8

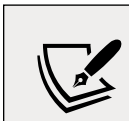
Built-In JavaScript Methods

We have just covered most of the basic building blocks in JavaScript. Now it's time to look at some powerful built-in methods that will make your life easier that we haven't seen yet. Built-in methods are functionality that we get out of the box with JavaScript. We can use these methods without having to code them first. This is something we have done a lot already, for example, `console.log()` and `prompt()`.

Many built-in methods belong to built-in classes as well. These classes and their methods can be used at any time because JavaScript has already defined them. These classes exist for our convenience, since they are very common things to need, such as the `Date`, `Array`, and `Object` classes.

The ability to harness the capabilities that are already built into JavaScript can improve the effectiveness of the code, save time, and comply with various best practices for developing solutions. We are going to address some of the common uses for such functions, such as manipulating text, mathematical computations, dealing with date and time values, interactions, and supporting robust code. Here are the topics covered in this chapter:

- Global JavaScript methods
- String methods
- Math methods
- Date methods
- Array methods
- Number methods



Note: exercise, project and self-check quiz answers can be found in the *Appendix*.

Introduction to built-in JavaScript methods

We have seen many built-in JavaScript methods already. Any method that we didn't define ourselves is a built-in method. Some examples include `console.log()`, `Math.random()`, `prompt()`, and many more—think about methods on arrays for example. The difference between a method and a function is that a function is defined anywhere in the script, and a method is defined inside a class. So methods are pretty much functions on classes and instances.

Methods can often be chained as well; this is only true for methods returning a result. The next method will then be performed on the result. So for example:

```
let s = "Hello";
console.log(
  s.concat(" there!")
  .toUpperCase()
  .replace("THERE", "you")
  .concat(" You're amazing!")
);
```

We create a variable, `s`, and we store `Hello` in there on the first line. Then we want to be logging something. This code has been divided over different lines for readability, but it's actually one statement. We first perform a `concat()` method on our `s` variable, which appends a string to our string. So after that first operation the value is `Hello there!`. Then we transform this to uppercase with the next method. At that point the value is `HELLO THERE!`. Then we proceed to replace `THERE` with `you`. After that, the value becomes `HELLO you!`. We then append a string to it again and finally the value will be logged:

```
HELLO you! You're amazing!
```

We need to log or store the output in this example, because the original string value will not be updated by just calling methods on a string.

Global methods

The global JavaScript methods can be used without referring to the built-in object they are part of. This means that we can just use the method name as if it is a function that has been defined inside the scope we are in, without the "object" in front of it. For example, instead of writing:

```
let x = 7;
console.log(Number.isNaN(x));
```

You can also write:

```
console.log(isNaN(x));
```

So, the `Number` can be left out, because `isNaN` is made globally available without referring to the class it belongs to (in this instance, the `Number` class). In this case, both of these `console.log` statements will log `false` (they are doing the exact same thing), because `isNaN` returns `true` when it isn't a number. And `7` is a number, so it will log `false`.

JavaScript has been built to have these available directly, so to achieve this, some magic is going on beneath the surface. The JavaScript creators chose the methods that they thought were most common. So the reasons why some of them are available as global methods and others are not might seem a bit arbitrary. It's just the choice of some very bright developers at a certain point in time.

We'll address the most common global methods below. We start with decoding and encoding URIs, escaped and unescaped, followed by parsing numbers, and finally evaluate.

Decoding and encoding URIs

Sometimes you will need to encode or decode a string. Encoding is simply converting from one shape to another. In this case we will be dealing with percent encoding, also called URL encoding. Before we start, there might be some confusion on the URI and URL meaning. A **URI** (**uniform resource identifier**) is an identifier of a certain resource. **URL** (**uniform resource locator**) is a subcategory of URI that is not only an identifier, but also holds the information on how to access it (location).

Let's talk about encoding and decoding these URIs (and also URLs, since they are a subset). An example of when you'd need this is when you are sending variables over the URL using the `get` method in a form. These variables that you are sending via the URL are called query parameters.

If something contains a space, this will be decoded, because you cannot use spaces in your URL. They will be converted to `%20`. The URL might look something like:

```
www.example.com/submit?name=maaike%20van%20putten&love=coding
```

All characters can be converted to some `%`-starting format. However, this is not necessary in most cases. URIs can contain a certain number of alphanumeric characters. The special characters need to be encoded. An example, before encoding, is:

```
https://www.example.com/submit?name=maaike van putten
```

The same URL after encoding is:

```
https://www.example.com/submit?name=maaike%20van%20putten
```

There are two pairs of encode and decode methods. We will discuss them and their use cases here. You cannot have a URI with spaces, so working with these methods is crucial in order to work with variables containing spaces.

decodeUri() and encodeUri()

The `decodeUri()` and `encodeUri()` are actually not really encoding and decoding, they are more so fixing broken URIs. It is like the previous example with the spaces. This method pair is really good at fixing broken URIs and decoding them back into a string. Here you can see them in action:

```
let uri = "https://www.example.com/submit?name=maaike van putten";
let encoded_uri = encodeURI(uri);
console.log("Encoded:", encoded_uri);
let decoded_uri = decodeURI(encoded_uri);
console.log("Decoded:", decoded_uri);
```

And here is the output:

```
Encoded: https://www.example.com/submit?name=maaike%20van%20putten
Decoded: https://www.example.com/submit?name=maaike van putten
```

As you can see, it has replaced the spaces in the encoded version and removed them again in the decoded version. All the other characters get to stay the same—this encode and decode do not take special characters into account, and therefore leave them in the URI. Colons, question marks, equal signs, slashes, and ampersands can be expected.

This is great for fixing broken URIs, but it's actually a bit useless whenever you need to encode strings that contain any of these characters: `/ , ? : @ & = + $ #`. These can be used in URIs as part of the URI and are therefore skipped. This is where the next two built-in methods come in handy.

decodeUriComponent() and encodeUriComponent()

So, the methods `decodeURI()` and `encodeURI()` can be very useful to fix a broken URI, but they are useless when you only want to encode or decode a string that contains a character with a special meaning, such as `=` or `&`. Take the following example:

```
https://www.example.com/submit?name=this&that=some thing&code=love
```

Weird value, we can agree on that, but it will demonstrate our problem. Using encodeURI on this will leave us with:

```
https://www.example.com/submit?name=this&that=some%20thing&code=love
```

There are actually 3 variables in here according to URI standards:

- `name` (value is `this`)
- `that` (value is `some thing`)
- `code` (value is `love`)

While we intended to send in one variable, `name`, with the value `this&that=some thing&code=love`.

In this case, you will need `decodeUriComponent()` and `encodeUriComponent()`, because you would need the `=` and `&` in the variable part encoded as well. Right now, this is not the case and it will actually cause problems in interpreting the query parameters (the variables after the `?`). We only wanted to send in one parameter: `name`. But instead we sent in three.

Let's have a look at another example. Here is what the example of the previous section would have done with this component encoding:

```
let uri = "https://www.example.com/submit?name=maaike van putten";
let encoded_uri = encodeURIComponent(uri);
console.log("Encoded:", encoded_uri);
let decoded_uri = decodeURIComponent(encoded_uri);
console.log("Decoded:", decoded_uri);
```

The resulting output is as follows:

```
Encoded: https%3A%2F%2Fwww.example.com%2Fsubmit%3Fname%3Dmaaike%20
van%20putten
Decoded: https://www.example.com/submit?name=maaike van putten
```

Clearly, you don't want this as your URI, but the component methods are useful to encode, for example, a URL variable. If the URL variable were to contain a special character, like `=` and `&`, this would change the meaning and break the URI if these characters don't get encoded.

Encoding with escape() and unescape()

These are still global methods available to do something similar to encode (escape) and decode (unescape). Both methods are strongly discouraged to use and they might actually disappear from future JavaScript versions or may not be supported by browsers for good reasons.

Practice exercise 8.1

Output the `decodeURIComponent()` for the string `How's%20it%20going%3F` to the console. Also, encode the string `How's it going?` to be output into the console. Create a web URL and encode the URI:

1. Add the strings as variables in the JavaScript code
2. Using `encodeURIComponent()` and `decodeURIComponent()` output the results into the console
3. Create a web URI with request parameters `http://www.basescripts.com?=Hello World";`
4. Encode and output the web URI into the console

Parsing numbers

There are different ways to parse strings to numbers. In many situations you will have to translate a string to a number, for example reading input boxes from an HTML web page. You cannot calculate with strings, but you can with numbers. Depending on what exactly you need to do, you will need either one of these methods.

Making integers with parseInt()

With the method `parseInt()` a string will be changed to an integer. This method is part of the `Number` class, but it is global and you can use it without the `Number` in front of it. Here you can see it in action:

```
let str_int = "6";
let int_int = parseInt(str_int);
console.log("Type of ", int_int, "is", typeof int_int);
```

We start off with a string containing a `6`. Then we convert this string to an integer using the `parseInt` method, and when we log the result, we will get in the console:

```
Type of 6 is number
```

You can see that the type has changed from `string` to `number`. At this point, you may wonder what will happen if `parseInt()` tries to parse other types of numbers, like string versions of floats or binary numbers. What do you think will happen when we do this?

```
let str_float = "7.6";
let int_float = parseInt(str_float);
console.log("Type of", int_float, "is", typeof int_float);

let str_binary = "0b101";
let int_binary = parseInt(str_binary);
console.log("Type of", int_binary, "is", typeof int_binary);
```

This will log:

```
Type of 7 is number
Type of 0 is number
```

Can you figure out the logic here? First of all, JavaScript doesn't like crashing or using errors as a way out, so it is trying to make it work to the best of its abilities. The `parseInt()` method simply stops parsing when it runs into a non-numeric character. This is the specified behavior, and you need to keep that in mind while working with `parseInt()`. In the first case, it stops parsing as soon as it finds the dot, so the result is `7`. And in the binary number case, it stops parsing as soon as it hits the `b`, and the result is `0`. By now you can probably figure out what this does:

```
let str_nan = "hello!";
let int_nan = parseInt(str_nan);
console.log("Type of", int_nan, "is", typeof int_nan);
```

Since the first character is non-numeric, JavaScript will convert this string to `NaN`. Here is the result that you will get in the console:

```
Type of NaN is number
```

So `parseInt()` can be a bit quirky, but it's very valuable. In the real world, it is used a lot to combine the input of users via web pages and calculations.

Making floats with parseFloat()

Similarly, we can use `parseFloat()` to parse a string to a float. It works exactly the same, except it can also understand decimal numbers and it doesn't quit parsing as soon as it runs into the first dot:

```
let str_float = "7.6";
let float_float = parseFloat(str_float);
console.log("Type of", float_float, "is", typeof float_float);
```

This will log:

```
Type of 7.6 is number
```

With the `parseInt()`, this value became `7`, because it would stop parsing as soon as it finds a non-numeric character. However, `parseFloat()` can deal with one dot in the number, and the numbers after that are interpreted as decimals. Can you guess what happens when it runs into a second dot?

```
let str_version_nr = "2.3.4";
let float_version_nr = parseFloat(str_version_nr);
console.log("Type of", float_version_nr, "is", typeof float_version_
nr);
```

This will log:

```
Type of 2.3 is number
```

The strategy is similar to the `parseInt()` function. As soon as it finds a character it cannot interpret, a second dot in this case, it will stop parsing and just return the result so far. Then one more thing to note. It is not going to append a `.0` to integers, so `6` is not going to become `6.0`. This example:

```
let str_int = "6";
let float_int = parseFloat(str_int);
console.log("Type of", float_int, "is", typeof float_int);
```

Will log:

```
Type of 6 is number
```

Lastly, the behavior for binary numbers and strings is the same. It is going to stop parsing as soon as it runs into a character it cannot interpret:

```
let str_binary = "0b101";
let float_binary = parseFloat(str_binary);
console.log("Type of", float_binary, "is", typeof float_binary);

let str_nan = "hello!";
let float_nan = parseFloat(str_nan);
console.log("Type of", float_nan, "is", typeof float_nan);
```

This will log:

```
Type of 0 is number
Type of NaN is number
```

You will use the `parseFloat()` whenever you need a decimal number. However, it will not work with binary, hexadecimal, and octal values, so whenever you really need to work with these values or integers you'll have to use `parseInt()`.

Executing JavaScript with eval()

This global method executes the argument as a JavaScript statement. This means that it will just do whatever JavaScript is inserted in there, as if that JavaScript were written directly on the spot instead of `eval()`. This can be convenient for working with injected JavaScript, but injected code comes with great risks. We'll deal with these risks later; let's first explore an example. Here is a fabulous website:

```
<html>
  <body>
    <input onchange="go(this)"></input>
    <script>
      function go(e) {
        eval(e.value);
      }
    </script>
  </body>
</html>
```

This is a basic HTML web page with an input box on it.

You'll learn more about HTML in *Chapter 9, The Document Object Model*.

Whatever you insert in the input box will get executed. If we were to write this in the input box:

```
document.body.style.backgroundColor = "pink";
```

The website background would change to pink. That looks like fun, right? However, we cannot stress enough how careful you should be using `eval()`. They might as well have called it *evil* according to many developers. Can you reason why this might be?

The answer is security! Yes, this is probably the worst thing security-wise you can do in most situations. You are going to execute external code. This code could be malicious. It is a method for supporting code injection. The well-respected **OWASP** (**Open Web Application Security Project**) Foundation creates top 10s for security threats every 3 years. Code injection has been on it since their first top 10 and it is still in the OWASP top 10 security threats now. Running it server side can cause your server to crash and your website to go down, or worse. There are almost always better solutions to what you want to do than using `eval()`. Next to the security risks, it is terrible performance-wise. So just for this reason already you might want to avoid using it.

Alright, so one last note on this. If you know what you are doing you might want to use it in very specific cases. Even though it is "evil", it has a lot of power. It can be okay to use in certain cases, for example when you are creating template engines, your own interpreter, and all other JavaScript core tools. Just beware of the danger and control access to this method carefully. And one last bonus tip, when you feel like you really have to use eval, do a quick search on the web. Chances are that you will find a better approach.

Array methods

We have seen arrays already—they can contain multiple items. We have also seen quite a few built-in methods on arrays, like `shift()` and `push()`. Let's look at a few more in the following sections.

Performing a certain action for every item

There is a reason we are starting with this method. You might be thinking of loops at this point, but there is a built-in method that you can use to execute a function for every element in the array. This is the `forEach()` method. We mentioned this briefly in *Chapter 6*, *Functions*, but let's consider it in some more detail. It takes the function that needs to be executed for every element as input. Here you can see an example:

```
let arr = ["grapefruit", 4, "hello", 5.6, true];

function printStuff(element, index) {
  console.log("Printing stuff:", element, "on array position:", index);
}

arr.forEach(printStuff);
```

This code snippet will write to the console:

```
Printing stuff: grapefruit on array position: 0
Printing stuff: 4 on array position: 1
Printing stuff: hello on array position: 2
Printing stuff: 5.6 on array position: 3
Printing stuff: true on array position: 4
```

As you can see, it called the `printStuff()` function for every element in the array. And we can also use the index, it is the second parameter. We don't need to control the flow of the loop here and we cannot get stuck at a certain point. We just need to specify what function needs to be executed for every element. And the element will be input for this function. This is used a lot, especially for a more functional programming style in which many methods get chained, for example, to process data.

Filtering an array

We can use the built-in `filter()` method on an array to alter which values are in the array. The filter method takes a function as an argument, and this function should return a Boolean. If the Boolean has the value `true`, the element will end up in the filtered array. If the Boolean has the value `false`, the element will be left out. You can see how it works here:

```
let arr = ["squirrel", 5, "Tjed", new Date(), true];

function checkString(element, index) {
  return typeof element === "string";
}

let filterArr = arr.filter(checkString);
console.log(filterArr);
```

This will log to the console:

```
[ 'squirrel', 'Tjed' ]
```

It is important to realize that the original array has not changed, the `filter()` method returns a new array with the elements that made it through the filter. We capture it here in the variable `filterArr`.

Checking a condition for all elements

You can use the `every()` method to see whether something is true for all elements in the array. If that is the case, the `every()` method will return `true`, else it will return `false`. We are using the `checkString()` function and array from the previous example here:

```
console.log(arr.every(checkString));
```

This will log `false`, since not all elements are of type `string` in the array.

Replacing part of an array with another part of the array

The `copyWithin()` method can be used to replace a part of the array with another part of the array. In the first example we specify 3 arguments. The first one is the target position, to which the values get copied. The second one is the start of what to copy to the target position and the last one is the end of the sequence that will be copied to the target position; this last index is not included. Here we are only going to override position 0 with whatever is in position 3:

```
arr = ["grapefruit", 4, "hello", 5.6, true];
arr.copyWithin(0, 3, 4);
```

`arr` becomes:

```
[ 5.6, 4, 'hello', 5.6, true ]
```

If we specify a range with length `2`, the first two elements after the starting position get overridden:

```
arr = ["grapefruit", 4, "hello", 5.6, true];
arr.copyWithin(0, 3, 5);
```

And now `arr` becomes:

```
[ 5.6, true, 'hello', 5.6, true ]
```

We can also not specify an end at all; it will take the range to the end of the string:

```
let arr = ["grapefruit", 4, "hello", 5.6, true, false];
arr.copyWithin(0, 3);
console.log(arr);
```

This will log:

```
[ 5.6, true, false, 5.6, true, false ]
```

It is important to keep in mind that this function changes the *content* of the original array, but will never change the *length* of the original array.

Mapping the values of an array

Sometimes you'll need to change all the values in an array. With the array `map()` method you can do just that. This method will return a new array with all the new values. You'll have to say how to create these new values. This can be done with the arrow function. It is going to execute the arrow function for every element in the array, so for example:

```
let arr = [1, 2, 3, 4];
let mapped_arr = arr.map(x => x + 1);
console.log(mapped_arr);
```

This is what the console output with the new mapped array looks like:

```
[ 2, 3, 4, 5 ]
```

Using the arrow function, the `map()` method has created a new array, in which each of the original array values has been increased by 1.

Finding the last occurrence in an array

We can find occurrences with `indexOf()` as we have seen already. To find the last occurrence, we can use the `lastIndexOf()` method on an array, just as we did for `string`.

It will return the index of the last element with that value, if it can find it at all:

```
let bb = ["so", "bye", "bye", "love"];
console.log(bb.lastIndexOf("bye"));
```

This will log `2`, because the index 2 holds the last `bye` variable. What do you think you'll get when you ask for the last index of something that's not there?

```
let bb = ["so", "bye", "bye", "love"];
console.log(bb.lastIndexOf("hi"));
```

That's right (hopefully)! It's `-1`.

Practice exercise 8.2

Remove duplicates from the array using `filter()` and `indexOf()`. The starting array is:

```
["Laurence", "Mike", "Larry", "Kim", "Joanne", "Laurence", "Mike",
"Laurence", "Mike", "Laurence", "Mike"]
```

Using the array `filter()` method, this will create a new array using the elements that pass the test condition implemented by the function. The final result will be:

```
[ 'Laurence', 'Mike', 'Larry', 'Kim', 'Joanne' ]
```

Take the following steps:

1. Create an array of names of people. Make sure you include duplicates. The exercise will remove the duplicate names.
2. Using the `filter()` method, assign the results of each item from the array as arguments within an anonymous function. Using the value, index, and array arguments, return the filtered result. You can set the return value to `true` temporarily as this will build the new array with all the results in the original array.
3. Add a `console.log` call within the function that will output the index value of the current item in the array. Also add the value so you can see the results of the item value that has the current index number and the first matching result from the array's index value.

4. Using `indexOf()` the current value returns the index value of the item and applies the condition to check to see if it matches the original index value. This condition will only be true on the first result so all subsequent duplicates will be false and not get added to the new array. `false` will not return the value into the new array. The duplicates will all be false since the `indexof()` only gets the first match in the array.
5. Output the new, unique value array onto the console.

Practice exercise 8.3

Using the array `map()` method, update an array's contents. Take the following steps:

1. Create an array of numbers.
2. Using the array `map` method and an anonymous function, return an updated array, multiplying all the numbers in the array by 2. Output the result into the console.
3. As an alternative method, use the arrow function format to multiply each element of the array by 2 with the array `map()` method in one line of code.
4. Log the result onto the console.

String methods

We have worked with strings already and chances are that you have run into some of the methods on strings by now. There are a few we didn't address specifically just yet and we are going to discuss them in this section.

Combining strings

When you want to concatenate strings, you can use the `concat()` method. This does not change the original string(s); it returns the combined result as a string. You will have to capture the result in a new variable, else it will get lost:

```
let s1 = "Hello ";
let s2 = "JavaScript";
let result = s1.concat(s2);
console.log(result);
```

This will log:

```
Hello JavaScript
```

Converting a string to an array

With the `split()` method we can convert a string to an array. Again, we will have to capture the result; it is not changing the original string. Let's use the previous result containing `Hello JavaScript`. We will have to tell the `split` method on what string it should split. Every time it encounters that string, it will create a new array item:

```
let result = "Hello JavaScript";
let arr_result = result.split(" ");
console.log(arr_result);
```

This will log:

```
[ 'Hello', 'JavaScript' ]
```

As you can see, it creates an array of all the elements separated by a space. We can split by any character, for example a comma:

```
let favoriteFruits = "strawberry,watermelon,grapefruit";
let arr_fruits = favoriteFruits.split(",");
console.log(arr_fruits);
```

This will log:

```
[ 'strawberry', 'watermelon', 'grapefruit' ]
```

It has now created an array with 3 items. You can split on anything, and the string you are splitting on is left out of the result.

Converting an array to a string

With the `join()` method you can convert an array to a string. Here is a basic example:

```
let letters = ["a", "b", "c"];
let x = letters.join();
console.log(x);
```

This will log:

```
a,b,c
```

The type of `x` is `string`. If you want something else other than a comma, you can specify that, like this:

```
let letters = ["a", "b", "c"];
let x = letters.join('-');
console.log(x);
```

This will use the – instead of the comma. This is the result:

```
a-b-c
```

This can be nicely combined with the `split()` method that we covered in the previous section, which does the reverse and converts a string into an array.

Working with index and positions

Being able to find out what index a certain substring is at within your string is very useful. For example, when you need to search for a certain word through the user input of a log file and create a substring starting at that index. Here is an example of how to find the index of a string. The `indexOf()` method returns the index, a single number, of the first character of the substring:

```
let poem = "Roses are red, violets are blue, if I can do JS, then you
            can too!";
let index_re = poem.indexOf("re");
console.log(index_re);
```

This is logging `7` to the console, because the first occurrence of `re` is in `are`, and the `re` begins at index `7`. When it can't find an index, it will return `-1`, like this example:

```
let indexNotFound = poem.indexOf("python");
console.log(indexNotFound);
```

It is logging `-1` to indicate that the string we are searching for doesn't occur in the target string. Often you will write an `if` check to see whether it's `-1` before dealing with the result. For example:

```
if(poem.indexOf("python") != -1) {
  // do stuff
}
```

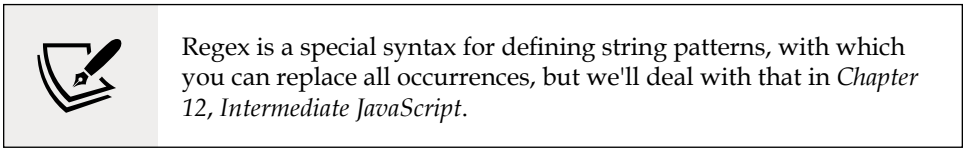
An alternative way of searching for a particular substring within a string is to use the `search()` method:

```
let searchStr = "When I see my fellow, I say hello";
let pos = searchStr.search("lo");
console.log(pos);
```

This will log `17`, because that is the index of `lo` in `fellow`. Much like `indexOf()`, if it cannot find it, it will return `-1`. This is the case for this example:

```
let notFound = searchStr.search("JavaScript");
console.log(notFound);
```

`search()` will accept a regex format as input, whereas `indexOf()` just takes a string. `indexOf()` is faster than the `search()` method, so if you just need to look for a string, use `indexOf()`. If you need to look for a string pattern, you'll have to use the `search()` method.



Regex is a special syntax for defining string patterns, with which you can replace all occurrences, but we'll deal with that in *Chapter 12, Intermediate JavaScript*.

Moving on, the `indexOf()` method is returning the index of the first occurrence, but similarly, we also have a `lastIndexOf()` method. It returns the index where the argument string occurs last. If it cannot find it, it returns `-1`. Here is an example:

```
let lastIndex_re = poem.lastIndexOf("re");
console.log(lastIndex_re);
```

This returns `24`; this is the last time `re` appears in our poem. It is the second `are`.

Sometimes you will have to do the reverse; instead of looking for what index a string occurs at, you will want to know what character is at a certain index position. This is where the `charAt(index)` method comes in handy, where the specified index position is taken as an argument:

```
let pos1 = poem.charAt(10);
console.log(pos1);
```

This is logging `r`, because the character at index 10 is the `r` of `red`. If you are asking for the position of an index that is out of the range of the string, it will return an empty string, as is happening in this example:

```
let pos2 = poem.charAt(1000);
console.log(pos2);
```

This will log an empty line to the screen, and if you ask for the type of `pos2`, it will return `string`.

Creating substrings

With the `slice(start, end)` method we can create substrings. This does not alter the original string, but returns a new string with the substring. It takes two parameters, the first is the index at which it starts and the second is the end index. If you leave out the second index it will just continue until the end of the string from the start. The end index is not included in the substring. Here is an example:

```
let str = "Create a substring";
let substr1 = str.slice(5);
let substr2 = str.slice(0,3);
console.log("1:", substr1);
console.log("2:", substr2);
```

This will log:

```
1: e a substring
2: Cre
```

The first one only has one argument, so it starts at index 5 (which holds an `e`) and grabs the rest of the string from there. The second one has two arguments, `0` and `3`. `C` is at index 0 and `a` is at index 3. Since the last index is not included in the substring, it will only return `Cre`.

Replacing parts of the string

If you need to replace a part of the string, you can use the `replace(old, new)` method. It takes two arguments, one string to look for in the string and one new value to replace the old value with. Here is an example:

```
let hi = "Hi buddy";
let new_hi = hi.replace("buddy", "Pascal");
console.log(new_hi);
```

This will log to the console `Hi Pascal`. If you don't capture the result, it is gone, because the original string will not get changed. If the string you are targeting doesn't appear in the original string, the replacement doesn't take place and the original string will be returned:

```
let new_hi2 = hi.replace("not there", "never there");
console.log(new_hi2);
```

This logs `Hi buddy`.

One last note here, it is only changing the first occurrence by default. So this example will only replace the first `hello` in the new string:

```
let s3 = "hello hello";
let new_s3 = s3.replace("hello", "oh");
console.log(new_s3);
```

This logs `oh hello`. If we wanted to replace all the occurences, we could use the `replaceAll()` method. This will replace all occurrences with the specified new string, like this:

```
let s3 = "hello hello";
let new_s3 = s3.replaceAll("hello", "oh");
console.log(new_s3);
```

This logs `oh oh`.

Uppercase and lowercase

We can change the letters of a string with the `toUpperCase()` and `toLowerCase()` built-in methods on `string`. Again, this is not changing the original string, so we'll have to capture the result:

```
let low_bye = "bye!";
let up_bye = low_bye.toUpperCase();
console.log(up_bye);
```

This logs:

```
BYE!
```

It converts all the letters to uppercase. We can do the opposite with `toLowerCase()`:

```
let caps = "HI HOW ARE YOU?";
let fixed_caps = caps.toLowerCase();
console.log(fixed_caps);
```

This will log:

```
hi how are you?
```

Let's make it a bit more complicated and say that we'd like the first letter of the sentence to be capitalized. We can do this by combining some of the methods we have seen already right now:

```
let caps = "HI HOW ARE YOU?";
let fixed_caps = caps.toLowerCase();
let first_capital = fixed_caps.charAt(0).toUpperCase().concat(fixed_
caps.slice(1));
console.log(first_capital);
```

We are chaining the methods here; we first grab the first character of `fixed_caps` with `charAt(0)` and then make it uppercase by calling `toUpperCase()` on it. We then need the rest of the string and we get it by concatenating `slice(1)`.

The start and end of a string

Sometimes you would want to check what a string starts or ends with. You've guessed it, there are built-in methods for this on `string`. We can imagine this chapter is tough to work through, so here is a little encouragement and an example at the same time:

```
let encouragement = "You are doing great, keep up the good work!";
let bool_start = encouragement.startsWith("You");
console.log(bool_start);
```

This will log `true` to the console, because the sentence starts with `You`. Careful here, because it is case sensitive. So the following example will log `false`:

```
let bool_start2 = encouragement.startsWith("you");
console.log(bool_start2);
```

If you don't care about uppercase or lowercase, you can use the previously discussed `toLowerCase()` method here, so that it will not take uppercase or lowercase into account:

```
let bool_start3 = encouragement.toLowerCase().startsWith("you");
console.log(bool_start3);
```

We are now converting the string to lowercase first, so we know we are only working with lowercase characters here. However, an important side note here is that this will affect performance for huge strings.

Again, a more performance-friendly alternative is to use regex. Getting excited for *Chapter 12*, *Intermediate JavaScript*, yet?

To end this section, we can do the same thing for checking whether a string ends with a certain string. You can see it in action here:

```
let bool_end = encouragement.endsWith("Something else");
console.log(bool_end);
```

Since it doesn't end with `Something else`, it will return `false`.

Practice exercise 8.4

Using string manipulation, create a function that will return a string with the first letter of all the words capitalized and the rest of the letters in lowercase. You should transform the sentence `thIs will be capiTalized for each word` into `This Will Be Capitalized For Each Word`:

1. Create a string with several words that have letters with different cases, a mix of upper and lowercase words.
2. Create a function that gets a string as an argument, which will be the value that we will manipulate.
3. In the function first transform everything to lowercase letters.
4. Create an empty array that can hold the values of the words when we capitalize them.
5. Convert the phrase into words in an array using the `split()` method.
6. Loop through each one of the words that are now in the new array, so you can select each one independently. You can use `forEach()` for this.

7. Using `slice()` isolate the first letter in each word, then transform it to uppercase. Again using `slice()`, get the remaining value of the word without including the first letter. Then concatenate the two together to form the word that is now capitalized.
8. Add the new capitalized word into the blank array that you created. By the end of the loop you should have an array with all the words as separate items in the new array.
9. Take the array of updated words and using the `join()` method, transform them back into a string with spaces between each word.
10. Return the value of the newly updated string with capitalized words that can then be output into the console.

Practice exercise 8.5

Using the `replace()` string method, complete this vowel replacer exercise by replacing the vowels in a string with numbers. You can start with this string:

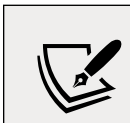
```
I love JavaScript
```

And turn it into something like the following:

```
2 l3v1 j0v0scr2pt
```

Take the following steps:

1. Create the previously specified string, and convert it to lowercase.
2. Create an array containing the vowels: a, e, i, o, u.
3. Loop through each letter you have in the array, and output the current letter into the console so that you can see which letter will be converted.
4. Within the loop, using `replaceAll()` update each vowel substring with the index value of the letter from the vowel array.



Using `replace()` will only replace the first occurrence; if you use `replaceAll()` this will update all matching results.

5. Once the loop completes output the result of the new string into the console.

Number methods

Let's move on to some built-in methods on the `Number` object. We have seen a few already, these are so popular that some of them have been made into global methods.

Checking if something is (not) a number

This can be done with `isNaN()`. We have seen this already when we talked about global methods, we can use this method without `Number` in front of it. Often you will want to do the opposite, you can negate the function with an `!` in front of it:

```
let x = 34;
console.log(isNaN(x));
console.log(!isNaN(x));
let str = "hi";
console.log(isNaN(str));
```

This will log to the console:

```
false
true
true
```

Since `x` is a number, `isNaN` will be `false`. But this result negated becomes `true`, since `x` is a number. The string `hi` is not a number, so it will become `false`. And this one?

```
let str1 = "5";
console.log(isNaN(str1));
```

Some funky stuff is going on here, even though `5` is between quotation marks, JavaScript still sees that it's a 5 and it will log `false`. At this point, I'm sure you'd wish your partner, family, and coworkers are as understanding and forgiving as JavaScript.

Checking if something is finite

By now you might be able to guess the name of the method on `Number` that checks whether something is finite. It is a very popular one and has been made into a global method as well, and its name is `isFinite()`. It returns `false` for `NaN`, `Infinity`, and `undefined`, and `true` for all other values:

```
let x = 3;
let str = "finite";
console.log(isFinite(x));
console.log(isFinite(str));
console.log(isFinite(Infinity));
console.log(isFinite(10 / 0));
```

This will log:

```
true
false
false
false
```

The only finite number in this list is `x`. The others are not finite. A string is a not a number and is therefore not finite. `Infinity` is not finite and `10` divided by `0` returns `Infinity` (not an error).

Checking if something is an integer

Yes, this is done with `isInteger()`. Unlike `isNaN()` and `isFinite()`, `isInteger()` has not been made global and we will have to refer to the `Number` object to use it. It really does what you think it would: it returns `true` if the value is an integer and `false` when it's not:

```
let x = 3;
let str = "integer";
console.log(Number.isInteger(x));
console.log(Number.isInteger(str));
console.log(Number.isInteger(Infinity));
console.log(Number.isInteger(2.4));
```

This will log:

```
true
false
false
false
```

Since the only integer in the list is `x`.

Specifying a number of decimals

We can tell JavaScript how many decimals to use with the `toFixed()` method. This is different from the rounding methods in `Math`, since we can specify the number of decimals here. It doesn't change the original value, so we'll have to store the result:

```
let x = 1.23456;
let newX = x.toFixed(2);
```

This will only leave two decimals, so the value of `newX` will be `1.23`. It rounds the number normally; you can see this when we ask for one more decimal:

```
let x = 1.23456;
let newX = x.toFixed(3);
console.log(x, newX);
```

This logs `1.23456 1.235` as output.

Specifying precision

There is also a method to specify precision. Again this is different from the rounding methods in the `Math` class, since we can specify the total number of numbers to look at. This comes down to JavaScript looking at the total number of numbers. It is also counting the ones before the dot:

```
let x = 1.23456;
let newX = x.toPrecision(2);
```

So the value of newX will be `1.2` here. And also here, it is rounding the numbers:

```
let x = 1.23456;
let newX = x.toPrecision(4);
console.log(newX);
```

This will log `1.235`.

Now, let's move on and talk about some related mathematical methods!

Math methods

The `Math` object has many methods that we can use to do calculations and operations on numbers. We will go over the most important ones here. You can see all the available ones when you use an editor that shows suggestions and options during typing.

Finding the highest and lowest number

There is a built-in method `max()` to find the highest number among the arguments. You can see it here:

```
let highest = Math.max(2, 56, 12, 1, 233, 4);
console.log(highest);
```

It logs `233`, because that's the highest number. In a similar way, we can find the lowest number:

```
let lowest = Math.min(2, 56, 12, 1, 233, 4);
console.log(lowest);
```

This will log `1`, because that is the lowest number. If you try to do this with non-numeric arguments, you will get `NaN` as a result:

```
let highestOfWords = Math.max("hi", 3, "bye");
console.log(highestOfWords);
```

It is not giving `3` as output, because it is not ignoring the text but concluding that it cannot determine whether `hi` should be higher or lower than `3`. So it returns `NaN` instead.

Square root and raising to the power of

The method `sqrt()` is used to calculate the square root of a certain number. Here you can see it in action:

```
let result = Math.sqrt(64);
console.log(result);
```

This will log `8`, because the square root of `64` is `8`. This method works just like the mathematics you learned in school. In order to raise a number to a certain power ($base^{exponent}$, for example 2^3), we can use the `pow(base, exponent)` function. Like this:

```
let result2 = Math.pow(5, 3);
console.log(result2);
```

We are raising `5` to the power of `3` here (5^3), so the result will be `125`, which is the result of 5*5*5.

Turning decimals into integers

There are different ways to turn decimals into integers. Sometimes you will want to round a number. This you can do with the `round()` method:

```
let x = 6.78;
let y = 5.34;

console.log("X:", x, "becomes", Math.round(x));
console.log("Y:", y, "becomes", Math.round(y));
```

This will log:

```
X: 6.78 becomes 7
Y: 5.34 becomes 5
```

As you can see it is using normal rounding here. It is also possible that you don't want to round down, but up. For example, if you need to calculate how many wood boards you need and you conclude that you need `1.1`, `1` is not going to be enough to do the job. You'll need `2`. In this case, you can use the `ceil()` method (referring to ceiling):

```
console.log("X:", x, "becomes", Math.ceil(x));
console.log("Y:", y, "becomes", Math.ceil(y));
```

This will log:

```
X: 6.78 becomes 7
Y: 5.34 becomes 6
```

The `ceil()` method is always rounding up to the first integer it encounters. We have used this before when we were generating random numbers! Careful with negative numbers here, because `-5` is higher than `-6`. This is how it works, as you can see in this example:

```
let negativeX = -x;
let negativeY = -y;

console.log("negativeX:", negativeX, "becomes", Math.ceil(negativeX));
console.log("negativeY:", negativeY, "becomes", Math.ceil(negativeY));
```

This will log:

```
negativeX: -6.78 becomes -6
negativeY: -5.34 becomes -5
```

The `floor()` method is doing the exact opposite of the `ceil()` method. It rounds down to the nearest integer number, as you can see here:

```
console.log("X:", x, "becomes", Math.floor(x));
console.log("Y:", y, "becomes", Math.floor(y));
```

This will log:

```
X: 6.78 becomes 6
Y: 5.34 becomes 5
```

Again, careful with negative numbers here, because it can feel counterintuitive:

```
console.log("negativeX:", negativeX, "becomes", Math.floor(negativeX));
console.log("negativeY:", negativeY, "becomes", Math.floor(negativeY));
```

This logs:

```
negativeX: -6.78 becomes -7
negativeY: -5.34 becomes -6
```

And then one last method, `trunc()`. This gives the exact same result as `floor()` for positive numbers, but it gets to these results differently. It is not rounding down, it is simply only returning the integer part:

```
console.log("X:", x, "becomes", Math.trunc(x));
console.log("Y:", y, "becomes", Math.trunc(y));
```

This will log:

```
X: 6.78 becomes 6
Y: 5.34 becomes 5
```

When we use negative numbers for `trunc()` we can see the difference:

```
negativeX: -6.78 becomes -6
negativeY: -5.34 becomes -5
```

So whenever you need to round down, you'll have to use `floor()`, if you need the integer part of the number, you'll need `trunc()`.

Exponent and logarithm

The exponent is the number to which a base is being raised. We use `e` (Euler's number) a lot in mathematics, this is what the `exp()` method in JavaScript does. It returns the number to which `e` must be raised to get the input. We can use the `exp()` built-in method of `Math` to calculate the exponent, and the `log()` method to calculate the natural logarithm. You can see an example here:

```
let x = 2;
let exp = Math.exp(x);
console.log("Exp:", exp);
let log = Math.log(exp);
console.log("Log:", log);
```

This will log:

```
Exp: 7.38905609893065
Log: 2
```

Don't worry if you can't follow along mathematically at this point. You'll figure this out whenever you'll need it for your programming.

Practice exercise 8.6

Experiment with the `Math` object with these steps:

1. Output the value of `PI` into the console using `Math`.
2. Using `Math` get the `ceil()` value of `5.7`, get the `floor()` value of `5.7`, get the round value of `5.7`, and output it into the console.
3. Output a random value into the console.
4. Use `Math.floor()` and `Math.random()` to get a number from 0 to 10.
5. Use `Math.floor()` and `Math.random()` to get a number from 1 to 10.
6. Use `Math.floor()` and `Math.random()` to get a number from 1 to 100.
7. Create a function to generate a random number using the parameters of `min` and `max`. Run that function 100 times, returning into the console a random number from 1 to 100 each time.

Date methods

In order to work with dates in JavaScript we use the built-in `Date` object. This object contains a lot of built-in functions to work with dates.

Creating dates

There are different ways to create a date. One way to create dates is by using the different constructors. You can see some examples here:

```
let currentDateTime = new Date();
console.log(currentDateTime);
```

This will log the current date and time, in this case:

```
2021-06-05T14:21:45.625Z
```

But, this way we are not using the built-in method, but the constructor. There is a built-in method, `now()`, that returns the current date and time, similar to what the no argument constructor does:

```
let now2 = Date.now();
console.log(now2);
```

This will log the current time, represented in seconds since January 1st 1970. This is an arbitrary date representing the Unix epoch. In this case:

```
1622902938507
```

We can add 1,000 milliseconds to the Unix epoch time:

```
let milliDate = new Date(1000);
console.log(milliDate);
```

It will log:

```
1970-01-01T00:00:01.000Z
```

JavaScript can also convert many string formats to a date. Always mind the order in which days and months of dates are presented in the date format and the interpreter of JavaScript. This can vary depending on the region:

```
let stringDate = new Date("Sat Jun 05 2021 12:40:12 GMT+0200");
console.log(stringDate);
```

This will log:

```
2021-06-05T10:40:12.000Z
```

And lastly, you can also specify a certain date using the constructor:

```
let specificDate = new Date(2022, 1, 10, 12, 10, 15, 100);
console.log(specificDate);
```

This will log:

```
2022-02-10T12:10:15.100Z
```

Please mind this very important detail here, the second parameter is the month. `0` is for January and `11` is for December.

Methods to get and set the elements of a date

Now we have seen how to create dates, we'll learn how to get certain parts of dates. This can be done with one of the many `get` methods. Which you will use depends on the part you need:

```
let d = new Date();
console.log("Day of week:", d.getDay());
console.log("Day of month:", d.getDate());
console.log("Month:", d.getMonth());
console.log("Year:", d.getFullYear());
console.log("Seconds:", d.getSeconds());
console.log("Milliseconds:", d.getMilliseconds());
console.log("Time:", d.getTime());
```

This will log right now:

```
Day of week: 6
Day of month: 5
Month: 5
Year: 2021
Seconds: 24
Milliseconds: 171
Time: 1622903604171
```

The time is so high because it's the number of milliseconds since January 1st 1970. You can change the date in a similar way with a `set` method. Important to note here is that the original date object gets changed with these set methods:

```
d.setFullYear(2010);
console.log(d);
```

We have changed the year of our date object to 2010. This will output:

```
2010-06-05T14:29:51.481Z
```

We can also change the month. Let's add the below snippet to our change of the year code. This will change it to October. Please mind that while I'm doing this, I run the code again and again, so the minutes and smaller units of time will vary in the examples when I haven't set these yet:

```
d.setMonth(9);
console.log(d);
```

It will log:

```
2010-10-05T14:30:39.501Z
```

This is a weird one, in order to change the day, we have to call the `setDate()` method and not the `setDay()` method. There is no `setDay()` method, since the day of the week is deducted from the specific date. We cannot change that September 5th 2021 is a Sunday. We can change the number of days of the month though:

```
d.setDate(10);
console.log(d);
```

This will log:

```
2010-10-10T14:34:25.271Z
```

We can also change the hours:

```
d.setHours(10);
console.log(d);
```

Now it will log:

```
2010-10-10T10:34:54.518Z
```

Remember how JavaScript doesn't like to crash? If you call `setHours()` with a number higher than 24, it will roll over to the next date (1 per 24 hours) and after using the modulo operator, whatever is left over from `hours % 24` will be the hours. The same process applies for minutes, seconds, and milliseconds.

The `setTime()` actually overrides the complete date with the inserted epoch time:

```
d.setTime(1622889770682);
console.log(d);
```

This will log:

```
2021-06-05T10:42:50.682Z
```

Parsing dates

With the built-in `parse()` method we can parse epoch dates from strings. It accepts many formats, but again you will have to be careful with the order of days and months:

```
let d1 = Date.parse("June 5, 2021");
console.log(d1);
```

This will log:

```
1622851200000
```

As you can see it ends with many zeros, because no time or seconds are specified in our string. And here is another example of a completely different format:

```
let d2 = Date.parse("6/5/2021");
console.log(d2);
```

This will also log:

```
1622851200000
```

The input for the parse is ISO formats of dates. Quite a few formats can be parsed to string, but you'll have to be careful. The result might depend on the exact implementation. Make sure that you know what the format of the incoming string is, so that you don't confuse months and days, and make sure that you know the behavior of the implementations. This can only be done reliably if you know what the string format is. So for example when you need to convert data coming from your own database or website's date picker.

Converting a date to a string

We can also convert dates back to strings. For example with these methods:

```
console.log(d.toDateString());
```

This will log the day in written format:

```
Sat Jun 05 2021
```

This is another method that converts it differently:

```
console.log(d.toLocaleDateString());
```

It will log:

```
6/5/2021
```

Practice exercise 8.7

Output the date with the full month name into the console. When converting to or from arrays, remember that they are zero-based:

1. Set up a date object, which can be any date in the future or past. Log the date out into the console to see how it is typically output as a date object.
2. Set up an array with all the named months of the year. Keep them in sequential order so that they will match the date month output.
3. Get the day from the date object value, using `getDate()`.
4. Get the year from the date object value, using `getFullYear()`.
5. Get the month of the date object value, using `getMonth()`.
6. Set up a variable to hold the date of the date object and output the month using the numeric value as the index for the array month name. Due to arrays being zero-based and the month returning a value of 1-12, the result needs to be subtracted by one.
7. Output the result into the console.

Chapter projects

Word scrambler

Create a function that returns a value of a word and scrambles the letter order with `Math.random()`:

1. Create a string that will hold a word value of your choice.
2. Create a function that can intake a parameter of the string word value.

3. Just like an array, strings also have a length by default. You can use this length to set the loop maximum value. You will need to create a separate variable to hold this value as the length of the string will be decreasing as the loop continues.
4. Create an empty temporary string variable that you can use to hold the new scrambled word value.
5. Create a `for` loop that will iterate the number of letters within the string parameter starting at 0 and iterating until the original length value of the string is reached.
6. Create a variable that will randomly select one letter using its index value, with `Math.floor()` and `Math.random()` multiplied by the current length of the string.
7. Add the new letter to the new string and remove it from the original string.
8. Using `console.log()` output the newly constructed string from the random letters and output to the console both the original string and the new one as the loop continues.
9. Update the original string by selecting the substring from the index value and adding it to the remaining string value from the index plus one onward. Output the new original string value with the removed characters.
10. As you loop through the content you will see a countdown of the remaining letters, the new scrambled version of the word as it is built, and the decreasing letters in the original word.
11. Return the final result and invoke the function with the original string word as an argument. Output this to the console.

Countdown timer

Create code for a countdown timer that can be executed in the console window, and will show the total milliseconds, days, hours, minutes, and seconds remaining until a target date is reached:

1. Create an end date that you want to count down to. Format it in a date type format within a string.
2. Create a countdown function that will parse the `endDate()` and subtract the current date from that end date. This will show the total in milliseconds. Using `Date.parse()` you can convert a string representation of a date to a numeric value as a number of milliseconds since January 1, 1970, 00:00:00 UTC.

3. Once you have the total milliseconds, to get the days, hours, minutes, and seconds you can take the following steps:
    - To get days you can divide the number of milliseconds in a date, removing the remainder with `Math.floor()`.
    - To get hours you can use modulus to capture just the remainder once the total days are removed.
    - To get minutes you can use the value of milliseconds in a minute and using the modulus capture the remainder.
    - Do the same for seconds by dividing the number by seconds in milliseconds and getting the remainder. If you use `Math.floor()` you can round down removing any remaining decimal places that will be shown in the lower values.
4. Return all the values within an object with property names indicating what the unit of time the values refer to.
5. Create a function to use a `setTimeout()` to run the `update()` function every second (1,000 milliseconds). The `update()` function will create a variable that can temporarily hold the object return values of `countdown()`, and create an empty variable that will be used to create the output values.
6. Within the same function, using the `for` loop get all the properties and values of the `temp` object variable. As you iterate through the object update the output string to contain the property name and property value.
7. Using `console.log()`, print the output result string into the console.

Self-check quiz

1. Which method will decode the following?

```
var c = "http://www.google.com?id=1000&n=500";
var e = encodeURIComponent(c);
```

    a. `decodeURIComponent(e)`
    b. `e.decodeUriComponent()`
    c. `decoderURIComponent(c)`
    d. `decoderURIComponent(e)`

2. What will be output into the console from the following syntax?

```
const arr = ["hi","world","hello","hii","hi","hi World","Hi"];
console.log(arr.lastIndexOf("hi"));
```

3. What is the result of the below code in the console?

```
const arr = ["Hi","world","hello","Hii","hi","hi World","Hi"];
arr.copyWithin(0, 3, 5);
console.log(arr);
```

4. What is the result of the below code in the console?

```
const arr = ["Hi","world","hello","Hii","hi","hi World","Hi"];
const arr2 = arr.filter((element,index)=>{
    const ele2 = element.substring(0, 2);
    return (ele2 == "hi");
});
console.log(arr2);
```

Summary

In this chapter we have dealt with many built-in methods. These are methods that are handed to us by JavaScript and that we can use for things that we'll often need. We went over the most used global built-in methods, which are so common they can be used without being prepended by the object they belong to.

We also discussed array methods, string methods, number methods, math methods, and date methods. You'll find yourself using these methods a lot and chaining them (whenever they return a result) when you get more comfortable with JavaScript.

Now we've become familiar with many of JavaScript's core features, we'll spend the next couple of chapters diving into how it works alongside HTML and the browser to bring web pages to life!