Extra Notes - JavaScript - Data Structures

- Dr Nick Hayward

A brief introduction to basic data structures in JavaScript.

Contents

Intro In JavaScript, and many other programming languages, we may store strings and numbers as individual values in a single variable.

Whilst this is useful for storing smaller chunks of data, such as a word or phrase, we also need to be able to store larger amounts of data. For example, multiple values in a single variable.

This data will, customarily, be organised in a logical manner. Perhaps, we might require a numerical index for the values, or a key and value pair to easily reference and search for a particular stored value.

This is the role of data structures, including

- indexed collections - **arrays**…
  - keyed collection - **maps**, **sets**…

Further details,

- MDN - JavaScript data types and data structure

Arrays - intro An *array* allows us to store multiple values in a single variable with an associated numerical index.

They are also one of the most common data types we use in programming, and specifically JavaScript. Using an array, we may now handle various collections of items.

For example, instead of creating separate variables to store each name in a sports team, we can create a single variable to store an array with all of the names.

Another benefit is that the size of the array in JavaScript will be dynamic. If a new player is added to the team, we can also add their name to the array. Likewise, if they leave, we simply remove their name from the array, and so on.

Arrays are important to the way we program with JavaScript, and many other programming languages.

As with other data types in JavaScript, arrays are nothing more than objects. A benefit of this feature is that arrays can access methods and properties. For example, we can use the available `length` property to quickly find the total number of items in an array. This makes our lives a lot easier as we work with arrays in JavaScript.

Further details,

- W3Schools - Arrays
  - MDN - Array

Arrays - creating an array We can create an array in JavaScript using either of the following two options,

- the built-in Array constructor
- array literals `[]`

e.g.

```
// using array literals to create new array
var players = ["Amelia", "Emma", "Daisy", "Yvaine"];
// using Array constructor to create new array
var places = new Array("Paris", "Nice", "Marseille");
```

However, *array literals* tend to be the more common option for JS development. The Array constructor is useful if we need to extend the functionality of an array beyond the default options.

Arrays - access To access values stored in an array, we can use the *index* of an item. This follows the same pattern as finding a single character in a string.

For example,

```
// using array literals to create new array
var players = ["Amelia", "Emma", "Daisy", "Yvaine"];
var places = ["Paris", "Nice", "Marseille"];
players[0];
// value is "Amelia"
places[1];
// value is "Nice"
```

n.b. array index begins at `0`

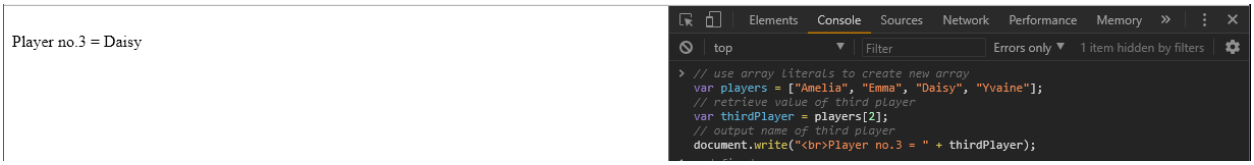

Figure 1: JavaScript - array access

Arrays - set, change, add elements The simplest option for modifying data in an array is using a specific index number, e.g.

```
players[3] = "Rose";
```

This will update the value stored in the `players` array from the current value `Yvaine` to the new value `Rose` .

If we specify an index position beyond the current bounds of the array, such as

```
players[5] = "Violet";
```

the array will dynamically expand to add this new value.

However, we may end up creating a hole in the array, and the item at index 4 will now be set as `undefined`. The length property of the array will also be updated to reflect the new size.

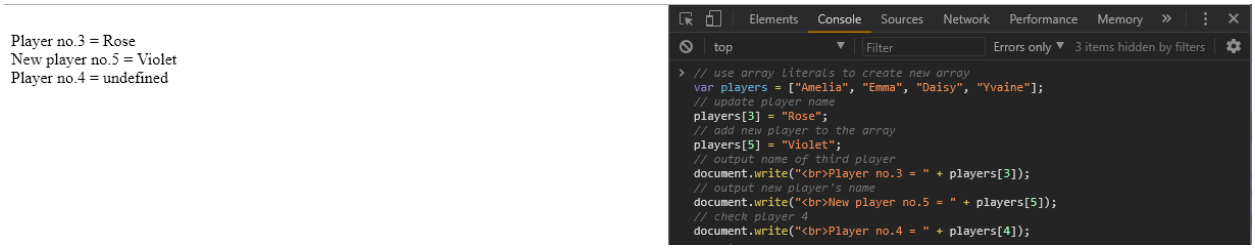

Figure 2: JavaScript - modify an array by adding or updating values

Figure 3: JavaScript - add new items to array - dynamically expand

Arrays - mix data types Another benefit of storing data in an array is that we can mix data types. We can store numbers with strings, for example.

```
var players = [1, "Amelia", 42, "Yvaine", "Daisy"];
```

We can also store an array within an array, creating what we call a **multi-dimensional array**. So, we might then add an array as well,

```
var players = [6, "names", ["Amelia", "Emma", "Daisy", "Yvaine", "Rose", "Violet"]];
```

So, in this example, we're able to store three separate values each with different values and data type. We have a number, a string, and another array. This is a simple example of a multi-dimensional array.

We may consider the `players` array as follows,

- index `0` - stores a number value `6`
- index `1` - stores a string value `"names"`
- index `2` - stores another array

Arrays - multi-dimensional access We may then access the value in an inner array using the familiar pattern of index positions, e.g.

```
// create new multi-dimensional array
var players = [6, "names", ["Amelia", "Emma", "Rose", "Yvaine", "Daisy", "Violet"]];
// get value from inner array - fifth name
var fifthName = players[2][4];
```

In effect, we access the array at index position `2` , and then access index position `4` in the inner array.

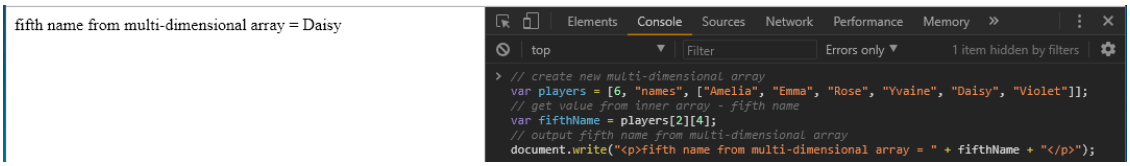

Figure 4: JavaScript - access the inner array of a multi-dimensional array

Arrays - practical abstraction & usage There are many practical uses for an array data structure.

We may also use an array data structure as a building block for other data structures.

example 1 - create a stack For example, we might begin with a simple **stack** to store some data.

With a stack, we are storing and accessing data in a known, predictable pattern. In effect, the last data in the stack will be the first data out. So, we often think in terms of the acronym,

- **LIFO** - Last In, First Out

```
> // create first array of values
  var playersAll = ["Amelia", "Yvaine", "Emma", "Daisy"];
  // push a new player to the stack
  playersAll.push("Violet");
  // push another player to the stack
  playersAll.push("Ruby");
  // pop the last player added to the stack
  playersAll.pop();
< "Ruby"
> // check stack values
  playersAll;
< ▼(5) ["Amelia", "Yvaine", "Emma", "Daisy", "Violet"]
    0: "Amelia"
    1: "Yvaine"
    2: "Emma"
    3: "Daisy"
    4: "Violet"
    length: 5
  ▶ __proto__: Array(0)
>
```

Figure 5: JavaScript - arrays - use push() and pop() methods to create LIFO

example 2 - create a queue We can also create the opposite of a stack with a **queue**. In effect, the first data in the queue will also be the first data out. So, we can use the acronym

- **FIFO** - First In, First Out

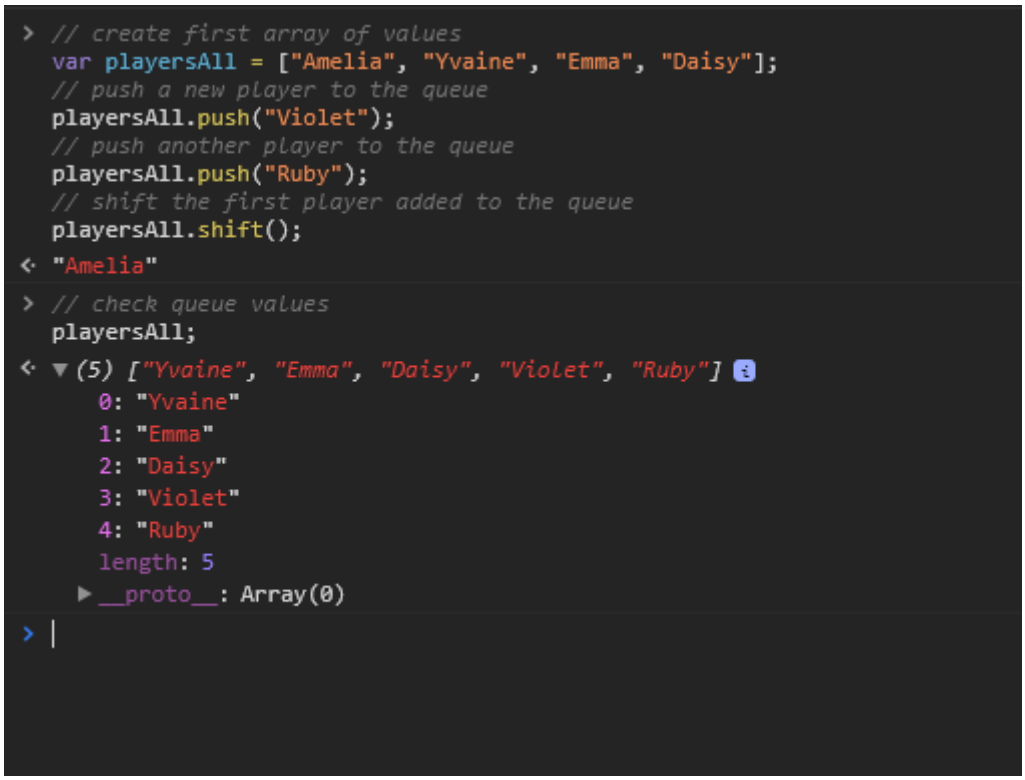

Figure 6: JavaScript - arrays - use push() and shift() methods to create FIFO

References

- MDN - JavaScript data types and data structure
- W3Schools - Arrays
  - MDN - Array