Data Structures: Arrays

Working With and Manipulating Indexed Data

We'll cover the following

There are many situations in which you will have a *collection* of related data.

Say, for example, we had a roster of students in a classroom. One way we could store information about the individual students is to assign each to a variable.

```
var student1 = "Mary";
var student2 = "Barbara";
var student3 = "David";
var student4 = "Alex";
```

But what about if you had hundreds of students? Assigning each student to a separate variable quickly becomes tedious.

Thankfully, Javascript has a few dedicated variable types for handling collections of data. Let's begin to discuss one of them.

Arrays

A Javascript **array** is a variable that can store multiple values. Those values are

accessed using a *numbered* indexing scheme.

To make an empty array, assign brackets (`[]`) after the variable declaration:

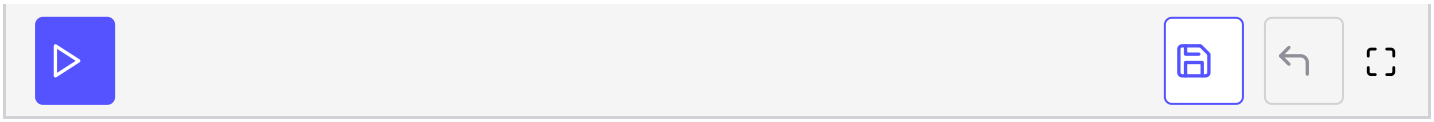
```
var emptyArray = [];
```

You can also assign values when creating the array by including them in between the brackets, with commas separating the values. Going back to our example, we can create an array named `students` like so:

```
var students = ["Mary", "Barbara", "David", "Alex"];
```

The values in an array can be accessed by writing the name of the array followed by an **index position** with the following syntax:

```
//Logs "Barbara" to the console
console.log(students[1]);
```



Arrays are **zero indexed**, meaning the first item in an array will have an index of `0`, the second item's index will be `1`, and so forth.

In an array, you can store any valid Javascript data type, and you are not limited to storing data of a single type.

So, you could store both **Strings** and **Numbers** in an array to indicate a student's name and age:

```
var students = ["Mary", 10, "Barbara", 11, "David", 12, "Alex", 11];
```

Nested arrays

You can also store arrays within an array, referred to as **nested arrays**. A better way of defining the `students` array above would be like so:

```
var students = [
  ["Mary", 10],
  ["Barbara", 11],
  ["David", 12],
  ["Alex", 11]
];
```

Now, each individual student's information is more organized as *indexed* data. For each array in `students`, the student's name is at index position `0` and the student's age is at index position `1`.

To access the *third student's age*, use the following syntax:

```
var thirdStudent = students[2][1];
```

`students[2]` will access the *third array* in `students`, and the additional bracketed index (`[1]`), accesses the *second element* in the *third array*.

```
var students = [
  ["Mary", 10],
  ["Barbara", 11],
  ["David", 12],
  ["Alex", 11]
];
```

Check your Understanding

1

```
var students = ["Mary", "Barbara", "David", "Alex"];
```

How would you access the last element in the following `students` array?

2

The following statement:

```
student[0];
```

will access the *first* element in the array.

3

```
var students = [ ["Mary", 10], ["Barbara", 11], ["David", 12], [
"Alex", 11] ];
```

How would you access the *second* student's name in the following `students` array?

Retake Quiz

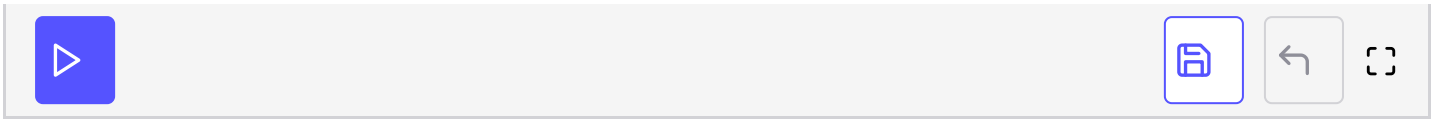
Manipulating arrays

There are several functions (called array **methods**) you can use to manipulate arrays in various ways.

Converting to a String

The `Array.toString()` method will take an array and convert it into a string, with items separated by commas.

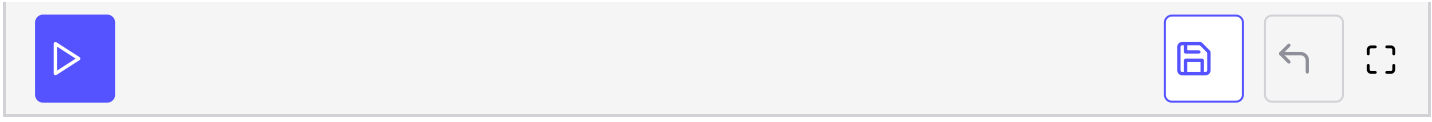
```
var studentString = students.toString();
console.log(studentString);
```



Array Size

`Array.length` will return the number of items currently in the array. Take notice that we do not use the parentheses operator as `length` is an array **property**, not a function.

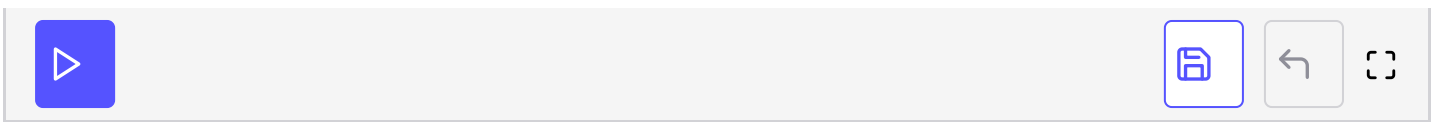
```
var classSize = students.length;
console.log(classSize);
```



Adding / removing items

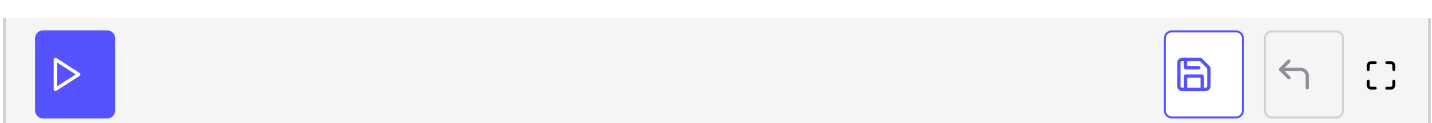
Use the `Array.push()` method to **add** items to *the end* of an array.

```
students.push("Sallie");
console.log(students[4]);
```



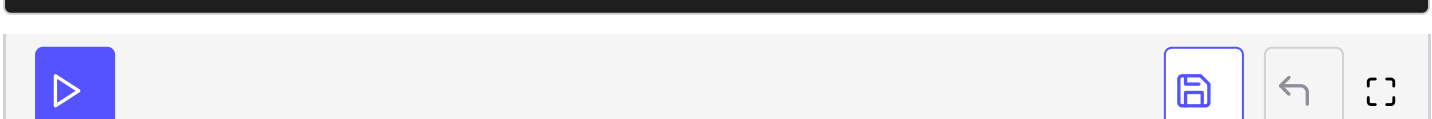
Use `Array.pop()` to **remove** the *last item* in the array. The `Array.pop()` method returns the removed item.

```
var removedStudent = students.pop();
console.log(students.toString());
console.log(removedStudent);
```



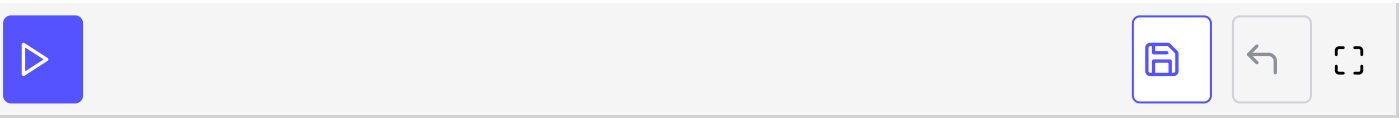
Use `Array.shift()` to **remove** the *first item* in the array. `Array.shift()` returns the value that was removed as well.

```
var removedStudent = students.shift();
console.log(students.toString());
console.log(removedStudent);
```



`Array.unshift()` **adds** an item to *the beginning* of an array. `Array.unshift()` returns the new size of the array.

```
var newSize = students.unshift("Joey");
console.log(newSize);
console.log(students.toString());
```

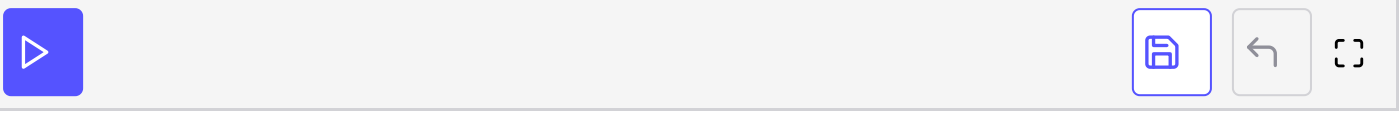


Iterating Over Arrays

There are some tasks that involve looking at all the values in an array.

Continuing with our example, say we want to create a function that prints out the name of every student in the class. If we know the size of the array, we could just access each value by its index:

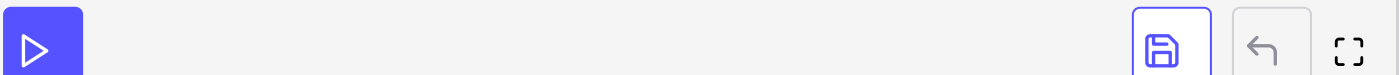
```
var printStudents = function(students) {
  console.log(students[0]);
  console.log(students[1]);
  console.log(students[2]);
  console.log(students[3]);
}

printStudents(students);
```



However, what if we add an item to the array using `Array.push()`? This code will break in any case that the `students` array does not have exactly four items.

A better solution would be to **iterate** over all the values of the array using a `for` loop:

```
var printStudents = function(students){
  for(var i = 0; i < students.length; i++) {
   console.log(students[i]);
  }
}

printStudents(students);
```



The initial variable, `i`, is set to 0 so that the accessing of array items *starts at the beginning*. The last index position of the array is always one less than the total number of items in the array, so we set the loop condition to be `i < students.length`. The loop is incremented by 1 to access every item in the array.

Now try adding and/or removing items from `students`, then call `printStudents()` again. You should notice that using a `for` loop to iterate over the array's contents means the function will work for an array of any size.

Exercise

Write a function (named `arraySum`) that takes an array as an argument and returns *the sum of all the items*. Assume your array's values are all **Numbers**.

```
arraySum([4, 5, 6, 7]); /* returns 22 */
arraySum([-6, 10, 0, 4]); /* returns 8 */
```

```
var arraySum = function(array){
  var sum = 0;
  /* write your code here */
  return sum;
}
```

Iterating over nested arrays

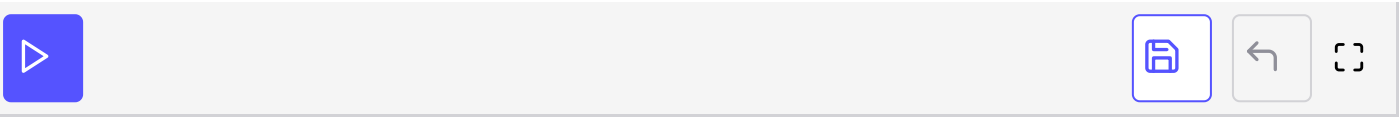
Let's take a look again at the `students` array with array items:

```
var students = [
  ["Mary", 10],
  ["Barbara", 11],
  ["David", 12],
  ["Alex", 11]
];
```

How would we iterate over the contents of this array? We could iterate over the `students` array using a `for` loop:

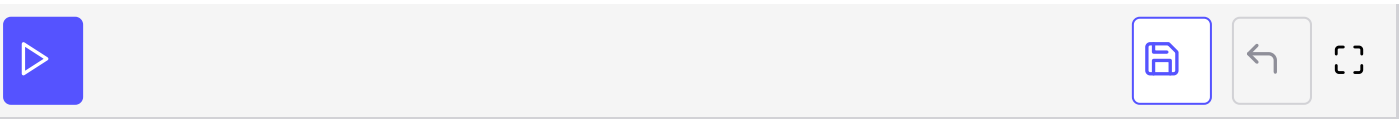
```
var printStudents = function(students){
  for(var i = 0; i < students.length; i++){
    console.log("Student " + i + ":");
```

```
    console.log(students[i][0]);
    console.log(students[i][1]);


  }
}
```



While this solution works fine for the above `students` array, what if we decide to add an additional piece of information to students, such as their gender?

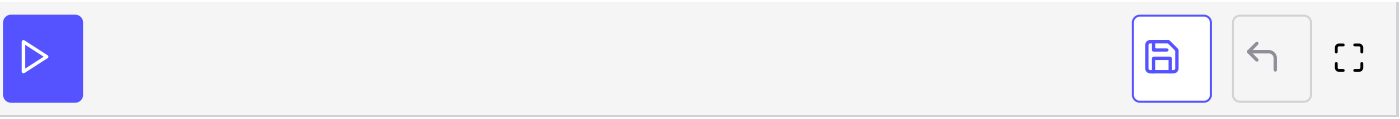
```
var students = [
  ["Mary", "Female", 10],
  ["Barbara", "Female", 11],
  ["David", "Male", 12],
  ["Alex", "Female", 11]
];

printStudents(students);
```



Now the `printStudents()` function breaks because it is hard-coded to print sub-arrays with only two items. Ideally, our code should be able to handle a sub-array of any size, and even sub-arrays of different sizes.

A better way to write the `printStudents()` function would be to use **another `for` loop** within the first `for` loop to *iterate* over the *sub-arrays*.

```
var printStudents = function(students){
  //iterate over the students array
  for(var i = 0; i < students.length; i++){
    //print student number
    console.log("Student " + i + ": ");

    //iterate over each sub-array
    for(var j = 0; j < students[i].length; j++){
      //print sub-array contents
      console.log(students[i][j]);
    }
  }
}

printStudents(students);
```



For the *nested* `for` loop, we initialize a different variable name, `j`, since `i` is

already taken by the first loop.

The second `for` loop will print **all** the contents of each sub-array since it uses the sub-array's `length` property to determine how many times to execute the loop.

Exercise

Given an array of arrays, each of which contains a set of *numbers*, write a function that returns an array where each item is the **sum** of all the items in the sub-array. For example,

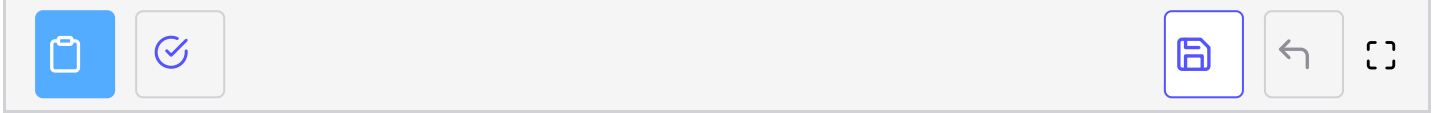
```
var numbers = [
  [1, 2, 3, 4],
  [5, 6, 7],
  [8, 9, 10, 11, 12]
];

arraySum(numbers);
```

Would return the following array:

```
[10, 18, 42]
```

A helpful hint: **nested `for` loops** are your friend :).

```
var arraySum = function(numbers) {
  var sums = [];
  //write your code here
  return sums;
}
```



Now that you have learned about arrays, let's learn a structured approach in programming using objects in the next lesson.