Number Variables

Now that we've gone over string variables, let's go over **number variables** in a little more detail. Like string variables, number variables are primitives. Unlike some programming languages, in JavaScript all number variables are treated as floating point values.

The difference between an integer and a floating point value is that an integer is a round number whereas a floating point value is tracked beyond the decimal point to sub-values. If you compare a variable that's in the set to a whole number and a variable that's been set to a floating point number, JavaScript will consider them the same value.

Number Primitives

Number variables are also primitives just like string variables. That means that they passed their value when they're copied, leaving the original unchanged.

Let's demonstrate that:

```
var number1 = 123;
var number2 = number1;
console.log(number1);
console.log(number2);
```

They have the same value.

Change `number2` to be `456`:

```
number2 = 456;
```

`number1` has not been changed, but `number2` has been changed to `456`. This may seem like an intuitive way for variables to work, but not all variables in JavaScript will work this way. We're going to be talking about a few examples later.

Number Methods

Just like variables of type string, variables of type number also have a set of specific methods that can be used to process them.

For example, you can use the `parseint` method to pull out the integer value, or whole number value, from a floating point variable number.

You can use `parseFloat` to pull out the floating point value of a number, to assert.

The `toString` method will convert a number to a string. What this does is take the value of the number, and output it as a string between quotes which can no longer be used as a number, but can be used to present information inside of another string.

The `toFixed` method creates a string out of the value of a number, but limits it to a certain number of digits past the decimal point.

Examples of Number Methods

As I mentioned before,all numbers in JavaScript treated as floating point values. Even if they're integers and have no value after the decimal point. To pull out the whole number integer value of this variable, essentially dropping off everything after the decimal point we can use `parseInt`:

```
var example = 5.5;
console.log(example.toString());
console.log(parseInt(example, 10));
```

We want it to parse the integer in base 10. If you don't pass the 10, you risk the possibility that it might be processed in a different base, which would give you a different result.

These methods process the value of the variable but they don't actually affect the variable itself:

```
console.log(example.toString());
```

We can do a `parseFloat`:

```
console.log(parseFloat(example));
```

In this case we don't need to pass in a base.

Sometimes when you're creating a string out of a variable you want to have a certain number of digits in the result, for presentation purposes. For that reason, JavaScript includes the `toFixed` and `toPrecision` methods, which also convert to strings but establish exactly how many digits will be shown:

```
console.log(example.toFixed());
console.log(example.toPrecision(3));
```

`toPrecision` is very convenient for doing currency for example.

Numbers That Aren't Numbers

All of these number methods assume that you're starting with a variable that is actually a number to begin with. But sometimes you can get strange results if you try to apply them to variables that aren't numbers or if you use mathematical operations on variables that are not numbers. An impossible value for a number will result in a value `NaN`, which means "not a number".

```
var notNumber = 5 * "Word";
console.log(notNumber);
```

`NaN` means this is not a number so the result actually can't be parsed and processed by JavaScript. If you've ever been on a web page where you were looking for results from a pricing table, for example, and you saw `NaN` show up anywhere, that means that the programmers didn't check to make sure that their results were actually generating real numbers when they tried to present them.

One interesting thing about `NaN`:

```
console.log(typeof(notNumber));
```

That's one of those little idiosyncrasies about JavaScript. It's important to know if you ever start testing for numbers to make sure that they're actually valid. Not a number will show as a valid number, so be careful.