Using Operators with Different Types

Learn what happens when we try to add, subtract, and divide strings. We'll cover how JavaScript handles 'abc' + 20. Learn what NaN is. We'll also discuss type coercion, a very import concept in JavaScript.

Different Variable Types

We've shown these operators being used on numbers only. They can also be used with other variable types.

```javascript
let string1 = 'Hello ';
let string2 = 'there!';
console.log(string1 + string2); // -> Hello there!
```

String addition just joins the strings together into a new, larger string.

`NaN`

What happens if we try to use the other operators with strings? We get a new value: `NaN`.

```javascript
let string1 = 'Hello ';
let string2 = 'there!';

console.log(string1 - string2); // -> NaN
console.log(string1 * string2); // -> NaN
console.log(string1 / string2); // -> NaN
console.log(string1 % string2); // -> NaN
```

`NaN` is technically a number type variable, but it stands for Not-a-Number. It's meant to show us that we messed up our math and attempted to perform some nonsensical operation.

Adding two strings to join them together makes sense. Dividing, multiplying, subtracting, or finding remainders does not. When we try, the JavaScript engine is nice enough to tell us that our mathematical operation failed. We get Not-a-Number.

Type Coercion

We can try this with other variable types as well. We usually get `NaN`, but sometimes, something a little unexpected happens.

```
console.log(undefined - null); // -> NaN
console.log('abc' * undefined); // -> NaN
console.log(9 % null); // -> NaN
console.log(null + 4); // -> 4
```

Notice that the last line above logged `4`. We're adding `null`, something that is not a number type, to a number. We get 4 back.

When we to use different types in an operation, JavaScript will do its best to get us a real value. It will try to get us something that is not `NaN`. If it thinks it can safely convert something to another type and then perform its operation, it'll do so. This is called *type coercion*.

In this case, it converts `null` to 0. Then it adds 0 and 4.

Here are some more cases of type coercion.

```
console.log(null - 7);      // -> -7
console.log('abc' + null); // -> abcnull
console.log(20 + 'abc');    // -> 20abc
console.log(9 + true);      // -> 10
```

Here are the conversions in each of the lines above.

1. `null` -> `0` (`null` -> number)

2. `null` -> `'null'` (`null` -> string)
3. `20` -> `'20'` (number -> string)
4. `true` -> `1` (boolean -> number)

This often gets tricky and confusing and unfortunately, there's no great way to deal with this. You can google JavaScript type coercion to see more examples.

We'll move on to other types of operators in the next section.

Quiz

Feel free to test your understanding.

1

What will this code print?

```
let str1 = 'Hello';
let str2 = 'there!';
console.log(str1 + str2);
```

- A) Hello
- B) there!
- C) Hello there!
- D) Hellothere!
- E) NaN

2

What will this code print?

```
let str = 'Hello';
let bool = true;
console.log(str + bool);
```

- ○ A) Hello
- ○ B) true
- ○ C) Hellotrue
- ○ D) Hello true
- ○ E) NaN

3

What will this code print?

```
let str = 'Hello';
let bool = 0;
console.log(str - bool);
```

- ○ A) Hell
- ○ B) Hello0
- ○ C) 0

D) NaN

E) error

4

What will this code print? (This may not be intuitive! The result is due to a coercion. You may want to google what number `true` coerces to.)

```
let bool = true;
let num = 0;
console.log(bool - num);
```

A) true

B) 0

C) 1

D) NaN

E) error

5

What will this code print?

```
let num1 = 2;
let num2 = 5;

console.log(num2 % num1);
```

- ◯ A) 1
- ◯ B) 2
- ◯ C) 3
- ◯ D) 4
- ◯ E) 5

CHECK ANSWERS