JavaScript™

ABSOLUTE BEGINNER'S GUIDE

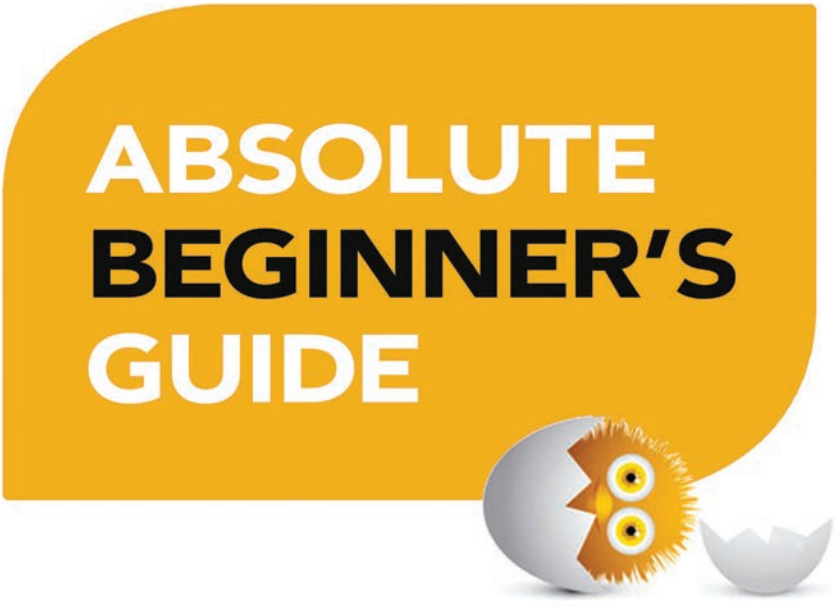
No experience necessary!

Third Edition

Kirupa Chinnathambi

JavaScript™

Third Edition



Kirupa Chinnathambi

Pearson

For example, this is all valid:

```
let textA = "Please";
let textB = new String("stop!");
let combined = textA + " make it " + textB;

console.log(combined);
```

Despite all the mixing going on, the type of the combined variable is simply a **string** primitive.

For combining strings, we also have the concat method. We can call this method from any string and specify a sequence of string primitives, literals, and objects that we want to combine into one megastring:

```
let foo = "I really";
let blah = "why anybody would";
let blarg = "do this";

let result = foo.concat(" don't know", " ", blah, " ", blarg);

console.log(result);
```

For the most part, just use the + and += approach for combining strings. It is faster than the `concat` approach. With everything else being equal, who wouldn't want some extra speed in their code?

One thing to point out is that there is a much more modern way to combine strings, especially if what we are trying to do is combine strings and variables together. We'll look at that in the next chapter.

Getting Substrings Out of Strings

Sometimes what we are interested in is a sequence of characters somewhere in the middle of our string. The two properties that help satisfy this interest are `slice` and `substr`. Let's say we have the following string:

```
let theBigString = "Pulp Fiction is an awesome movie!";
```

Let's mess with this string for a bit.

The slice Method

The `slice` method allows us to specify the start and end positions of the part of the string that we want to extract:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.slice(5, 12));
```

In this example, we extract the characters between index positions 5 and 12. The end result is that the word **Fiction** is returned.

The start and end position values do not have to be positive. If you specify a negative value for the end position, the end position for your string is what is left when you count backwards from the end:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.slice(0, -6));
```

If we specify a negative start position, our start position is the count of whatever we specify starting from the end of the string:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.slice(-14, -7));
```

We just saw three variations of how the `slice` method can be used. I've never used anything but the first version with a positive start and end position, and you'll probably be in a similar boat.

The substr Method

The next approach we will look at for splitting up a string is the `substr` method. This method takes two arguments as well:

```
let newString = substr(start, length);
```

The first argument is a number that specifies our starting position, and the second argument is a number that specifies the length of our substring. This makes more sense when we look at some examples:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.substr(0, 4)); // Pulp
```

We start the substring at the 0 position and count four characters up. That is why **Pulp** is returned. If we want to just extract the word **Fiction**, this is what our code would look like:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.substr(5, 7)); // Fiction
```

If we don't specify the length, the substring that gets returned is the string that goes from the start position to the end:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.substr(5)); // Fiction is an awesome
movie!
```

There are a few more variations of values we can pass in for substr, but these are the big ones.

Splitting a String with split

That which you can concatenate, you can also split apart. I am pretty sure a wise person once said that. Another way we can split apart a string is by using the `split` method. Calling this method on a string returns an array of substrings. These substrings are separated by a character or regular expression (aka RegEx) that we use to determine where to split apart our string.

Let's look at a simple example where this makes more sense:

```
let inspirationalQuote = "That which you can concatenate, you can
also split apart.";
```

```
let splitWords = inspirationalQuote.split(" ");

console.log(splitWords.length); // 10
```

In this example, we are splitting the `inspirationalQuote` text on the space character. Every time a space character is encountered, we break our string and make the text prior to the space an array item. This repeats until we reach the end of the string. What we see at the end is an array of strings whose contents are the individual pieces of text we had separated by a space earlier.

Here is another example:

```
let days = "Monday,Tuesday,Wednesday,Thursday,Friday,
Saturday,Sunday";
let splitWords = days.split(",");

console.log(splitWords[6]); // Sunday
```

We have the `days` variable, which stores a string of days separated only by a comma. If we wanted to separate out each day, we could use the `split` method with the separator character being the comma. The end result is an array of seven items, where each item is the day of the week from the original string.

You'll be surprised at how often you find yourself using the `split` method to break apart a sequence of characters, which can be as simple as a sentence or something more complex like data returned from a web service.

Finding Something Inside a String

If we ever need to find a character or characters inside a string, we can use the `indexOf`, `lastIndexOf`, and `match` methods. Let's look at the `indexOf` method first.

What the `indexOf` method does is take the character(s) we are looking for as its argument. If what we are looking for is found, it returns the index position in the string where the first occurrence…occurs. If no matches are found, this method gifts you with a −1. Let's look at an example:

```
let question = "I wonder what the pigs did to make these birds so
angry?";
console.log(question.indexOf("pigs")); // 18
```

We are trying to see if pigs exist in our string. Because what we are looking for does exist, the `indexOf` method lets us know that the first occurrence of this word can be found at the 18th index position. If we look for something that doesn't exist, like the letter **z** in this example, a –1 gets returned:

```
let question = "I wonder what the pigs did to make these birds so
angry?";
console.log(question.indexOf("z")); // -1
```

The `lastIndexOf` method is very similar to `indexOf`. As you can sorta maybe guess by the name, `lastIndexOf` returns the last occurrence of what you are looking for:

```
let question = "How much wood could a woodchuck chuck if a
woodchuck could chuck wood?";
console.log(question.lastIndexOf("wood")); // 65
```

There is one more argument you can specify to both `indexOf` and `lastIndexOf`. In addition to providing the characters to search for, you can also specify an index position on your string to start your search from:

```
let question = "How much wood could a woodchuck chuck if a
woodchuck could chuck wood?";
console.log(question.indexOf("wood", 30)); // 43
```

The last thing to mention about the `indexOf` and `lastIndexOf` methods is that you can match any instance of these characters appearing in your string. These functions do not differentiate between whole words and a substring of a larger set of characters. Be sure to take that into account.

Before we wrap this up, let's look at the `match` method. With the `match` method, you have a little more control. This method takes a RegEx as its argument:

```
let phrase = "There are 3 little pigs.";
let regexp = /[0-9]/;
```

```
let numbers = phrase.match(regexp);

console.log(numbers[0]); // 3
```

What gets returned is also an array of matching substrings, so you can use your array ninja skills to make working with the results a breeze. Learning how to work with regular expressions is something that goes beyond what we'll look at in this book, but the following documentation on MDN is a great starting point: https://bit.ly/kirupaRegEx

Uppercasing and Lowercasing Strings

Finally, let's end this coverage on strings with something easy that doesn't require anything complicated. To uppercase or lowercase a string, we can use the appropriately named `toUpperCase` and `toLowerCase` methods. Let's look at an example:

```
let phrase = "My name is Bond. James Bond.";

console.log(phrase.toUpperCase()); // MY NAME IS BOND. JAMES BOND.
console.log(phrase.toLowerCase()); // my name is bond. james bond.
```

See, told you this was easy!

THE ABSOLUTE MINIMUM

Strings are one of the handful of basic data types you have available in JavaScript, and you just saw a good overview of the many things you can do using them. One issue that I skirted around is where your string primitives seem to mysteriously have all these properties that are common only to objects. We'll look at that in the next chapter!

? Ask a question: **https://forum.kirupa.com**

✔ Practice by building real apps: **https://bit.ly/coding_exercises**

Errors/known issues: **https://bit.ly/javascript_errata**