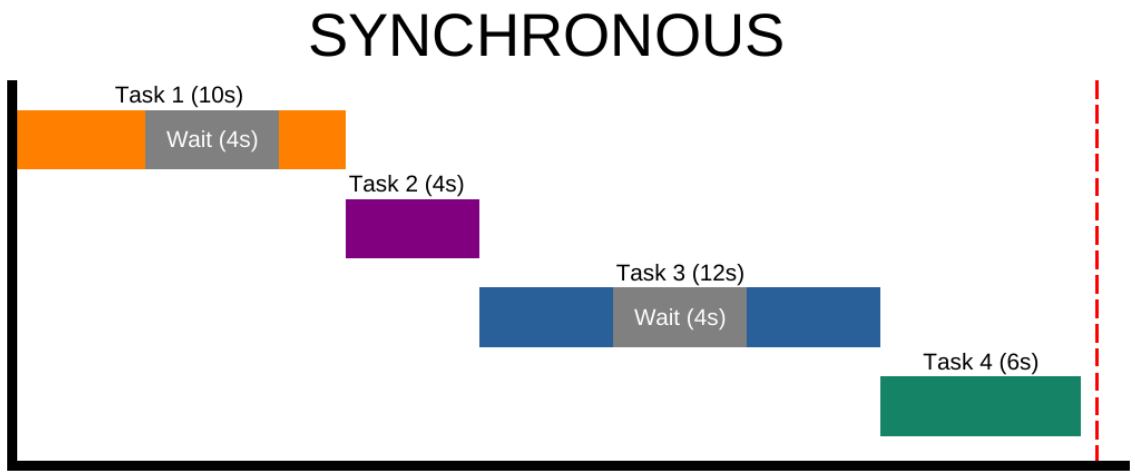
IBM

Synchronous vs Asynchronous events

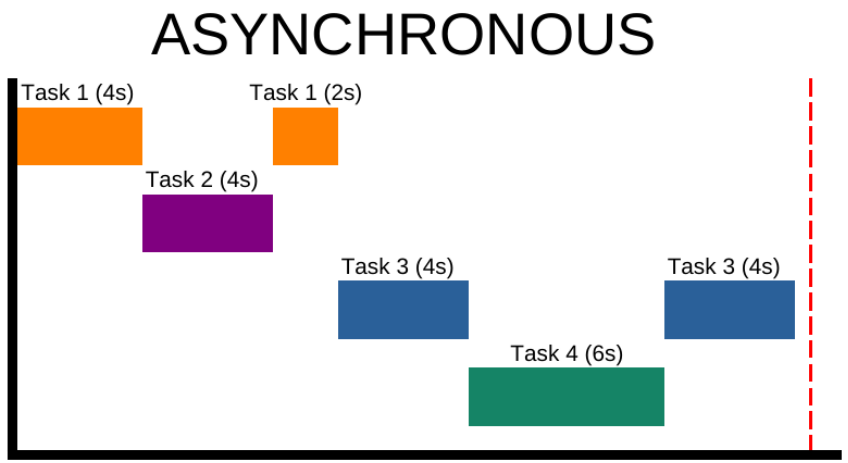


- In a synchronous program, each operation is dependent on the completion of the one before it
- In an asynchronous program, operations can be completed in any order, sometimes even concurrently
- By design, JavaScript is a synchronous, single-threaded, language, however, by using **callbacks** and the **promise** object, asynchronous actions can be implemented

Synchronous vs Asynchronous events (cont.)



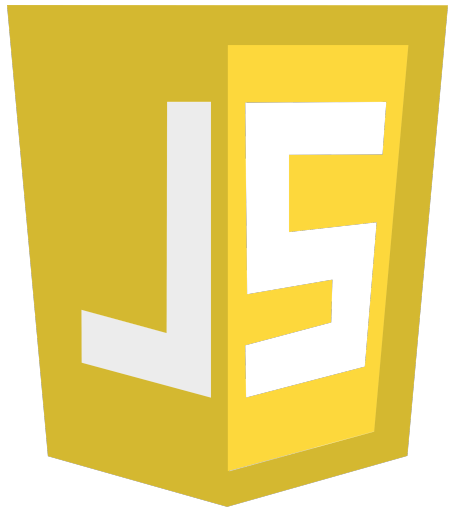
Time Taken: 32 Seconds



Time Taken: 24 Seconds

Demo: blocks1.js

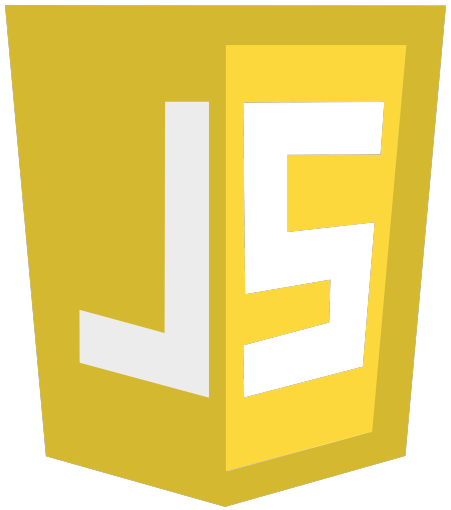
JavaScript Callbacks



- A callback is simply a function that is passed into another function as an argument
- The receiving function is expected to “call back” the callback function as part of its job
- A callback is often back on the level of the original caller
- Nesting callbacks is a powerful pattern, but quickly becomes unwieldy after a few layers
  - Often referred to as ‘callback hell’

Demo: blocks2.js, blocks3.js

JavaScript Promises



- The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value
- Asynchronous operations can generally be split into two parts
  - Producing code: code that takes some time to execute (e.g. API call, expensive calculation)
  - Consuming code: code that must wait for the result
- The JavaScript Promise object provides an interface to link producing and consuming code

JavaScript Promises (cont.)

- Promises can be created using the Promise object constructor
- The promise constructor takes in a function, called an **executor**
  - The executor function receives two params from the Promise object
  - First param is a function to be called by the developer when the promise is resolved
  - Second param is a function to be called by the developer when the promise is rejected
- The then() method on a promise is used to queue up handlers after a promise is resolved or rejected
  - Takes in two functions as params
  - First param is called when the promise is resolved
  - Second param is called when the promise is rejected

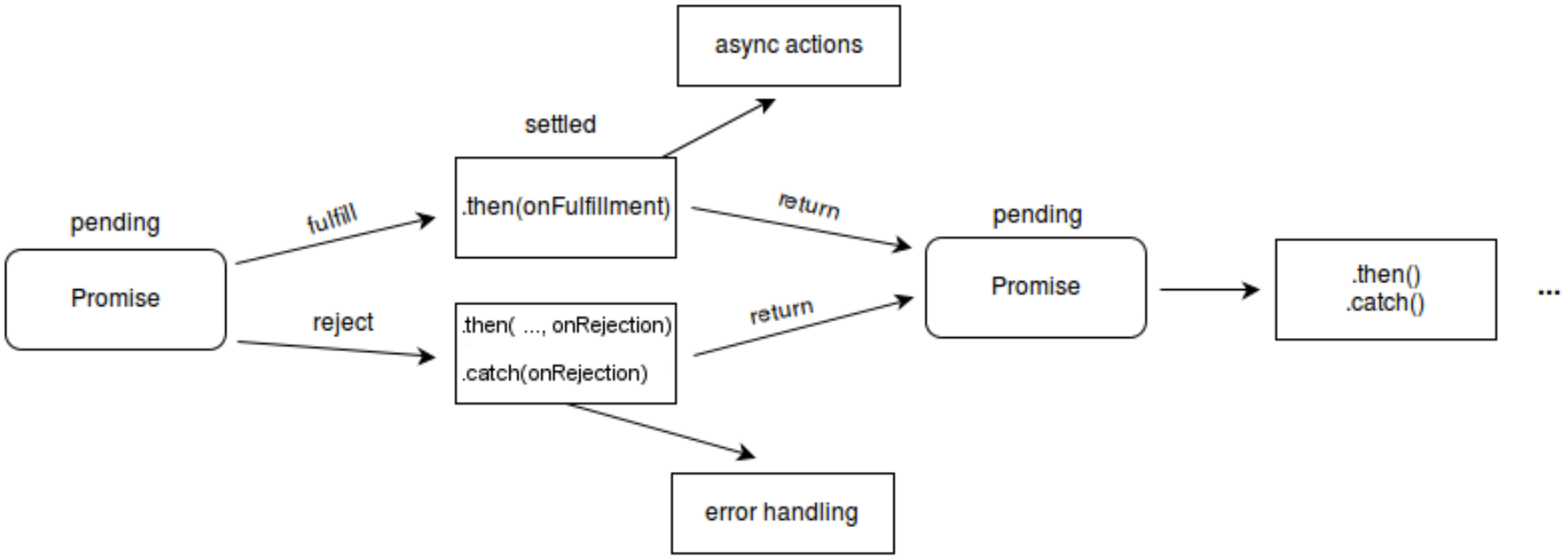
```
let myPromise = new Promise(
function(resolve, reject) {
// "Producing Code" (May take some time)

  resolve(); // when successful
  reject();  // when error
}
);

// "Consuming Code" (Must wait for a fulfilled
Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```


Demo: promises.js

JavaScript Promises (cont. 2)

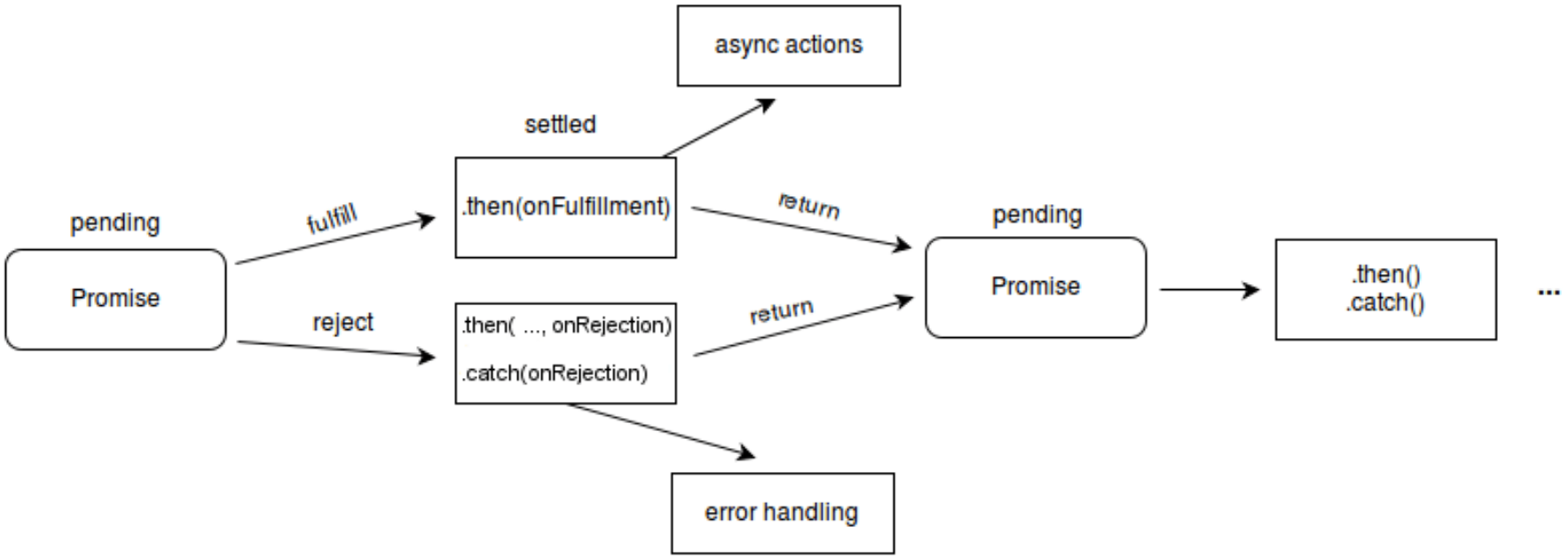
- A promise is in one of three states:
  - **pending**: initial state, neither fulfilled nor rejected.
  - **fulfilled**: meaning that the operation was completed successfully.
  - **rejected**: meaning that the operation failed.

- A promise is said to be **settled** if it is either fulfilled or rejected, but not pending.



JavaScript Promises (cont. 3)

- The function then() is used to associate further action (consuming code) with a promise that has settled
  - then() takes in two arguments, the first is a callback for the *fulfilled* promise and the second is a callback for a *rejected* promise
  - then() always returns another promise, allowing promises to be chained
- When working with chained promises, catch() and finally() can be used to assist in promise processing
  - catch(): handles *rejected* promises for any promise in the chain (equivalent to second param on then())
  - finally(): always ran last, regardless of whether previous promises are *fulfilled* or *rejected*



Demo: blocks4.js

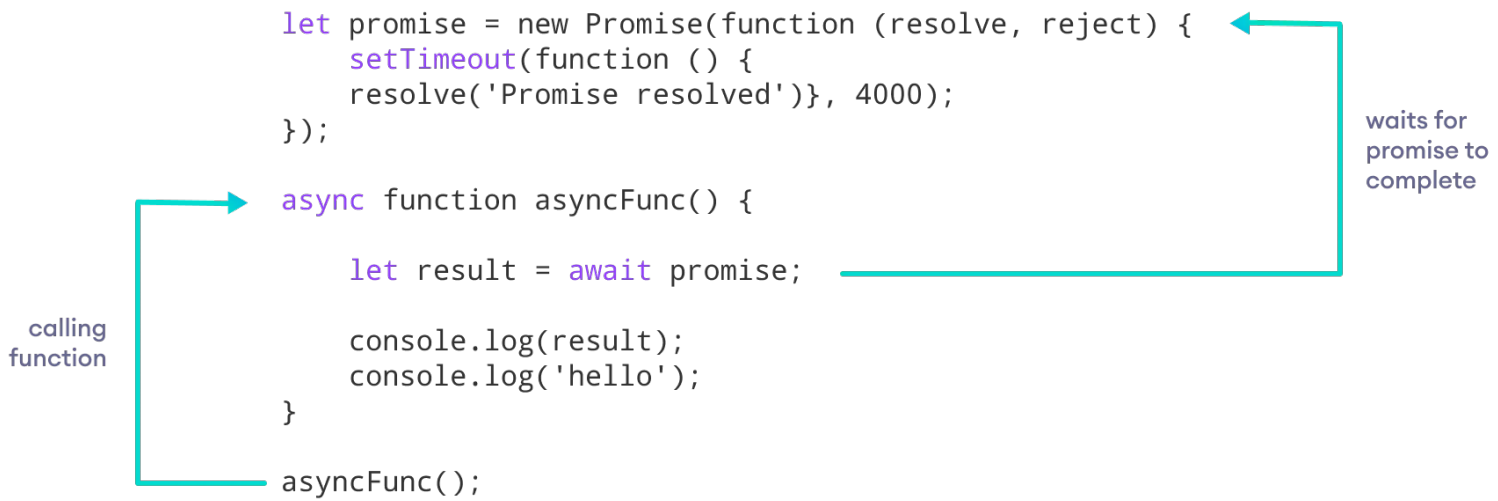
JavaScript Promises (cont. 4)

- The promise object also provides several static methods that take in an array of promises, attempts to fulfill them, and then returns a single promise
  - Promise.all()
    - If all promises are fulfilled, returns a promise with an array of fulfillment values
    - If any promises are rejected, immediately rejects with the first rejection
  - Promise.allSettled()
    - When all promises are settled, returns a promise with an array of settled values, including both fulfills and rejects
  - Promise.any()
    - Fulfills when the first supplied promise fulfills
    - Only rejects if every supplied promise rejects
  - Promise.race()
    - Returns when the first supplied promise is settled

Demo: blocks5.js

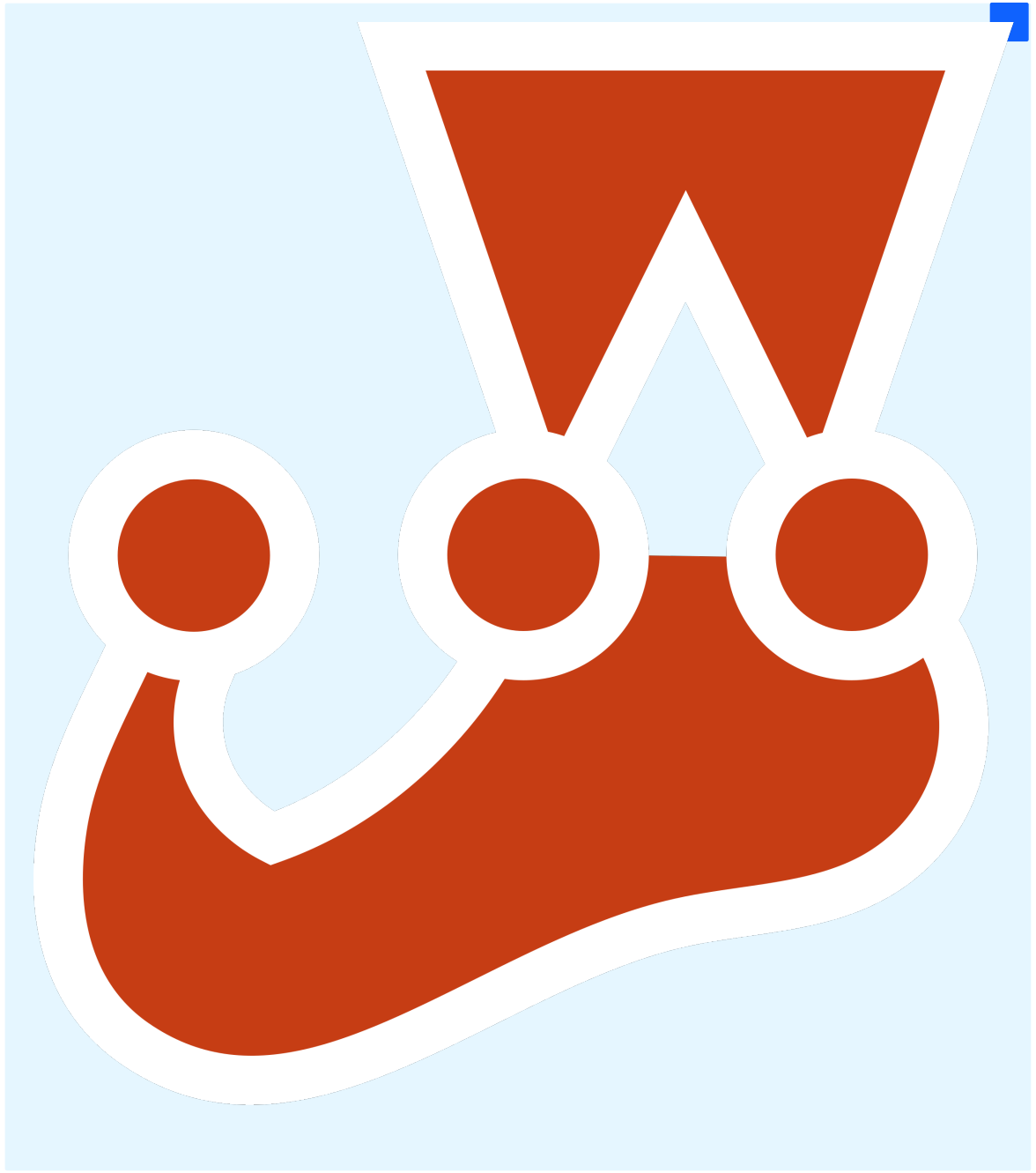
JavaScript Promises, Async - Await

- Async and await build on promises
- The **async** function declaration declares an async function where the **await** keyword is permitted within the function body
- The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains



Demo: blocks6.js

Software Testing



Why we test

Direct cost of fixing Bugs

- Bugs caught by a customer in a production environment are extremely disruptive
- Requires a rushed design, development, and test cycle to triage
- Can interrupt other projects, developments

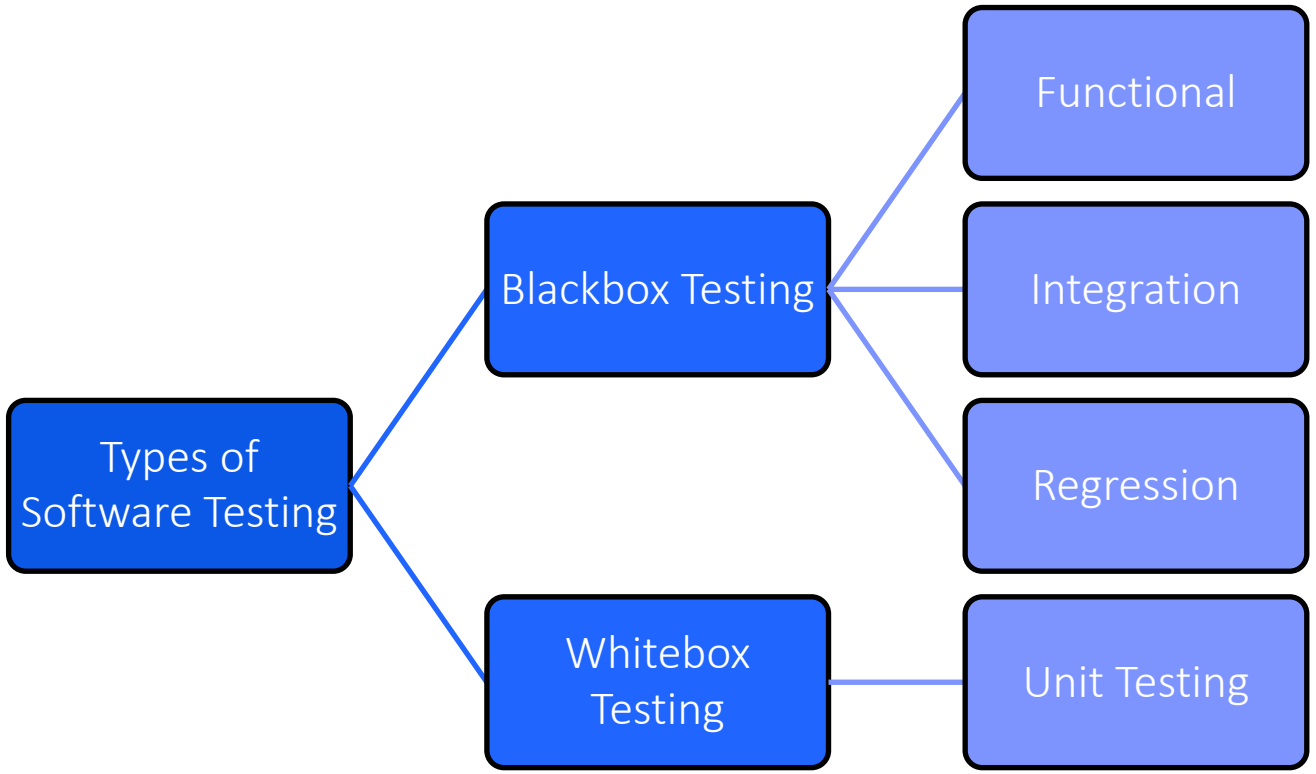
Indirect costs

- Warranty and replacement costs for defective products
- Extra development time required to fix, may even need to send physical technicians
- Potential for lawsuits

Intangible costs

- Damage to reputation

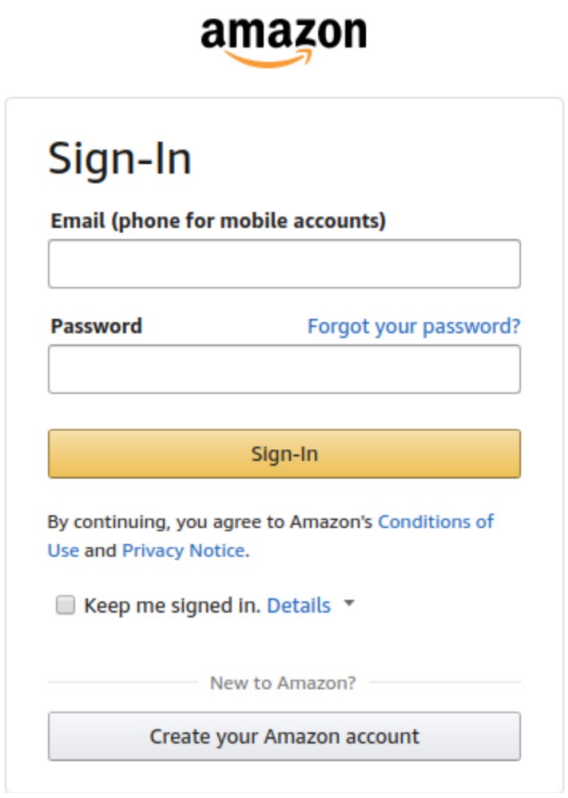
Types of Software testing



* Please note that there are more levels of black box testing, not illustrated here

Types of Software testing (cont.)

Functional Testing



Functional testing

Checking functions by emulating business scenarios, based on functional requirements

Regression testing

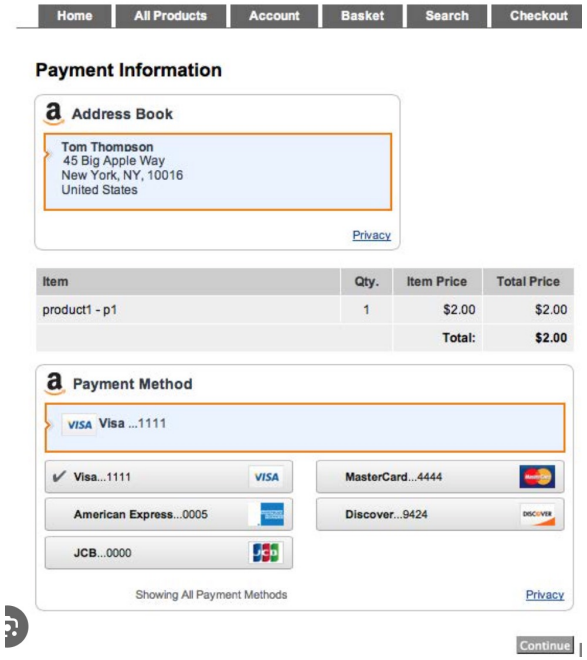
Checking whether new features break or degrade functionality.

Performed in parallel to development

Integration testing

Ensuring that software components or functions operate together

Integration Testing



Testing JavaScript and React

- **Jest** is arguably the most popular JavaScript testing framework and is developed and maintained by Meta, the developer and maintainer of React

- In conjunction with Jest, Dom Testing Library, or more specifically **React Testing Library** provides sets of utility functions for testing against the DOM (Document Object Model – essentially how the browser sees a web page)

Testing JavaScript

sumNums.js

```
function sumNums(a, b) {
  if (isNaN(a) || isNaN(b)) throw new Error("Parameter is not a number!");

  return a + b;
}

export default sumNums;

```

- Test suites in Jest are defined by creating a file with *.test* in the extension and defining test() functions in the form of test(*test name*, *test function*)
- Test suites can be run using the command: `npm test`

sumNums.test.j
s

```
import sumNums from "./sumNums";

test("adds 1 + 2 to equal 3", () => {
  expect(sumNums(1, 2)).toBe(3);
});

test("add string to a number to throw an error", () => {
  expect(() => sumNums("x", 1)).toThrow();
});
```

- Inside of a Jest test, expect() can be used in conjunction with matchers, such as toBe(), to ensure a value meets certain conditions
- There are over 30 other matchers, including modifiers like .not
  - See docs: https://jestjs.io/docs/getting-started

Demo: sumNums.js, sumNums.test.js

Testing React

- React testing library provides a ‘virtual’ render method and a set of queries to find and test elements on a web page
  - Many queries and matchers available with different use cases
  - See Docs: https://testing-library.com/docs/

```
import {render, screen} from '@testing-library/react' // (or /dom, /vue, ...)

test('should show login form', () => {
  render(<Login />)
  const input = screen.getByLabelText('Username')
  // Events and assertions...
})
```

- React testing library also provides a function fireEvent(), which can be used to simulate user interactions such as mouse clicks, scrolling, and text input

```
fireEvent(node: HTMLElement, event: Event)
```

```
// <button>Submit</button>
fireEvent(
  getByText(container, 'Submit'),
  new MouseEvent('click', {
    bubbles: true,
    cancelable: true,
  }),
)
```

Testing React

App.js

```
1  import * as React from "react";
2
3  import sumNums from "./function/sumNums";
4
5  import "./App.css";
6
7  function App() {
8    const [showMessage, setShowMessage] = React.useState(false);
9
10   return (
11     <div className="App">
12       <h3>Check out this slick test app!</h3>
13       <hr />
14       <h3>1 + 1 = {sumNums(2, 3)}</h3>
15       <hr />
16       <button onClick={() => setShowMessage((lastState) => !lastState)}>
17         Show secret message
18       </button>
19       <hr />
20       {showMessage && (
21         <h4 style={{ fontStyle: "italic" }}>This is a secret message</h4>
22       )}
23     </div>
24   );
25 }
26
27 export default App;
```

App.test.js

```
1  import { render, fireEvent, screen } from "@testing-library/react";
2  import App from "./App";
3
4  test("renders test app", () => {
5    render(<App />);
6    const headerElement = screen.queryByText("Check out this slick test app!");
7    expect(headerElement).toBeInTheDocument();
8  });
9
10 test("show / hide secret message", () => {
11   render(<App />);
12
13   const button = screen.getByText("Show secret message");
14
15   fireEvent.click(button);
16   expect(screen.queryByText("This is a secret message")).toBeInTheDocument();
17
18   fireEvent.click(button);
19   expect(
20     screen.queryByText("This is a secret message")
21   ).not.toBeInTheDocument();
22 });
23
```


Demo: App.js, App.test.js

Testing React: pro-tip

Can’t find the component you’re looking to test using queryBy* or getBy*?

Try giving it a test id:

- In your React code’s render method, add a field “data-testid” to the component you are looking to test

- In your test, use the function getByTestId() to find the component with the specified data-testid

App.js

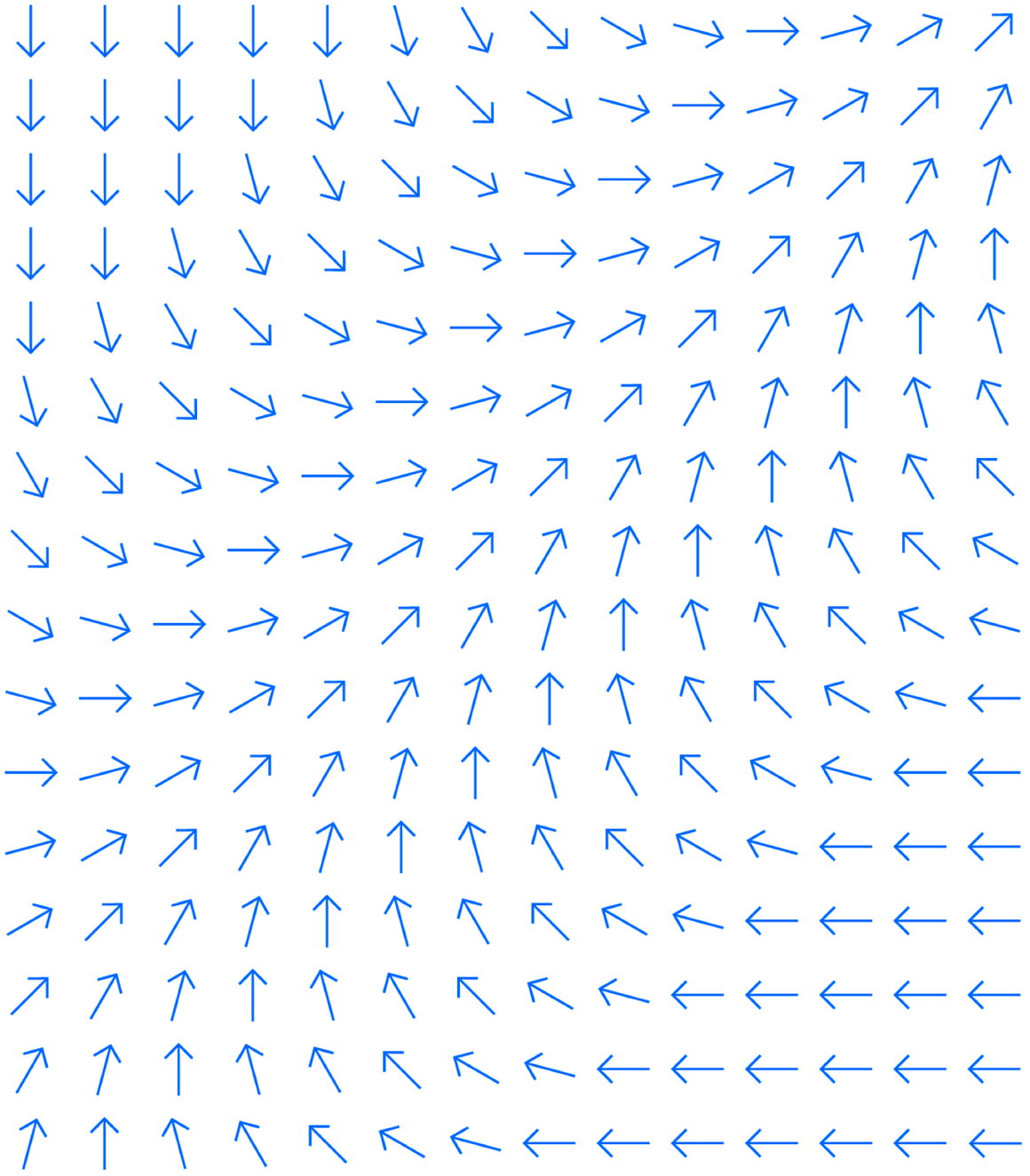
```
return (
  <div className="App">
    <h3>Check out this slick test app!</h3>
    <hr />
    <h3>2 + 3 = {sumNums(2, 3)}</h3>
    <hr />
    <button
      data-testid="custom-button-1"
      onClick={() => setShowMessage((lastState) => !lastState)}
    >
      Show secret message
    </button>
    <hr />
    {showMessage && (
      <h4 style={{ fontStyle: "italic" }}>This is a secret message</h4>
    )}
  </div>
);
```

App.test.js

```
test("show / hide secret message", () => {
  render(<App />);

  const button = screen.getByTestId("custom-button-1");

  fireEvent.click(button);
  expect(screen.queryByText("This is a secret message")).toBeInTheDocument();

  fireEvent.click(button);
  expect(
    screen.queryByText("This is a secret message")
  ).not.toBeInTheDocument();
});
```

Q&A

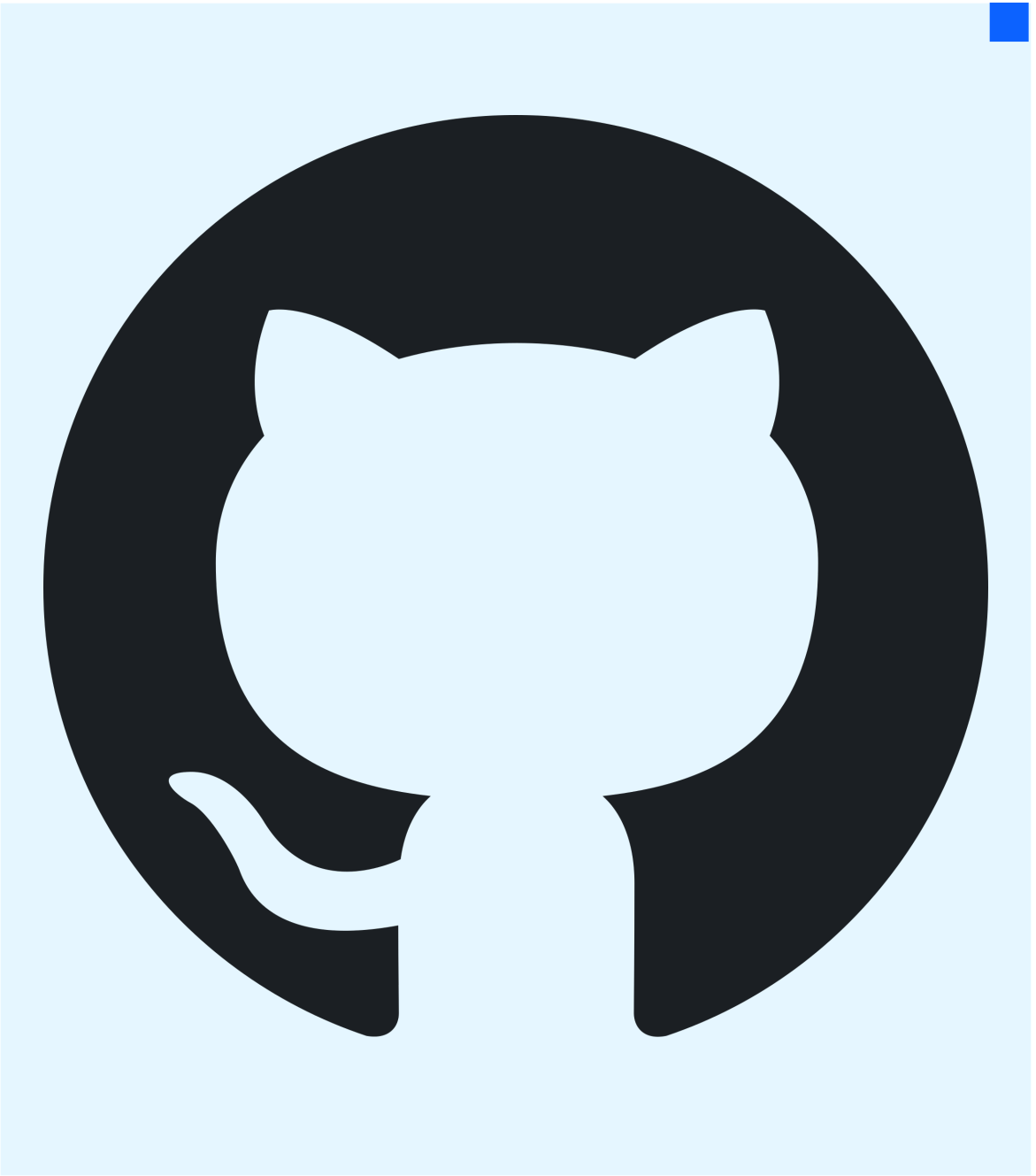
Austin Eberle
Sr. Firmware Developer,
Flash Systems Storage,
IBM Infrastructure



IBM

Project/Lab and GitHub Classroom

Sujeily Fonseca-Gonzalez
Software Engineering Manager,
Technical Leader,
IBM Software,
SW Track Leader



To-do-list Lab:
Review

- Submissions for Lab/Project 3 will close Monday, June 24, 2024, 11:59 PM ET
    - We will pull the code from your **master/main** branch at this time to provide feedback
    - (Make sure you merge any branches before then)
- Use the Slack channel and office hours for any questions
- When starting the lab, pull a fresh copy of the Lab 3 branch.

To-do-list Lab:
Instructions

Feature Requirements:

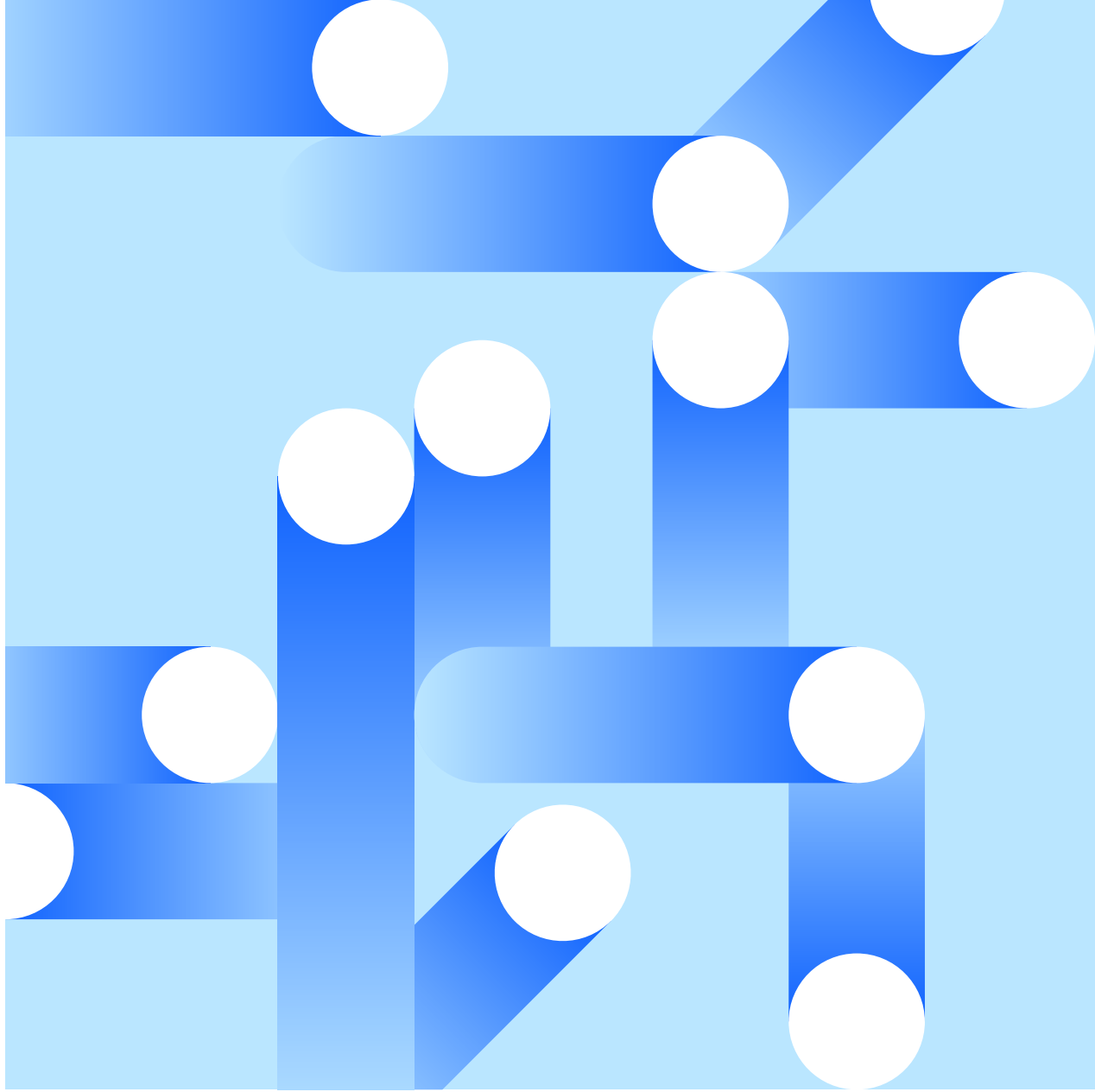
- Provide the due date for the task
- Change the color of overdue tasks

Implementation Requirements:

- Write test cases for your application

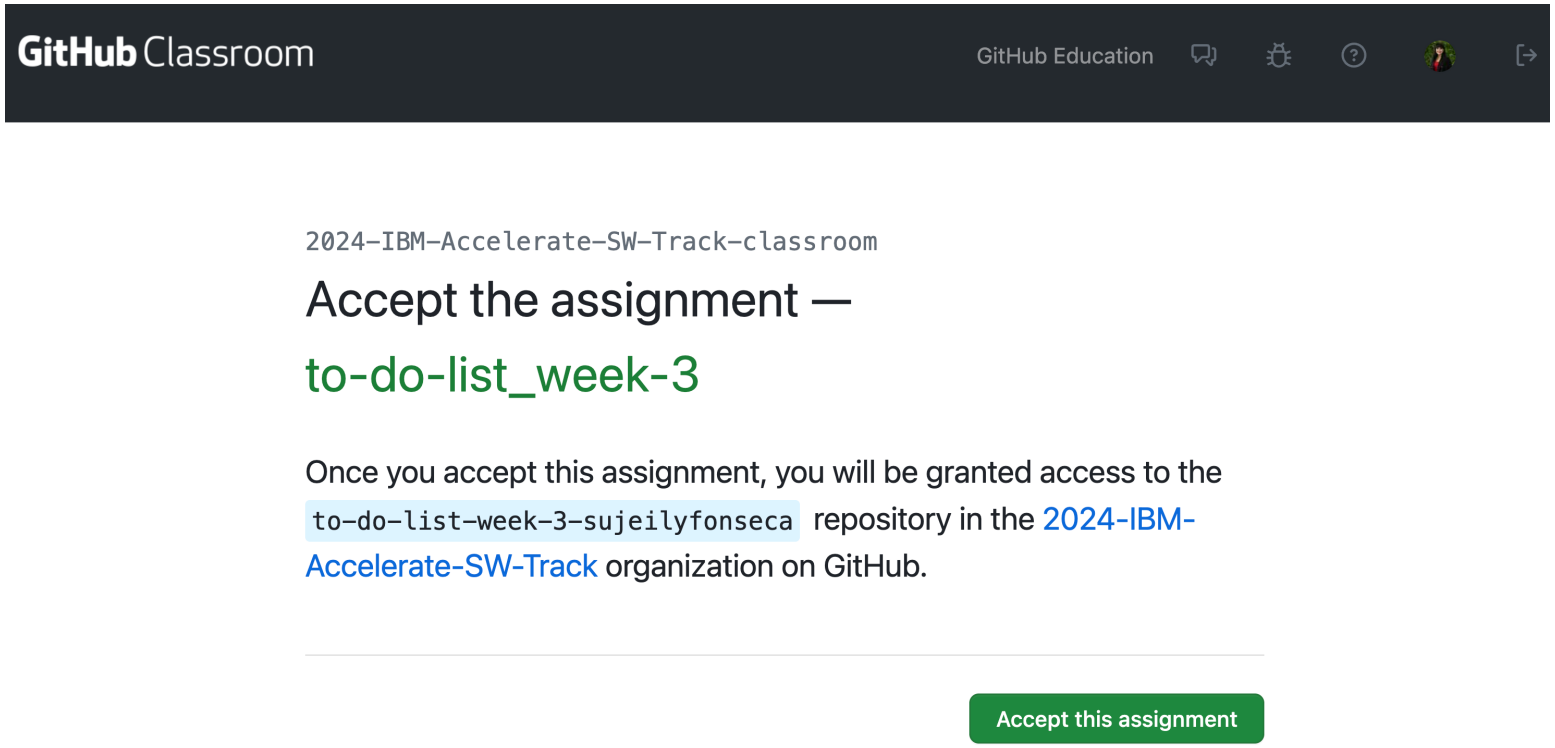
GitHub Classroom:

Let’s walk through the process together!



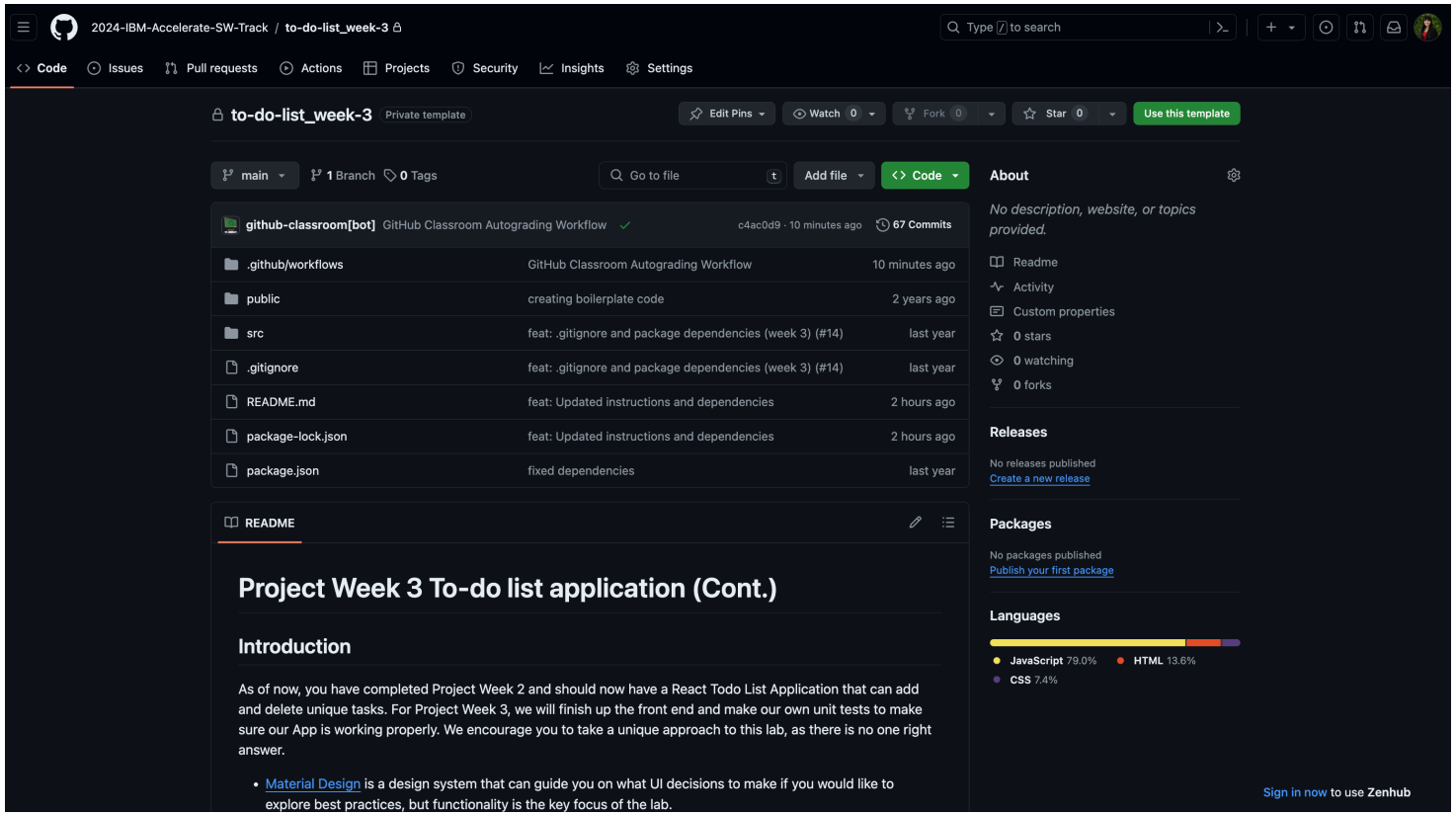
GitHub Classroom:

Let’s walk through the process together!



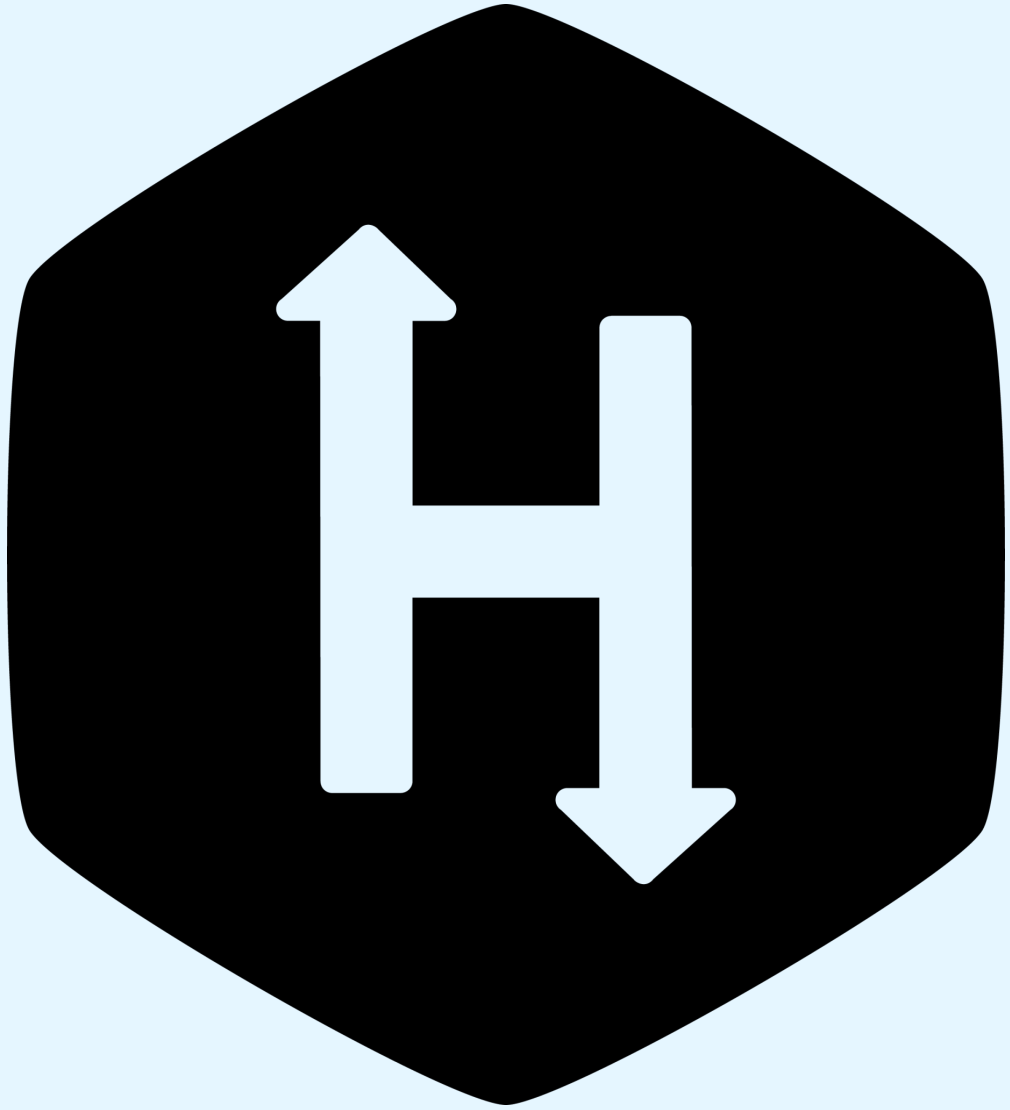
- Short link: https://ibm.biz/accelerate-sw-week3
- Long link: https://classroom.github.com/a/XXiOzV4T

GitHub Classroom:
Automated feedback and code validation



- Automated grading/feedback was enabled.
- It will also give help you understand how you're doing.
- You can run the tests locally by executing "npm test".

Required HackerRank Test



IBM

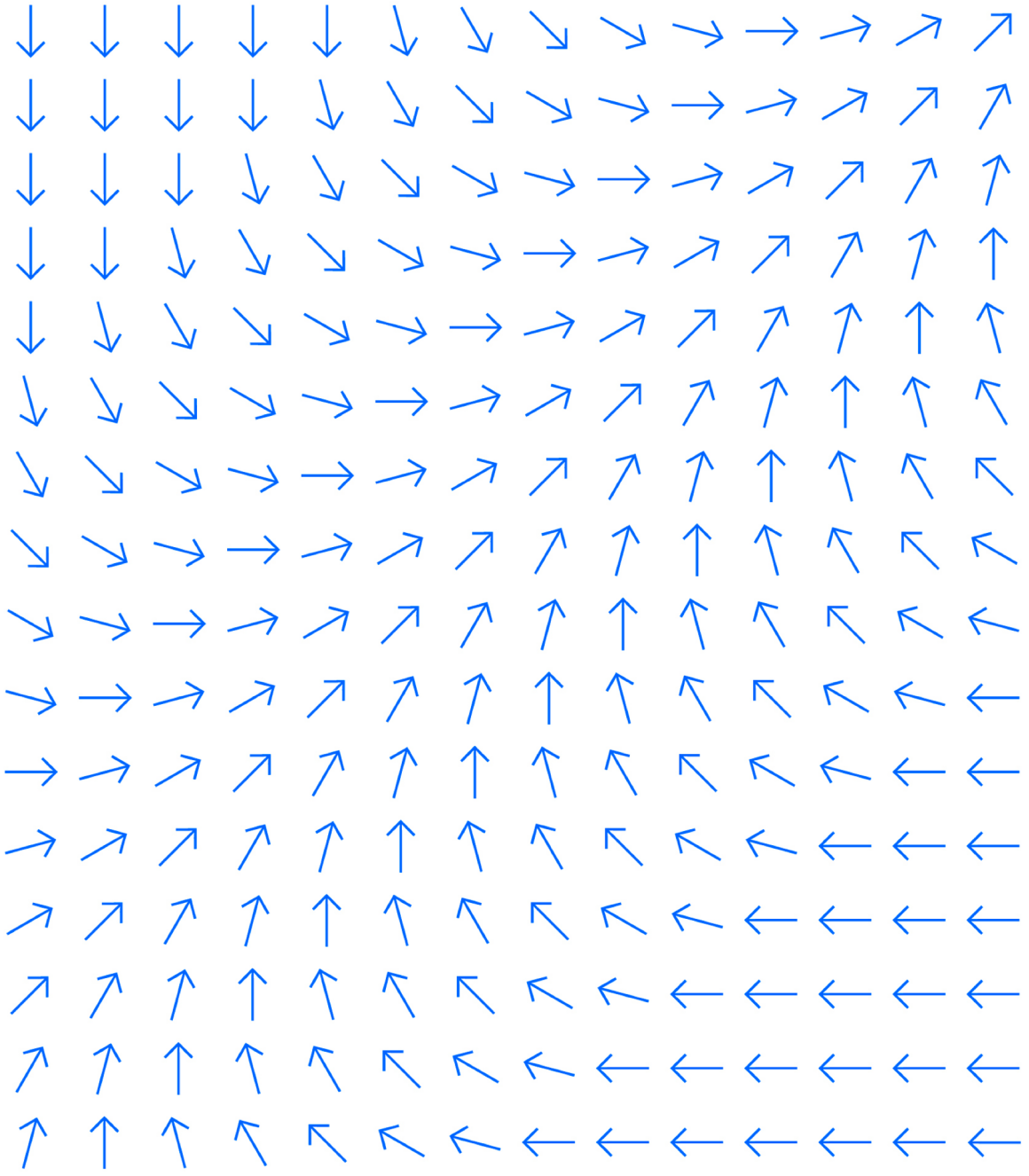
Required HackerRank test

- Tomorrow morning (Thursday), you should receive an email containing a link to a HackerRank quiz covering content from Weeks 1-3
  - From IBM Corporation Hiring Team – support@hackerrankforwork.com
- Test will include **6 questions, 2 from each week**
- **Required to earn the IBM Accelerate Badge**
- **Passing score: 4/6**
- **Due by next Tuesday, June 25 12:00 PM ET**

Required HackerRank test: preperation

- Review labs and presentations from prior weeks
- Complete optional HackerRank tests from weeks 1 and 2
  - Same form and style as required test
- Use Slack and/or Office Hours to answer any questions

Next Steps



Next Steps

1. Finish your lab (https://ibm.biz/accelerate-sw-week3) by Monday, June 24, 11:59 PM ET.
2. If you need further help, ask in the slack channel (#ibm-accelerate-2024-software) or join the office hours (Thursday at 6:00 PM ET).
3. **Complete the required HackerRank test by Tuesday, June 25, 12:00 PM ET.**
4. Check out the solutions for the lab and HackerRank test once available.
5. Slides and video recordings of the session are available in our Box folder.
6. Watch for Week 4 materials to start to arrive by Monday.
7. Network and connect with your peers today and after today's session.