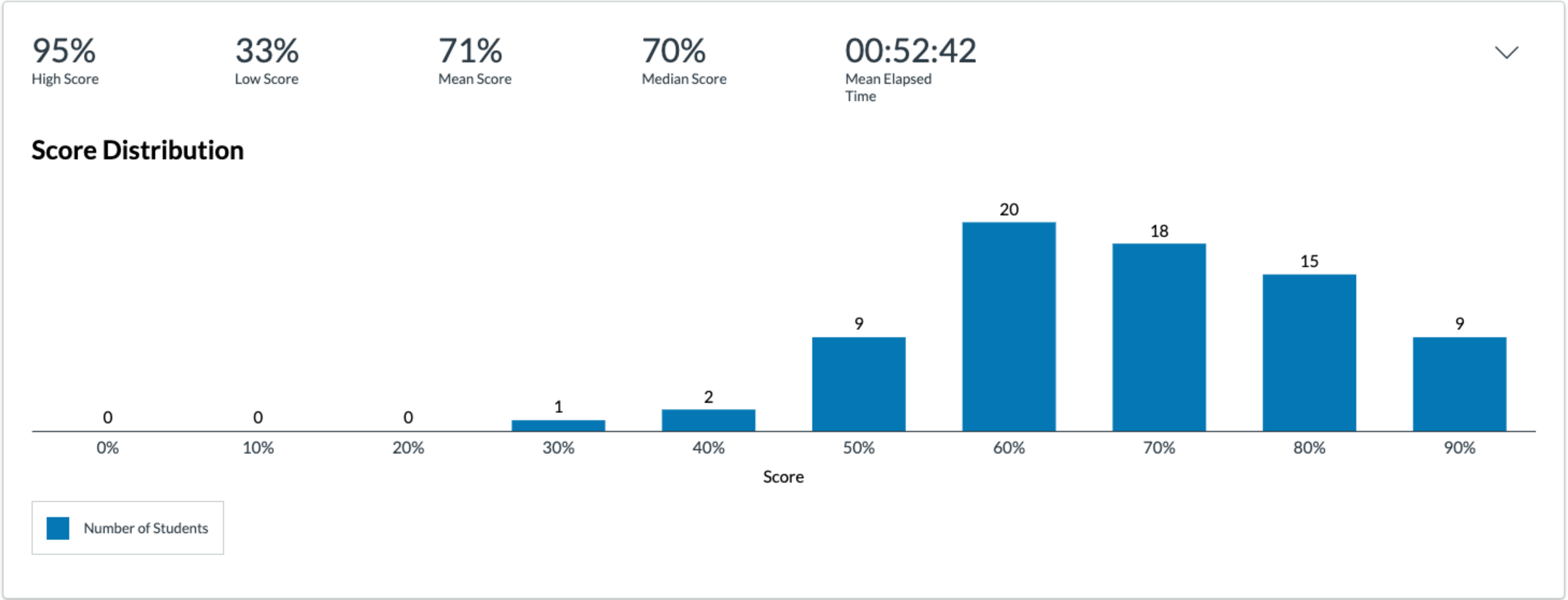
Midterm Review

Midterm



95% High Score

33% Low Score

71% Mean Score

70% Median Score

00:52:42 Mean Elapsed Time

Score Distribution

The midterm results are visible

- We can have a discussion if you like.

Generators in JavaScript

CPEN320

Outline

What are generators ?

Creating a simple generator

Iterating with a generator

Combining generators with promises

What is a generator ?

What is a generator ?

A feature (introduced in ES6) that allows a function to generate a sequence of values over time, pausing and resuming its execution, instead of computing them all at once.

- Used to “remember” state of the function at the point where it left off.
- Allows us to control the function’s execution and its returned values.
- Useful when you want to create iterators or handle asynchronous programming in a more flexible way.

Enter generators

This is achieved using a new kind of function called a generator function, which is declared using the **function*** syntax (note the asterisk after the function keyword) and the **yield** keyword inside its body.

```
function* simpleGenerator() {

  yield alert(1)

}
```

Yield and done

‘yield’ pauses the execution at that point, and resumes the execution on a subsequent call to the generator. Yield can also return a value to the caller - typically value being iterated on.

When end of function is reached, it returns ‘undefined’ (can also specify a return) value, and the generator is DONE. Cannot be resumed after that point ever.

```
function* simpleGenerator() {
  yield alert(1)
  yield alert(2)
}
```

Yield

First time the next is called: executes until a ‘yield’ is encountered

- Control is then ceded back to the caller function (but state is remembered)

Each subsequent call to the yield resumes execution from after where the first ‘yield’ left off, and executes until the next yield statement (state is remembered), or a return statement or throw occurs in the function (the generator is terminated)

IMPORTANT: Yield can only be called **within generator function itself**, not even in nested functions or from asynchronous callbacks.

Value returned by yield

Yield returns an object called ***IteratorResult*** consisting of two properties:

1. **value**: represents the actual value returned by the yield statement
2. **done**: a boolean flag representing whether or not the iteration is finished

It’s caller’s responsibility to check value of ‘done’. In addition, the generator can

a. Throw an exception - this can be caught via a try-catch statement
b. Return a value (using ‘return’) - sets the value of *IteratorResult* to value and the done property to *true*. Note that not returning a value will still set the done property to *true* and the value to ‘undefined’

SimpleGenerator

```
function* simpleGenerator() {
    yield 1
    yield 2
    yield 3
}

const g = simpleGenerator();
console.log(g.next()); ?
```

SimpleGenerator

```
function* simpleGenerator() {
    yield 1
    yield 2
    yield 3
}

const g = simpleGenerator();
console.log(g.next()); // { value: 1, done: false }
console.log(g.next()); ?
```

SimpleGenerator

```
function* simpleGenerator() {
    yield 1
    yield 2
    yield 3
}

const g = simpleGenerator();
console.log(g.next()); // { value: 1, done: false }
console.log(g.next()); // { value: 2, done: false }
console.log(g.next()); ?
```

SimpleGenerator

```
function* simpleGenerator() {
    yield 1
    yield 2
    yield 3
}

const g = simpleGenerator();
console.log(g.next()); // { value: 1, done: false }
console.log(g.next()); // { value: 2, done: false }
console.log(g.next()); // { value: 3, done: false }
console.log(g.next()); ???
```

SimpleGenerator

```
function* simpleGenerator() {
    yield 1
    yield 2
    yield 3
}

const g = simpleGenerator();
console.log(g.next()); // { value: 1, done: false }
console.log(g.next()); // { value: 2, done: false }
console.log(g.next()); // { value: 3, done: false }
console.log(g.next()); // { value: undefined, done: true }
```

Using yield to handle async flows

```
function* fetchData() {
  const user = yield fetch('https://api.example.com/user’);

  const posts = yield
      fetch(`https://api.example.com/posts?user=${user.id}`);

  return posts;
}
```

Use Cases (Iteration)

```
function* generateID() {
  let id = 1

  while(true) {
    yield id
    id++
  }
}

const g = generateID();
console.log(g.next()); // 1
console.log(g.next()); // 2
…
```

Class activity - 1

Write a simple generator function to **yield** the **factors of a natural number** n (including itself)

The function takes as input n and only calculates the factors on demand and not ahead of time (one at a time). You can assume that the input has been validated already.

Solution

```
function* factors(n) {
  for (let i = 1; i <= n; i++) {
    if (n % i === 0) {
      yield i
    }
  }
}

// Example usage:
const factorGenerator = factors(8);
console.log(factorGenerator.next().value); // 1
console.log(factorGenerator.next().value); // 2
console.log(factorGenerator.next().value); // 4
console.log(factorGenerator.next().value); // 8
console.log(factorGenerator.next().done); // true
```

Exercise

Write a generator function that iterates over a list of names with a generator.

The generator function takes the list as an argument, and yields one name at a time, in the list order.

Use the generator function to log the following name list using **value** and **done:**

```
['Alice', 'Bob', 'Charlie', 'Dana’]
```

Also, use the generator function to log “all done” when done.

Solution

```
function* nameIterator(names) {
  for (const name of names) {
    yield name;
  }
  return “all done”
}
// Usage:
const namesList = ['Alice', 'Bob', 'Charlie', 'Dana'];
const names = nameIterator(namesList);
let result = names.next();
while (!result.done) {
  console.log(result.value); // Logs each name in the list
  result = names.next();
}
console.log(result.value); // logs “all done”
```

Exercise

Write a **generator function findWordOccurrences** to iterate over a large string, and find all occurrences of a specific word in it. The generator should match the word on demand only and not pre-compute the matches. Also, after each match, it should start from the next location until the end of the string is reached. It should yield the number of characters traversed in the string after the previous match (until the current one)

Now, write a function **countWordOccurrences** to iterate using the generator function and count *all* occurrences of a given word in a string.

Solution

```
function* findWordOccurrences(text, word) {
  let index = 0;
  word = word.toLowerCase(); // Assuming case-insensitive search
  let textLower = text.toLowerCase();

  while (index < text.length) {
    let foundAt = textLower.indexOf(word, index);
    if (foundAt === -1) break; // No more occurrences found

    yield foundAt + word.length - index; // Yield the number of characters traversed
    index = foundAt + word.length; // Move index past the current match
  }
}
```

Solution

```
function countWordOccurrences(text, word) {
  const occurrencesGenerator = findWordOccurrences(text, word);
  let count = 0;

  for (const o of occurrencesGenerator) {
    count++; // Increment count for each yielded value (each occurrence)
  }

  return count;
}

// Example usage:
const text = "This is a test. This test is only a test.";
const word = "test";
const count = countWordOccurrences(text, word);
console.log(`The word "${word}" appears ${count} times.`);
```

Generators and asynchronous call-backs

One of the main advantages of Generators is that they can be used to write code in a “natural” style, without using callbacks and closures

However, generators cannot (easily) call asynchronous functions…

Recall that callbacks defined inside the generators cannot call yield (as they’re not executing in the context of the generator function)

One possible solution: Using promises together with generators

Generators and Promises

Yield a promise in each call of the next() method

- Promise can be resolved or rejected later
- Attach then and catch handlers to the promises as usual

You can also send the results of the resolved promise to the generator as an argument of the next function to chain them (outside the scope of this class)

Makes for cleaner code than attaching multiple layers of call-backs to promises

Exercise

Write a generator function that takes an array of function as the first argument and executes them asynchronously, each after a specified delay (second argument).

The generator function should return a set of promises, one after the other, for each function in the list with the appropriate delays. It should also pass to each function a parameter representing the current index of the function in the list.

Iterate over the list using the generator and attach a **.then** and **.catch** clause to each of the promises yielded by the generator.

Solution: asyncGenerator.js

```
function* asyncFunctionExecutor(fnArray, delay) {
  for (let i = 0; i < fnArray.length; i++) {
    // Yield a promise
    yield new Promise((resolve, reject) => {
      setTimeout(() => {
        try {
          resolve(fnArray[i](i)); // Pass the current index
        } catch (error) {
          reject(error);
        }
      }, delay);
    });
  }
}
```

```
const fnArray = [
  (index) => `Function ${index} executed`,
  (index) => `Function ${index} executed`
];
const delay = 1000; // Delay in milliseconds

const executor = asyncFunctionExecutor(fnArray, delay);

for (const promise of executor) {
  promise.then(result => {
    console.log('Success:', result);
  }).catch(error => {
    console.error('Error:', error.message);
  });
}
```