**Day 4: Array Buffer and Binary Array**

In web-development, binary data is generally used while working with files ( for instance, creating, uploading, downloading). [JavaScript](https://www.w3docs.com/learn-javascript.html) allows doing it all.

Multiple classes exist, among them are:

* **ArrayBuffer**
* **ArrayBuffer, Uint8Array, DataView**
* **Uint8Array**
* **Blob, and so on.**

In JavaScript, binary data is performed in a non-standard way. But, it becomes simple after getting into some details.

**ArrayBuffer**

ArrayBuffer is known as the basic binary object. It is a reference to a fixed-length contiguous memory area.

ArrayBuffer can be generated in the following way:

let buffer = new ArrayBuffer(16); // create a buffer of length 16

console.log(buffer.byteLength); // 16

Run >Reset

A contiguous memory area of 16 bytes and pre-filled with zeros by this. Be careful not to confuse ArrayBuffer with [Array](https://www.w3docs.com/learn-javascript/javascript-array.html). they don’t have anything in common.

The most specific things about ArrayBuffer is that:

* Its length is fixed, it can’t be increased or decreased.
* Exactly that much memory is taken by it in the memory.
* For accessing individual bytes, another “view” object, not buffer, is needed.

For manipulating an ArrayBuffer, a “view” object is needed.

On its own, a view object contains nothing. It gives an interpretation of the bytes, stored in the ArrayBuffer.

For example,

* **Uint8Array** treats every byte inside ArrayBuffer as a particular number with possible values from 0 to 255 ( one byte is equal to 8 bites). A value like this is known as “8-bit unsigned integer”.
* **Uint16Array** treats any two bytes like an integer, with possible values from 0 to 65535 (“16-bit unsigned integer”).
* **Uint32Array** treats any four bytes like an integer with possible values from 0 to 4294967295 (“32-bit unsigned integer”).
* **Float64Array** treats any eight bytes like a floating-point number with possible values from 5.0x10-324 to 1.8x10308.

ArrayBuffer is the raw binary data. It’s the basic object, the root of everything.

For writing it or iterating over it, and for any other operation, a view should be used like this:

console.log(view.length); // 4, it stores so many integers

console.log(view.byteLength); // 16, size in bytes

// let's write the value

view[0] = 123456;

// iterate over the values

for (let num of view) {

console.log(num); // 123456, then 0, 0, 0 (4 values total)

}

Run >Reset

**TypedArray**

TypedArray is the common term for all the views (Uint8Array, Uint32Array, and more). The same methods and properties are used by them.

There is no constructor TypedArray. It’s just an “umbrella” term for representing one of the views over ArrayBuffer. For example, new new TypedArray means one of the new Int8Array, new Uint8Array, and more.

The Typed array is similar to the regular arrays: they include iterable and indexes.

Five types of arguments exist:

new TypedArray(buffer, [byteOffset], [length]);

new TypedArray(object);

new TypedArray(typedArray);

new TypedArray(length);

new TypedArray();

A TypedArray can be created directly, without referring to ArrayBuffer. But no view can exist without an underlying ArrayBuffer. So, it’s generated automatically in any case, except when it’s provided.

The properties for accessing ArrayBuffer are the following:

* **arr.buffer:** it references the ArrayBuffer.
* **arr.byteLength:** it’s the length of the ArrayBuffer.

So, it can be moved from one view to another like this:

let arr8 = new Uint8Array([0, 1, 2, 3]);

// another view at the same data

let arr16 = new Uint16Array(arr8.buffer);

The list of the typed arrays is demonstrated below:

* **Uint8Array, Uint16Array, Uint32Array:** these are used for integer numbers of 8,16, and 32 bits.
* **Uint8ClampedArray:** it is used for 8-bit integers.
* **Int8Array, Int16Array, Int32Array** are used for signed integer numbers and can be negative.
* **Float32Array, Float64Array** are applied for the signed floating-point numbers of 32 and 64 bits.

Also, take into account that no single-value type (int or int8) exists in JavaScript.

**Out-of-bounds Behavior**

If you try to write an out-of-bounds value into a typed array, no error will occur, but the bytes will be cut off.

For example, let’s attempt to store 256 into Uint8Array. Within the binary form, 256 is 9 bits (100000000). But, as it was stated above, Uint8Array can contain 8 bits per value.

The demo will look like this:

let uint8array = new Uint8Array(16);

let number = 256;

console.log(number.toString(2)); // 100000000, binary representation

uint8array[0] = 256;

uint8array[1] = 257;

console.log(uint8array[0]); // 0

console.log(uint8array[1]); // 1

Run >Reset

In this point,Uint8ClampedArray is unique: it behaves differently. It can save 255 for each number greater than 255 and 0 for negative numbers.

**The Methods of TypedArray**

Regular [array methods](https://www.w3docs.com/learn-javascript/array-methods.html) are used by ArrayBuffer. But, there are remarkable exceptions.

For example, [slice](https://www.w3docs.com/learn-javascript/array-methods.html#slice), [map](https://www.w3docs.com/learn-javascript/array-methods.html#map), [find](https://www.w3docs.com/learn-javascript/array-methods.html#find-and-findindex), and other methods can be iterated. But, there are things that can not be done. For example, there is no [concat](https://www.w3docs.com/learn-javascript/array-methods.html#concat) method and no splice.

Instead, there are to additional methods such as:

* **arr.set(fromArr, [offset]):** it copies all the elements from **fromArr** to the arr, beginning at the **offset** position.
* **arr.subarray([begin, end]):** it generates a new same-type view from begin to end.

This method is like the slice method but it doesn’t copy anything. Wit it, you just can create a new view, operating on a particular piece of data.

**DataView**

There is a flexible untyped view over the ArrayBuffer, known as DataView. With it, you can access the data on any offset in any format. The constructor indicates what the format is for typed arrays. The whole array should be uniform, and arr[i] is the i-th number. The data can be accessed using **.getUint8(i)** or **.getUint16(i)** method with DataView.

The syntax of DataView is the following:

new DataView(buffer, [byteOffset], [byteLength]);

Let’s try to extract numbers in distinct formats from the same buffer, like this:

// binary array of 4 bytes, all have a maximal value of 255

let bufferArr = new Uint8Array([255, 255, 255, 255]).buffer;

let dataView = new DataView(bufferArr);

// get an 8-bit number with an offset of 0

console.log(dataView.getUint8(0)); // 255

// now get a 16-bit number with an offset of 0, it consists of 2 bytes, together iterpreted as 65535

console.log(dataView.getUint16(0)); // 65535 (biggest 16-bit unsigned int)

// get 32-bit number with an offset of 0

console.log(dataView.getUint32(0)); // 4294967295 ,biggest unsigned int 32-bit

Run >Reset

So, the DataView is great for storing mixed-format data in the same buffer.

**Summary**

ArrayBuffer is the basic binary object. It is considered as a reference to a fixed-length contiguous memory area. A view is necessary for almost all the operations on ArrayBuffer.

It can be either a TypedArray or a DataView.

In the majority of cases, you can create and operate on the typed arrays directly, leaving ArrayBuffer under cover. It can be accessed as a .buffer , making another view, if it’s necessary.