Array Operations

Rich Web Application Technologies

Array Constructors

- Arrays can be constructed from iterables, strings, maps or sets

```
const arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; simple way

const arr = Array.from('Hello'); // => ['H', 'e', 'l', 'l', 'o’]  from string

const set = new Set([1, 2, 2, 3, 3, 3, 4, 5]);
const arr = Array.from(set);   ; // => [1, 2, 3, 4, 5] from set without duplicate

const map = new Map([ ['a', 1], ['b', 2] ]); create map with 2 elements [key, value]
map.set('c', 3); set third element
const arr = Array.from(map);// => [ ['a', 1], ['b', 2], ['c', 3] ] return array of arrays
```

Array Traversal

- You can *walk* an array, visiting each element of an array using a variety of approaches

```
const arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; // arr.length === 10 create array
arr[100] = 100;                             // arr.length === 101 extend the size

Iterating the array:
for (let i = 0; i < arr.length; i++) { if (i in a) console.log(a[i]); } using for loop

for (let i in arr) {console.log(arr[i]);} “in” operator – return only existing elements of the
arr by their index

for (let e of arr) { if (e) console.log(e);} “of” operator – return the values, including
undefined (you need to check it first)

arr.forEach(e => console.log(e)); “forEach” function – only elements that exist
```

Array Slicing

- A slice is a copy subset of the elements of an array specified by the start and end index

```
const nums = ['One', 'Two', 'Three', 'Four', 'Five'];

const copy = nums.slice(); // => ['One', 'Two', 'Three', 'Four', 'Five’] making a copy

const some = nums.slice(1, 3); // => ['Two', 'Three’] start and end index

const end = nums.slice(4);     // => ['Five’] slice from an specific index
```

Adding Array Elements

- Adding an element has a different form depending on where in the array you are adding

```
let arr = [1, 2, 4, 5];

arr.push(6);          // => [1, 2, 4, 5, 6] add at the end (append)

arr.unshift(0);       // => [0, 1, 2, 4, 5, 6] add at the beginning (prepend)

arr.splice(3, 0, 3); // => [0, 1, 2, 3, 4, 5, 6] add in the middle (where, how many things
to remove, new value)
```

- In each case here, the original array `arr` is mutated in place (arrays and objects are mutable)

Removing or Replacing Array Elements

- The splice method on arrays can be used to insert, replace or remove array elements

```
let arr = [0, 1, 2, 3, 4, 5, 6];

arr.splice(3, 1, 'x'); // => [0, 1, 2, 'x', 4, 5, 6] replace in the middle
(where, how many things to remove, new value)

arr.splice(3, 1);      // => [0, 1, 2, 4, 5, 6] removing element (where,
how many things to remove)

arr.splice(3);         // => [0, 1, 2] removing element (where)
```

- In each case here, the original array `arr` is mutated in place

Visitor Pattern Using `forEach()`

The forEach() method assigns a visitor pattern on JS arrays. Visitor pattern defines a new operation on an object without changing the object itself. The allows a flawless way for clients to implement future extensions (follows open/closed principle: software entities are closed for modification but open for extension).

Visitor Pattern Using `forEach()`

- Arrays come with a forEach() method already defined. The definition looks something like the following (simplified):

```
Array.prototype.forEach = (callback) => {
  if (typeof callback !== 'function') {
    throw new TypeError(callback + ' is not a function’); check if callback is a function
  }
  let O = Object(this); turn array into object
  for (let k = 0; k < O.length; k++) {
    if (k in O) callback.call(k, O[k], O); loop through the array
  }
}
```

- `forEach()` implements a general *visitor pattern* that others can be built on

Functional Programming Approach

- Commonly, developers take a non-mutating, functional approach to array processing building on the visitor pattern
- Some of the most important of these are shown here

arr.**map**(*callback*)	*arr*.**reduce**(*callback*)
arr.**filter**(*callback*)	*arr*.**every**(*callback*)
arr.**some**(*callback*)	*arr*.**find**(*callback*)

Array map( )

- Map( ) creates a new array with the results of calling a function for every array element

```
const numbers = [4, 9, 16, 25];
const newArr = numbers.map(Math.sqrt)

//outcome: 2,3,4,5
```

Array filter( )

- The filter( ) method creates an array filled with all array elements that passed a certain test

```
const ages = [32, 33, 16, 40];

ages.filter(checkAdult)    // Returns [32, 33, 40]

function checkAdult(age) {
  return age >= 18;
}
```

Array some( )

- Every( ) calls a given function on each of the elements of an array returning true if at least one element matches the function predicate

```
Array.prototype.filter = (callback) => {
  let O = Object(this), matches = false;
  O.forEach(x => {
    if (callback(x) { matches = true; return; }
  });
  return matches;
}

let arr = [10, 5, 30];
arr.some(x => x < 10); // => true
```

Array reduce( )

- Reduce( ) executes a reducer function for each value of the array, returning the accumulated result
- In the following example, the reduce function is subtracting the numbers in the array starting from the left

```
const numbers = [175, 50, 25];

document.getElementById("demo").innerHTML = numbers.reduce(myFunc);

function myFunc(total, num) {
  return total - num;
}
```

Array every()

- Every( ) calls a given function on each of the elements of an array returning true if every element matches the function predicate

```
Array.prototype.every = (callback) => {
  let O = Object(this), matches = true;
  O.forEach(x => {
    if (!callback(x) { matches = false; return; }
  });
  return matches;
}

let arr = [10, 20, 30];
arr.every(x => x > 0); // => true
```

Array `find()`

- Find( ) calls a given function on each of the elements of an array returning the first value matching the function predicate

```
Array.prototype.map = (callback) => {
  let O = Object(this), value;
  O.forEach(x => {
    if (callback(x) { value = x; return; }
  });
  return value;
}

let arr = [10, 5, 30, 1];
arr.find(x => x < 20); // => 5
```

Combining Functional Primitives

- The functional primitives (like the examples seen) can be combined together to build more complex functional elements to operate on arrays

```
const inventory = [                array of objects
    {name: 'apples', quantity: 2},
    {name: 'bananas', quantity: 7},
    {name: 'cherries', quantity: 5}
];
inventory.filter(product => product.name !== 'apples’)    first filter not equal to
apple
   .map(product => product.quantity)                map only the quantity
   .reduce((quantity, total) => total + quantity, 0);  reduce to the total quantity
// => 12
```

- Function chaining is available when the previous stage returns an array

Summary

- Arrays (or lists) are a powerful and flexible data structure for processing data in JS
- Arrays come with a traditional set of functions for creating and transforming array contents by index
- But there is a functional style that is arguably more powerful allowing processing by values in an immutable way
- The functional style supports operator composition allow complex transformations on arrays is a very readable form