Notes - JavaScript - Collections - Stack

- Dr Nick Hayward

A brief outline of implementing a *stack* data structure in JavaScript.

Contents

Intro

A *stack* data structure may commonly be represented as a modified, restricted array or list in various programming and scripting languages.

However, a *stack* is an efficient data structure for many development purposes. Data may only be added and removed from the top of the structure, thereby affording ease of implementation and speed.

So, we commonly refer to a stack as *last in, first out*, or *LIFO*.

A stack of plates in a restaurant kitchen is a good analogy of this structure's usage. Dirty plates are added to the top of the stack. A dishwasher will wash these plates from the top down, so last plate on the stack will be washed first.

A *push* method may add a value to the end of an array. Likewise, a complementary *pop* method may be used to remove the last value in the array.

So, a stack is a data structure, which allows values to be pushed into it, and popped from it as needed. Last item added is now the first removed.

Stacks may be used for many different purposes in development. From execution requests to function calls, a stack is a common strcuture for storing lists of items for ordered usage.

JavaScript call stack

In JavaScript, for example, the order of execution for functions and application code is defined by the *call stack*. This stack provides the context for ordered execution of code.

For example, a conceptual stack of ordered execution might be as follows,

```
not in function
    in greetings function
        in console.log
    in greetings function
not in function
    in console.log
not in function
```

This stack represents the following sample code,

```
function greetings(name) {
  console.log("Hello " + name);
}

greet("Daisy");
console.log("Goodbye");
```

So, the context for this code's execution will be stored in the *call stack*.

recursion and the call stack

A stack may be used, for example, to represent execution logic for a recursive function.

The following code example uses a *call stack* to ensure expected execution,

```
function findSolution(target) {
    function find(current, history) {
        if (current == target) {
            return history;
        } else if (current > target) {
            return null;
        } else {
            return find(current + 5, `(${history} + 5)`) || find(current * 3,
`(${history} * 3)`);
        }
    }
    return find(1, "1");
}

console.log(findSolution(24));
```

The initial `findSolution()` function is called with the passed parameter of `24`, the value to check.

This function returns an executed `find()` function with initial test values for current and history.

As part of this function's execution, it checks these initial values until it reaches the `else` part of the conditional statement.

This returns the `find()` function, called recursively by itself, initially checking against an addition of 5. It will continue to check possible values with `+ 5` until it either succeeds or moves onto the right side of logical OR, `||`, and a check with `* 3`.

Again, it will either succeed or fail with these recursive checks.

However, the structure that permits this recursiion to execute is the *call stack*. It provides a defined pattern to execution, which allows the code to run as expected.

Stack operations

So, we may define a stack as a simple list of elements, which may be accessed from only one end. This is known as the top of the stack.

This is why we refer to this data structure as *last in, first out*.

A known limitation of this structure is the lack of access to elements not at the top of the stack. This is a simple difference between a basic list or array, and a specific stack data structure.

To access the bottom element, all of the elements above must first be *popped*.

Therefore, stack operations are simple, and may be defined as follows

- add elements to the top
- pop elements from the top

However, this also means that specific restrictions must be in place to ensure *only* these operations are allowed for the data structure. If not, it ceases to be a *stack*.

Complementary operations are commonly available, although these may vary relative to language implementation. For example, a stack may permit the following

- view the top element in the stack
    - this is not *pop*, as the element is not removed from the stack
    - operation known as *peeking*
- *clear* operation will remove all elements from the stack
- `length` property returns the number of elements in the stack
- `empty` returns whether the stack has any values or not

Example implementations

We may choose various existing data structures to define a custom stack. For example, we might use an array or list, and the choice will often depend on support in the chosen programming language.

For JavaScript, a common option is an `array` object.

We'll define a constructor for a `stack` object, and then extend the prototype for custom properties and methods.

stack constructor

The initial constructor is as follows

```
// CONSTRUCTOR = Stack object
function Stack() {
    /* define instance properties for stack
    * - empty array for instantiated stack
    * - options might include max length, restricted data type &c.
    */
    this.store = [];
}
```

As we instantiate a basic Stack object, we are simply defining an empty array as the store for the Stack data structure.

This constructor may be updated to include type checks and restrictions, initial values for the Stack, required access context, and so on.

extend the prototype

However, we may initially extend the Prototype for this object to add the required functionality for a basic stack. For example, we may define functions for the following

- add data
- delete data
- get the size of the Stack

prototype - add data

The *add data* function will simply add passed data to the top of the stack,

```
// PROTOTYPE - add method for value pushed to top of stack
Stack.prototype.add = function (value) {
    this.store.push(value);
    console.log(`value added = ${value}`);
}
```

As the underlying `store` object is an array for the Stack, we may use the default `push()` method to add the required data.

prototype - delete data

The *delete data* function may then be defined as follows,

```
Stack.prototype.delete = function () {
    const deletedValue = this.store.pop();
    console.log(`last value deleted = ${deletedValue}`);
}
```

As with *add data*, we may use the default `pop()` method for the `store` array in the Stack data structure.

prototype - size of Stack

We may also define a simple `size()` function for the Stack, which will likewise use a built-in Array property for `length`,

```
Stack.prototype.size = function () {
    const size = this.store.length;
    console.log(`store size = ${size}`);
}
```

prototype - peek Stack

Another useful initial option, as noted above, is *peeking* at the top of the Stack.

For example,

```
Stack.prototype.peek = function () {
    const peekValue = this.store[(this.store.length-1)]
    console.log(`top value = ${peekValue}`);
}
```

This function will return a copy of the top value, but will not delete the item from the Stack and the underlying `store` array.

prototype - clear stack

A common operation for a Stack is to clear all entries, yet preserve the Stack itself. In effect, we're resetting the store array for the instantiated Stack object.

For example,

```
Stack.prototype.clear = function () {
    // resets Stack's array store - clears all items
    this.store = [];
}
```

prototype - check empty stack

We may also check an instantiated Stack object for entries, effectively determining whether the stack is empty or not.

```
Stack.prototype.empty = function () {
    if (this.store.length === 0) {
        return true;
    } else {
        return false;
    }
}
```

The conditional logic has been placed in this function, and not passed down the chain of logic to the requesting application call. This means the function is self-contained, and returns a valid response regardless of execution context.

As we develop the Stack's Prototype methods, we may add further restrictions and controls to clearly define how to use this data structure. We may also define what and how may be returned for this custom data structure.

Control access to the Stack

Whilst the Stack object, and its methods, are now working as expected, it is still open to mis-use due to the array object in the Stack.

However, we may also restrict and control access to this Stack data structure using a Proxy.

proxy wrapper for the stack

To use a Proxy with our Stack constructor, we may define a custom `construct` trap. We may also use the Reflect API to define defaults for handlers.

The `construct` trap intercepts calls to the defined `new` operator for a given constructor.

For example. we may define an initial Proxy wrapper for a passed constructor

```
/*
* PROXY
*/
function proxyConstruct(constructor) {

    const handler = {
        construct(constructor, args) {
            console.log('proxy constructor...');
            // const stack = Reflect.construct(constructor, args);
            return new constructor(...args);
        }
    };
    return new Proxy(constructor, handler);
}
```

and then pass our basic `Stack` constructor to the proxy

```
// proxy wrapper for Stack constructor
const proxiedStack = new proxyConstruct(Stack);
// instantiate proxied Stack & check store...
console.log(new proxiedStack().store);
```

This instantiation of a proxied Stack object allows us to wrap the constructor for the Stack. However, we may still use prototype methods for a successfully instantiated Stack object.

The benefit of using a proxy for the constructor is control of the initial object instantiation. If the object cannot be instantiated, access to the Prototype methods becomes irrelevant.