Improved Array Capabilities

The array is a foundational JavaScript object. But while other aspects of JavaScript have evolved over time, arrays remained the same until ECMAScript 5 introduced several methods to make them easier to use. ECMAScript 6 continues to improve arrays by adding a lot more functionality, like new creation methods, several useful convenience methods, and the ability to make typed arrays.

Creating Arrays

Prior to ECMAScript 6, there were two primary ways to create arrays: the `Array` constructor and array literal syntax. Both approaches require listing array items individually and are otherwise fairly limited. Options for converting an array-like object (that is, an object with numeric indices and a `length` property) into an array were also limited and often required extra code. To make JavaScript arrays easier to create, ECMAScript 6 adds the `Array.of()` and `Array.from()` methods.

The Array.of() Method

One reason ECMAScript 6 adds new creation methods to JavaScript is to help developers avoid a quirk of creating arrays with the `Array` constructor. The `new Array()` constructor actually behaves differently based on the type and number of arguments passed to it. For example:

```
let items = new Array(2);
console.log(items.length);          // 2
console.log(items[0]);              // undefined
console.log(items[1]);              // undefined

items = new Array("2");
console.log(items.length);          // 1
console.log(items[0]);              // "2"

items = new Array(1, 2);
console.log(items.length);          // 2
console.log(items[0]);              // 1
console.log(items[1]);              // 2

items = new Array(3, "2");
console.log(items.length);          // 2
console.log(items[0]);              // 3
console.log(items[1]);              // "2"
```

When the `Array` constructor is passed a single numeric value, the `length` property of the array is set to that value. If a single non-numeric value is passed, then that value becomes the one and only item in the array. If multiple values are passed (numeric or not), then those values become items in the array. This behavior is both confusing and risky, as you may not always be aware of the type of data being passed.

ECMAScript 6 introduces `Array.of()` to solve this problem. The `Array.of()` method works similarly to the `Array` constructor but has no special case regarding a single numeric value. The `Array.of()` method always creates an array containing its arguments regardless of the number of arguments or the argument types. Here are some examples that use the `Array.of()` method:

```
let items = Array.of(1, 2);
console.log(items.length);          // 2
console.log(items[0]);              // 1
console.log(items[1]);              // 2

items = Array.of(2);
console.log(items.length);          // 1
console.log(items[0]);              // 2

items = Array.of("2");
console.log(items.length);          // 1
console.log(items[0]);              // "2"
```

To create an array with the `Array.of()` method, just pass it the values you want in your array. The first example here creates an array containing two numbers, the second array contains one number, and the last array contains one string. This is similar to using an array literal, and you can use an array literal instead of `Array.of()` for native arrays most of the time. But if you ever need to pass the `Array` constructor into a function, then you might want to pass `Array.of()` instead to ensure consistent behavior. For example:

```
function createArray(arrayCreator, value) {
    return arrayCreator(value);
}

let items = createArray(Array.of, value);
```

In this code, the `createArray()` function accepts an array creator function and a value to insert into the array. You can pass `Array.of()` as the first argument to `createArray()` to create a new array. It would be dangerous to pass `Array` directly if you cannot guarantee that `value` won't be a number.

I> The `Array.of()` method does not use the `Symbol.species` property (discussed in Chapter 9) to determine the type of return value. Instead, it uses the current constructor (`this` inside the `of()` method) to determine the correct data type to return.

The Array.from() Method

Converting non-array objects into actual arrays has always been cumbersome in JavaScript. For instance, if you have an `arguments` object (which is array-like) and want to use it like an array, then you'd need to convert it first. To convert an array-like object to an array in ECMAScript 5, you'd write a function like the one in this example:

```
function makeArray(arrayLike) {
    var result = [];
    for (var i = 0, len = arrayLike.length; i < len; i++) {
        result.push(arrayLike[i]);
    }

    return result;
}

function doSomething() {
    var args = makeArray(arguments);

    // use args
}
```

This approach manually creates a `result` array and copies each item from `arguments` into the new array. That works but takes a decent amount of code to perform a relatively simple operation. Eventually, developers discovered they could reduce the amount of code by calling the native `slice()` method for arrays on array-like objects, like this:

```
function makeArray(arrayLike) {
    return Array.prototype.slice.call(arrayLike);
}

function doSomething() {
    var args = makeArray(arguments);

    // use args
}
```

This code is functionally equivalent to the previous example, and it works because it sets the `this` value for `slice()` to the array-like object. Since `slice()` needs only numeric indices and a `length` property to function correctly, any array-like object will work.

Even though this technique requires less typing, calling `Array.prototype.slice.call(arrayLike)` doesn't obviously translate to, "Convert `arrayLike` to an array." Fortunately, ECMAScript 6 added the `Array.from()` method as an obvious, yet clean, way to convert objects into arrays.

Given either an iterable or an array-like object as the first argument, the `Array.from()` method returns an array. Here's a simple example:

```
function doSomething() {
    var args = Array.from(arguments);

    // use args
}
```

The `Array.from()` call creates a new array based on the items in `arguments`. So `args` is an instance of `Array` that contains the same values in the same positions as `arguments`.

I> The `Array.from()` method also uses `this` to determine the type of array to return.

Mapping Conversion

If you want to take array conversion a step further, you can provide `Array.from()` with a mapping function as a second argument. That function operates on each value from the array-like object and converts it to some final form before storing the result at the appropriate index in the final array. For example:

```
function translate() {
    return Array.from(arguments, (value) => value + 1);
}

let numbers = translate(1, 2, 3);

console.log(numbers);               // 2,3,4
```

Here, `Array.from()` is passed `(value) => value + 1` as a mapping function, so it adds 1 to each item in the array before storing the item. If the mapping function is on an object, you can also optionally pass a third argument to `Array.from()` that represents the `this` value for the mapping function:

```
let helper = {
    diff: 1,

    add(value) {
        return value + this.diff;
    }
};

function translate() {
    return Array.from(arguments, helper.add, helper);
}

let numbers = translate(1, 2, 3);

console.log(numbers);               // 2,3,4
```

This example passes `helper.add()` as the mapping function for the conversion. Since `helper.add()` uses the `this.diff` property, you need to provide the third argument to `Array.from()` specifying the value of `this`. Thanks to the third argument, `Array.from()` can easily convert data without calling `bind()` or specifying the `this` value in some other way.

Use on Iterables

The `Array.from()` method works on both array-like objects and iterables. That means the method can convert any object with a `Symbol.iterator` property into an array. For example:

```
let numbers = {
    *[Symbol.iterator]() {
        yield 1;
        yield 2;
        yield 3;
    }
};

let numbers2 = Array.from(numbers, (value) => value + 1);

console.log(numbers2);              // 2,3,4
```

Since the `numbers` object is an iterable, you can pass `numbers` directly to `Array.from()` to convert its values into an array. The mapping function adds one to each number so the resulting array contains 2, 3, and 4 instead of 1, 2, and 3.

I> If an object is both array-like and iterable, then the iterator is used by `Array.from()` to determine the values to convert.

New Methods on All Arrays

Continuing the trend from ECMAScript 5, ECMAScript 6 adds several new methods to arrays. The `find()` and `findIndex()` methods are meant to aid developers using arrays with any values, while `fill()` and `copyWithin()` are inspired by use cases for *typed arrays*, a form of array introduced in ECMAScript 6 that uses only numbers.

The find() and findIndex() Methods

Prior to ECMAScript 5, searching through arrays was cumbersome because there were no built-in methods for doing so. ECMAScript 5 added the `indexOf()` and `lastIndexOf()` methods, finally allowing developers to search for specific values inside an array. These two methods were a big improvement, yet they were still fairly limited because you could only search for one value at a time. For example, if you wanted to find the first even number in a series of numbers, you'd need to write your own code to do so. ECMAScript 6 solved that problem by introducing the `find()` and `findIndex()` methods.

Both `find()` and `findIndex()` accept two arguments: a callback function and an optional value to use for `this` inside the callback function. The callback function is passed an array element, the index of that element in the array, and the array itself--the same arguments passed to methods like `map()` and `forEach()`. The callback should return `true` if the given value matches some criteria you define. Both `find()` and `findIndex()` also stop searching the array the first time the callback function returns `true`.

The only difference between these methods is that `find()` returns the value whereas `findIndex()` returns the index at which the value was found. Here's an example to demonstrate:

```
let numbers = [25, 30, 35, 40, 45];

console.log(numbers.find(n => n > 33));         // 35
console.log(numbers.findIndex(n => n > 33));    // 2
```

This code calls `find()` and `findIndex()` to locate the first value in the `numbers` array that is greater than 33. The call to `find()` returns 35 and `findIndex()` returns 2, the location of 35 in the `numbers` array.

Both `find()` and `findIndex()` are useful to find an array element that matches a condition rather than a value. If you only want to find a value, then `indexOf()` and `lastIndexOf()` are better choices.

The fill() Method

The `fill()` method fills one or more array elements with a specific value. When passed a value, `fill()` overwrites all of the values in an array with that value. For example:

```
let numbers = [1, 2, 3, 4];

numbers.fill(1);

console.log(numbers.toString());    // 1,1,1,1
```

Here, the call to `numbers.fill(1)` changes all values in `numbers` to `1`. If you only want to change some of the elements, rather than all of them, you can optionally include a start index and an exclusive end index, like this:

```
let numbers = [1, 2, 3, 4];

numbers.fill(1, 2);

console.log(numbers.toString());    // 1,2,1,1

numbers.fill(0, 1, 3);

console.log(numbers.toString());    // 1,0,0,1
```

In the `numbers.fill(1,2)` call, the `2` indicates to start filling elements at index 2. The exclusive end index isn't specified with a third argument, so `numbers.length` is used as the end index, meaning the last two elements in `numbers` are filled with `1`. The `numbers.fill(0, 1, 3)` operation fills array elements at indices 1 and 2 with `0`. Calling `fill()` with the second and third arguments allows you to fill multiple array elements at once without overwriting the entire array.

I> If either the start or end index are negative, then those values are added to the array's length to determine the final location. For instance, a start location of `-1` gives `array.length - 1` as the index, where `array` is the array on which `fill()` is called.

The copyWithin() Method

The `copyWithin()` method is similar to `fill()` in that it changes multiple array elements at the same time. However, instead of specifying a single value to assign to array elements, `copyWithin()` lets you copy array element values from the array itself. To accomplish that, you need to pass two arguments to the `copyWithin()` method: the index where the method should start filling values and the index where the values to be copied begin.

For instance, to copy the values from the first two elements in an array to the last two items in the array, you can do the following:

```
let numbers = [1, 2, 3, 4];

// paste values into array starting at index 2
// copy values from array starting at index 0
numbers.copyWithin(2, 0);

console.log(numbers.toString());    // 1,2,1,2
```

This code pastes values into `numbers` beginning from index 2, so both indices 2 and 3 will be overwritten. Passing `0` as the second argument to `copyWithin()` indicates to start copying values from index 0 and continue until there are no more elements to copy into.

By default, `copyWithin()` always copies values up to the end of the array, but you can provide an optional third argument to limit how many elements will be overwritten. That third argument is an exclusive end index at which copying of values stops. Here's an example:

```
let numbers = [1, 2, 3, 4];

// paste values into array starting at index 2
// copy values from array starting at index 0
// stop copying values when you hit index 1
numbers.copyWithin(2, 0, 1);

console.log(numbers.toString());    // 1,2,1,4
```

In this example, only the value in index 0 is copied because the optional end index is set to `1`. The last element in the array remains unchanged.

I> As with the `fill()` method, if you pass a negative number for any argument to the `copyWithin()` method, the array's length is automatically added to that value to determine the index to use.

The use cases for `fill()` and `copyWithin()` may not be obvious to you at this point. That's because these methods originated on typed arrays and were added to regular arrays for consistency. As you'll learn in the next section, however, if you use typed arrays for manipulating the bits of a number, these methods become a lot more useful.

Typed Arrays

Typed arrays are special-purpose arrays designed to work with numeric types (not all types, as the name might imply). The origin of typed arrays can be traced to WebGL, a port of Open GL ES 2.0 designed for use in web pages with the `<canvas>` element. Typed arrays were created as part of the port to provide fast bitwise arithmetic in JavaScript.

Arithmetic on native JavaScript numbers was too slow for WebGL because the numbers were stored in a 64-bit floating-point format and converted to 32-bit integers as needed. Typed arrays were introduced to circumvent this limitation and provide better performance for arithmetic operations. The concept is that any single number can be treated like an array of bits and thus can use the familiar methods available on JavaScript arrays.

ECMAScript 6 adopted typed arrays as a formal part of the language to ensure better compatibility across JavaScript engines and interoperability with JavaScript arrays. While the ECMAScript 6 version of typed arrays is not exactly the same as the WebGL version, there are enough similarities to make the ECMAScript 6 version an evolution of the WebGL version rather than a different approach.

Numeric Data Types

JavaScript numbers are stored in IEEE 754 format, which uses 64 bits to store a floating-point representation of the number. This format represents both integers and floats in JavaScript, with conversion between the two formats happening frequently as numbers change. Typed arrays allow the storage and manipulation of eight different numeric types:

1. Signed 8-bit integer (int8)
2. Unsigned 8-bit integer (uint8)
3. Signed 16-bit integer (int16)
4. Unsigned 16-bit integer (uint16)
5. Signed 32-bit integer (int32)
6. Unsigned 32-bit integer (uint32)
7. 32-bit float (float32)
8. 64-bit float (float64)

If you represent a number that fits in an int8 as a normal JavaScript number, you'll waste 56 bits. Those bits might better be used to store additional int8 values or any other number that requires less than 56 bits. Using bits more efficiently is one of the use cases typed arrays address.

All of the operations and objects related to typed arrays are centered around these eight data types. In order to use them, though, you'll need to create an array buffer to store the data.

I> In this book, I will refer to these types by the abbreviations I showed in parentheses. Those abbreviations don't appear in actual JavaScript code; they're just a shorthand for the much longer descriptions.

Array Buffers

The foundation for all typed arrays is an *array buffer*, which is a memory location that can contain a specified number of bytes. Creating an array buffer is akin to calling `malloc()` in C to allocate memory without specifying what the memory block contains. You can create an array buffer by using the `ArrayBuffer` constructor as follows:

```
let buffer = new ArrayBuffer(10);   // allocate 10 bytes
```

Just pass the number of bytes the array buffer should contain when you call the constructor. This `let` statement creates an array buffer 10 bytes long. Once an array buffer is created, you can retrieve the number of bytes in it by checking the `byteLength` property:

```
let buffer = new ArrayBuffer(10);   // allocate 10 bytes
console.log(buffer.byteLength);     // 10
```

You can also use the `slice()` method to create a new array buffer that contains part of an existing array buffer. The `slice()` method works like the `slice()` method on arrays: you pass it the start index and end index as arguments, and it returns a new `ArrayBuffer` instance comprised of those elements from the original. For example:

```
let buffer = new ArrayBuffer(10);   // allocate 10 bytes


let buffer2 = buffer.slice(4, 6);
console.log(buffer2.byteLength);    // 2
```

In this code, `buffer2` is created by extracting the bytes at indices 4 and 5. Just like when you call the array version of this method, the second argument to `slice()` is exclusive.

Of course, creating a storage location isn't very helpful without being able to write data into that location. To do so, you'll need to create a view.

I> An array buffer always represents the exact number of bytes specified when it was created. You can change the data contained within an array buffer, but never the size of the array buffer itself.

Manipulating Array Buffers with Views

Array buffers represent memory locations, and *views* are the interfaces you'll use to manipulate that memory. A view operates on an array buffer or a subset of an array buffer's bytes, reading and writing data in one of the numeric data types. The `DataView` type is a generic view on an array buffer that allows you to operate on all eight numeric data types.

To use a `DataView`, first create an instance of `ArrayBuffer` and use it to create a new `DataView`. Here's an example:

```
let buffer = new ArrayBuffer(10),
    view = new DataView(buffer);
```

The `view` object in this example has access to all 10 bytes in `buffer`. You can also create a view over just a portion of a buffer. Just provide a byte offset and, optionally, the number of bytes to include from that offset. When a number of bytes isn't included, the`DataView` will go from the offset to the end of the buffer by default. For example:

```
let buffer = new ArrayBuffer(10),
    view = new DataView(buffer, 5, 2);      // cover bytes 5 and 6
```

Here, `view` operates only on the bytes at indices 5 and 6. This approach allows you to create several views over the same array buffer, which can be useful if you want to use a single memory location for an entire application rather than dynamically allocating space as needed.

Retrieving View Information

You can retrieve information about a view by fetching the following read-only properties:

- `buffer` - The array buffer that the view is tied to
- `byteOffset` - The second argument to the `DataView` constructor, if provided (0 by default)
- `byteLength` - The third argument to the `DataView` constructor, if provided (the buffer's `byteLength` by default)

Using these properties, you can inspect exactly where a view is operating, like this:

```
let buffer = new ArrayBuffer(10),
    view1 = new DataView(buffer),           // cover all bytes
    view2 = new DataView(buffer, 5, 2);     // cover bytes 5 and 6

console.log(view1.buffer === buffer);       // true
console.log(view2.buffer === buffer);       // true
console.log(view1.byteOffset);              // 0
console.log(view2.byteOffset);              // 5
console.log(view1.byteLength);              // 10
console.log(view2.byteLength);              // 2
```

This code creates `view1`, a view over the entire array buffer, and `view2`, which operates on a small section of the array buffer. These views have equivalent `buffer` properties because both work on the same array buffer. The `byteOffset` and `byteLength` are different for each view, however. They reflect the portion of the array buffer where each view operates.

Of course, reading information about memory isn't very useful on its own. You need to write data into and read data out of that memory to get any benefit.

Reading and Writing Data

For each of JavaScript's eight numeric data types, the `DataView` prototype has a method to write data and a method to read data from an array buffer. The method names all begin with either "set" or "get" and are followed by the data type abbreviation. For instance, here's a list of the read and write methods that can operate on int8 and uint8 values:

- `getInt8(byteOffset)` - Read an int8 starting at `byteOffset`
- `setInt8(byteOffset, value)` - Write an int8 starting at `byteOffset`
- `getUint8(byteOffset)` - Read an uint8 starting at `byteOffset`

- `setUint8(byteOffset, value)` - Write an uint8 starting at `byteOffset`

The "get" methods accept a single argument: the byte offset to read from. The "set" methods accept two arguments: the byte offset to write at and the value to write.

Though I've only shown the methods you can use with 8-bit values, the same methods exist for operating on 16- and 32-bit values. Just replace the `8` in each name with `16` or `32`. Alongside all those integer methods, `DataView` also has the following read and write methods for floating point numbers:

- `getFloat32(byteOffset, littleEndian)` - Read a float32 starting at `byteOffset`
- `setFloat32(byteOffset, value, littleEndian)` - Write a float32 starting at `byteOffset`
- `getFloat64(byteOffset, littleEndian)` - Read a float64 starting at `byteOffset`
- `setFloat64(byteOffset, value, littleEndian)` - Write a float64 starting at `byteOffset`

The float-related methods are only different in that they accept an additional optional boolean indicating whether the value should be read or written as little-endian. (*Little-endian* means the least significant byte is at byte 0, instead of in the last byte.)

To see a "set" and a "get" method in action, consider the following example:

```
let buffer = new ArrayBuffer(2),
    view = new DataView(buffer);

view.setInt8(0, 5);
view.setInt8(1, -1);

console.log(view.getInt8(0));       // 5
console.log(view.getInt8(1));       // -1
```

This code uses a two-byte array buffer to store two int8 values. The first value is set at offset 0 and the second is at offset 1, reflecting that each value spans a full byte (8 bits). Those values are later retrieved from their positions with the `getInt8()` method. While this example uses int8 values, you can use any of the eight numeric types with their corresponding methods.

Views are interesting because they allow you to read and write in any format at any point in time, regardless of how data was previously stored. For instance, writing two int8 values and reading the buffer with an int16 method works just fine, as in this example:

```
let buffer = new ArrayBuffer(2),
    view = new DataView(buffer);

view.setInt8(0, 5);
view.setInt8(1, -1);

console.log(view.getInt16(0));      // 1535
console.log(view.getInt8(0));       // 5
console.log(view.getInt8(1));       // -1
```

The call to `view.getInt16(0)` reads all bytes in the view and interprets those bytes as the number 1535. To understand why this happens, take a look at Figure 10-1, which shows what each `setInt8()` line does to the array buffer.

```
new ArrayBuffer(2)      0000000000000000
view.setInt8(0, 5);     0000010100000000
view.setInt8(1, -1);    0000010111111111
```

The array buffer starts with 16 bits that are all zero. Writing `5` to the first byte with `setInt8()` introduces a couple of 1s (in 8-bit representation, 5 is 00000101). Writing -1 to the second byte sets all bits in that byte to 1, which is the two's complement representation of -1. After the second `setInt8()` call, the array buffer contains 16 bits, and `getInt16()` reads those bits as a single 16-bit integer, which is 1535 in decimal.

The `DataView` object is perfect for use cases that mix different data types in this way. However, if you're only using one specific data type, then the type-specific views are a better choice.

Typed Arrays Are Views

ECMAScript 6 typed arrays are actually type-specific views for array buffers. Instead of using a generic `DataView` object to operate on an array buffer, you can use objects that enforce specific data types. There are eight type-specific views corresponding to the eight numeric data types, plus an additional option for `uint8` values.

Table 10-1 shows an abbreviated version of the complete list of type-specific views from section 22.2 of the ECMAScript 6 specification.

Constructor Name	Element Size (in bytes)	Description	Equivalent C Type
`Int8Array`	1	8-bit two's complement signed integer	`signed char`
`Uint8Array`	1	8-bit unsigned integer	`unsigned char`
`Uint8ClampedArray`	1	8-bit unsigned integer (clamped conversion)	`unsigned char`
`Int16Array`	2	16-bit two's complement signed integer	`short`
`Uint16Array`	2	16-bit unsigned integer	`unsigned short`
`Int32Array`	4	32-bit two's complement signed integer	`int`
`Uint32Array`	4	32-bit unsigned integer	`int`
`Float32Array`	4	32-bit IEEE floating point	`float`

Constructor Name	Element Size (in bytes)	Description	Equivalent C Type
`Float64Array`	8	64-bit IEEE floating point	`double`

The left column lists the typed array constructors, and the other columns describe the data each typed array can contain. A `Uint8ClampedArray` is the same as a `Uint8Array` unless values in the array buffer are less than 0 or greater than 255. A `Uint8ClampedArray` converts values lower than 0 to 0 (-1 becomes 0, for instance) and converts values higher than 255 to 255 (so 300 becomes 255).

Typed array operations only work on a particular type of data. For example, all operations on `Int8Array` use `int8` values. The size of an element in a typed array also depends on the type of array. While an element in an `Int8Array` is a single byte long, `Float64Array` uses eight bytes per element. Fortunately, the elements are accessed using numeric indices just like regular arrays, allowing you to avoid the somewhat awkward calls to the "set" and "get" methods of `DataView`.

A> ### Element Size A> A> Each typed array is made up of a number of elements, and the element size is the number of bytes each element represents. This value is stored on a `BYTES_PER_ELEMENT` property on each constructor and each instance, so you can easily query the element size: A> A> `js A> console.log(UInt8Array.BYTES_PER_ELEMENT); // 1 A> console.log(UInt16Array.BYTES_PER_ELEMENT); // 2 A> A> let ints = new Int8Array(5); A> console.log(ints.BYTES_PER_ELEMENT); // 1 A>`

Creating Type-Specific Views

Typed array constructors accept multiple types of arguments, so there are a few ways to create typed arrays. First, you can create a new typed array by passing the same arguments `DataView` takes (an array buffer, an optional byte offset, and an optional byte length). For example:

```js
let buffer = new ArrayBuffer(10),
    view1 = new Int8Array(buffer),
    view2 = new Int8Array(buffer, 5, 2);

console.log(view1.buffer === buffer);       // true
console.log(view2.buffer === buffer);       // true
console.log(view1.byteOffset);              // 0
console.log(view2.byteOffset);              // 5
console.log(view1.byteLength);              // 10
console.log(view2.byteLength);              // 2
```

In this code, the two views are both `Int8Array` instances that use `buffer`. Both `view1` and `view2` have the same `buffer`, `byteOffset`, and `byteLength` properties that exist on `DataView` instances. It's easy to switch to using a typed array wherever you use a `DataView` so long as you only work with one numeric type.

The second way to create a typed array is to pass a single number to the constructor. That number represents the number of elements (not bytes) to allocate to the array. The constructor will create a new buffer with the correct number of bytes to represent that number of array elements, and you can access the number of elements in the array by using the `length` property. For example:

```
let ints = new Int16Array(2),
    floats = new Float32Array(5);

console.log(ints.byteLength);       // 4
console.log(ints.length);           // 2

console.log(floats.byteLength);     // 20
console.log(floats.length);         // 5
```

The `ints` array is created with space for two elements. Each 16-bit integer requires two bytes per value, so the array is allocated four bytes. The `floats` array is created to hold five elements, so the number of bytes required is 20 (four bytes per element). In both cases, a new buffer is created and can be accessed using the `buffer` property if necessary.

W> If no argument is passed to a typed array constructor, the constructor acts as if `0` was passed. This creates a typed array that cannot hold data because zero bytes are allocated to the buffer.

The third way to create a typed array is to pass an object as the only argument to the constructor. The object can be any of the following:

- **A Typed Array** - Each element is copied into a new element on the new typed array. For example, if you pass an int8 to the `Int16Array` constructor, the int8 values would be copied into an int16 array. The new typed array has a different array buffer than the one that was passed in.
- **An Iterable** - The object's iterator is called to retrieve the items to insert into the typed array. The constructor will throw an error if any elements are invalid for the view type.
- **An Array** - The elements of the array are copied into a new typed array. The constructor will throw an error if any elements are invalid for the type.
- **An Array-Like Object** - Behaves the same as an array.

In each of these cases, a new typed array is created with the data from the source object. This can be especially useful when you want to initialize a typed array with some values, like this:

```
let ints1 = new Int16Array([25, 50]),
    ints2 = new Int32Array(ints1);

console.log(ints1.buffer === ints2.buffer);     // false

console.log(ints1.byteLength);      // 4
console.log(ints1.length);          // 2
console.log(ints1[0]);              // 25
console.log(ints1[1]);              // 50

console.log(ints2.byteLength);      // 8
console.log(ints2.length);          // 2
console.log(ints2[0]);              // 25
console.log(ints2[1]);              // 50
```

This example creates an `Int16Array` and initializes it with an array of two values. Then, an `Int32Array` is created and passed the `Int16Array`. The values 25 and 50 are copied from `ints1` into `ints2` as the two typed arrays have completely separate buffers. The same numbers are represented in both typed arrays, but `ints2` has eight bytes to represent the data while `ints1` has only four.

Similarities Between Typed and Regular Arrays

Typed arrays and regular arrays are similar in several ways, and as you've already seen in this chapter, typed arrays can be used like regular arrays in many situations. For instance, you can check how many elements are in a typed array using the `length` property, and you can access a typed array's elements directly using numeric indices. For example:

```
let ints = new Int16Array([25, 50]);

console.log(ints.length);          // 2
console.log(ints[0]);              // 25
console.log(ints[1]);              // 50

ints[0] = 1;
ints[1] = 2;

console.log(ints[0]);              // 1
console.log(ints[1]);              // 2
```

In this code, a new `Int16Array` with two items is created. The items are read from and written to using their numeric indices, and those values are automatically stored and converted into int16 values as part of the operation. The similarities don't end there, though.

I> Unlike regular arrays, you cannot change the size of a typed array using the `length` property. The `length` property is not writable, so any attempt to change it is ignored in non-strict mode and throws an error in strict mode.

Common Methods

Typed arrays also include a large number of methods that are functionally equivalent to regular array methods. You can use the following array methods on typed arrays:

- `copyWithin()`
- `entries()`
- `fill()`
- `filter()`
- `find()`
- `findIndex()`
- `forEach()`
- `indexOf()`
- `join()`
- `keys()`
- `lastIndexOf()`

- `map()`
- `reduce()`
- `reduceRight()`
- `reverse()`
- `slice()`
- `some()`
- `sort()`
- `values()`

Keep in mind that while these methods act like their counterparts on `Array.prototype`, they are not exactly the same. The typed array methods have additional checks for numeric type safety and, when an array is returned, will return a typed array instead of a regular array (due to `Symbol.species`). Here's a simple example to demonstrate the difference:

```
let ints = new Int16Array([25, 50]),
    mapped = ints.map(v => v * 2);

console.log(mapped.length);        // 2
console.log(mapped[0]);            // 50
console.log(mapped[1]);            // 100

console.log(mapped instanceof Int16Array);  // true
```

This code uses the `map()` method to create a new array based on the values in `ints`. The mapping function doubles each value in the array and returns a new `Int16Array`.

The Same Iterators

Typed arrays have the same three iterators as regular arrays, too. Those are the `entries()` method, the `keys()` method, and the `values()` method. That means you can use the spread operator and `for-of` loops with typed arrays just like you would with regular arrays. For example:

```
let ints = new Int16Array([25, 50]),
    intsArray = [...ints];

console.log(intsArray instanceof Array);    // true
console.log(intsArray[0]);                  // 25
console.log(intsArray[1]);                  // 50
```

This code creates a new array called `intsArray` containing the same data as the typed array `ints`. As with other iterables, the spread operator makes converting typed arrays into regular arrays easy.

of() and from() Methods

Lastly, all typed arrays have static `of()` and `from()` methods that work like the `Array.of()` and `Array.from()` methods. The difference is that the methods on typed arrays return a typed array instead of a regular array. Here are some examples that use these methods to create typed arrays:

```
let ints = Int16Array.of(25, 50),
    floats = Float32Array.from([1.5, 2.5]);

console.log(ints instanceof Int16Array);        // true
console.log(floats instanceof Float32Array);    // true

console.log(ints.length);       // 2
console.log(ints[0]);           // 25
console.log(ints[1]);           // 50

console.log(floats.length);     // 2
console.log(floats[0]);         // 1.5
console.log(floats[1]);         // 2.5
```

The `of()` and `from()` methods in this example are used to create an `Int16Array` and a `Float32Array`, respectively. These methods ensure that typed arrays can be created just as easily as regular arrays.

Differences Between Typed and Regular Arrays

The most important difference between typed arrays and regular arrays is that typed arrays are not regular arrays. Typed arrays don't inherit from `Array` and `Array.isArray()` returns `false` when passed a typed array. For example:

```
let ints = new Int16Array([25, 50]);

console.log(ints instanceof Array);     // false
console.log(Array.isArray(ints));       // false
```

Since the `ints` variable is a typed array, it isn't an instance of `Array` and cannot otherwise be identified as an array. This distinction is important because while typed arrays and regular arrays are similar, there are many ways in which typed arrays behave differently.

Behavioral Differences

While regular arrays can grow and shrink as you interact with them, typed arrays always remain the same size. You cannot assign a value to a nonexistent numeric index in a typed array like you can with regular arrays, as typed arrays ignore the operation. Here's an example:

```
let ints = new Int16Array([25, 50]);

console.log(ints.length);          // 2
console.log(ints[0]);              // 25
console.log(ints[1]);              // 50

ints[2] = 5;
```

```
console.log(ints.length);           // 2
console.log(ints[2]);               // undefined
```

Despite assigning `5` to the numeric index `2` in this example, the `ints` array does not grow at all. The `length` remains the same and the value is thrown away.

Typed arrays also have checks to ensure that only valid data types are used. Zero is used in place of any invalid values. For example:

```
let ints = new Int16Array(["hi"]);

console.log(ints.length);        // 1
console.log(ints[0]);            // 0
```

This code attempts to use the string value `"hi"` in an `Int16Array`. Of course, strings are invalid data types in typed arrays, so the value is inserted as `0` instead. The `length` of the array is still one, and even though the `ints[0]` slot exists, it just contains `0`.

All methods that modify values in a typed array enforce the same restriction. For example, if the function passed to `map()` returns an invalid value for the typed array, then `0` is used instead:

```
let ints = new Int16Array([25, 50]),
    mapped = ints.map(v => "hi");

console.log(mapped.length);         // 2
console.log(mapped[0]);             // 0
console.log(mapped[1]);             // 0

console.log(mapped instanceof Int16Array);  // true
console.log(mapped instanceof Array);       // false
```

Since the string value `"hi"` isn't a 16-bit integer, it's replaced with `0` in the resulting array. Thanks to this error correction behavior, typed array methods don't have to worry about throwing errors when invalid data is present, because there will never be invalid data in the array.

Missing Methods

While typed arrays do have many of the same methods as regular arrays, they also lack several array methods. The following methods are not available on typed arrays:

- `concat()`
- `pop()`
- `push()`
- `shift()`
- `splice()`
- `unshift()`

Except for the `concat()` method, the methods in this list can change the size of an array. Typed arrays can't change size, which is why these aren't available for typed arrays. The `concat()` method isn't available because the result of concatenating two typed arrays (especially if they deal with different data types) would be uncertain, and that would go against the reason for using typed arrays in the first place.

Additional Methods

Finally, typed arrays methods have two methods not present on regular arrays: the `set()` and `subarray()` methods. These two methods are opposites in that `set()` copies another array into an existing typed array, whereas `subarray()` extracts part of an existing typed array into a new typed array.

The `set()` method accepts an array (either typed or regular) and an optional offset at which to insert the data; if you pass nothing, the offset defaults to zero. The data from the array argument is copied into the destination typed array while ensuring only valid data types are used. Here's an example:

```
let ints = new Int16Array(4);

ints.set([25, 50]);
ints.set([75, 100], 2);

console.log(ints.toString());   // 25,50,75,100
```

This code creates an `Int16Array` with four elements. The first call to `set()` copies two values to the first and second elements in the array. The second call to `set()` uses an offset of `2` to indicate that the values should be placed in the array starting at the third element.

The `subarray()` method accepts an optional start and end index (the end index is exclusive, as in the `slice()` method) and returns a new typed array. You can also omit both arguments to create a clone of the typed array. For example:

```
let ints = new Int16Array([25, 50, 75, 100]),
    subints1 = ints.subarray(),
    subints2 = ints.subarray(2),
    subints3 = ints.subarray(1, 3);

console.log(subints1.toString());   // 25,50,75,100
console.log(subints2.toString());   // 75,100
console.log(subints3.toString());   // 50,75
```

Three typed arrays are created from the original `ints` array in this example. The `subints1` array is a clone of `ints` that contains the same information. Since the `subints2` array copies data starting from index 2, it only contains the last two elements of the `ints` array (75 and 100). The `subints3` array contains only the middle two elements of the `ints` array, as `subarray()` was called with both a start and an end index.

Summary

ECMAScript 6 continues the work of ECMAScript 5 by making arrays more useful. There are two more ways to create arrays: the `Array.of()` and `Array.from()` methods. The `Array.from()` method can also convert iterables and array-like objects into arrays. Both methods are inherited by derived array classes and do not use the `Symbol.species` property to determine what type of value should be returned (other inherited methods do use `Symbol.species` when returning an array).

There are also several new methods on arrays. The `fill()` and `copyWithin()` methods allow you to alter array elements in-place. The `find()` and `findIndex()` methods are useful for finding the first element in an array that matches some criteria. The former returns the first element that fits the criteria, and the latter returns the element's index.

Typed arrays are not technically arrays, as they do not inherit from `Array`, but they do look and behave a lot like arrays. Typed arrays contain one of eight different numeric data types and are built upon `ArrayBuffer` objects that represent the underlying bits of a number or series of numbers. Typed arrays are a more efficient way of doing bitwise arithmetic because the values are not converted back and forth between formats, as is the case with the JavaScript number type.