# ES2017: String Padding, Object.entries(), and More

Let's discover the many new and interesting features that ES2017 brought.

WE'LL COVER THE FOLLOWING



ES2017 introduced many cool new features, which we'll check out below.

## String padding (`.padStart()` and `.padEnd()`) #

We can now add some padding to our strings, either at the end (`.padEnd()`) or at the beginning (`.padStart()`) of them.

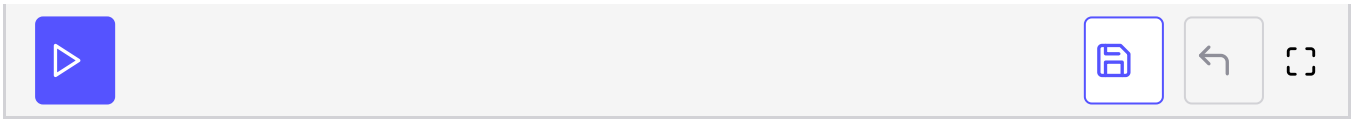
```
console.log("hello".padStart(6));
// " hello"
console.log("hello".padEnd(6));
// "hello "
```

We specified that we want 6 as our padding, so then why in both cases did we only get 1 space? It happened because `padStart` and `padEnd` will go and fill the **empty spaces**. In our example "hello" is 5 letters, and our padding is 6, which

leaves only 1 empty space.

Look at this example:

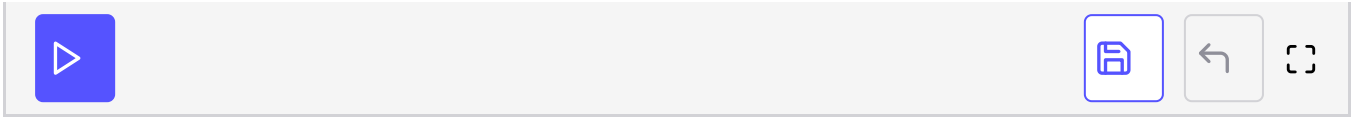
```
console.log("hi".padStart(10));
// 10 - 2 = 8 empty spaces
// "        hi"
console.log("welcome".padStart(10));
// 10 - 6 = 4 empty spaces
// "    welcome"
```



## Right align with `padStart`

We can use `padStart` if we want to right align something.

```
const strings = ["short", "medium length", "very long string"];

const longestString = strings.sort(str => str.length).map(str => str.length)[0];

strings.forEach(str => console.log(str.padStart(longestString)));

// very long string
//    medium length
//             short
```



First we grabbed the longest of our strings and measured its length. We then applied a `padStart` to all the strings based on the length of the longest so that we now have all of them perfectly aligned to the right.

## Add a custom value to the padding

We are not bound to just add a white space as a padding- we can pass both strings and numbers.

```
console.log("hello".padEnd(13," Alberto"));
// "hello Alberto"
```

```
// "001"
console.log("99".padStart(3,0));
// "099"
```

## `Object.entries()` and `Object.values()` #

Let's first create an object.

```
const family = {
  father: "Jonathan Kent",
  mother: "Martha Kent",
  son: "Clark Kent",
}
```

In previous versions of JavaScript, we would have accessed the values inside the object like this:

```
Object.keys(family);
// ["father", "mother", "son"]
family.father;
"Jonathan Kent"
```

`Object.keys()` returned only the keys of the object that we then had to use to access the values.

We now have two more ways of accessing our objects:

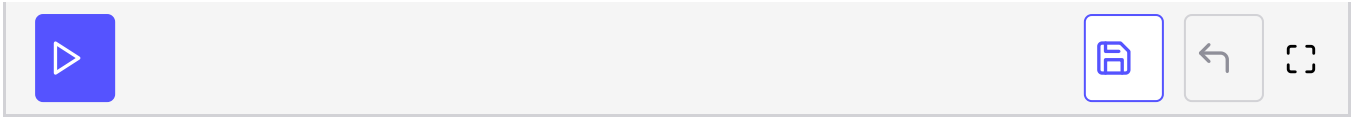
```
const family = {
  father: "Jonathan Kent",
  mother: "Martha Kent",
  son: "Clark Kent",
}
console.log(Object.values(family));
// ["Jonathan Kent", "Martha Kent", "Clark Kent"]

console.log(Object.entries(family));
// ["father", "Jonathan Kent"]
// ["mother", "Martha Kent"]
// ["son", "Clark Kent"]
```

`Object.values()` returns an array of all the values while `Object.entries()` returns an array of arrays containing both keys and values.

## `Object.getOwnPropertyDescriptors()` #

This method will return all the own property descriptors of an object. The attributes it can return are `value`, `writable`, `get`, `set`, `configurable` and `enumerable`.

```
const myObj = {
  name: "Alberto",
  age: 25,
  greet() {
    console.log("hello");
  },
}
console.log(Object.getOwnPropertyDescriptors(myObj));
// age:{value: 25, writable: true, enumerable: true, configurable: true}

// greet:{value: ƒ, writable: true, enumerable: true, configurable: true}

// name:{value: "Alberto", writable: true, enumerable: true, configurable: true}
```



## Trailing commas in function parameter lists and calls #

This is just a minor change to a syntax. Now, when writing objects we can leave a trailing comma after each parameter even if it's not the last one.

```
// from this
const object = {
  prop1: "prop",
  prop2: "propop"
}

// to this
const object = {
  prop1: "prop",
  prop2: "propop",
```

```
}
```

Notice how I wrote a comma at the end of the second property. It will not throw any error if you don't add it, but it's a better practice to follow as it will make your colleague's or team member's life easier.

```
// I write
const object = {
  prop1: "prop",
  prop2: "propop"
}

// my colleague updates the code, adding a new property
const object = {
  prop1: "prop",
  prop2: "propop"
  prop3: "propopop"
}
// suddenly, he gets an error because he did not notice that I forgot to leave a comma at the
```

## Shared memory and `Atomics` #

From MDN:

> When memory is shared, multiple threads can read and write the same data in memory. **Atomic** operations make sure that predictable values are written and read, that operations are finished before the next operation starts and that operations are not interrupted.

`Atomics` is not a constructor. All of its properties and methods are static (just like `Math`) therefore we cannot use it with a new operator or invoke the `Atomics` object as a function.

Examples of its methods are:

- add / sub
- and / or / xor
- load / store

Atomics are used with `SharedArrayBuffer` (generic fixed-length binary data

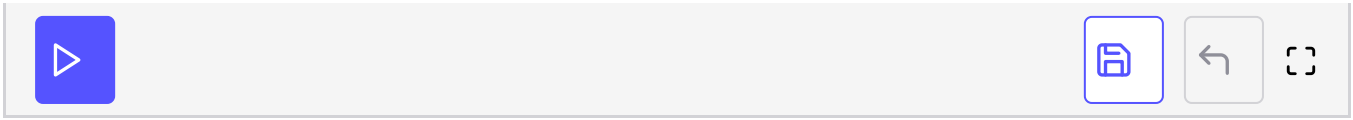
buffer) objects which represent generic, fixed-length raw binary data buffer.

Let's have a look at some examples of `Atomics` methods:

## `Atomics.add()` & `Atomics.load()` #

`Atomics.add()` will take three arguments, an array, an index and a value and will return the previous value at that index before performing an addition.

```
// create a `SharedArrayBuffer`
const buffer = new SharedArrayBuffer(16);
const uint8 = new Uint8Array(buffer);

// add a value at the first position
uint8[0] = 10;

console.log(Atomics.add(uint8, 0, 5));
// 10

// 10 + 5 = 15
console.log(uint8[0])
// 15
console.log(Atomics.load(uint8,0));
// 15
```



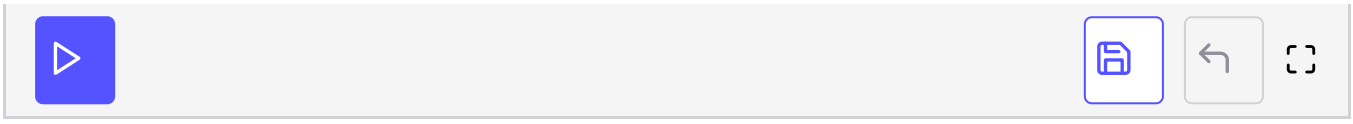
As you can see, calling `Atomics.add()` will return the previous value at the array position we are targeting. When we call `uint8[0]` again we see that the addition was performed and we got 15.

To retrieve a specific value from our array we can use `Atomics.load` and pass two arguments- an array and an index.

`Atomics.sub()` works the same way as `Atomics.add()` but it will subtract a value.

```
// create a `SharedArrayBuffer`
const buffer = new SharedArrayBuffer(16);
const uint8 = new Uint8Array(buffer);

// add a value at the first position
uint8[0] = 10;

console.log(Atomics.sub(uint8, 0, 5));
// 10

// 10 - 5 = 5
```

```
console.log(uint8[0])
// 5
console.log(Atomics.store(uint8,0,3));

// 3
console.log(Atomics.load(uint8,0));
// 3
```



Here we are using `Atomics.sub()` to substract 5 from the value at position `uint8[0]`, which is equivalent to 10 - 5. Same as with `Atomics.add()`, the method will return the previous value at that index, in this case 10.

We are then using `Atomics.store()` to store a specific value, in this case 3- at a specific index of the array, or in this case 0, the first position. `Atomics.store()` will return the value that we just passed, in this case 3. You can see that when we call `Atomics.load()` on that specific index we get 3 and not 5 anymore.

## `Atomics.and()`, `Atomics.or()` and `Atomics.xor()` #

These three methods all perform bitwise AND, OR and XOR operations at a given position of the array. You can read more about bitwise operations on Wikipedia.

In the next lesson, we will cover proxies and how to define custom behaviors and operations using them.