Arrays

Lecture 5

Go to poll.unc.edu

Sign-in via this website then go to pollev.com/compunc

VSCode: Open Project -> View Terminal -> npm run pull -> npm start

UNC DEPARTMENT OF COMPUTER SCIENCE

Data Buddies Survey

CRA Computing Research Association

SCAN ME

https://l.ead.me/cs-survey

What is it?

- Anonymous survey provided by CRA open now through February 6th

Why is it important?

- Your feedback gives department real-time data on curriculum, pedagogy, student support and cultural climate from student POV

What's in it for you?

- Completion of survey means raffle entry and chance to win $500 in Amazon gift cards

Check your email for more details

Upcoming Deliverables

- PS2 - Array Utils
  - Releases at 5pm today
  - Due Fri 2/7 at 11:59pm

Tutoring and OH Conceptual Help

- Looking for extra conceptual help outside of lecture?
  - Conceptual help only! For help with problem sets and unsubmitted worksheets, office hours is our only personalized resource.
  - Great way to go over quiz questions you did not understand.
- Free tutoring from the COMP110 UTA team is available:
  - Tuesdays 5-7pm
  - Wednesdays 5-7pm
  - Thursdays 5-7pm
- All in Fred Brooks 007 as part of the CS Learning Lab
- Can't make tutoring? Come to office hours and request conceptual help.
  - If you do this on days where there is not a queue we can work with you for longer than 15 minutes 1-on-1.

Graded warm-up questions…

Warm-up Questions

1. What is A in: for (A ; B ; C) { D } **- Counter variable initialization**
2. What is B in: for (A ; B ; C) { D } **- Boolean test**
3. What is C in: for (A ; B ; C) { D } **- Variable modification**
4. The for loop's counter variable is only defined inside the for loop. **true**
5. A for loop is more difficult to use than a while loop because you are more likely to write an infinite loop. **false**
6. An array is a variable with a name that holds many values addressed by an index. **true**
7. Each item in an array is called: **an element**
8. The first index in an array is always 1. **false**
9. Which of the following is the type "array of numbers": **number[]**
10. What property tells you how many values an array named a holds? **a.length**

1. What is the printed output when **main** runs?

```
export let main = async () => {
    test("double(3)", 6, double(3));
};

let double = (x: number): number => {
    return x ** 2;
};

let test = (s: string, e: number, a: number): void => {
    if (e === a) {
        print("PASS: " + s);
    } else {
        print("FAIL: " + s);
    }
};
```

Big Idea: We can write code to test the correctness of programs we're working on.

- This is generally called *testing* in industry
  - Helps you confirm correctness during development
  - Helps you avoid accidentally breaking things that were previously working

- The idea is what was illustrated in the last PollEv:
  1. Implement the "skeleton" of the function you are working on
     - Name, parameters, return type, and some dummy (wrong/naive!) return value
  2. Think about good examples in how the function could be used (what arguments?) and what you would expect it to return back
  3. Write a "test case" that actually performs your example function call and compares your expected return value with the actual result
  4. Once you have failing tests, then you go actually try to correctly implement the function's body

- This gives you a framework for knowing your code is behaving as you expect

Example: Writing and Testing a **sum** Function (1/2)

Let's write a function to add up all elements of a number array!

Step 0) Implement the function skeleton:

```
let sum = (a: number[]): number => {
   return -1; // return a dummy value (wrong but correct type)
}
```

Step 1) Think of some example uses...

sum([1, 2, 3]) *should* return **6**

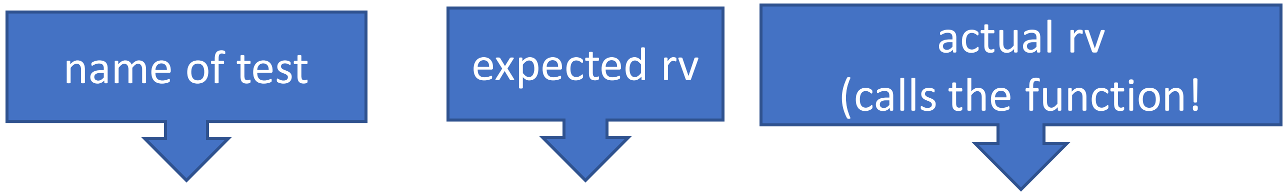
sum([110]) *should* return **110**

sum([]) *should* return **0**

Example: Writing and Testing a **sum** Function (2/2)

Step 2) Write test cases that encode the example uses you produced

- A test has a **name**, an **expected return value**, and an **actual return value**.
  - **How do you name a test?** We'll use a string that looks like the function call so we can easily find the test in our code if we need to.
  - **How do you get the actual RV?** You perform an actual function call
  - ***What is a test?*** Just a call to a function that compares expected vs actual and prints diagnostic output. Take a look at the **testNumber** function to see what's happening.

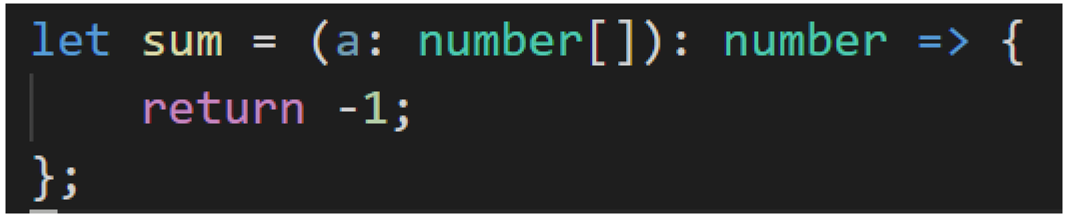


```
testNumber("sum([1, 2, 3])", 6, sum([1, 2, 3]));
```

Step 3) Once your tests run without any black screens of death, ***then*** you go work on correctly implementing the function being tested (**sum**). You get *immediate* feedback on whether your tests are passing or not!

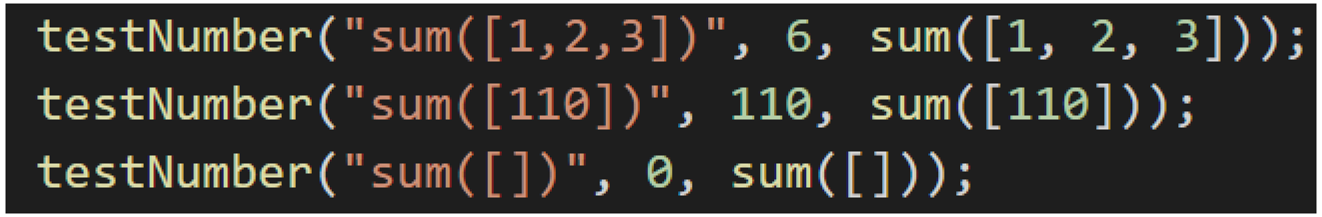
Follow-Along: Testing **`sum`**

- Let's implement a function to sum the elements of an array
- Function Skeleton:



```
let sum = (a: number[]): number => {
    return -1;
};
```

- What are our test cases?



```
testNumber("sum([1,2,3])", 6, sum([1, 2, 3]));
testNumber("sum([110])", 110, sum([110]));
testNumber("sum([])", 0, sum([]));
```

- Notice the `sum` function takes an array of `number` values as a parameter and returns a `number`!

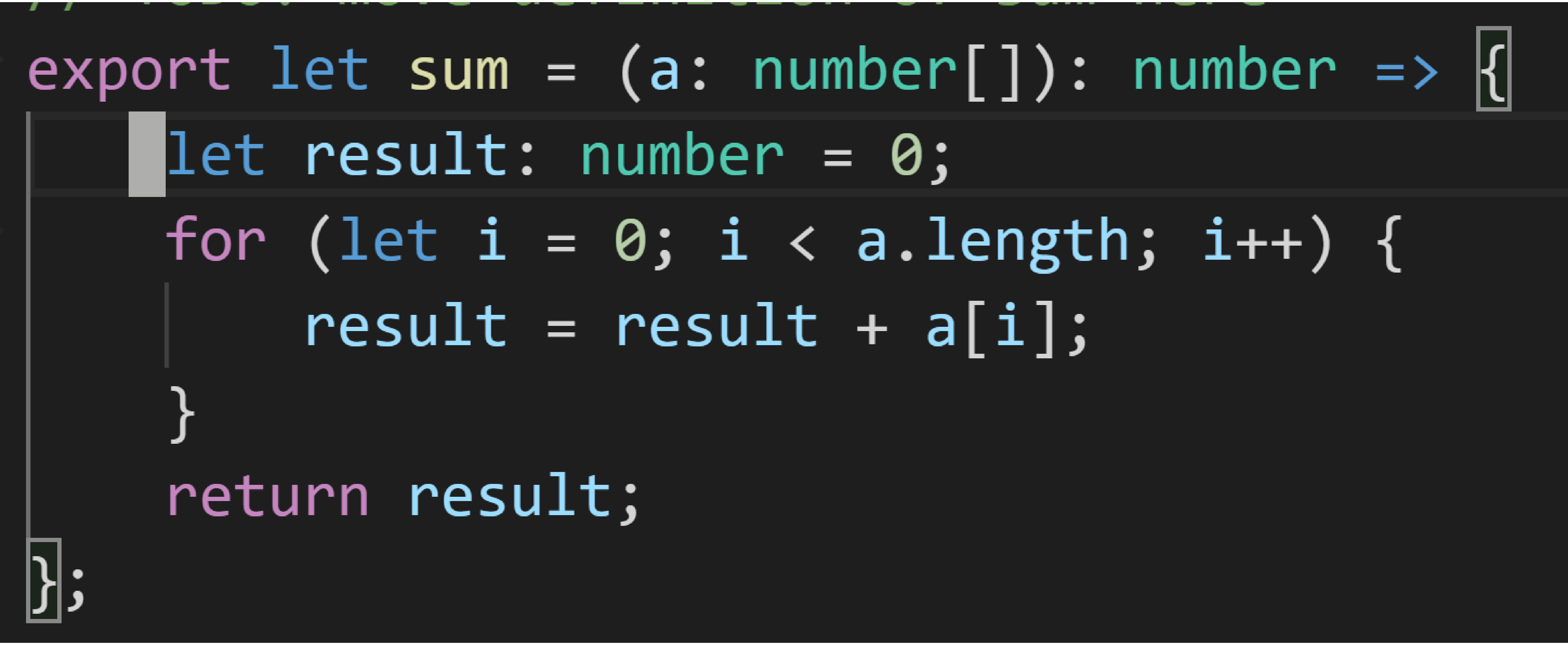
Hands-on: Implementing **sum**

- Try implementing 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

- Save to have your tests run against your implementation of sum.

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



```typescript
export let sum = (a: number[]): number => {
    let result: number = 0;
    for (let i = 0; i < a.length; i++) {
        result = result + a[i];
    }
    return result;
};
```

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* 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

Multiple File Example

- Let's try reorganizing our array-practice-app.ts to clean it up

1. Remove the comment and code for **testNumber** function definition
   - At the top of the file add an import to import the same function from test-util.ts

   ```
   import { testNumber } from "./test-util";
   ```

2. Move the sum function definition to the top of array-functions.ts, then
   - add the keyword **export** before: **let sum = (...**
   - Back in **array-practice-app.ts** add an import for sum:

     ```
     import { sum } from "./array-functions";
     ```

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?

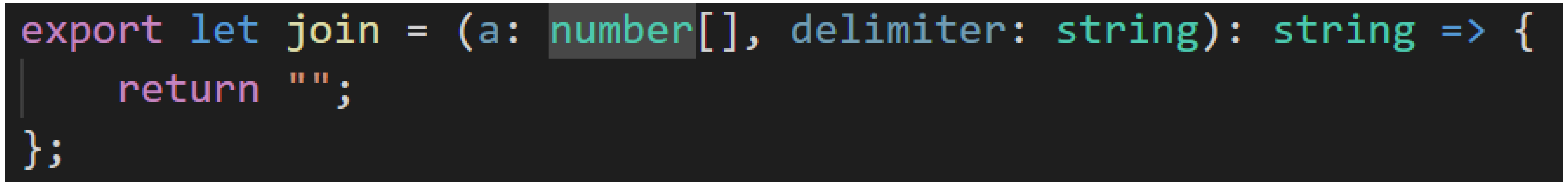
Test-Driven Programming: Case Study **join**

- Suppose you want to write a function named **`join`**
- Its purpose is to make form a string out of a number array **`a's`** values where each element is separated by some delimiter.
  Example: joining an array with 1, 2, 3 and delimiter "-" returns "1-2-3"
- Its signature is this: **`join = (a: number[], delimiter: string): string`**

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?

Testing Use/Edge Cases Programmatically

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



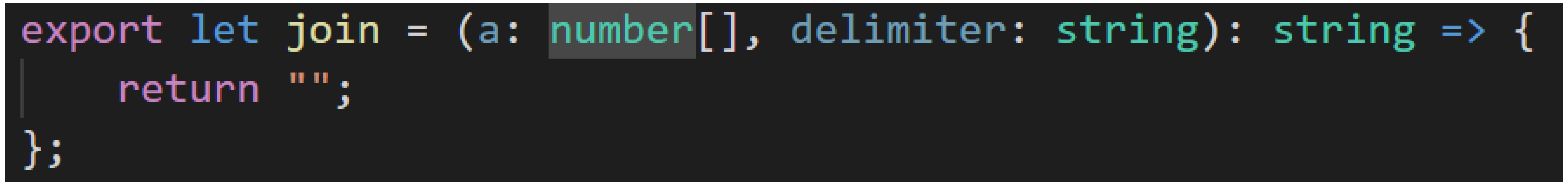
```
export let join = (a: number[], delimiter: string): string => {
    return "";
};
```

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

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

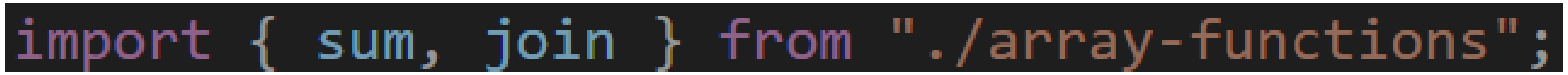
Hands-on: Implement **join**

- Add a skeleton definition of join to array-functions.ts

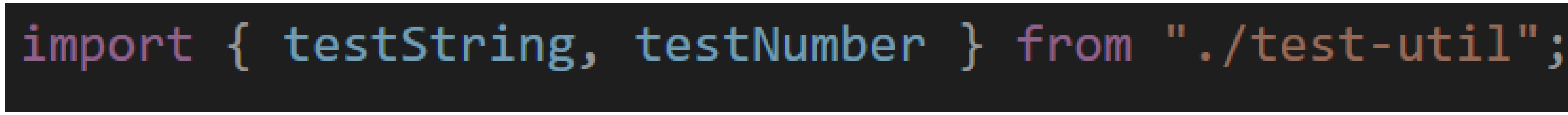


```
export let join = (a: number[], delimiter: string): string => {
    return "";
};
```

- Import join in array-practice-app.ts, import testString

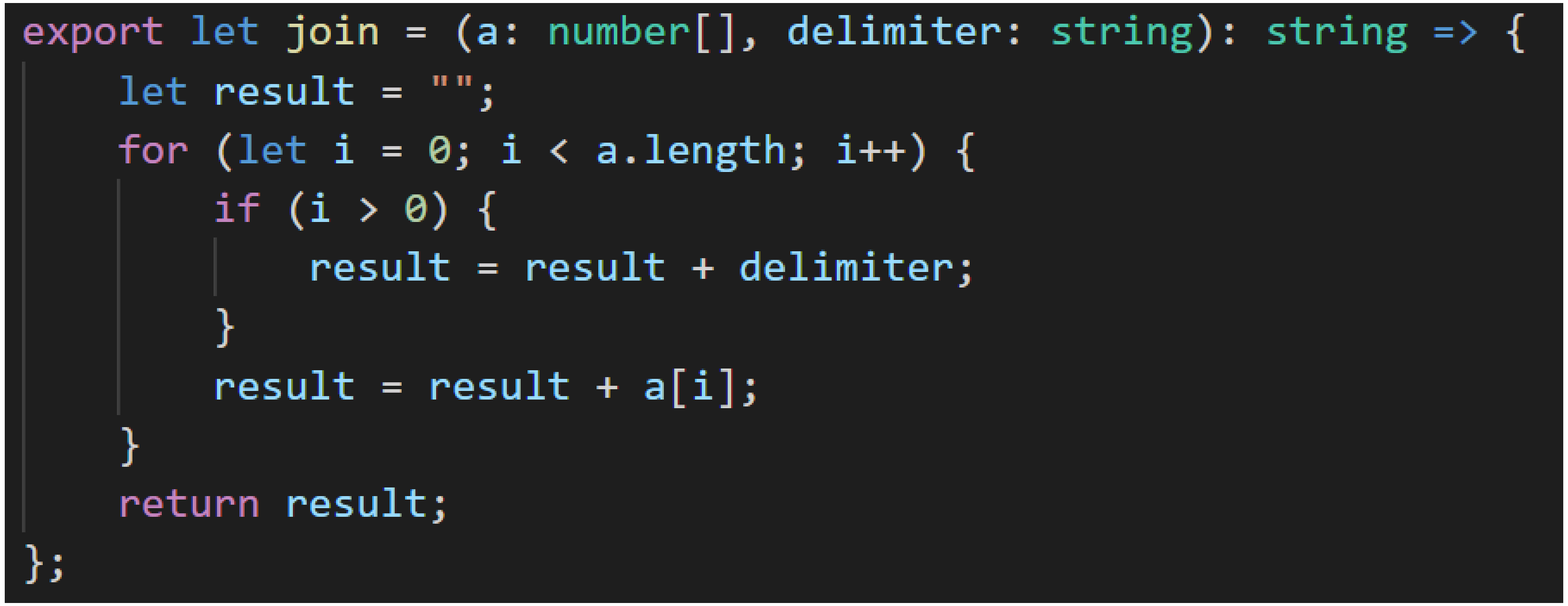


```
import { sum, join } from "./array-functions";
```



```
import { testString, testNumber } from "./test-util";
```

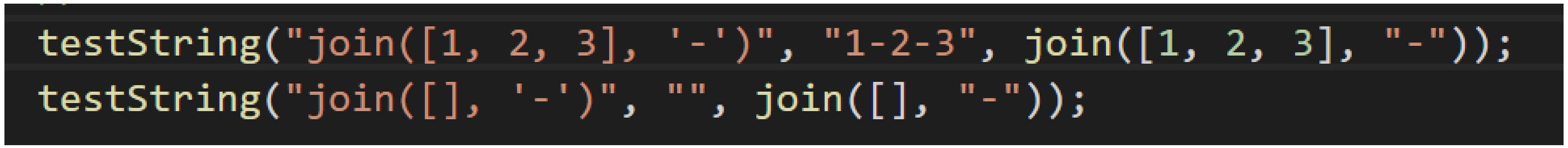
- In `array-functions.ts`, write the **join** function to build a string.
  1. Declare a string result variable. Initialize it to an empty string.
  2. Write a loop that iterates while counter variable is less than a.length
     1. If i is greater than 0, then append the delimiter to your result string
     2. In all cases in the repeat block of the loop, append a[i] to your result string
  3. Return the resulting string



```
export let join = (a: number[], delimiter: string): string => {
    let result = "";
    for (let i = 0; i < a.length; i++) {
        if (i > 0) {
            result = result + delimiter;
        }
        result = result + a[i];
    }
    return result;
};
```

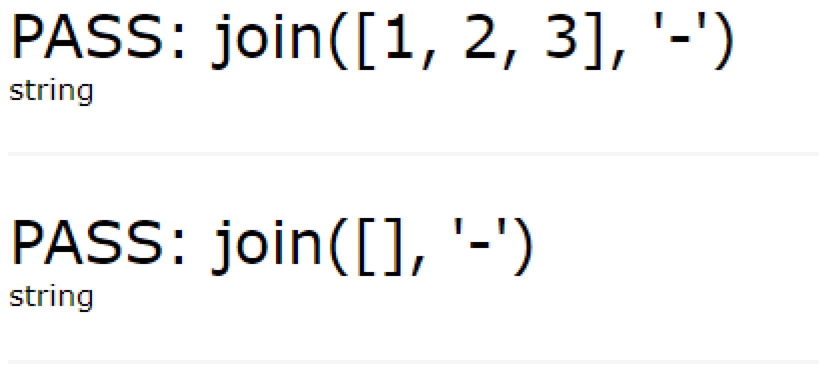
Programmatic Tests Give You Instant Feedback

Test:



```
testString("join([1, 2, 3], '-')", "1-2-3", join([1, 2, 3], "-"));
testString("join([], '-')", "", join([], "-"));
```

Result:



PASS: join([1, 2, 3], '-')
string

PASS: join([], '-')
string

Testing is no substitute for critical thinking…

- Passing your own tests doesn't ensure your function is correct!
  - Your tests must cover a useful range of cases

- 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.

Challenge: 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>;
```

Suppose you're writing a **fillRange** function

- Its signature is:

```
fillRange(low: number, high: number): number[]
```

- Its purpose is to generate an array of consecutive integers increasing from low and ending with high, inclusive.

- Select the test case (input parameters and expected return value) which you believe is the best example of an **edge case**.

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

- The function generates 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 another **use case** you can imagine: inputs 1, 3 – output: [1, 2, 3]
  2. Write a test case for an **edge case – input: 3, 1 – []**

- Once you have two failing tests, one passing, check-in on pollev.com/compunc

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

1. Open **`array-functions.ts`**
2. Hint #1: Look to `fillZeros` as a starting point.
3. Hint #2: What should your loop's counting variable's initial value be?
4. Hint #3: You can append to an array named a with: **`a[a.length] = <num>`**
5. **Check-in once you have your tests passing** and a working **`fillRange`**.
6. Done? Try improving with a version that rounds down decimals and still works.