LEARN STACKS: JAVASCRIPT

Introduction

You have an understanding of how stacks work in theory, so now let's see how they can be useful out in the wild — with Javascript!

Remember that there are three main methods that we want our stacks to have:

- Push - adds data to the top of the stack
- Pop - provides and removes data from the top of the stack
- Peek - provides data from the top of the stack without removing it

We also need to consider the stack's size and tweak our methods a bit so that our stack does not *overflow*.

Let's get started building out our `Stack` class.

Instructions

1.

In **Stack.js**, an empty `Stack` class has been created for you. Inside the `.constructor()` method, instantiate the `.stack` property with a construction of a new `LinkedList` class that has been imported for you.

Hint

To create a new linked list, use the syntax `new LinkedList()`.

2.

Below `.constructor()`, define another method `.peek()` that returns the `data` assigned to the `stack`'s top element. Remember that `stack` is a `LinkedList` instance that is made up of `Node` instances. The `LinkedList` class has a `head` property pointing to a `Node` instance. To access `data`, you first have to get to the `head` property.

Hint

Browse through the **LinkedList.js** and **Node.js** files to see how `head` and `data` relate. Sample code to access nested members is shown below:

```
peek() {
  return this.stackProperty.linkedListProperty.nodeProperty;
}
```

Stack.js

```
const LinkedList = require('./LinkedList');

class Stack {
  constructor() {
    this.stack = new LinkedList();
  }
  peek() {
    return this.stack.head.data;
  }
}

module.exports = Stack;
```

LinkedList.js

```
const Node = require('./Node');

class LinkedList {
  constructor() {
    this.head = null;
  }

  addToHead(value) {
    const nextNode = new Node(value);
    const currentHead = this.head;
    this.head = nextNode;
    if (currentHead) {
      this.head.setNextNode(currentHead);
    }
  }

  addToTail(value) {
    let lastNode = this.head;
    if (!lastNode) {
      this.head = new Node(value);
    } else {
      let temp = this.head;
      while (temp.getNextNode() !== null) {
        temp = temp.getNextNode();
```

```
      }
      temp.setNextNode(new Node(value));
    }
  }

  removeHead() {
    const removedHead = this.head;
    if (!removedHead) return;

    if (removedHead.next) {
      this.head = removedHead.next;
    }
    return removedHead.data;
  }

}

module.exports = LinkedList;
```

Node.js

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }

  setNextNode(node) {
    if (!(node instanceof Node)) {
      throw new Error('Next node must be a member of the Node class');
    }
    this.next = node;
  }

  setNext(data) {
    this.next = data;
  }

  getNextNode() {
    return this.next;
```

```
  }
}

module.exports = Node;
```

Push and Pop

The stack's `.push()` and `.pop()` methods are our tools to add and remove items from it. `.pop()` additionally returns the value of the item it is removing. Keep in mind that we can only make modifications to the top of the stack.

Instructions

1.

Below `.constructor()`, define a method `.push()` for `Stack` that takes the parameter `value`. Inside `.push()`, use a `LinkedList` method to add a new value to the head of `this.stack`.

Hint

Open **LinkedList.js** to see `LinkedList`'s `.addToHead()` method and how it works.

```
push(value) {
  this.stack.addToHead(value);
}
```

2.

Below `.push()`, define a `.pop()` method for `Stack`. Inside `.pop()`:

- figure out which of the `LinkedList` class methods will remove the head of the list and return its value.
- create a `const` variable `value` and set it equal to the removed value.
- return the `value`.

Hint

You can open **LinkedList.js** to see `LinkedList`'s `.removeHead()` method and how it works.

```
pop() {
  const value = this.stack.removeHead();
  return value;
}
```

Stack.js

```
const LinkedList = require('./LinkedList');

class Stack {
  constructor() {
    this.stack = new LinkedList();
  }

  push(value) {
    this.stack.addToHead(value);
  }

  pop() {
    const value = this.stack.removeHead();
    return value;
  }

  peek() {
    return this.stack.head.data;
  }
}

module.exports = Stack;
```

Size I

With stacks, size matters. If we're not careful, we can accidentally over-fill them with data. Since we don't want any stack overflow, we need to go back and make a few modifications to our methods that help us track and limit the stack size so we can keep our stacks healthy.

What do we do if someone tries to `.peek()` or `.pop()` when our stack is empty?

How do we keep someone from `.push()`ing to a stack that has already reached its limit?

How do we even know how large our stack has gotten?

Instructions

1.

Let's begin tracking the size of our stack. In `.constructor()`, we'll add a new property `size` and initialize it to `0`.

Hint

Assign a class property to some value as follows.

```
class Person {
  this.name = 'Harry';
}
```

2.

In the same `.constructor()`, let's add:

- a parameter `maxSize` with a default value of `Infinity` so that the stack is unlimited unless we define a size for it upon instantiation later.
- a property `.maxSize` and initialize it to the parameter of the same name.

Hint

Assign a default value to a function as follows:

```
function setTemperature(temperature = 80) {
  const localTemperature = temperature;
}
```

3.

Let's make sure that `.peek()` returns a valid value when it is not empty.
In `.peek()`, wrap the current body with an `if` statement that checks if the `size` of the stack is greater than `0`.

Hint

Here is an example of wrapping the body of a function with `if` statement:

```
function status() {
  if (availabity == true) {
    console.log("I'm available!");
  }
}
```

4.

If the stack is empty, we need to return `null` instead. In `.peek()` outside the `if` statement, add an `else` statement to accomplish this.

Hint

Here is an example of adding an `else` statement outside of the `if` statement in the body of a function:

```
function status() {
  if (availabity == true) {
    console.log("I'm available!");
  } else {
    console.log("Sorry, not available!");
  }
}
```

5.

Similarly in `.pop()`, we want to check if the stack is empty before returning a value. Wrap the current body in an `if` statement that checks if the `size` of the stack is greater than `0`.

Hint

An example of wrapping the body of a function with `if-else` statement block is as follows:

```
function isWeekend(day) {
  if ((day == 'Saturday') || (day == 'Sunday')) {
     return true;
  }
  else {
     return false;
  }
}
```

6.

In `.pop()`, just before the return statement, reduce the `size` of the stack by `1`.

7.

If the `Stack` is empty, we want to let the user know with a message. Add an `else` statement that logs a message `'Stack is empty.'`.

Stack.js

```
const LinkedList = require('./LinkedList');

class Stack {
  constructor(maxSize = Infinity) {
    this.stack = new LinkedList();
    this.size = 0;
    this.maxSize = maxSize;
  }
```

```
  push(value) {
    this.stack.addToHead(value);
  }

  pop() {
    if (this.size > 0) {
       const value = this.stack.removeHead();
       this.size--;
       return value;
    }
    else {
      console.log('Stack is empty');
    }
  }

  peek() {
    if (this.size > 0) {
       return this.stack.head.data;
    }
    else {
      return null;
    }
  }

}

module.exports = Stack;
```

Size II

It's time to add a couple helper methods.

Helper methods simplify the code we've written by abstracting and labeling chunks of code into a new function.

First, we want one that checks if our stack has room for more items. We can use this in `.push()` to guard against pushing items to our stack when it's full.

Second, it's helpful to have a method that checks if the stack is empty...

Instructions

1.

Define a new method `.hasRoom()` in `Stack`. The method should return `true` if its current size is less than its maximum size, otherwise `false`.

2.

Go back to your `.push()` method — we need to make sure we're keeping track of our stack size when we add new items. At the end of your method body, increment `this.size` by 1.

3.

Now add a conditional statement at the top of `.push()` that checks if your stack has room (using your newly created helper method).

- If there's room, the rest of the body of the method should execute
- If there's no room, we want to `throw` an `Error()` letting users know that the stack is already full with a message of `'Stack is full'`.

4.

Finally, let's define a new method `.isEmpty()` in `Stack`.

The method should return `true` if the stack's `size` is 0 and `false` otherwise.

Hint

An example of returning a conditional that would evaluate to `true` in a return statement would be as follows.

```
function isHot() {
  return temperature > 99;
}
```

5.

Now we can use our `.isEmpty()` method in our class. Rewrite your `.pop()` and `.peek()` methods to use `.isEmpty()` in their conditionals.

Stack.js

```
const LinkedList = require('./LinkedList');

class Stack {
  constructor(maxSize = Infinity) {
    this.stack = new LinkedList();
    this.maxSize = maxSize;
    this.size = 0;
  }

  hasRoom() {
    return this.size < this.maxSize;
  }

  isEmpty() {
    return this.size === 0;
  }

  push(value) {
    if (this.hasRoom()) {
      this.stack.addToHead(value);
      this.size++;
    } else {
        throw new Error('Stack is full')
    }
  }

  pop() {
    if (!this.isEmpty()) {
      const value = this.stack.removeHead();
      this.size--;
      return value;
    }
    else {
      console.log('Stack is empty!');
    }
  }

  peek() {
    if (!this.isEmpty()) {
```

```
      return this.stack.head.data;
    } else {
      return null;
    }
  }

}

module.exports = Stack;
```

Review

Nice work — you've built out a `Stack` class that can:

- add a new item to the top via a `.push()` method
- remove an item from the top and returns its value with a `.pop()` method
- return the value of the top item using a `.peek()` method
- allow a stack instance to maintain an awareness of its size to prevent stack overflow.

So how does your code stack up against pizza delivery?

Instructions

1.

In **main.js** file, instantiate a `Stack` class with a size of `6` and assign it to a variable `pizzaStack` using the `const` keyword.

Hint

```
// Define an empty pizza stack with a maxSize of 6
const pizzaStack = new Stack(6);
```

2.

Use a `for` loop to push six pizzas to `pizzaStack`, naming each pizza `'Pizza #n'` where `n` is from `1` to `6`.

Hint

```
// Add pizzas as they are ready until we fill up the stack
for (let i = 1; i < 7; i++) {
  pizzaStack.push('Pizza #' + i);
}
```

3.

Try adding 'Pizza #7' to `pizzaStack` to see if the program will throw any error. Add this code inside a `try` and `catch` block.

Hint

```
// Try pushing another pizza to check for overflow
try {
  pizzaStack.push('Pizza #7');
} catch(e) {
  console.log(e);
}
```

4.

Before we deliver our pizzas, let's take a peek at which pizza is at the top of the stack. Log a message `The first pizza to deliver is` followed by the value of the top pizza.

5.

We are ready to deliver the pizzas off our stack. Deliver all the pizzas from the stack from top down. Which method would you use to do so?

Hint

```
for (let i=0; i<6; i++) {
  pizzaStack.pop();
}
```

6.

Is the pizza stack now empty? Try removing one more pizza off the stack. Catch any error that might be thrown. Do this in a `try` and `catch` block.

Hint

```
// 5. Try popping another pizza to check for empty stack
try {
  pizzaStack.pop();
} catch(e) {
  console.log(e);
}
```

main.js

```
const Stack = require('./Stack');

// 1. Define an empty pizza stack with a maxSize of 6
const pizzaStack = new Stack(6);

// 2. Add pizzas as they are ready until we fill up the stack
for (let i=1; i < 7; i++) {
  pizzaStack.push('Pizza #'+i);
}

// 3. Try pushing another pizza to check for overflow
try {
  pizzaStack.push('Pizza #7');
} catch(e) {
  console.log(e);
}

// 4. Peek at the pizza on the top of stack and log its value
console.log('The first pizza to deliver is', pizzaStack.peek());

// 5. Deliver all the pizzas from the top of the stack down
for (let i=0; i < 6; i++) {
  pizzaStack.pop();
}

// 6. Try popping another pizza to check for empty stack
try {
  pizzaStack.pop();
} catch(e) {
  console.log(e);
}
```