Type Coercion

Type Coercion is when data types are converted implicitly by JavaScript. Although the conversion that we did in the previous video can also be called explicit coercion. There's so many different words for the same thing in programming.

Anyway, there are a few situations where implicit coercion can occur. For the most part, you have 3 types of coercion

- to string
- to number
- to boolean

Typically coercion happens when you apply operators to values of different types.

Let's take a look at some examples. I will warn you, some of this will get weird. But again, this stuff usually isn't a big issue in everyday development.

Let's look at our first example

```
5 + '5'; // 55 (string)
```

So as you can see, if we use the **+** operator on the **number** 5 and a **string** with the character '5', we get a **string** of 55. Some of you may have expected 10, which is the answer to 5 + 5.

The reason that this happened is because the **number** 5 is being implicitly coerced into a **string** due to the + operator also being used for concatenation, as I talked about in the last video.

This is an example of automatic or implicit coercion because we applied an operator to values of different types.

If I wanted to add these together, I would first convert the string '55' to a number like this

```
5 + Number('5'); // 10
```

Although the string would most likely be a variable in this case.

Now, let's take the **number** 5 and multiply by the **string** 5.

```
5 * '5'; // 25 (number)
```

For this expression, we do get the result of 5 * 5, which is a **number** of 25. So in this case, JavaScript looked at the expression and coerced the **string** of '5' into the **number** 5.

This happened because it makes the most sense. The * can not do anything else but multiply in this expression.

Let's try some more weird stuff and see the results

```
5 + null; // 5
```

So in this case, we get 5. Reason being that null is coerced to a number of 0. We can see what a value would be as a number by doing the following

```
Number(null); // 0
```

Let's see what the boolean values of true and false would be as a number

```
Number(true); // 1
Number(false); // 0
```

So with that what do you think the answer would be if we added 5 + true and 5 + false?

```
5 + true; // 51
5 + false; // 50
```

The null and false being 0, brings us to something called "falsey values". We'll get more into that when we talk about conditionals though.

Now let's look at the following expression

```
    5 + undefined; // NaN
```

So the result is NaN or **not a number**. We talked a little about this in the last video. The reason is because NaN is the result of a failed number operation. If we run undefined through the Number() method, we also get NaN. If we try and add NaN like this, we also get NaN.

```
5 + NaN; // NaN
```

Want to see something really strange?

```
NaN == NaN // false
```

You read that right. NaN is NOT equal to NaN. Kind of mind blowing. This is because not all NaN numbers are created equal. You can read more about it here, but just know that this will always equate to false.

Now there may be times when you need to check for a NaN value. Again, I know we haven't got into functions or conditionals yet, but we do have a function called **isNaN** that we can use like this.

```
  isNan(NaN); // true
```

However, this will return true for ANYTHING that is "not a number", such as

```
isNan('Hello') // true
```

If you really want to check for the specific value of NaN, you can use the isNan method on the Number object like this

```
Number.isNan(NaN) // true
Number.isNaN('Hello') //false
```