

## STRINGS

So far, we've just been working with numbers. Now let's look at another type of data: *strings*. Strings in JavaScript (as in most programming languages) are just sequences of characters, which can include letters, numbers, punctuation, and spaces. We put strings between quotes so JavaScript knows where they start and end. For example, here's a classic:

---

```
"Hello world!";  
"Hello world!"
```

---

To enter a string, just type a double quotation mark (") followed by the text you want in the string, and then close the string with another double quote. You can also use single quotes ('), but to keep things simple, we'll just be using double quotes in this book.

You can save strings into variables, just like numbers:

---

```
var myAwesomeString = "Something REALLY awesome!!!";
```

---

There's also nothing stopping you from assigning a string to a variable that previously contained a number:

---

```
var myThing = 5;  
myThing = "this is a string";  
"this is a string"
```

---

What if you put a number between quotes? Is that a string or a number? In JavaScript, a string is a string (even if it happens to have some characters that are numbers). For example:

---

```
var numberNine = 9;  
var stringNine = "9";
```

---

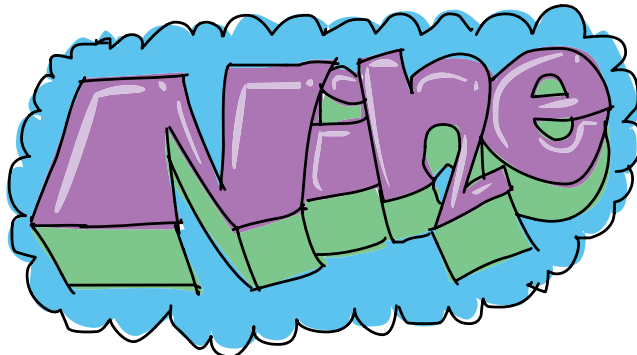
numberNine is a number, and stringNine is a string. To see how these are different, let's try adding them together:

---

```
numberNine + numberNine;  
18  
stringNine + stringNine;  
"99"
```

---

When we add the number values 9 and 9, we get 18. But when we use the + operator on "9" and "9", the strings are simply joined together to form "99".



## JOINING STRINGS

As you just saw, you can use the + operator with strings, but the result is very different from using the + operator with numbers. When you use + to join two strings, you make a new string with the second string attached to the end of the first string, like this:

---

```
var greeting = "Hello";  
var myName = "Nick";  
greeting + myName;  
"HelloNick"
```

---

Here, we create two variables (greeting and myName) and assign each a string value ("Hello" and "Nick", respectively). When we add these two variables together, the strings are combined to make a new string, "HelloNick".

That doesn't look right, though—there should be a space between Hello and Nick. But JavaScript won't put a space there unless we specifically tell it to by adding a space in one of the original strings:

---

```
❶ var greeting = "Hello ";  
var myName = "Nick";  
greeting + myName;  
"Hello Nick"
```

---

The extra space inside the quotes at ❶ puts a space in the final string as well.

You can do a lot more with strings other than just adding them together. Here are some examples.

## FINDING THE LENGTH OF A STRING

To get the length of a string, just add .length to the end of it.

---

```
"Supercalifragilisticexpialidocious".length;  
34
```

---

You can add .length to the end of the actual string or to a variable that contains a string:

---

```
var java = "Java";  
java.length;  
4
```

---

```
var script = "Script";
script.length;
6
var javascript = java + script;
javascript.length;
10
```

---

Here we assign the string "Java" to the variable `java` and the string "Script" to the variable `script`. Then we add `.length` to the end of each variable to determine the length of each string, as well as the length of the combined strings.

Notice that I said you can add `.length` to “the actual string *or to a variable* that contains a string.” This illustrates something very important about variables: anywhere you can use a number or a string, you can also use a variable containing a number or a string.

## GETTING A SINGLE CHARACTER FROM A STRING

Sometimes you want to get a single character from a string. For example, you might have a secret code where the message is made up of the second character of each word in a list of words. You’d need to be able to get just the second characters and join them all together to create a new word.

To get a character from a particular position in a string, use square brackets, `[ ]`. Just take the string, or the variable containing the string, and put the number of the character you want in a pair of square brackets at the end. For example, to get the first character of `myName`, use `myName[0]`, like this:

---

```
var myName = "Nick";
myName[0];
"N"
myName[1];
"i"
myName[2];
"c"
```

---

Notice that to get the first character of the string, we use 0 rather than 1. That’s because JavaScript (like many other programming languages) starts counting at zero. That means when

you want the first character of a string, you use 0; when you want the second one, you use 1; and so on.

Let's try out our secret code, where we hide a message in some words' second characters. Here's how to find the secret message in a sequence of words:

---

```
var codeWord1 = "are";
var codeWord2 = "tubas";
var codeWord3 = "unsafe";
var codeWord4 = "?!";
codeWord1[1] + codeWord2[1] + codeWord3[1] + codeWord4[1];
"run!"
```


---

Again, notice that to get the second character of each string, we use 1.

## CUTTING UP STRINGS

To “cut off” a piece of a big string, you can use `slice`. For example, you might want to grab the first bit of a long movie review to show as a teaser on your website. To use `slice`, put a period after a string (or a variable containing a string), followed by the word `slice` and opening and closing parentheses. Inside the parentheses, enter the start and end positions of the slice of the string you want, separated by a comma. Figure 2-2 shows how to use `slice`.

These two numbers  
set the **start** and **end** of the slice.



```
"a string".slice(1, 5)
```

Figure 2-2: How to use `slice` to get characters from a string

For example:

---

```
var longString = "My long string is long";
longString.slice(3, 14);
"long string"
```

---

The first number in parentheses is the number of the character that begins the slice, and the second number is the number of

the character *after* the last character in the slice. Figure 2-3 shows which characters this retrieves, with the start value (3) and stop value (14) highlighted in blue.

M	y		l	o	n	g		s	t	r	i	n	g		i	s		l	o	n	g
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Figure 2-3: In the example above, *slice* grabs the characters shown in the gray box.

Here we basically tell JavaScript, “Pull a slice out of this longer string starting at the character at place 3 and keep going until you hit place 14.”

If you include only one number in the parentheses after *slice*, the string that it slices will start from that number and continue all the way to the end of the string, like this:

---

```
var longString = "My long string is long";
longString.slice(3);
"long string is long"
```

---

## CHANGING STRINGS TO ALL CAPITAL OR ALL LOWERCASE LETTERS

If you have some text that you just want to shout, try using *toUpperCase* to turn it all into capital letters.

---

```
"Hello there, how are you doing?".toUpperCase();
"HELLO THERE, HOW ARE YOU DOING?"
```

---

When you use *toUpperCase()* on a string, it makes a new string where all the letters are turned into uppercase.

You can go the other way around, too:

---

```
"hELLO THERE, hOW ARE yOu doING?".toLowerCase();
"hello there, how are you doing?"
```

---

As the name suggests, *toLowerCase()* makes all of the characters lowercase. But shouldn't sentences always start with a capital letter? How can we take a string and make the first letter uppercase but turn the rest into lowercase?

**NOTE**

*See if you can figure out how to turn "hELlo THERE, hOW ARE yOu doING?" into "Hello there, how are you doing?" using the tools you just learned. If you get stuck, review the sections on getting a single character and using slice. Once you're done, come back and have a look at how I did it.*

Here's one approach:

---

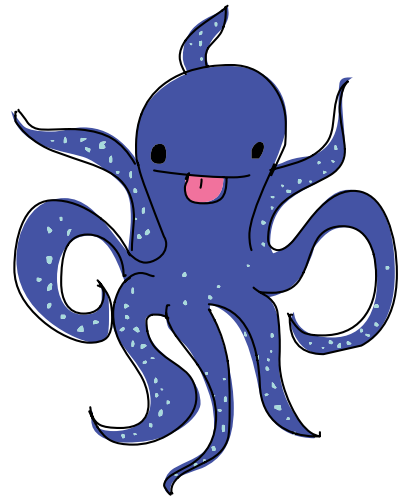
```
❶ var sillyString = "hELlo THERE, hOW ARE yOu doING?";
❷ var lowerString = sillyString.toLowerCase();
❸ var firstCharacter = lowerString[0];
❹ var firstCharacterUpper = firstCharacter.toUpperCase();
❺ var restOfString = lowerString.slice(1);
❻ firstCharacterUpper + restOfString;
"Hello there, how are you doing?"
```

---

Let's go through this line by line. At ❶, we create a new variable called `sillyString` and save the string we want to modify to that variable. At ❷, we get the lowercase version of `sillyString` ("hello there how are you doing?") with `.toLowerCase()` and save that in a new variable called `lowerString`.

At ❸, we use `[0]` to get the first character of `lowerString` ("h") and save it in `firstCharacter` (0 is used to grab the first character). Then, at ❹, we create an uppercase version of `firstCharacter` ("H") and call that `firstCharacterUpper`.

At ❺, we use `slice` to get all the characters in `lowerString`, starting from the second character ("ello there how are you doing?") and save that in `restOfString`. Finally, at ❻, we add `firstCharacterUpper` ("H") to `restOfString` to get "Hello there, how are you doing?".



Because values and variables can be substituted for each other, we could turn lines ❷ through ❸ into just one line, like this:

---

```
var sillyString = "hElLo THERE, hOW ARE yOu doINg?";  
sillyString[0].toUpperCase() + sillyString.slice(1).toLowerCase();  
"Hello there, how are you doing?"
```

---

It can be confusing to follow along with code written this way, though, so it's a good idea to use variables for each step of a complicated task like this—at least until you get more comfortable reading this kind of complex code.

## BOOLEANS

Now for Booleans. A *Boolean* value is simply a value that's either true or false. For example, here's a simple Boolean expression.

---

```
var javascriptIsCool = true;  
javascriptIsCool;  
true
```

---

In this example, we created a new variable called `javascriptIsCool` and assigned the Boolean value `true` to it. On the second line, we get the value of `javascriptIsCool`, which, of course, is `true`!

## LOGICAL OPERATORS

Just as you can combine numbers with mathematical operators (+, -, \*, /, and so on), you can combine Boolean values with Boolean operators. When you combine Boolean values with Boolean operators, the result will always be another Boolean value (either true or false).

The three main Boolean operators in JavaScript are `&&`, `||`, and `!`. They may look a bit weird, but with a little practice, they're not hard to use. Let's try them out.

### **&& (AND)**

`&&` means “and.” When reading aloud, people call it “and,” “and-and,” or “ampersand-ampersand.” (*Ampersand* is the name of the character `&`.) Use the `&&` operator with two Boolean values to see if they're *both* true.



For example, before you go to school, you want to make sure that you've had a shower *and* you have your backpack. If both are true, you can go to school, but if one or both are false, you can't leave yet.

---

```
var hadShower = true;  
var hasBackpack = false;  
hadShower && hasBackpack;  
false
```

---

Here we set the variable `hadShower` to `true` and the variable `hasBackpack` to `false`. When we enter `hadShower && hasBackpack`, we are basically asking JavaScript, “Are both of these values true?” Since they aren't both true (you don't have your backpack), JavaScript returns `false` (you're not ready for school).

Let's try this again, with both values set to `true`:

---

```
var hadShower = true;  
var hasBackpack = true;  
hadShower && hasBackpack;  
true
```

---

Now JavaScript tells us that `hadShower && hasBackpack` is `true`. You're ready for school!



## II (OR)

The Boolean operator `||` means “or.” It can be pronounced “or,” or even “or-or,” but some people call it “pipes,” because programmers call the `|` character a *pipe*. You can use this operator with two Boolean values to find out whether *either* one is true.

For example, say you're still getting ready to go to school and you need to take a piece of fruit for lunch, but it doesn't matter whether you take an apple or an orange or both. You can use JavaScript to see whether you have at least one, like this:

---

```
var hasApple = true;  
var hasOrange = false;
```

---

```
hasApple || hasOrange;  
true
```

---

`hasApple || hasOrange` will be true if either `hasApple` or `hasOrange` is true, or if both are true. But if *both* are false, the result will be false (you don't have any fruit).

## ! (NOT)

`!` just means “not.” You can call it “not,” but lots of people call it “bang.” (An exclamation point is sometimes called a *bang*.) Use it to turn false into true or true into false. This is useful for working with values that are opposites. For example:

```
var isWeekend = true;  
var needToShowerToday = !isWeekend;  
needToShowerToday;  
false
```

---

In this example, we set the variable `isWeekend` to true. Then we set the variable `needToShowerToday` to `!isWeekend`. The bang converts the value to its opposite—so if `isWeekend` is true, then `!isWeekend` is *not* true (it's false). So when we ask for the value of `needToShowerToday`, we get false (you don't need to shower today, because it's the weekend).

Because `needToShowerToday` is false, `!needToShowerToday` will be true:

```
needToShowerToday;  
false  
!needToShowerToday;  
true
```

---

In other words, it's *true* that you do *not* need to shower today.

## COMBINING LOGICAL OPERATORS

Operators get interesting when you start combining them. For example, say you should go to school if it's *not* the weekend *and* you've showered *and* you have an apple *or* you have an orange. We could check whether all of this is true with JavaScript, like this:

```
var isWeekend = false;  
var hadShower = true;  
var hasApple = false;
```

---

```
var hasOrange = true;  
var shouldGoToSchool = !isWeekend && hadShower && (hasApple || hasOrange);  
shouldGoToSchool;  
true
```

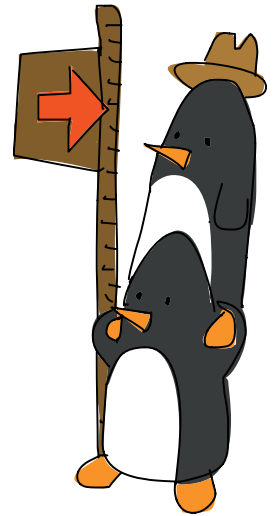
---

In this case, it's not the weekend, you have showered, and you don't have an apple but you do have an orange—so you should go to school.

`hasApple || hasOrange` is in parentheses because we want to make sure JavaScript works out that bit first. Just as JavaScript calculates `*` before `+` with numbers, it also calculates `&&` before `||` in logical statements.

## COMPARING NUMBERS WITH BOOLEANS

Boolean values can be used to answer questions about numbers that have a simple yes or no answer. For example, imagine you're running a theme park and one of the rides has a height restriction: riders must be at least 60 inches tall, or they might fall out! When someone wants to go on the ride and tells you their height, you need to know if it's greater than this height restriction.



### GREATER THAN

We can use the greater-than operator (`>`) to see if one number is greater than another. For example, to see if the rider's height (65 inches) is greater than the height restriction (60 inches), we could set the variable `height` equal to 65 and the variable `heightRestriction` equal to 60, and then use `>` to compare the two:

```
var height = 65;  
var heightRestriction = 60;  
height > heightRestriction;  
true
```

---

With `height > heightRestriction`, we're asking JavaScript to tell us whether the first value is greater than the second. In this case, the rider is tall enough!

What if a rider were exactly 60 inches tall, though?

---

```
var height = 60;
var heightRestriction = 60;
height > heightRestriction;
false
```

---

Oh no! The rider isn't tall enough! But if the height restriction is 60, then shouldn't people who are exactly 60 inches be allowed in? We need to fix that. Luckily, JavaScript has another operator, `>=`, which means "greater than or equal to":

---

```
var height = 60;
var heightRestriction = 60;
height >= heightRestriction;
true
```

---

Good, that's better—60 *is* greater than or equal to 60.

## LESS THAN

The opposite of the greater-than operator (`>`) is the less-than operator (`<`). This operator might come in handy if a ride were designed only for small children. For example, say the rider's height is 60 inches, but riders must be no more than 48 inches tall:

---

```
var height = 60;
var heightRestriction = 48;
height < heightRestriction;
false
```

---

We want to know if the rider's height is *less* than the restriction, so we use `<`. Because 60 is not less than 48, we get `false` (someone whose height is 60 inches is too tall for this ride).

And, as you may have guessed, we can also use the operator `<=`, which means "less than or equal to":

---

```
var height = 48;
var heightRestriction = 48;
height <= heightRestriction;
true
```

---

Someone who is 48 inches tall is still allowed to go on the ride.

## EQUAL TO

To find out if two numbers are exactly the same, use the triple equal sign (`===`), which means “equal to.” But be careful not to confuse `===` with a single equal sign (`=`), because `===` means “are these two numbers equal?” and `=` means “save the value on the right in the variable on the left.” In other words, `===` asks a question, while `=` assigns a value to a variable.



When you use `=`, a variable name has to be on the left and the value you want to save to that variable must be on the right. On the other hand, `===` is just used for comparing two values to see if they’re the same, so it doesn’t matter which value is on which side.

For example, say you’re running a competition with your friends Chico, Harpo, and Groucho to see who can guess your secret number, which is 5. You make it easy on your friends by saying that the number is between 1 and 9, and they start to guess. First you set `mySecretNumber` equal to 5. Your first friend, Chico, guesses that it’s 3 (`chicoGuess`). Let’s see what happens next:

---

```
var mySecretNumber = 5;
var chicoGuess = 3;
mySecretNumber === chicoGuess;
false
var harpoGuess = 7;
mySecretNumber === harpoGuess;
false
var grouchoGuess = 5;
mySecretNumber === grouchoGuess;
true
```

---

The variable `mySecretNumber` stores your secret number. The variables `chicoGuess`, `harpoGuess`, and `grouchoGuess` represent your friends’ guesses, and we use `===` to see whether each guess is the same as your secret number. Your third friend, Groucho, wins by guessing 5.

When you compare two numbers with `===`, you get `true` only when both numbers are the same. Because `grouchoGuess` is 5 and `mySecretNumber` is 5, `mySecretNumber === grouchoGuess` returns `true`. The other guesses didn’t match `mySecretNumber`, so they returned `false`.

You can also use `===` to compare two strings or two Booleans. If you use `===` to compare two different types—for example, a string and a number—it will always return `false`.

## DOUBLE EQUALS

Now to confuse things a bit: there's another JavaScript operator (double equals, or `==`) that means “equal-ish.” Use this to see whether two values are the same, even if one is a string and the other is a number. All values have some kind of type. So the number 5 is different from the string “5”, even though they basically look like the same thing. If you use `===` to compare the number 5 and the string “5”, JavaScript will tell you they’re not equal. But if you use `==` to compare them, it will tell you they’re the same:

---

```
var stringNumber = "5";
var actualNumber = 5;
stringNumber === actualNumber;
false
stringNumber == actualNumber;
true
```

---

At this point, you might be thinking to yourself, “Hmm, it seems much easier to use double equals than triple equals!” You have to be very careful, though, because double equals can be very confusing. For example, do you think 0 is equal to `false`? What about the string “false”? When you use double equals, 0 is equal to `false`, but the string “false” is not:

---

```
0 == false;
true
"false" == false;
false
```

---

This is because when JavaScript tries to compare two values with double equals, it first tries to make them the same type. In this case, it converts the Boolean into a number. If you convert Booleans to numbers, `false` becomes 0, and `true` becomes 1. So when you type `0 == false`, you get `true`!

Because of this weirdness, it’s probably safest to just stick with `===` for now.

## TRY IT OUT!

You've been asked by the local movie theater managers to implement some JavaScript for a new automated system they're building. They want to be able to work out whether someone is allowed into a PG-13 movie or not.

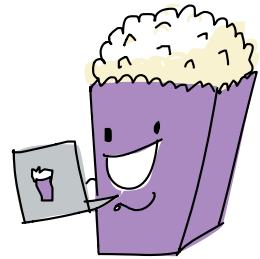
The rules are, if someone is 13 or over, they're allowed in. If they're not over 13, but they are accompanied by an adult, they're also allowed in. Otherwise, they can't see the movie.

---

```
var age = 12;  
var accompanied = true;  
???
```

---

Finish this example using the age and accompanied variables to work out whether this 12-year-old is allowed to see the movie. Try changing the values (for example, set age to 13 and accompanied to false) and see if your code still works out the right answer.



## UNDEFINED AND NULL

Finally, we have two values that don't fit any particular mold. They're called undefined and null. They're both used to mean "nothing," but in slightly different ways.

undefined is the value JavaScript uses when it doesn't have a value for something. For example, when you create a new variable, if you don't set its value to anything using the = operator, its value will be set to undefined:

---

```
var myVariable;  
myVariable;  
undefined
```

---

The null value is usually used when you want to deliberately say “This is empty.”

---

```
var myNullVariable = null;  
myNullVariable;  
null
```

---

At this point, you won’t be using undefined or null very often. You’ll see undefined if you create a variable and don’t set its value, because undefined is what JavaScript will always give you when it doesn’t have a value. It’s not very common to set something to undefined; if you feel the need to set a variable to “nothing,” you should use null instead.

null is used only when you actually want to say something’s not there, which is very occasionally helpful. For example, say you’re using a variable to track what your favorite vegetable is. If you hate all vegetables and don’t have a favorite, you might set the favorite vegetable variable to null.

Setting the variable to null would make it obvious to anyone reading the code that you don’t have a favorite vegetable. If it were undefined, however, someone might just think you hadn’t gotten around to setting a value yet.

## WHAT YOU LEARNED

Now you know all the basic data types in JavaScript—numbers, strings, and Booleans—as well as the special values null and undefined. Numbers are used for math-type things, strings are used for text, and Booleans are used for yes or no questions. The values null and undefined are there to give us a way to talk about things that don’t exist.

In the next two chapters, we’ll look at arrays and objects, which are both ways of joining basic types to create more complex collections of values.