ARRAYS, LOOPS, AND ITERATORS

ARRAYS

The purpose of an **array**?

We store collections of data in an array, which is great for enumerating or recording data.

Each item in an array is called an **element**, and each element has an **index**.

The **index** always starts at 0

```
let names = ['Jerry', 'George', 'Elaine', 'Kramer'];
```

We can target each **element** in the array via its **index**:

```
let first = names[0];
names;
```

What does **names[2]** return?

We can assign **values** to an array via the **index**

```
let names = ['Jerry', 'George', 'Elaine', 'Kramer'];
names[0] = 'Justin';
```

What will **names[0]** return? And what does the array contain now?

```
names;
=> ['Justin', 'George', 'Elaine', 'Kramer']
```

We can also find the number of **elements** in an **array** using the **length** property

```
let names = ['Jerry', 'George', 'Elaine', 'Kramer'];
names.length
=> 4
```

The **length** property will always give us a value one digit greater than the last **index**

To say this explicitly, **length** returns the number of items in the **array**, not the **index**

So the **index** of the last element is **length-1**

```
let names = ['Jerry', 'George', 'Elaine', 'Kramer'];
let lastIndexedItem = names.length-1;
lastIndexedItem;
=> 3
```

There are also **types** within **arrays**.

They can contain any type of **element** or **data** in JavaScript, and they can shrink or grow.

```
let sillyArray = [ 'Hello World!', true, undefined,
null, 42, ['Look!', 'a', 'nested Array!'], false ];
```

Side note, this code is *very* poor; you're just making your life difficult.

Keep different **data types** in separate **arrays**

Strings are similar to **arrays** in that we can find the **length** of them the same way we operate on **arrays**

```
let string = "Hello World!";
string.length;
=> 12

string[0];
=> H
```

We can also create arrays

```
let a = new Array();
a[0] = "dog";

a;
=> ["dog"]

let pets = new Array("dog", "cat", "unicorn");
pets;
=> ["dog", "cat", "unicorn"]
```

There are also a bunch of helper **methods**

The toString() method returns a string with each element separated by a comma:

```
array.toString();
```

The join() method returns a string with each element separated by a **parameter**:

```
array.join( param );
```

The pop() method returns the last item from the **array**:

```
array.pop();
```

The push() method adds one or more items to the end and returns the new **length**:

```
array.push( item1, item2, ..., itemN );
```

We can reverse the **array**:

```
array.reverse();
```

We can remove and return the first item:

```
array.shift();
```

We can add one or more **elements** to the front and return the new length:

```
array.unshift( item1, item2, ..., itemN );
```

An example, create an **array** and add **elements** to it using the **push method** in repl.it

```
let message = [];
message.push(1);

message.push( 'e', 'g', 'a', 's', 's' );
=> 6

message.push( 'e', 'm', 'T', 'E', 'R', 'C', 'E', 'S', 'X' );
=> 15
```

pop(), shift(), unshift()

```
message.pop();
=> 'X'

message.shift();
=> 1

message.unshift( 'duh' );
=> 14
```

Array reversal using reverse()

```
message.reverse();
['S','E','C','R','E','T','m','e','s','s','a','g','e','duh']
```

Turn that **array** into a string

```
message.join(' ');
'S E C R E T m e s s a g e duh'
```

LOOPS AND ITERATING

Okay, let's **loop** back to **loops**, and specifically the **while loop**.

We can use the **while statement** to run a block of code as long as the conditions are **true**. The condition is evaluated *before* executing the code.

```
while ( condition ) {
// statement
}
```

In a basic sense, loops execute blocks of code a set number of times.

An **infinite loop** is when we don't give the code a stopping point.

That will break your code, and I'm sure you'll all accidentally do it soon enough.

But, to reiterate (pun?), the **loop's** power is in the ability to run the same code over and over and over again.

These are tricky because you can easily get stuck in an infinite loop again:

```
while ( true ) {
// infinite loop
}

while ( false ) {
// the loop will never run
}
```

Another example, adding **numbers** to an **array** using a **while loop**.

```
let num = 1;
let numArray = [];
while ( num < 11 ) {
numArray.push( num );
num++
}
console.log( numArray );
```

Try that in repl.it

There's a **do-while loop**, which runs a block of code *until* the **condition** is **false**. The **condition** is **evaluated** after executing the **statement** once.

```
let num = 10;
let numArray = [];
do {
numArray.push( num );
num -= 1;
} while ( num > 0 );
console.log( numArray );
```

Try that in repl.it too

Let's look at a for loop:

```
let a = [ 1, 2, 3, 4, 5 ];
for ( let i = 0; i < a.length; i++ ) {
console.log( a[i] );
}
```

We can also cache the array's length, to save some time:

```
let a = [ 1, 2, 3, 4, 5 ];
let arrayLength = a.length;
for ( let i = 0; i < arrayLength; i++ ) {
console.log( a[i] );
}
```

And one more:

```
let pets = [ "dog", "cat", "turle", "bunny" ];

pets.forEach(
function( currentValue, index) {
console.log( "I want a ", currentValue );
console.log( index );
}
);
```

```
let departments = ['Fine Art', 'Illustration', 'Cartooning'];

for ( let i = 0; i < departments.length; i++ ) {
 let department = departments[i];
 console.log( department );
}
```

JavaScript **arrays** have several **iterator methods**. Many of the methods require a **function** to be passed in as an **argument**

Each **element** in the **array** has the **statement** in the **function body** applied to it individually.

For example, the forEach() method is a cleaner approach to the previous code:

```
let departments = ['Fine Art', 'Illustration', 'Cartooning'];

departments.forEach( function( department ) {
 console.log( department );
});
```

In the previous example, **'department'** was just an **element**; it was arbitrary

And the **function** is called a callback

In brief, a **callback** is a **function** to execute for each **element**

The **callback** also takes three **arguments**, the element value, the element index, the array being traversed

So, this:

```
departments.forEach( function(department) {
        console.log(department);
});
```

And this:

```
function useThisLater(element, index, array) {
        console.log("element: " + element);
        console.log("index: " + index);
        console.log(" ");
}

departments.forEach( useThisLater );
```

Function the same way.

Similar to how we discussed the difference in **variables** in **ES5** versus **ES6** (var versus let & const), there are are options to how we might write a **function** or **callback** in **ES6**.

We're specifically covering **functions** in all their complexity and nuance next week, but I'll give a quick rundown so you're not confused by documenation.

There are **function declartions**:

```
function myFuncName(param) {
return param;
}
```

Function expressions:

```
let myFuncName = function(param) {
return param;
}
```

And arrow functions:

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: => { return expression; }

// Parentheses are optional when there's
// only one parameter name:
(singleParam) => { statements }
singleParam => { statements }

// The parameter list for a function with no parameters
// should be written with a pair of parentheses.
() => { statements }
```

In a practical context, this would manifest like this:

```
let materials = ['Hydrogen','Helium','Lithium','Beryllium'];

console.log(materials.map(material => material.length));
// expected output: Array [8, 6, 7, 9]
```

Taken from MDN Arrow Functions using the Map method.

A direct comparison:

```
let materials = ['Hydrogen','Helium','Lithium','Beryllium'];

materials.map(material => material.length);
```

```
let materials = ['Hydrogen','Helium','Lithium','Beryllium'];

materials.map(function(material){
        return material.length;
})
```

Again, don't be confused when you read **documentation** and they're using **arrow function callbacks**.

Well, you can be confused but they're doing it on purpose because **arrow functions** are best practice in that context.

So, we just covered a lot of ground and remembering all of the particular **syntax** and names of these **methods** is exceedingly difficult to memorize--which is totally fine and normal.

Because of this, we constantly need to reference **documentation**.

If you recall, many documentation websites are on the syllabus.

In any case, let's take roughly 10 or so minutes to skim over some documention on the Mozilla developer site.

Go here: https://developer.mozilla.org/en-US/docs/Web/JavaScript and track down the documention for:

- .every()
- .some()
- .filter()
- .map()

After you've looked over the **documentation**, open repl.it and create these **arrays**:

```
let evens = [];
evens.push( 2, 4, 6, 8, 10 );

let odds = [];
odds.push( 1, 3, 5, 7, 9 );
```

The **every() method** tests whether **ALL** elements in an array pass the test implemented by the provided **function**

```
let evenResult = evens.every(num => num % 2 === 0);
let allDivisibleByFour = evens.every(num => num % 4 === 0);

console.log("evenResult", evenResult);
console.log("allDivisibleByFour", allDivisibleByFour);
```

The **some() method** tests whether **AN** element in the array passes the test implemented by the provided **function**

```
let someDivisibleByFour = evens.some(num => num % 4 === 0);

console.log("someDivisibleByFour", someDivisibleByFour);
```

The **filter() method** creates **a new array** with all elements that pass the test implented by the provided **function**

Note, this **method** does not mutate the original **array**

```
let bigNums = evens.filter(num => num > 5);
let smallNums = odds.filter(num => num < 5);

console.log("bigNums", bigNums);
console.log("smallNums", smallNums);
```

The **map() method** creates **a new array** with the results of calling a provided **function** on every element in the original **array**

```
let timesFive = evens.map(num => num * 5);
let timesTen = odds.map(num => num * 10);

console.log("timesFive", timesFive);
console.log("timesTen", timesTen);
```

As usual, we'll employ these techniques in an in-class demo, and if you have questions in the meantime let me know!