03. DATASTRUCTS Stacks.

Lists are a natural form of organization for data. We have already seen how to use the List class to organize data into a list. When the order of the data being stored doesn't matter, or when you don't have to search the data stored, lists work wonderfully. For other applications, however, plain lists are too simple and we need a more complex, *listlike* data structure. A list-like structure that can be used to solve many problems in computing is the **stack**.

Stacks are efficient data structures because data can be added or removed only from the top of a stack, making these procedures fast and easy to implement. Stacks are used extensively in programming language implementations for everything from expression evaluation to handling function calls.

3.1. Stack Operations

A **stack** is a list of elements that are accessible only from one end of the list, which is called the top. One common, real-world example of a stack is the stack of trays at a cafeteria. Trays are **always removed from the top**, and when trays are put back on the stack after being washed, they are **placed on the top of the stack**.

The stack is known as a last-in, first-out (LIFO) data structure. Because of the last-in, first-out nature of the stack, any element that is not currently at the top of the stack cannot be accessed. To get to an element at the bottom of the stack, you have to dispose of all the elements above it first.

The two primary operations of a stack are adding elements to a stack and taking elements off a stack.

- Elements are added to a stack using the **push** operation.
- Elements are taken off a stack using the **pop** operation.
- Another common operation on a stack is **viewing the element at the top of a stack**. The **pop** operation visits the top element of a stack, but it permanently removes the element from a stack.

The **peek** operation returns the value stored at the top of a stack without removing it from the stack. To keep track of where the top element is, as well as keeping track of where to add a new element, we use a top variable that is incremented when we push new elements onto the stack and is decremented when we pop elements off the stack.

While **pushing**, **popping**, and **peeking** are the primary operations associated with a stack, there are other operations we need to perform and properties we need to examine.

- The **clear** operation removes all the elements from a stack.
- The **length** property holds the number of elements contained in a stack.
- We also define an **empty** property to let us know if a stack has no elements in it, though we can use the length property for this as well.

***top** (property)*	Current position in the stack
***length** (property)*	Returns the number of elements in the stack
***clear** (function)*	Clears all elements from stack
***push** (function)*	Inserts new element at the top of the stack
***pop** (function)*	Removes the element at the top of the stack
***peek** (function)*	Returns the value stored at the top of the stack

3.2. A Stack Class Implementation

To build a **stack**, we first need to decide on the underlying data structure we will use to store the stack elements. We will use an **array** in our implementation. We begin our stack implementation by defining the constructor function for a **Stack** class:

Example: Stack class constructor.

```
class Stack {

   constructor(){
       this.dataStore = [];
       this.top = 0;
   }
…

}
```

The array that stores the stack elements is named **dataStore**. The constructor sets it to an empty array. The *top* variable keeps track of the top of the stack and is initially set to 0 by the constructor, indicating that the 0 position of the array is the top of the stack, at least until an element is pushed onto the stack.

3.2.1. Push: Adding an Element to a Stack

The first function to implement is the **push**() function. When we push a new element onto a stack, we have to **store it in the top position and increment the top variable** so that the new top is the next empty position in the array. Here is the code:

Example: push() method.

```
push(element) {

     this.dataStore[this.top++] = element;

}
```

Pay particular attention to the placement of the increment operator after the call to **this.top**. Placing the operator there ensures that the current value of top is used to place the new element at the top of the stack before top is incremented.

3.2.2. Pop: Removing an Element from a Stack.

The **pop**() function does the reverse of the *push*() function—it returns the element in the top position of the stack and then decrements the top variable:

Example: *pop*() method.

```
pop(element) {

    return this.dataStore[--this.top];
}
```

The **peek**() function returns the top element of the stack by accessing the element at the top-1 position of the array:

Example: *peek*() method.

```
peek(element) {

    return this.dataStore[this.top-1];
}
```

If you call the **peek**() function on an empty stack, you get undefined as the result. That's because there is no value stored at the top position of the stack since it is empty. There will be situations when you need to know how many elements are stored in a stack. The **length**() function returns this value by returning the value of top:

Example: *length*() method.

```
length() {

    return this.top;
}
```

Finally, we can **clear a stack by simply setting the top variable back to 0**:

Example: *clear*() method.

```
clear() {

    return this.top=0;
}
```

3.2.3. Full Stack Class Implementation

The following code shows the complete implementation of the Stack class.

Example: The *Stack* class.

```
class Stack {

   constructor(){
       this.dataStore = [];
       this.top = 0;
   }

   push(element) {
       this.dataStore[this.top++] = element;
   }

   peek() {
       return this.dataStore[this.top-1];
   }

   pop() {
       return this.dataStore[--this.top];
  }

  clear() {
      this.top = 0;
  }

  length() {
      return this.top;
  }

}
```

3.2.4.Testing Stack class implementation.

The following code demonstrates a program that tests this implementation.

Example: Testing the *Stack* class implementation.

```
var s = new Stack();

s.push("David");
s.push("Raymond");
s.push("Bryan");
```

```
console.log("length: " + s.length());
console.log(s.peek());

var popped = s.pop();
console.log("The popped element is: " + popped);
console.log(s.peek());


s.push("Cynthia");
console.log(s.peek());

s.clear();
console.log("length: " + s.length());
console.log(s.peek());

s.push("Clayton");
console.log(s.peek());
```

The output from previous code is:

Output:

```
length: 3
Bryan
The popped element is: Bryan
Raymond
Cynthia
length: 0
undefined
Clayton
```

3.3. Using the Stack Class

There are several problems for which a stack is the perfect data structure needed for the solution. In this section, we look at several such problems.

3.3.1. Multiple Base Conversions

A **stack** can be used to **convert a number from one base to another base**.

Given a number, **n**, which we want to convert to a **base, b**, here is the algorithm for performing the conversion:

1. The rightmost digit of n is *n % b*. Push this digit onto the stack.
2. Replace n with *n / b*.
3. Repeat steps 1 and 2 until *n = 0* and there are no significant digits remaining.
4. Build the converted number string by popping the stack until the stack is empty.

NOTE: This algorithm will work only with bases 2 through 9.

We can implement this algorithm very easily using a stack in JavaScript. Here is the definition of a function for converting a number to any of the bases 2 through 9:

Example:

```
function mulBase(num, base) {

   var s = new Stack();
   do {
      s.push(num % base);
      num = Math.floor(num /= base);
   } while (num > 0);

   var converted = "";
   while (s.length() > 0) {
      converted += s.pop();
   }

   return converted;
}
```

The following code demonstrates **how to use this function for base 2 and base 8 conversions**.

Example: Converting numbers to base 2 and base 8

```
function mulBase(num, base) {
   var s = new Stack();
   do {
      s.push(num % base);
      num = Math.floor(num /= base);
   } while (num > 0);

   var converted = "";
   while (s.length() > 0) {
      converted += s.pop();
   }
   return converted;
}

// Converting 32 to base 2
var num = 32;
var base = 2;
var newNum = mulBase(num, base);
console.log(num + " converted to base " + base + " is " +
newNum)

// Converting 125 to base 8
num = 125;
base = 8;
var newNum = mulBase(num, base);
console.log(num + " converted to base " + base + " is " +
newNum);
```

The output from previous code is:

Output:

```
32 converted to base 2 is 100000
125 converted to base 8 is 175
```

3.3.2. Palindrome

A **palindrome** is a word, phrase, or number that is spelled the same forward and backward. For example, "*dad*" is a palindrome; "*racecar*" is a palindrome; "*A man, a plan, a canal: Panama*" is a palindrome if you take out the spaces and ignore the punctuation; and 1,001 is a numeric palindrome.

We can use a stack to determine whether or not a given string is a palindrome. We take the original string and push each character onto a stack, moving from left to right. When the end of the string is reached, the stack contains the original string in reverse order, with the last letter at the top of the stack and the first letter at the bottom of the stack

Once the complete original string is on the stack, we can create a new string by popping each letter the stack. This process will create the original string in reverse order. We then simply compare the original string with the reversed work, and if they are equal, the string is a palindrome.

Example: This code presents a program that determines if a given string is a palindrome.

```
function isPalindrome(word) {
   var s = new Stack();
   for (var i = 0; i < word.length; ++i) {
      s.push(word[i]);
   }

   var rword = "";
   while (s.length() > 0) {
      rword += s.pop();
   }
   if (word == rword) {
```

```
            return true;
          }
        else {
            return false;
        }
      }

    // Check if "hello" is palindrome
    var word = "hello";
     if (isPalindrome(word)) {
       console.log(word + " is a palindrome.");
     }
     else {
       console.log(word + " is not a palindrome.");
     }

    // Check if "racecar" is palindrome
     word = "racecar";
     if (isPalindrome(word)) {
       console.log(word + " is a palindrome.");
     } else {
       console.log(word + " is not a palindrome.");
     }
```

The output from this program is:

Output:

```
hello is not a palindrome.
racecar is a palindrome.
```

3.3.3. Demonstrating Recursion.

Stacks are often used in the implementation of computer programming languages. One area where stacks are used is in implementing recursion. It is beyond the scope of this book to demonstrate exactly how stacks are used to implement recursive procedures, but we can use stacks to simulate recursive processes. If you are interested in learning more about recursion, a good starting point is this web page that actually uses JavaScript to describe how recursion works.

To demonstrate how recursion is implemented using a stack, **let's consider a recursive definition of the factorial function**. First, here is a definition of factorial for the number 5:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Example: Here is a recursive function to compute the factorial of any number:

```
function factorial(n) {
   if (n === 0) {
      return 1;
   }
   else {
      return n * factorial(n-1);
   }
}
```

When called with the argument 5, the function returns 120.

To simulate computing 5! using a stack, first push the numbers 5 through 1 onto a stack. Then, inside a loop, pop each number and multiply the number by the running product, resulting in the correct answer, 120.

Example: Simulating recursive processes using a stack.

```
function fact(n) {
   var s = new Stack();
   while (n > 1) {
      s.push(n--);
   }

   var product = 1;
   while (s.length() > 0) {
      product *= s.pop();
   }
   return product;
 }

console.log(factorial(5)); // displays 120

console.log(fact(5)); // displays 120
```