Review: Conditionals

(https://github.com/learn-co-curriculum/phase-1-review-conditionals) (https://github.com/learn-co-curriculum/phase-1-review-conditionals/issues/new)

Learning Goals

- Review what constitutes an expression in JavaScript
- Organize code using block statements
- Review the difference between truthy and falsy values
- Review how to use conditional statements

Introduction

If you're hungry, you make a sandwich. If the traffic light is green, you press the gas pedal. If your rent is due, then you pay your rent. This breaks down into a lot of conditional choices:

- `if` hungry → make sandwich.
  - `else` → don't make sandwich.
- `if` light is green → press gas pedal.
  - `else` → press brake pedal.
- `if` it's the first of the month → pay rent.
  - `else` → don't pay rent.

Writing code involves the same type of logic — we only want an action to happen *if* a certain condition is met. In the programming world, this is called **control flow** because, well, it helps *control* the *flow* of an application.

Before we dive into JavaScript's conditional structures, let's review a few concepts that provide the syntactic underpinnings.

Review What Constitutes an Expression in JavaScript

A JavaScript expression is **a unit of code that returns a value**. Primitive values are expressions because they resolve to a value:

```
9;
// => 9

('Hello, world!');
// => "Hello, world!"

false;
// => false
```

So are arithmetic and string operations. This code resolves to the number `64`:

```
8 * 8;
// => 64
```

And this resolves to the string `"Hello, world!"`:

```
'Hello, ' + 'world!';
// => "Hello, world!"
```

Ditto for comparison and assignment operations. This comparison resolves to the boolean `true`:

```
2 > 1;
// => true
```

Variable declarations are NOT expressions...

```
const greeting = "Hello!";
let answer;
```

...but variable assignments ARE, resolving to the assigned value (`42` , in this case):

```
answer = 42;
// => 42
```

Finally, variable lookups are also expressions, resolving to the value contained in the variable:

```
const fullName = 'Ada Lovelace';

fullName;
// => "Ada Lovelace"
```

Organize Code Using Block Statements

A block statement is a pair of curly braces (`{ }`) used to group JavaScript statements. It plays a role in conditional statements, loops, and functions.

```
{
    ('This line is a JavaScript statement nested inside a block statement!');

    // This is also a statement nested inside a block:
    5 * 5 - 5;

    // And so are these:
    const weCan = 'group multiple statements';

    const suchAs = 'these variable declarations';

    const insideA = 'block statement.';
}
// => 20
```

Block statements return the value of the *last evaluated expression* inside the curly braces. Remember, the variable declarations are *not* expressions, so the value of `5 * 5 - 5` is returned.

> **Note**: The statement above *implicitly* returns 20 (the value returned by `5 * 5 - 5`, when evaluated). Functions, which we will discuss in an upcoming lesson, also contain all of their code inside curly braces, but for functions, we need to *explicitly* use the word `return` to tell JavaScript what we want the return value to be (if we want one at all). Just remember that the *implicit return is something unique to block statements* like the ones we use for `if...else` and loop statements.

Review the Difference Between Truthy and Falsy Values

Truthiness and falsiness indicate what happens when the value is converted into a boolean. If, upon conversion, the value becomes `true`, we say that it's a **truthy** value. If it becomes `false`, we say that it's **falsy**.

In JavaScript, the following values are **falsy**:

- `false`
- `null`
- `undefined`
- `0`
- `NaN`
- An empty string (`''`, `""`)

Every other value is truthy.

To check whether a value is truthy or falsy in our browser's JS console, we can pass it to the global `Boolean` object, which converts the value into its boolean equivalent:

```
Boolean(false);
// => false

Boolean(null);
// => false
```

```
Boolean(undefined);
// => false

Boolean(0);
// => false

Boolean(NaN);
// => false

Boolean('');
// => false

Boolean(true);
// => true

Boolean(42);
// => true

Boolean('Hello, world!');
// => true

Boolean({ firstName: 'Brendan', lastName: 'Eich' });
// => true
```

> **NOTE**: `document.all` is also falsy, but don't worry about it for now. (Or ever, really — it's an imperfect solution for legacy code compatibility.)
> Ready to put that killer new vocabulary to the test? Here we go!

Review How to Use Conditional Statements

JavaScript includes three structures for implementing code conditionally: *if statements*, *switch statements*, and *ternary expressions*.

`if` statement

To write a basic `if` statement, we use the following structure:

```
if (condition) {
    // Block of code
}
```

It consists of the `if` keyword followed by the condition to be checked in parentheses. After that comes a *block statement* (more commonly called a *code block*): one or more JavaScript expressions or statements enclosed in `{}` . The *code block* contains the code we want to execute *if* the condition returns a truthy value:

```
const age = 30;

let isAdult;

if (age >= 18) {
    isAdult = true;
}
// => true

isAdult;
// => true
```

If the condition returns a **falsy** value, do nothing:

```
const age = 14;

let isAdult;

if (age >= 18) {
    isAdult = true;
```

```
}

isAdult;
// => undefined
```

`else`

Often we want to run one block of code when the condition returns a `truthy` value and a *different* block of code when it returns a `falsey` value. To do this, we use an `else` clause:

```
const age = 14;

let isAdult;

if (age >= 18) {
    isAdult = true;
} else {
    isAdult = false;
}
// => false

isAdult;
// => false
```

Note that the `else` clause **does not take a condition** — if the condition for the `if` returns a falsey value, we want the `else` code block to run **no matter what**. This means that exactly one of the code blocks will *always* run.

Ternary Expressions

Recall that this is the exact situation where we can use a ternary expression. Here's what the code above would look like using the ternary operator:

```
const age = 26;

let isAdult;

age >= 18 ? (isAdult = true) : (isAdult = false);
// => true

isAdult;
// => true
```

Here, we assign `isAdult` as `true` if the condition returns a truthy value and as `false` otherwise, exactly like the version using `if`.

Remember that a ternary is an *expression* — it returns a *value*. What this means is that we can simplify the code above a bit and assign the *result* of the ternary directly to a variable:

```
const age = 26;
const isAdult = age >= 18 ? true : false;

isAdult;
// => true
```

Advanced: What is the ternary above doing? Basically, it's saying: "when the conditional code returns `true`, return `true`, and when the conditional code returns `false`, return `false`." Sounds a bit redundant, doesn't it? When the return values are `true` and `false` as in the example above, you actually don't need to use a ternary — or an `if...else` — at all! This is because **the conditional is an expression as well**. The return value of `age >= 18` is a *Boolean value* (`true` or `false`), so it can be assigned directly to our `isAdult` variable:

```
const age = 6;
const isAdult = age >= 18;

isAdult;
//=> false
```

The ternary (or `if...else`) is only necessary if the desired return value is something other than a Boolean:

```
const age = 20;
const ageMessage = age >= 18 ? "Congratulations! You're an adult!" : "Enjoy your childhood while it lasts!";

ageMessage;
//=> "Congratulations! You're an adult!"
```

If it helps you visualize what's going on, you can wrap the condition, the expressions, or the entire ternary in parentheses:

```
const age = 17;

const isAdult = (age >= 18) ? true : false;

const canWork = (age >= 16) ? (1 === 1) : (1 !== 1);

const canEnlist = (isAdult ? true : false);

isAdult;
// => false

canWork;
// => true

canEnlist;
// => false
```

> **Top Tip:** Be careful to not overuse the ternary operator. It's fine for slimming down a simple `if...else`, but be conscious of how easy your code is to understand for an outsider. Remember, you generally write code once, but it gets read (by yourself and others) **far** more than once. The ternary is often more difficult to quickly interpret than a regular old `if...else`, so make sure the reduction in code is worth any potential reduction in readability.

`else if`

We've discussed the case where our condition is *binary* (one code block executes if the conditional returns true and a second executes otherwise), but sometimes we need to check multiple conditions. We can handle this situation by using one or more `else if` clauses.

Let's say that instead of just deciding whether the passed-in `age` meets the criterion for `isAdult`, we want to add in some other examples of adulthood (in American society, at least): `canWork`, `canEnlist`, and `canDrink`. 16-year-olds can legally work; 18-year-olds can do what 16-year-olds can do **plus** they can enlist and they are legal adults; 21-year-olds can do what 16- and 18-year-olds can do **plus** they can drink (at the federally set minimum age).

Here's how we can handle that using `else if` clauses:

```
const age = 20;

let isAdult, canWork, canEnlist, canDrink;

if (age >= 21) {
    isAdult = true;
    canWork = true;
    canEnlist = true;
    canDrink = true;
} else if (age >= 18) {
    isAdult = true;
    canWork = true;
    canEnlist = true;
} else if (age >= 16) {
    canWork = true;
}
// => true

isAdult;
// => true
```

```
canWork;
// => true

canEnlist;
// => true

canDrink;
// => undefined
```

Any time you use an `if...else if` construction, **at most one code block will be executed**. As soon as one of the conditions returns a truthy value, the attached code block runs and the conditional statement ends. In the example above, we have not included an `else` statement so, if none of the conditions is truthy, no code blocks will be run. If we had included an `else` clause, exactly one code block would be run.

Nested `if` Statements

You may have noticed that there is some redundancy in the example above: three of the four variables appear in more than one of the conditions. In this circumstance, we can streamline our code a bit by using nested conditional statements:

```
const age = 17;

let isAdult, canWork, canEnlist, canDrink;

if (age >= 16) {
  canWork = true;

  if (age >= 18) {
    isAdult = true;
    canEnlist = true;

    if (age >= 21) {
      canDrink = true;
    }
  }
```

```
}

canWork;
```

The first `if` condition checks for the "base level" of adulthood (`age >= 16`), and each subsequent nested `if` "adds on." Note that each inner `if` statement is nested **inside** the code block of the one before. This means that the inner `if` statements will only execute if the outer ones are truthy. This makes sense: if age is less than 16, we're done — there's no need to check the remaining conditions because we know they have to be false as well. Otherwise JavaScript will keep checking each subsequent condition until it either comes to one that is false or finishes running all the code blocks.

While nested `if` s are more efficient than `if...else if` s for handling overlapping categories, they are also more difficult to read. An `if...else if` construction will always work. You should consider the tradeoff of readability vs. efficiency in deciding which construction to use.

switch

Let's say we have a program that includes a variable containing a person's food order and we want to create a variable containing the appropriate ingredients. Using an `if...else if` construction, that might look like this:

```
const order = 'cheeseburger';

let ingredients;
if (order === 'cheeseburger') {
    ingredients = 'bun, burger, cheese, lettuce, tomato, onion';
} else if (order === 'hamburger') {
    ingredients = 'bun, burger, lettuce, tomato, onion';
} else if (order === 'malted') {
    ingredients = 'milk, ice cream, malted milk powder';
} else {
    console.log("Sorry, that's not on the menu right now.");
}
```

As we can see, there's quite a bit of repetition here: we always test `order` and we always compare with `===`. This is a pretty common selection need. It's so standard that the `switch` statement was created to enable us to streamline our code. Here's the `switch` version of the code above:

```
const order = 'cheeseburger';

let ingredients;

switch (order) {
    case 'cheeseburger':
        ingredients = 'bun, burger, cheese, lettuce, tomato, onion';
        break;
    case 'hamburger':
        ingredients = 'bun, burger, lettuce, tomato, onion';
        break;
    case 'malted':
        ingredients = 'milk, ice cream, malted milk powder';
        break;
    default:
        console.log("Sorry, that's not on the menu right now.");
        break;
}
```

The JavaScript engine compares the value passed in to the `switch` statement (here, `order`) against each of the `case` values *using strict equality* (`===`). When a match is found, the statements nested under that `case` are executed. In this example, by using the `switch` statement, we avoid the need to repeat the `if (order === _____)` line for each possibility.

We can also assign the same set of statements to multiple cases. In the following example, if the `age` variable contains any number between `13` and `19`, the `isTeenager` variable will be set to `true`. If it contains anything other than a number between `13` and `19`, none of our `case`s will hit, and it will end up at the `default`, which sets `isTeenager` to `false`:

```
const age = 15;

let isTeenager;

switch (age) {
    case 13:
    case 14:
    case 15:
    case 16:
    case 17:
    case 18:
    case 19:
        isTeenager = true;
        break;
    default:
        isTeenager = false;
}
```

The `default` and `break` keywords are both optional in basic `switch` statements, but useful. In more complicated statements, they become necessary to ensure the correct flow.

default

The `default` keyword is similar to the `else` clause in an `if...else` construction. It specifies a set of statements to run after all of the `switch` statement's `case` s have been checked. However, it is different from an `else` in that it will run unless the engine hits a `break` in one of the `case` statements. If you only want one code block in your `switch` statement to execute, you should always include the `break` keyword.

break

In the previous example, `break` is used to stop the `switch` statement from continuing to look at case statements once it finds a match. If we didn't have the `break` keyword inside the switch statement, it would get to a match at `case 15` , continue on through to `case 19` and set `isTeenager` to true, as desired. The code would then continue, however, executing the `default` case and resetting `isTeenager` to false. To

keep that from happening, we can use `break` to tell the JavaScript engine to stop executing the `switch` statement once it has found a match. You will often see switch statements where `break` is used in every case as a way to ensure there is no unexpected behavior from multiple cases executing.

Advanced: Sometimes we *want* to potentially match multiple cases, and we will need to leave out `break` in order to do this. We can refactor the `if...else if...else` example we saw earlier as a more compact, less repetitious `switch` statement. To make it work, we will employ a neat little trick: we'll use comparisons for our `case` statements instead of a simple value.

```
const age = 25;

let isAdult, canWork, canEnlist, canDrink;

switch (true) {
    case age >= 21:
        canDrink = true;
    case age >= 18:
        isAdult = true;
        canEnlist = true;
    case age >= 16:
        canWork = true;
}
// => true

isAdult;
// => true

canWork;
// => true

canEnlist;
// => true
```

```
canDrink;
// => true
```

We specified `true` as the value to `switch` on. All of our `case` s are *comparison expressions* that return `true` or `false` . Therefore, if a comparison returns `true` , its statements will be run. Because we did not include any `break` statements, once *one* case statement matches, all subsequent statements will execute. This is what we want here: if `age` is greater than 21, it's also greater than 18 and 16, so we want *all* the assignments to be made.

If we set `age` to `20` in the above example, the first `case` , `age >= 21` , returns `false` and the assignment of `canDrink` never happens. The engine then proceeds to the next `case` , `age >= 18` , which returns `true` , assigning the value `true` to `isAdult` and `canEnlist` . Since it encounters no `break` statement, it then proceeds to the last case statement where `canWork` is set to true as well.

Conclusion

You now have three different types of conditional statements available to you: the `if` statement, the `ternary` expression, and the `switch` statement. The `if` statement is the one you will use most often — in fact, you can *always* construct your conditional code using some combination of `if` , `else if` , and `else` . It may not be the most efficient way to write the code, but it will always do the trick.

As a rule of thumb, you may find it makes sense to start with `if` statements and, once you've got the code working, consider refactoring it to use a ternary or switch statement if they're better suited for what you need to do.

Resources

- MDN
  - **Expressions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions)**
  - **Block statement (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block)**
  - **Truthy (https://developer.mozilla.org/en-US/docs/Glossary/Truthy)** and **falsy (https://developer.mozilla.org/en-US/docs/Glossary/Falsy)**
  - **Conditional statements (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#Conditional_statements)**
  - **`if...else` statement (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else)**
  - **Conditional (ternary) operator (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)**

- `switch` statement (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch)