JS Conditions and Booleans

Learning Objectives

- using conditions to control the program flow
- understanding what booleans and truthy/falsy values are
- working with comparison and logical operators
- writing ternary expressions

Boolean Values

A boolean value, named after George Boole, only has two states. It can either be **true** or **false**. Booleans are often used in conditional statements which can execute different code depending on their value.

Truthy and Falsy Values

Sometimes you want to have a condition depending on another type of value. JavaScript can transform any value into a boolean with *type coercion*. That means that some values act as if they were true and others as if they were false: *Truthy* values become true, *falsy* values become false.

- *truthy* values:
  - non zero numbers: `1`, `2`, `-3`, etc.
  - non empty strings: `"hello"`
  - `true`
- *falsy* values:
  - `0` / `-0`
  - `null`
  - `false`
  - `undefined`
  - empty string: `""`

Comparison Operators

Comparison operators produce boolean values by comparing two expressions:

Operator	Effect
A `===` B	strict equal: is `true` if both values are equal (including their type).
A `!==` B	strict not equal: is `true` if both values are not equal (including their type).
A `>` B	strictly greater than: is `true` if A is greater than B.
A `<` B	strictly less than: is `true` if A is less than B.

Operator	Effect
A `>=` B	greater than or equal: is `true` if A is greater than or equal B.
A `<=` B	less than or equal: is `true` if A is less than or equal B.

> 💡 You might notice that JavaScript uses three equal signs (`===`) to check for equality. This can seem very strange at first.
>
> - `=` (`const x = 0`) is the assignment operator and has nothing to do with comparison.
> - `==` and `!=` are non-strict equality operators. You should **avoid them 99% of the time**. Non-strict equality tries to use type coercion to convert both values to the same type: `"3" == 3` is `true`, which is seldomly what you want.
> - `===` and `!==` are strict equality operators. **This is what you need almost always**. Strict equality checks if type *and* value are the same: `"3" === 3` is `false`.

Logical Operators

Logical operators combine up to two booleans into a new boolean.

Operator	Effect
`!`A	`not`: flips a `true` value to `false` and vice versa.
A `||` B	`or`: is `true` if either A `or` B is true.
A `&&` B	`and`: is `true` if both A `and` B is true.

> 💡 You can combine logical operators with brackets to define which operator should be evaluated first, e.g:
>
> - `(A || B) && (C || D)`
> - `!(A || B)`

> 💡 Be careful when using `&&` or `||` with non-boolean values. They actually return one of the original values. That can be useful, but can also quickly lead to confusion. This behaviour is called short-circuit evaluation and is a more advanced topic.
>
> - `"some string" || "some other string"` evaluates to `"some string"`
> - `0 || 100` evaluates to `100`
> - `null && "yet another string"` evaluates to `null`

Control Flow: `if / else`

With an if statement we can control whether a part of our code is executed or not, based on a condition.

```
const isSunShining = true;

if (isSunShining) {
  // code that is executed only if condition "isSunShining" is true
}
```

The else block is executed only if the condition is `false`.

```
const isSunShining = false;

if (isSunShining) {
  // code that is executed only if condition "isSunShining" is true
} else {
  // code that is executed only if condition "isSunShining" is false
}
```

The condition expression between the `()` brackets can be composed of logical or comparison operators as well. You can distinguish between more cases by chaining `else if` statements:

```
if (hour < 12) {
  console.log("Good Morning.");
} else if (hour < 18) {
  console.log("Good afternoon.");
} else if (hour === 24) {
  console.log("Good night.");
} else {
  console.log("Good evening.");
}
```

If the condition is not a boolean, it is converted into one by type coercion. This can be used to check whether a value is not 0 or an empty string:

```
const name = "Alex";
if (name) {
  console.log("Hi " + name + "!"); // only executed if name is not an
empty string
}
```

Ternary Operator: `? :`

With if / else statements whole blocks of code can be controlled. The ternary operator can be used if you want to decide between two *expressions*, e.g. which value should be stored in a variable:

```
const greetingText = time < 12 ? "Good morning." : "Good afternoon.";
```

The ternary operator has the following structure:

```
condition ? expressionIfTrue : expressionIfFalse;
```

If the condition is true, the first expression is evaluated, otherwise the second expression. The ternary operator can be used to decide which function should be called:

```
isUserLoggedIn ? logoutUser() : loginUser();
```

It can also distinguish which value should be passed as an argument to a function:

```
moveElement(xPos > 300 ? 300 : xPos); // the element can't be moved
further than 300.
```

> ❗ The operator can only distinguish between two *expressions* like values, math / logical operations or function calls, not between *statements* like variable declarations, if / else statements or multi-line code blocks.

Advanced: The strangeness of boolean coercion and making use of non-strict equality

▶ 😱 This is an advanced topic and not important for the challenges. Click to expand if you're curious.

Assume you want to check if a variable has a useful value for us to work with. `if(variable)` does in fact not check if `variable` is defined but rather if it is truthy. Take a look at these examples:

- `if(undefined)` → falsy, won't execute
- `if(null)` → falsy, won't execute
- `if("")` → falsy, won't execute, but might still be a useful variable (e.g. when user clears an input field)
- `if(0)` → falsy, won't execute, but might still be a useful variable (e.g. when user wants to set the volume to `0`)
- `if(" ")` → truthy, will execute
- `if(-1)` → truthy, will execute

It's useful to define a variable as not having a value when it's `undefined` or `null`. We can check for that like this:

```
if (variable != null) {
  console.log('This will be logged even if variable is 0 or ""');
}
```

This is one of the rare valid use cases for non-strict comparison (`!=` instead of `!==`).

JavaScript tries to coerce the compared values into the same type. And just like `"3" == 3` is `true`, `undefined == null` is also `true`. This also works with `!=` instead of `==`.

> ⚠ Remember that this is an exception for using non-strict equality. **Strict equality should otherwise always be preferred.**

Resources

Operators

MDN Comparison Operators

MDN Logical Operators

if / else statements

MDN about if else

Ternary Operator

MDN Ternary Operator