JavaScript Arrays 21st **July** 2020.

Arrays are list–like objects whose prototype has methods to perform traversal and mutation operations. Neither the length of a JavaScript array nor the types of its elements are fixed. Since an array's length can change at any time, and data can be stored at non–contiguous locations in the array, JavaScript arrays are not guaranteed to be dense; this depends on how the programmer chooses to use them. In general, these are convenient characteristics; but if these features are not desirable for your particular use, you might consider using typed arrays. JavaScript arrays are used to store multiple values in a single variable.

In layman's language array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of user names, for example), storing the users in single variables could look like this:

```
let userA = "Jane";
let userB = "Moses";
let userc = "Rose";
```

But what if you want to loop through the users and find a specific one? And what if you had not 3 users, but 300,000? Here the solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Syntax for creating an array:

let *array_name* = [*item1*, *item2*, ...];

var *array_name* = [*item1*, *item2*, ...];

The users example we has earlier as an array would be:

```
let users = ["Jane", "Moses", " Rose "];
```

Note that:

– Spaces and line breaks are not important. A declaration can span multiple lines:

```
let users = [
                    "Jane",
                    "Moses",
                    " Rose "
                    ];
```

– You can use new array() method to declarea new array in JavaScript:

```
let users = new array("Jane", "Moses", " Rose ");
```

– You access an array element by referring to the **index number**.

```
let user =users[0]
```

Use case:

```
<div id= “demo”> </div>
<script>
var users = ["Jane", "Moses", "Rose"];
document.getElementById("demo").innerHTML = users[0];
</script>
```

The above program will get the element of the array at index 0, Jane (Remember array is 0 indexed) and add itin the DOM in the div with the id demo.

- You can change array elements as shown below:

```
<div id= “demo”> </div>
<script>
var users = ["Jane", "Moses", "Rose"];
users[0] = “Alexander”
document.getElementById("demo").innerHTML = users[0];
</script>
```

Index 1 we will have Alexander and not Jane anymore.

Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

- **length Property**:

The length property of an array returns the length of an array (the number of array elements).

```
let users = ["Jane", "Moses", " Rose "];
users.length; /* outputs 3 */
```

Note that:

The length property is always one more than the highest array index.

– **Accessing the First Array Element**

```
let users = ["Jane", "Moses", " Rose "];

let first_user = users[0]; /* outputs Jane */
```

– **Accessing the Last Array Element**.

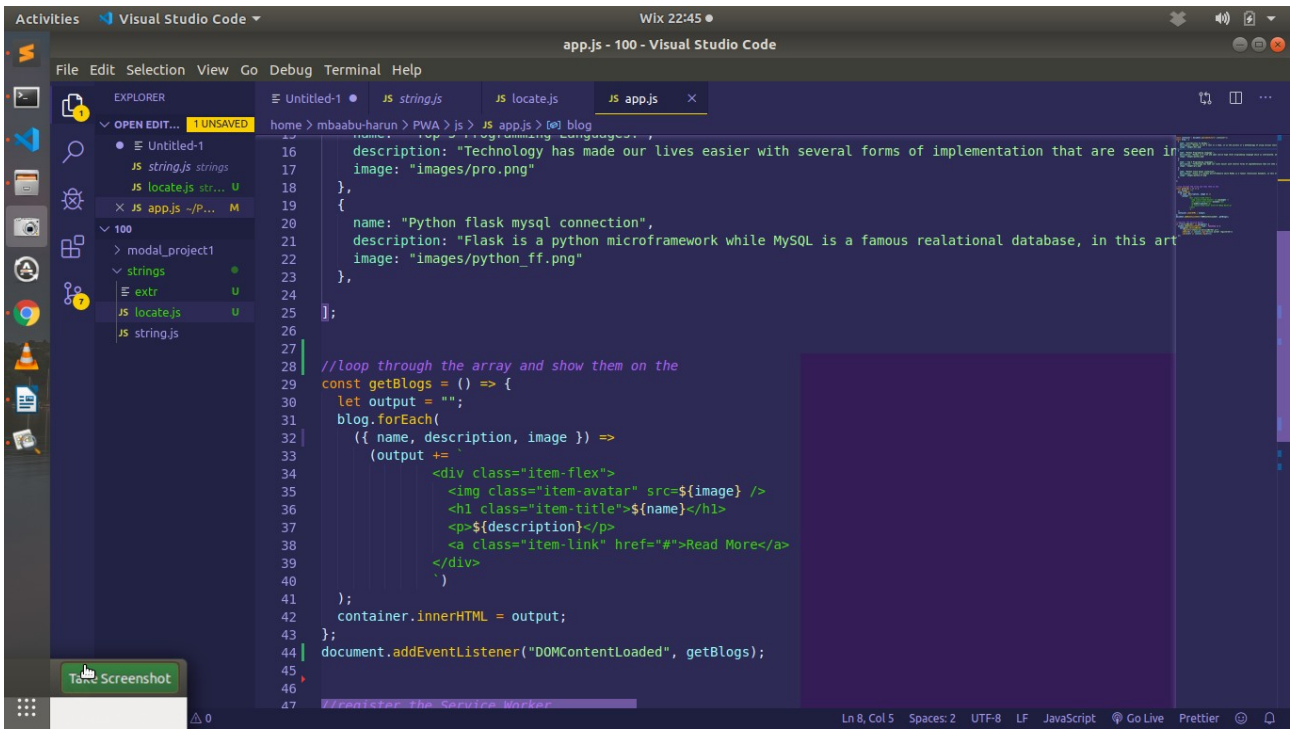
```
let users = ["Jane", "Moses", " Rose "];

let last_user = users(users.length -1); /* outputs Rose*/
```

– **Looping Array Elements**.

The safest way to loop through an array, is using a for loop:

```
let users, text, user_length, i;
let users = ["Jane", "Moses", " Rose "];
user_length = users.length;
text = "<ul>";
for(i = 0; i < user_length; i++) {
  text += "<li>" + users[i] + "</li>";
}
text += "</ul>";
```

Example below is a real life implementation in creating a blog:



Source Code on gitHub: https://github.com/HarunHM/-PWA-With-VanillaJS-CSS-AND-HTML

Live Demo: https://wonderfulpwa.netlify.app/

Adding Array Elements

The easiest way to add a new element to an array is using the push() method:

```
let users = ["Jane", "Moses", " Rose "];

users.push("Liam"); /* Adds a new element, Liam into the array*/
```

Associative Arrays

Many programming languages support arrays with named indexes. Arrays with named indexes are called associative arrays (or hashes). JavaScript does not support arrays with named indexes. In JavaScript,arrays always use numbered indexes. We will discuss this further when implementing hash & hash-table later. Remember There is no need to use the JavaScript's built-in array constructor new Array().

Queue & **Stack**.

Two of the most commonly used data structures in web development are **stacks and queues**. Many users of the Internet, including web developers, are unaware of this amazing fact. If you are one of these developers, then prepare yourself for two enlightening examples: the 'undo' operation of a text editor uses a stack to organize data; the event–loop of a web browser, which handles events (clicks, hoovers, etc.), uses a queue to process data.

Now pause for a moment and imagine how many times we, as both a user and developer, use stacks and queues. That is amazing, right? Due to their ubiquity and similarity in design, I have decided to use them to introduce you to data structures.

What is a Stack?

A stack is a type of data structure that is linear where order is conserved. For many people, stack is known to have a LIFO (**L**ast **I**n **F**irst **O**ut) mechanism. An example of a stack in real life: stack of books. The first book you put down will be used last, while the last you put down is the first you'll use. With Stacks, we therefore only insert and delete at one end of the list, which we refer to as the top. An insert is known as a push while delete is known as a pop.

Stack operations

- Pop: remove an element from the top of the stack.
- Push: add/insert an element to the top of the stack.
- Peek: return the element of the top without removing it.

- •Clear: removes all elements so you get a clear stack.
- •Size: the number of elements in the stack.
- •isEmpty: return true or false regarding if the stack is empty.
- •isFull: return true or false regarding if the stack is full.

Implement a Stack.

```
function Stack() {
      this.elements = [];
}
/* Add the element on top */
Stack.prototype.push = function(element) {
    this.elements.push(element)
}
/*  Remove the top element */
Stack.prototype.pop = function() {
    return this.elements.pop()
}
/* Return the top element without removing it */
Stack.prototype.peek = function() {
     return this.element[this.length - 1];
}
/* */
 Stack.prototype.getSize = function() {
    return this.elements.length;
}
```

The simplest method to implement a stack is to use array since it ready has a push() and pop() method.

```
let stack = [];
stack.push(2);
console.log(stack); /* [2] */
stack.push(5);
console.log(stack); /*[2, 5] */
stack.pop();
console.log(stack); /* [2] */
```

What is a Queue?

Similar to a stack, **queue** is a linear data structure where it obeys the FIFO (**F**irst **I**n **F**irst **O**ut) mechanism. You can think of a queue as a single line of people at a fast food restaurants.

Queue operations.

- Enqueue: add/insert an element at the rear (end of queue).
- Dequeue: delete an element at the front of the queue.
- Peek: return the element from the front without removing it.
- Clear: removes all elements so you get a clear queue.
- Size: the number of elements in the queue.
- isEmpty: return true or false regarding if the queue is empty.
- isFull: return true or false regarding if the queue is full.

Implement a Queue.

An Array in Javascript already has a push and shift method, which achieves what we want to achieve with an enqueue and dequeue, so creating your own Queue might seem to be unnecessary.

Sadly shift takes O(n) running time, and we want our dequeue to be O(1), so it might be a good idea to implement your own queue if you'll have a big input.

```
function Queue() {
    this.elements = [];
}
/* Add/insert an element at the rear (end of queue).  */
Queue.prototype.enqueue = function(element) {
    this.elements.push(element)
}
/*Delete an element at the front of the queue in O(n) running time*/
Queue.prototype.dequeue = function() {
    return this.elements.shift()
}
```

But most of the times when you are implementing a data structure you may want to achieve aO(1) runtime. One of the solutions we can use to achieve an O(1) runtime is to keep track of the rear and front index as shown below:

```
function Queue() {
   this.elements = {};
   this.rear = 0; // or head or end
   this.front = 0; // or tail or beginning
}
/* Add/insert an element at the rear (end of queue).*/
Queue.prototype.enqueue = function(element) {
   this.elements[this.rear++] = element;
}
/* Delete an element at the front of the queue in O(1) running time*/
Queue.prototype.dequeue = function() {
   if (this.rear === this.front)
       return undefined

   var element = this.elements[this.front];
   delete this.elements[this.front++];
   return element;
}
```

“How you look at it is pretty much how you’ll see it”

Best Wishes.

Lux Tech Academy.