{ Boolean Logic. }

Objectives:

By the end of this chapter, you should be able to:

- Write conditional logic using boolean operators
- List all of the falsey values in JavaScript
- Use if/else and switch statements to include conditional logic in your JavaScript code
- Explain the difference between `==` and `===` in JavaScript
- Convert between data types explicitly in JavaScript

Boolean Logic

An essential part of writing programs is being able to execute code that depends on certain conditions. There are many different examples when you'd want to conditionally execute code. Here are just a few:

- You want the navigation bar on your website to look different based on whether or not someone is logged in
- If someone enters their password incorrectly, you want to let them know; otherwise, you want to log them in
- You're building a tic-tac-toe game, and want to know whether it's X's turn or O's turn
- You're building a social network and want to keep person A from seeing person B's profile unless the two of them are friends

And so on, and so on. It's very hard to write any kind of interesting software without making use of conditionals and boolean logic.

So let's talk about how to write conditional logic in JavaScript. To do so, we'll make use of booleans (`true` and `false`), along with `if` statements and `switch` statements.

An `if` statement looks something like this:

```
let instructor = "Elie";

// we begin with an "if" statement followed by a condition in () and a block of code
inside of {}
if(instructor === "Elie") {
    console.log("Yes!");
} else {
    console.log("No");
}
```

Notice that we used a `===` instead of `=`. Anytime that we use more than one equals operator (we can either use `==` or `===`) we are doing what is called **comparison** (comparing values). When we use a single equals operator `=`, we are doing what is called **assignment** (setting a variable equal to some value).

This first example might appear a little strange, because the condition inside of the `if` statement (`instructor === "Elie"`) will always return true! Here's another example for you to consider. Run this code a couple of times and try to get both messages to log to the console based on what you enter into the prompt:

```
let favoriteFood = prompt("What's your favorite food?");

if(favoriteFood === "pizza") {
    console.log("Woah! My favorite food is pizza too!");
} else {
    console.log("That's cool. My favorite food is pizza.");
}
```

(If you're having trouble getting the first message to log, it may help to know that strings are *case-sensitive*: "pizza" and "PIZZA" are different strings.)

Now, what's the difference between `==` and `===`, you ask? Great question! We'll get to that down below. For now, though, it might be helpful to play around with these operators in the Chrome console, and see if you can come up with a guess as to how these operators behave differently.

```
let number = 55;

// we begin with an "if" statement followed by a condition in () and a block of code
inside of {}
```

```
if(number == "55") {
    console.log("Yes!");
} else {
    console.log("No");
}
```

Difference between `==` and `===`

In JavaScript we have two different operators for comparison: the double and triple equals. Both operators check whether the two things being compared have the same value, but there's one important difference. `==` allows for type coercion of the values, while `===` does not. So to understand the difference between these operators, we first need to understand what is meant by type coercion.

Consider the following examples:

```
// 1.
5 + "hi"; // "5hi"

// 2.
if ("foo") {
  console.log("this will show up!");
}

// 3.
if (null) {
  console.log("this won't show up!");
}

// 4.
+"304"; // 304
```

Let's figure out what's happening in each of these examples. In the first one, you've asked JavaScript to add a number and a string. In a lot of programming languages, this would throw an error, but JavaScript is more accomodating! It evaluates the expression 5 + "hi" by first *coercing* 5 into a string, and then interpreting the "+" operator as string concatenation. So it combines the string "5" with the string "hi" into the string "5hi".

The next two examples show a similar sort of *coercion*. JavaScript expects the values inside of parentheses that come after the keyword `if` to be booleans. If you pass in a value which is not a boolean, JavaScript will *coerce* the value to a boolean according to

the rules for truthy/falsey values defined below. Since "foo" is not a falsey value, it will be coerced to `true`, which is why the second example logs something to the console. `null`, however, is a falsey value, so it gets coerced to false and nothing shows up in the third example.

The last example shows a very common way to coerce a stringified number back into a number. By prefacing the string with the plus sign, JavaScript will perform a *coercion* on the value and convert it from a string value to a number value.

In essence, then, coercion is just the process of converting a value from one type to another. JavaScript uses coercion pretty liberally among programming languages, so if you don't understand how coercion in JavaScript works, it can be easy to introduce bugs into your code.

But what does all of this have to do with `==` and `===`? Let's look at some examples:

```
5 == "5"; // true
5 === "5"; // false
"true" === true; // false
"true" == true; // false
true == 1; // true
true === 1; // false
undefined === null; // false
undefined == null; // true
```

What's going on here? Let's deal with the expressions involving `===` first. As you can see, the expressions `5 === "5"`, `"true" === true`, `true === 1`, and `undefined === null` all evaluate to `false`. In some sense, perhaps this shouldn't be so surprising: none of the values being compared are the same! One way to think about this is to recall the types of the primitives being compared. In the first case, we're comparing a number to a string; in the second case, a boolean and a string; in the third case, a boolean and a number; and in the last case, `undefined` and `null`. How can these values be the same when the primitives involved aren't even of the same type??

From the above examples, you can see that the `==` operator is a little less strict (in fact, `===` is sometimes referred to as the "strict" equality operator, while `==` is sometimes

referred to as the "loose" equality operator). The reason that comparisons like `5 == "5"` evaluate to true is because `==` allows for type coercion!

But what gets coerced? Does `5` become `"5"` or does `"5"` become `5`? In this case, according to the specification the string gets coerced into a number, not the other way around. This might seem like an unimportant detail, but there are a couple of gotchas in the way coercion works that can be confusing when you first encounter them.

For example, it might seem like `"true" == true` should evaluate to `true`, since `"true"` is a truthy value! But in fact, what actually happens is that the boolean `true` gets coerced to a number (1), and then `"true"` is compared to `1`, which returns false. (This is also why `true == 1` evaluates to `true`.)

It's less important to memorize these rules for how coercion works with `==` than to recognize that `==` allows for coercion while `===` doesn't. If you don't want to have to think about coercion in your comparisons, stick to `===`.

If / else statements with other comparators

We previously saw what an `if` statement looks like. Let's examine this a bit more:

```
let x = 4;
if(x <= 5){
    console.log("x is less than or equal to five!");
} else {
    console.log("x is not less than or equal to five!");
}
```

We saw before that we can use `==` or `===` to compare values. We can also check for inequality, using

`<` - less than,

`<=` - less than or equal to,

`>` - greater than,

`>=` - greater than or equal to,

`!=` - not equal (loose), and

`!==` - not equal (strict).

Falsey Values

As we've alluded to already, another essential concept to understand in JavaScript is that some values (aside from `false`) are actually false as well, when they're used in a context where JavaScript expects a boolean value! Even if they do not have a "value" of false, these values will be translated (or "coerced") to false when evaluated in a boolean expression.

In JavaScript there are 6 falsey values:

- `0`
- `""`
- `null`
- `undefined`
- `false`
- `NaN` (short for not a number)

If you ever want to determine if a value is truthy or falsey, you can prefix it with `!!`. `!!` explicitly coerces a value into its boolean form.

What do these values return?

```
!!false
!!-1
!!-0
!![]
!!{}
!!""
!!null
```

You can read more about these here

!, || and &&

In our conditions (and assignments) we can use certain logical operators to write more complex statements. Here are some other useful operators:

`!` - the **not** operator, which flips the boolean value (`!true === false`). `!!` simply applies this operator twice, so `!!true === true`, and `!!false === false`.

`||` - the **or** operator, which in a boolean context returns true if either condition is true

`&&` - the **and** operator, which in a boolean context returns true if both conditions are true

You can read more about logical operators here.

Ternary Operators

Another common pattern you'll see in JavaScript is the use of *ternary operators*. A ternary operator is another way to write an if / else statement. For example, consider the following code:

```
let guess = prompt("Guess what number I'm thinking of!");

if (guess === "7") {
    console.log("Correct!");
} else {
    console.log("Incorrect!");
}
```

This code asks the user to guess a number. It logs "Correct!" if the user guesses 7, and "Incorrect!" otherwise.

If you wanted to, you could refactor this code using a ternary operator:

```
let guess = prompt("Guess what number I'm thinking of!");

// here's our first ternary
```

```
guess === "7" ? console.log("Correct!") : console.log("Incorrect!");
```

In general, a ternary operator has the form:

```
expression ? pathIfTrue : pathIfFalse
```

This is equivalent to:

```
if (expression) {
    pathIfTrue
} else {
    pathIfFalse
}
```

You can also store the value of a ternary in a variable:

```
let num = 3;
let comparison = num > 0 ? "Greater than 0" : "Less than or equal to 0";
comparison; // this will equal "Greater than 0", since 3 > 0.
```

Ternary operators can help you write less code, but be careful using them, as they also tend to make your code more difficult to read and reason about.

If / else if / else

Sometimes you may have more than two conditions to check. In this case, you can chain together multiple conditions using `else if`. Here's an example:

```
let number = prompt("What's your favorite number?");

if (number >= 1000) {
    console.log("Woah, that's a big number!");
} else if (number >= 0) {
    console.log("That's a cool number.");
} else {
    console.log("Negative numbers?! That's just bananas.");
}
```

Try this out with a few different numbers and see what happens!

Switch statements

Another way to write conditional logic is to use a `switch` statement. While these are used less frequently, they can be quite useful when there are multiple conditions that can be met. Notice that each `case` clause needs to end with a `break` so that we exit the `switch` statement. Here is an example:

```
let feeling = prompt("How are you feeling today?").toLowerCase();
// what do you think the .toLowerCase does at the end?

switch(feeling){
    case "happy":
        console.log("Awesome, I'm feeling happy too!");
        break;
    case "sad":
        console.log("That's too bad, I hope you feel better soon.");
        break;
    case "hungry":
        console.log("Me too, let's go eat some pizza!");
        break;
    default:
        console.log("I see. Thanks for sharing!");
}
```

Modulus Operator

Another very useful operator to use is the modulus operator which returns the remainder of a number when dividing by another number. The operator we use is `%`.

You'll often see this operator used when checking to see if a number is even or odd.

```
5 % 3 === 2 // true (the remainder when five is divided by 3 is 2)

let num = prompt("Please enter a whole number");
if ( num % 2 === 0 ) {
    console.log("the num variable is even!")
} else if ( num % 2 === 1) {
    console.log("the num variable is odd!")
} else {
    console.log("Hey! I asked for a whole number!");
}
```

A small note on using `let` inside of conditional logic

As you work with conditional statements more, you may come across a situation like this:

```
let num = prompt("Please enter a whole number");
if ( num % 2 === 0 ) {
    let message = "That is an even number!"
} else {
    let message = "That is an odd number!"
}

console.log("You typed", message)
```

Unfortunately, when you run this code, you will see: `Uncaught ReferenceError: message is not defined`. In our previous example, we just printed a string to the console, but here we are declaring a variable based on the condition and trying to use it outside of our `if`/`else` statement.

The reason here is because when you use the `let` (or `const`) keyword inside of a block, the variable you create can only be accessed inside of that block. You can define a block as a set of curly braces for an `if`, `else`, and `else if` statement.

If you wanted to create a variable based on a condition and change the value of the variable, make sure to declare it outside of your `if`/`else` statement. Here's what that might look like:

```
let num = prompt("Please enter a whole number");
let message;

if ( num % 2 === 0 ) {
    message = "That is an even number!"
} else {
    message = "That is an odd number!"
}

console.log("The message is:", message)
```

Notice here, we are defining a `message` variable with a value of `undefined` and then assigning a value to it depending on the condition.