It's important to understand that JavaScript is able to use variables in conditions - even without comparison operators.

This is kind of obvious, if we consider a boolean variable, for example:

```
let isLoggedIn = true;
if (isLoggedIn) {
    ...
}
```

Since if just wants a condition that returns true or false, it makes sense that you can just provide a boolean variable or value and it works - without the extra comparison (`if (isLoggedIn === true)` - that would also work but is redundant).

Whilst the above example makes sense, it can be confusing when you encounter code like this for the first time:

```
let userInput = 'Max';
if (userInput) {
    ... // this code here will execute because 'Max' is "truthy" (all strings but empty strings are)
}
```

JavaScript tries to coerce ("convert without really converting") the values you pass to if (or other places where conditions are required) to boolean values. That means that it tries to interpret `'Max'` as a boolean - and there it follows the rules outlined in the previous lecture (e.g. `0` is treated as `false`, all other numbers are treated as `true` etc.)

It's important to understand that JavaScript doesn't really convert the value though.

`userInput` still holds `'Max'` after being used in a condition like shown above - it's not converted to a boolean. That would be horrible because you'd invisibly lose the values stored in your variables.

Instead,

```
if (userInput) { ... }
```

is basically transformed (behind the scenes) to

```
if (userInput === true) {
```

And here, the `=== operator` generates and returns a boolean. It also doesn't touch the variable you're comparing - `userInput` stays a string. But it generates a new boolean which is temporarily used in the comparison.

And that's exactly what JavaScript automatically does when it finds something like this:

```
if (userInput) { ... }
```