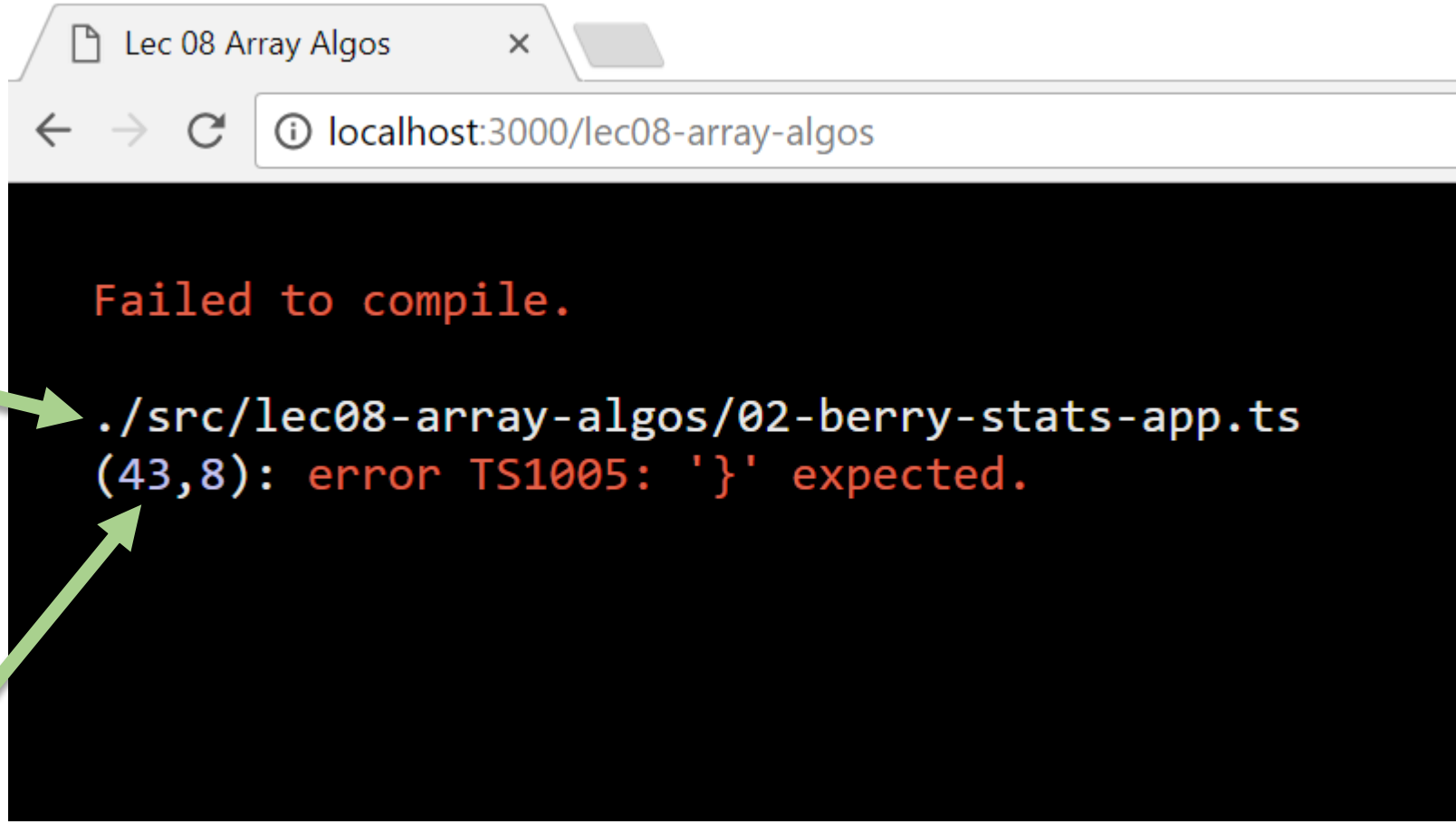
Arrays

Lecture 6

Fixing the "Black Screen of Death"

- When you see a screen that looks like the black screen to the right:
- The error may be in a file other than the one you are working on. **The file with the error is this one.**
- The simplest short-term fix in lecture is select all text in the file (Ctrl+A) and then comment it out (Ctrl+/) and save.
- The specific line # of the file that TypeScript *believes* the error is on is this number in parenthesis.



Challenge #1: What are the elements of **a**?

```
let a: number[] = [ 2 ]; // Notice initial element 2

for (let i = 0 ; i < 3; i++) {
   a[a.length] = (i + 1) * 2;
}

print(a);
```

How do we **append** an element to an array?

- Given an array **a**, what is the **next** index needed to append?
  - When it is **empty**, or has **0 elements**, the next index is **0**
  - When it has **1 element**, the next index is **1**
  - When it has **2 elements**, the next index is **2**

- **Because of 0-based indexing, we can use the # of elements in an array as the index to use to append a value to the array.**

- Append to an array:

```
a[a.length] = <value>;
```

Challenge #2. What are the elements of array **b** after the following code runs?

```
let a: number[] = [10, 20, 30];
let b: number[] = [];

for (let i = 0; i < a.length; i++) {
   b[i] = a[a.length - (i + 1)];
}
```

A. 10, 20, 30
B. 10, 10, 10
C. 30, 20, 10
D. 20, 20, 20

Challenge #3. Given the filter function to the left, which of the following definitions of **test** is correct?

```
let filter = (a: number[]): number[] => {
    let match: number[] = [];
    for (let i = 0; i < a.length; i++) {
        let element = a[i];
        if ( test(element) ) {
            match[match.length] = a[i];
        }
    }
    return match;
}
```

```
// A
let test = (e: number): void => {/*...*/}

// B
let test = (e: string): boolean => {/*...*/}

// C
let test = (e: number): boolean => {/*...*/}

// D
let test = (e: boolean): number => {/*...*/}
```

Organizing a Project into Multiple Files

- As our programs grow in size, we will organize them across multiple files
  - Each file will have related functions and functionality

- You can **export** *functions* and *classes* from one TypeScript file

```
export let aFunc = () => { ... }
```

- And **import** them into *another* TypeScript file

```
import { <function>, <function> } from "./<file>";
```

- Example: **`import { foo, bar } from "./library";`**
  - These functions would be *exported* from a file named library.ts
  - Note: Only the file with the **`main`** function needs to its filename to end with –app.ts

Test-driven Function Writing

- Before you implement a function, focus on concrete examples of *how the function should behave as if it were already implemented.*

- Key questions to ask:

1. **What are some *usual* input parameters?**
   - These are called *use cases.*

2. **What are some valid but *unusual* input parameters?**
   - These are your *edge cases.*

3. Given those input parameters,
   what is your <u>expected</u> return value for each set of inputs?

Challenge #4: Test-Driven Programming

- Suppose you want to write a function named **`fillZeros`**

- Its purpose is to create and fill an array of length **`n`** with all **`0`**s

- Its signature is this: **`fillZeros = (n: number): number[]`**

1. Think of two different, typical use case examples.
   What are their **parameters** and what is the **expected** return value for each?

2. Think of one unusual example as an edge case.
   What are its **parameters** and what is the **expected** return value?

- There are three survey questions. The first two are use cases. The second is an edge case.

  Answer in the form of: `fillZeros( ___ ) is [ ____ ]`

Testing Use/Edge Cases Programmatically

- After you have some use and edge cases, implement the skeleton of the function that is *syntactically valid* but *wrong*
  - Typically this means define the function and do nothing inside of the body except return a valid literal value. For example:

```
export let fillZeros = (n: number): number[] => {
        return [];
};
```

- Then, turn your use and edge cases into programmatic tests.

- How? With a function that compares an *expected* result with an *actual* result.

Example Test Functions

- Let's briefly look at **test-util.ts**

- It has two functions defined: `testArray, testNumber`
  - The first can test the results of a function that returns a number array
  - The second of a function that returns a number

- Let's try adding a test case inside the main function of array-practice-app.ts:

```
testArray("fillZeros 2", [0, 0], fillZeros(2));
```

This is the name of your test.

This is your expected result.

This is your actual result. Notice that it's calling the **`fillZeros`** function with an expected input.

Programmatic Tests Give You Instant Feedback

Test:

```
testArray("fillZeros 2", [0, 0], fillZeros(2));
```

Implementation:

```
export let fillZeros = (n: number): number[] => {
        return [];
};
```

Result:

FAIL: fillZeros 2
string

-- Expected: 0,0
string

-- Actually:
string

Testing is no substitute for critical thinking…

Is this **`fillZeros`** correct?

Test:

```
testArray("fillZeros 2", [0, 0], fillZeros(2));
```

Implementation:

```
export let fillZeros = (n: number): number[] => {
    return [0, 0];
};
```

Result:

PASS: fillZeros 2
string

- Passing a test doesn't ensure your function is correct!
- Rules of Thumb:
  - Test 2+ use cases and 1+ edge cases.
  - When a function has if-else statements, try to write a test that reaches each branch.

Follow-along: Implement **`fillZeros`**

- In array-practice-app.ts, add two more test cases:
  - Parameter Input: 1, Expected Result: [0]
  - Parameter Input: 0, Expected Result: []

- In `array-functions.ts`, we'll rewrite the body of the **fillZeros** function to build an array with **n** zeros in it.
  1. Declare a number array variable. Initialize it to an empty array.
  2. Write a loop that iterates n times.
  3. Append a 0 to the array.
  4. After the loop completes, return the array.

```
export let fillZeros = (n: number): number[] => {
    let a: number [] = [];
    for (let i = 0; i < n; i++) {
        a[i] = 0;
    }
    return a;
};
```

Hands-on: Write Tests for **`fillRange`**

- The purpose of this function is to fill an array of numbers from low to high, inclusive.

- One example use case: **`fillRange(0, 2)`** expects a return value of **`[0,1,2]`**

- In **`array-practice-app.ts`**:
  1. Write a test for the use case above.
  2. Write a test for another use case you can imagine.
  3. Write a test case for an edge case.

- Once you have three failing tests, check-in on pollev.com/compunc

Hands-on: Implement **`fillRange`**

- Hint: Use the implementation of `fillZeros` as a starting point
- Implement fillRange such that it will fill an array of numbers from low to high, inclusive.
  For example: **`fillRange(0, 3)`** returns **`[0, 1, 2, 3]`**
- Hint: you'll need to be clever with your counting variable(s).
- Check-in once you have your tests passing and a working **`fillRange`**.

Follow-Along: Testing **`sum`**

- Let's implement a function to total up a sum of all elements in an array

- What are our test cases?

```
testNumber("sum([])", 0, sum([]));
testNumber("sum([1])", 1, sum([1]));
testNumber("sum([2, 3])", 5, sum([2, 3]));
```

- Notice the `sum` function takes an array of `number` values as a parameter and returns a `number`!
  - So we'll test with the simple **`testNumber`** function defined in **`test-util.ts`**

Hands-on: Implementing **sum**

- In `array-functions.ts` implement the `sum` function

- Your algorithm should:
  1. Declare a variable to "accumulate" a sum
  2. Loop through each element in the input array a and add its value to your accumulating variable
  3. Return your accumulating variable

- Check-in when your **`sum`** tests are passing!

The world's 2nd worst magic trick…