Boolean Logic in JavaScript

Taken from a series of articles on Medium

Part 1: Boolean Operators & Truth Tables

Truthy & Falsy Values in JavaScript
Before we begin looking at logical expressions, which rely on the truthiness of statements to derive validity of the expression, we should have a solid understanding of what is *truthy* in JavaScript. Since JavaScript is loosely typed, values can be coerced into booleans to evaluate logical expressions. if conditions, `&&`, `||`, and the part of a ternary statement preceding the question mark (`_?_:_`) all coerce their evaluated values into booleans. (Note that this doesn't mean that they necessarily *return* a boolean from the operation.) The shortcut to knowing what is truthy is to know that there are only six *falsy* values — `false`, `null`, `undefined`, `NaN`, `0`, and `''` — and **everything else is truthy**. This means that `[]` and `{}` are both truthy, which tend to trip people up.

The Logical Operators
In formal propositional logic, there are only a few operators: negation, conjunction, disjunction, implication, and bicondition. These each have JavaScript equivalents: `!`, `&&`, `||`, `if (/* condition */) { /* then consequence */}`, and `===`, respectively. All other logical statements can be built up from these, including *exclusive or *(xor) and *if-then-else* (ternary) statements. We'll get to those in Part 2.

First, let's look at the **truth tables** for each of our basic operators. The truth tables tell us what the truthiness of an *expression* is based on the truthiness of its *parts*. For instance, the first row in the Negation truth table (below) should be read like this: "if statement A is True, then the expression !A is False." Truth tables are important because **if two expressions generate the same truth table, then those expressions are equivalent and can replace one another**.

Negation

A	!A
T	F
F	T

The **Negation** table is very straightforward. Negation is the only unary logical operator, meaning it acts on a single input. This means that `!A || B` should not be considered the same as `!(A || B)`. Parentheses act like grouping notation similar to what you'd find in mathematics.

Negating a simple statement is not difficult; the negation of "it is raining" is "it is **not** raining," and the negation of JavaScript's primitive `true` is, of course, `false`. However, negating complex statements or expressions is not so simple. What is the negation of "it is *always* raining" or `isFoo && isBar`? We will cover negating these and similar expressions in the next article.

Conjunction

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

The **Conjunction** table shows that the expression `A && B` is true only if *both* `A` and `B` are true. This should be very familiar from writing JavaScript.

Disjunction

A	B	A ∥ B
T	T	T
T	F	T
F	T	T
F	F	F

The **Disjunction** table should also be very familiar. A disjunction (logical `OR` statement) is true if *either* or *both* of `A` and `B` is true.

Implication

A	B	A → B
T	T	T
T	F	F
F	T	T
F	F	T

The **Implication** table is not as familiar. Since `A` *implies* `B`, A being true implies `B` is true. However, `B` can be true for reasons other than `A`, which is why the last two lines of the table are true. The only time implication is false is when `A` is true and `B` is false because then `A` doesn't imply `B`.

While `if` statements are used for implications in JavaScript, not all if statements work this way. Usually, we use if as a flow control, not as a truthiness check where the consequence also matters in the check. Here is the archetypical *implication* if statement:

```
function implication(A, B) {
  if (A) {
    return B;
  } else {
    /* if A is false, the implication is true */
    return true;
  }
}
```

Don't worry that this is somewhat awkward. There are easier ways to code implications. Because of this awkwardness, though, I will continue to use `→` as the symbol for implications throughout these articles.

Bicondition

A	B	A === B
T	T	T
T	F	F
F	T	F
F	F	T

The **Bicondition** operator, sometimes called if-and-only-if (IFF), evaluates to true only if the two operands, `A` and `B`, share the same truthiness value. Because of how JavaScript handles comparisons, the use of `===` for logical purposes should only be used on operands cast to booleans. That is, instead of `A === B`, we should use `!!A === !!B`.

A B	!A	A && B	A || B	A → B	A === B
T T	F	T	T	T	T
T F	F	F	T	F	F
F T	T	F	T	T	F
F F	T	F	F	T	T

The Complete Truth Table

Caveats

There are two big caveats to treating JavaScript code like propositional logic: **short circuiting** and **order of operations**.

Short circuiting is something that JavaScript engines do to save time by not evaluating something that will not change the output of the whole expression. The function `doSomething()` in the following examples is never called because no matter what it returned, the outcome of the logical expression wouldn't change:

```
// doSomething() is never called
false && doSomething();
true || doSomething();
```

Recall that conjunctions (`&&`) are true **only if both statements are true**, and disjunctions (`||`) are false **only if both statements are false**, so in each of these cases, after reading the first value, no more calculations need to be done to evaluate the logical outcome of the expressions. Because of this feature, JavaScript sometimes breaks logical commutativity. Logically `A && B` is equivalent to `B && A`, but you would break your program if you commuted `window && window.mightNotExist` into `window.mightNotExist && window`. That's not to say that the *truthiness* of a commuted expression is any different, just that JavaScript *may* throw an error trying to parse it.

JavaScript does not guarantee logical commutativity

The order of operations in JavaScript caught me by surprise because I was not taught that formal logic *had* an order of operations, other than by grouping and left-to-right. It turns out that many programming languages consider `&&` to have a higher precedence than `||`. This means that `&&` is grouped (not evaluated) first, left-to-right, and then `||` is grouped left-to-right. This means that `A || B && C` is *not* evaluated the same way as `(A || B) && C`, but rather as `A || (B && C)`[1].

> ***JavaScript does not group boolean operators strictly from left to right***

Fortunately, **grouping**, `()`, holds the topmost precedence in JavaScript, so we can avoid surprises and ambiguity by manually associating the statements we want evaluated together into discrete expressions. This is why many code linters prohibit having both `&&` and `||` within the same group.

Calculating Compound Truth Tables

Now that the truthiness of simple statements are known, the truthiness of more complex expressions can be calculated. To begin, count the number of variables in the expression and write a truth table that has 2^n rows. Next create columns for each of the variables and fill them with every possible combination of true/false values. I recommend filling the first half of the first column with `T` and the second half with `F`, then quartering the next column and so on until it looks like this:

A	B	C	
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

Then write the expression down and solve it in layers, from the innermost groups outward for each combination of truth values:

A	B	C	(!A || B) && (A || C)						
T	T	T	F	T	T	**T**	T	T	T
T	T	F	F	T	T	**T**	T	T	F
T	F	T	F	F	F	**F**	T	T	T
T	F	F	F	F	F	**F**	T	T	F
F	T	T	T	T	T	**T**	F	T	T
F	T	F	T	T	T	**F**	F	F	F
F	F	T	T	T	F	**T**	F	T	T
F	F	F	T	T	F	**F**	F	F	F

Notes

1. ↑ I would have expected `A() || B() && C()` to be evaluated the same way as `(A() || B()) && C()` in the following example, but it is not:

```
const A = () => {
  console.log('A');
  return true;
}
const B = () => {
  console.log('B');
```

```
  return true;
}
const C = () => {
  console.log('C');
  return false;
}

A() || B() && C()
// prints "A", returns true
(A() || B()) && C()
// prints "A C", returns false
```

Part 2: Logical Equivalencies

Photo by Chris Lawton on Unsplash

As stated in the previous article, expressions which produce the same truth table can be substituted for each other. This article covers several examples of Rules of Replacements that I often use. No truth tables are included here, but you can construct them yourself to prove that these rules are correct.

Double Negation

Logically, `A` and `!!A` are equivalent, so you can always remove a double negation or *add* a double negation to an expression without changing its truthiness. Adding a double-negation comes in handy when you want to negate part of a complex expression, perhaps using DeMorgan's Laws. The one caveat here is that in JavaScript, `!!` also acts to coerce a value into a boolean, which may be an unwanted side-effect.

```
A === !!A
```

Commutation

Any disjunction (`||`), conjunction (`&&`), or bicondition (`===`) can swap the order of its parts*.

```
(A || B) === (B || A)
```

```
(A && B) === (B && A)
```

```
(A === B) === (B === A)
```

Association

Disjunctions and conjunctions are binary operations, meaning they only operate on two inputs. While they can be coded in longer chains — `A || B || C || D` — they are implicitly associated from left to right — `((A || B) || C) || D`. The rule of association states that the order in which these groupings occur make *no difference* to the logical outcome.

```
((A || B) || C) === (A || (B || C))
```

```
((A && B) && C) === (A && (B && C))
```

Distribution

Association does not work across both conjunctions and disjunctions. That is, `(A && (B || C)) !== ((A && B) || C)`. In order to disassociate `B` and `C` in the previous example, you must *distribute* the conjunction — `(A && B) || (A && C)`.

This process also works in reverse. If you find a compound expression with a repeated disjunction or conjunction, you can un-distribute it, akin to factoring out a common factor in an algebraic expression.

```
(A && (B || C)) === ((A && B) || (A && C))
```

```
(A || (B && C)) === ((A || B) && (A || C))
```

Another common occurrence of distribution is double-distribution (similar to FOIL in algebra):

1. `((A || B) && (C || D)) === ((A || B) && C) || ((A || B) && D)`
2. `((A || B) && C) || ((A || B) && D) === ((A && C) || B && C)) || ((A && D) || (B && D))`

```
(A || B) && (C || D) === (A && C) || B && C) || (A && D) || (B && D)
```

```
(A && B) ||(C && D) === (A || C) && (B || C) && (A || D) && (B || D)
```

Material Implication

Implication expressions (`A → B`) typically get translated into code as if `(A) { B }` but that is not very useful if a compound expression has several implications in it. You would end up with nested `if` statements — a code smell. Instead, I often use the material implication rule of replacement, which says that `A → B` means either `A` is false or `B` is true.

```
(A → B) === (!A || B)
```

Tautology & Contradiction

Sometimes during the course of manipulating compound logical expressions, you'll end up with a simple conjunction or disjunction that only involves one variable and its negation or a boolean literal. In those cases, the expression is either always true (a tautology) or always false (a contradiction) and can be replaced with the boolean literal in code.

```
(A || !A) === true
```

```
(A || true) === true
```

```
(A && !A) === false
```

```
(A && false) === false
```

Related to these equivalencies are the disjunction and conjunction with the other boolean literal. These can be simplified to just the truthiness of the variable.

```
(A || false) === A
```

```
(A && true) === A
```

Transposition
When manipulating an implication (`A → B`), a common mistake people make is to assume that negating the first part, `A`, implies the second part, `B`, is also negated — `!A → !B`. This is called the *converse* of the implication and it is **not necessarily true**. That is, having the original implication does not tell us if the converse is true because `A` is not a *necessary* condition of `B`. (If the converse is also true — for independent reasons — then `A` and `B` are biconditional.)

What we can know from the original implication, though, is that the *contrapositive* is true. Since `B` *is* a necessary condition for `A` (recall from the truth table for implication that if `B` is true, `A` must also be true for the implication to be true), we can claim that `!B → !A`.

```
(A → B) === (!B → !A)
```

Material Equivalence
The name *biconditional* comes from the fact that it represents two conditional (implication) statements: `A === B` means that `A → B` **and** `B → A`. The truth values of `A` and `B` are locked into each other. This gives us the first material equivalence rule:

```
(A === B) === ((A → B) && (B → A))
```

Using material implication, double-distribution, contradiction, and commutation, we can manipulate this new expression into something easier to code:

1. `((A → B) && (B → A)) === ((!A || B) && (!B || A))`
2. `((!A || B) && (!B || A)) === ((!A && !B) || (B && !B)) || ((!A && A) || (B && A))`
3. `((!A && !B) || (B && !B)) || ((!A && A) || (B && A)) === ((!A && !B) || (B && A))`
4. `((!A && !B) || (B && A)) === ((A && B) || (!A && !B))`

`(A === B) === ((A && B) || (!A && !B))`

Exportation

Nested if statements, especially if there are no else parts, are a code smell. A simple nested if statement can be simplified into a single statement where the conditional is a conjunction of the two previous conditions:

```
if (A) {
  if (B) {
    C
  }
}
// is equivalent to
if (A && B) {
  C
}
```

`(A → (B → C)) === ((A && B) → C)`

DeMorgan's Laws

DeMorgan's Laws are essential to working with logical statements. They tell how to distribute a negation across a conjunction or disjunction. Consider the expression `!(A || B)`. DeMorgan's Laws say that when negating a disjunction or conjunction, negate each statement and change the `&&` to `||` or vice versa. Thus `!(A || B)` is the same as `!A && !B`. Similarly, `!(A && B)` is equivalent to `!A || !B`.

`!(A || B) === !A && !B`

`!(A && B) === !A || !B`

Ternary (If-Then-Else)

Ternary statements (`A ? B : C`) occur regularly in programming, but they're not quite implications. The translation from a ternary to formal logic is actually a conjunction of two implications, `A → B` and `!A → C`, which we can write as: `(!A || B) && (A || C)`, using material implication.

`(A ? B : C) === (!A || B) && (A || C)`

XOR (Exclusive Or)

Exclusive Or, often abbreviated **xor**, means, "one or the other, but not both." This differs from the normal *or* operator only in that both values cannot be true. This is often what we mean when we use "or" in plain English. JavaScript doesn't have a native xor operator, so how would we represent this?

1. *"A or B, but not both A and B"*
2. `(A || B) && !(A && B)` ***direct translation***
3. `(A || B) && (!A || !B)` ***DeMorgan's Laws***
4. `(!A || !B) && (A || B)` ***commutativity***
5. `A ? !B : B` ***if-then-else definition***

`A ? !B : B` *is exclusive or (xor) in JavaScript*

Alternatively,

1. *"A or B, but not both A and B"*
2. `(A || B) && !(A && B)` ***direct translation***
3. `(A || B) && (!A || !B)` ***DeMorgan's Laws***
4. `(A && !A) || (A && !B) || (B && !A) || (B && !B)` ***double-distribution***
5. `(A && !B) || (B && !A)` ***contradiction replacement***
6. `A === !B` *or* `A !== B` ***material equivalence***

`A === !B` *or* `A !== B` *is xor in JavaScript*

Part 3: Universal & Existential Statements

Photo by Sharon McCutcheon on Unsplash

So far we have been looking at statements about expressions involving two (or a few) values, but now we will turn our attention to sets of values. Much like how logical operators in compound expressions preserve truthiness in predictable ways, *predicate functions* on sets preserve truthiness in predictable ways. A **predicate function** is a function whose input is a value from a set and whose output is a boolean. For the code examples in this article, we will use an array of numbers for a set and two predicate functions:

- `isOdd = n => n % 2 !== 0;` and
- `isEven = n => n % 2 === 0;`

> ***A predicate function is a function whose input is a value from a set and whose output is a boolean.***

Universal Statements

A **universal** statement is one that applies to **all** elements in a set, meaning its predicate function returns true for every element. If the predicate returns false for any one (or more) element, then the universal statement is false. `Array.prototype.every` takes a predicate function and returns `true` only if every element of the array returns true for the predicate. It also terminates early (with `false`) if the predicate returns false, not running the predicate over any more elements of the array, so in practice *avoid side-effects in predicates*.

As an example, consider the array `[2, 4, 6, 8]`, and the universal statement, "every element of the array is even." Using `isEven` and JavaScript's built-in universal function, we can run [2, 4, 6, 8].every(isEven) `and find that this is` true`.

> *`Array.prototype.every` is JavaScript's Universal Statement*

Existential Statements
An **existential** statement makes a specific claim about a set: at least one element in the set returns true for the predicate function. If the predicate returns false for every element in the set, then the existential statement is false.

JavaScript also supplies a built-in existential statement: `Array.prototype.some`. Similar to `every`, `some` will return early (with true) if an element satisfies its predicate. As an example, `[1, 3, 5].some(isOdd)` will only run one iteration of the predicate `isOdd` (consuming `1` and returning `true`) and return `true`. `[1, 3, 5].some(isEven)` will return `false`.

> *`Array.prototype.some` is JavaScript's Existential Statement*

Universal Implication
Once you have checked a universal statement against a set, say `nums.every(isOdd)`, it is tempting to think that you can grab an element from the set that satisfies the predicate. However, there is one catch: in Boolean logic, a true universal statement **does not imply** that the set is non-empty. Universal statements about empty sets are *always true*, so if you wish to grab an element from a set satisfying some condition, use an existential check instead. To prove this, `run [].every(() => false`. It will be true.

*Universal statements about empty sets are **always true.***

Negating Universal and Existential Statements

Negating these statements can be surprising. The negation of a universal statement, say `nums.every(isOdd)`, is not `nums.every(isEven)`, but rather `nums.some(isEven)`. This is an existential statement with the predicate negated. Similarly, the negation of an existential statement is a universal statement with the predicate negated.

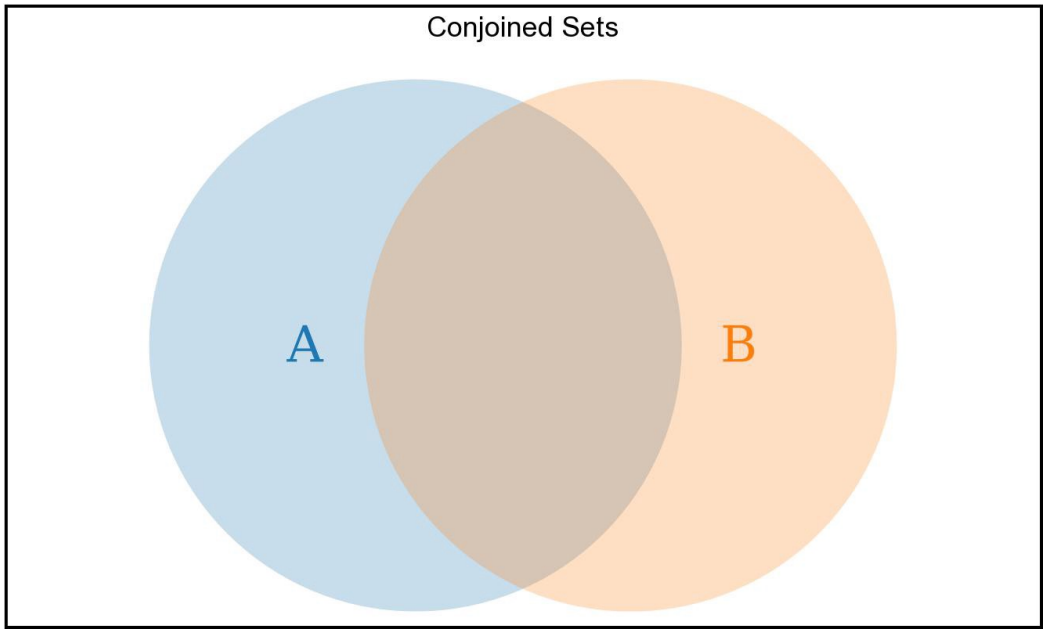
```
!arr.every(el => fn(el)) === arr.some(el => !fn(el))
```

```
!arr.some(el => fn(el)) === arr.every(el => !fn(el))
```

Set Intersections

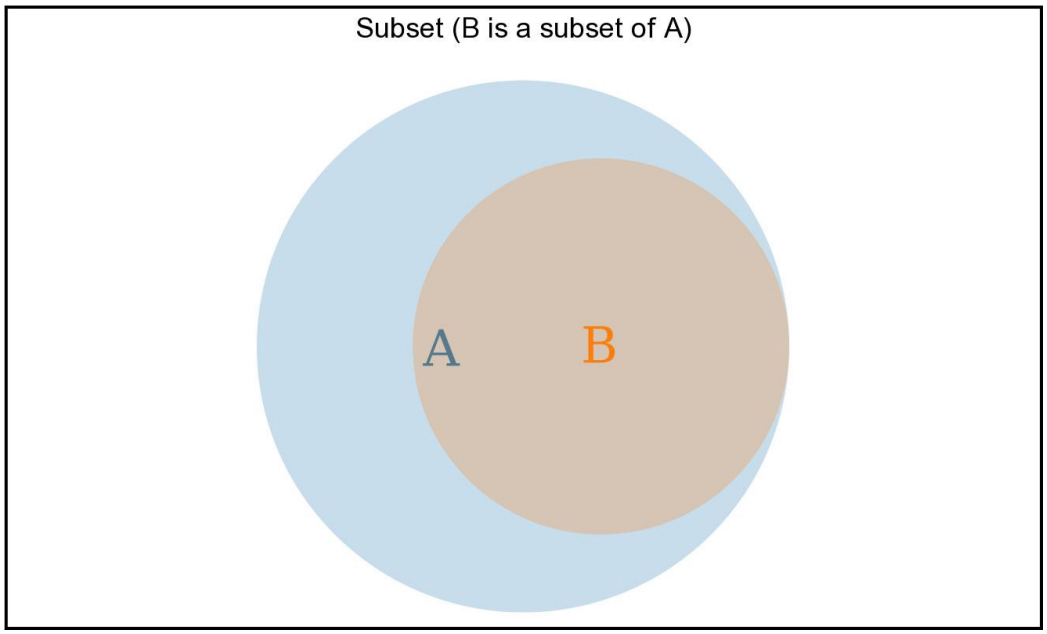
Two sets can only be related to each other in a few ways, with regards to their elements. These relationships are easily diagrammed with Venn Diagrams, and can (mostly) be determined in code using combinations of universal and existential statements.

Two sets can each share some but not all of their elements, like a typical *conjoined* Venn Diagram:



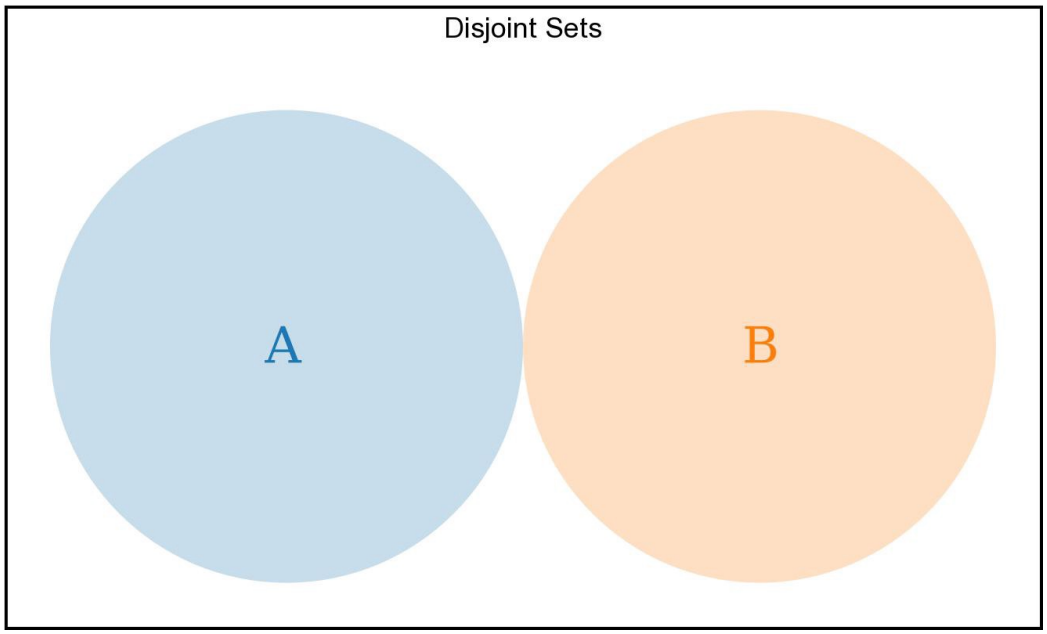
`A.some(el => B.includes(el)) && A.some(el => !B.includes(el)) && B.some(el => !A.includes(el))` ***describes a conjoined pair of sets***

One set can contain all of the other set's elements, but have elements not shared by the second set. This is a **subset** relationship, denoted as `Subset ⊆ Superset`.



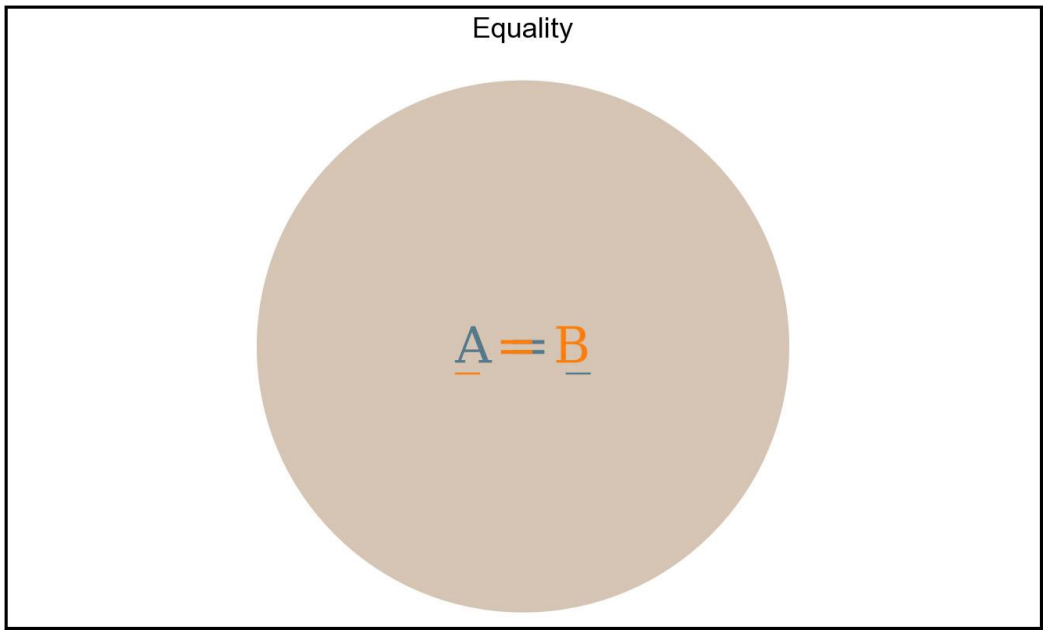
`B.every(el => A.includes(el))` ***describes the subset relationship B ⊆ A***

The two sets can share **no** elements. These are *disjoint* sets.



`A.every(el => !B.includes(el))` describes a disjoint pair of sets

Lastly, the two sets can share every element. That is, they are subsets of each other. These sets are *equal*. In formal logic, we would write `A ⊆ B && B ⊆ A` $\longleftrightarrow$ `A === B`, but in JavaScript, there are some complications with this. In JavaScript, an `Array` is an *ordered* set and may contain duplicate values, so we **cannot** assume that the bidirectional subset code `B.every(el => A.includes(el)) && A.every(el => B.includes(el))` implies the arrays `A` and `B` are equal. If `A` and `B` are Sets (meaning they were created with `new Set()`), then their values are unique and we can do the bidirectional subset check to see if `A === B`.



`(A === B) === (Array.from(A).every(el => Array.from(B).includes(el)) && Array.from(B).every(el => Array.from(A).includes(el))`, ***given that `A` and `B` are constructed using `new Set()`***

Next: Translating English to Code.

Part 4: English to Code Translations

This is part of a series. Read parts 1, 2, and 3.

This article is probably the most useful in the series. Here, now that we know the logical operators, their truth tables, and rules of replacement, we can learn how to translate an English phrase into code and *simplify* it. In learning this translation skill, you will also be able to *read* code better, storing complex logic in simple phrases in your mind.

Below is a table of logical code (left) and their English equivalents (right) that was heavily borrowed from the excellent book, *Essentials of Logic*.

Given A and B are truthy or falsy values	
Logic / Code	**English**
!A	not A
	it is not the case that A
A && B	A and B
	A but B
	A yet B
	A however B
	A inasmuch as B
	A although B
	both A and B
	A though B
	A even though B
	A nevertheless B
A II B	A or B
	either A or B
	B unless A
A → B	if A, then B
	A only if B
	B, if A
	B, given that A
	B, provided that A
	provided that A, B
	B, on the condition that A
	on the condition that A, B
	A implies B
	B is implied by A
	A entails B
	B is entailed by A
	B is a necessary condition for A
	A is a sufficient condition for B
	B in the event that A
	in the event that A, B
A === B	A if and only if B
	A just in case that B
	A is a necessary and sufficient condition for B
!A && !B	neither A nor B
!(A II B)	neither A nor B

Given X and Y are arrays	
Logic / Code	**English**
X.every(x => Y.includes(x))	all X are Y
	every X is Y
	for all x in X, x is in Y
	any X is a Y
	only Ys are Xs
	none but Ys are Xs
	the only X is a Y
	there is no X unless it is Y
	whatever is X is Y
X.every(x => !Y.includes(x))	X and Y are separate
	no X are Y
	none of the Xs are Ys
	not any X is a Y
X.some(x => Y.includes(x))	some X are Y
	there exists at least one X that is Y
	a few X are Y
	at least one X is Y
	many X are Y
	there is an x that is both X and Y
	there is an X that is Y
	several X are Y
	numerous X are Y
X.some(x => !Y.includes(x))	some X are not Y
	there is at least one X that is not Y
	a few X are not Y
	a X is not Y
	not all X are Y
	many X are not Y
	several X are not Y
	not only Y are X

View a screen-readable version of this code-to-English translation chart here.

Below, I will go through some real-world examples from my own work where I interpret from English to code, and vice-versa, and simplify code with the rules of replacement.

Example 1

Recently, to satisfy the EU's GDPR requirements, I had to create a modal that showed my company's cookie policy and allowed the user to set their preferences. To make this as unobtrusive as possible, we had the following requirements (in order of precedence): 1. If the user wasn't in the EU, **never** show the GDPR preferences modal. 2. If the app programmatically needs to show the modal (if a user action requires more permission than currently allowed), show the modal. 3. If

the user is allowed to have the less-obtrusive GDPR *banner*, do not show the modal. 4. If the user has **not** already set their preferences (ironically saved in a cookie), show the modal.

I started off with a series of `if` statements modeled directly after these requirements:

```
const isGdprPreferencesModalOpen = ({
  shouldModalBeOpen,
  hasCookie,
  hasGdprBanner,
  needsPermissions
}) => {
  if (!needsPermissions) {
    return false;
  }
  if (shouldModalBeOpen) {
    return true;
  }
  if (hasGdprBanner) {
    return false;
  }
  if (!hasCookie) {
    return true;
  }
  return false;
}
```

To be clear, the above code works and could be left alone, but I believe that returning boolean literals is a code smell. So I went through the following steps:

```
/* change to a single return, if-else-if structure */
let result;
if (!needsPermissions) {
  result = false;
} else if (shouldBeOpen) {
  result = true;
} else if (hasBanner) {
```

```
  result = false;
} else if (!hasCookie) {
  result = true
} else {
  result = false;
}
return result;

/* use the definition of ternary to convert to a single return */
return !needsPermissions ? false : (shouldBeOpen ? true : (hasBanner
? false : (!hasCookie ? true : false)))

/* convert from ternaries to conjunctions of disjunctions */
return (!!needsPermissions || false) && (!needsPermissions ||
((!shouldBeOpen || true) && (shouldBeOpen || ((!hasBanner || false)
&& (hasBanner || !hasCookie))))

/* simplify double-negations and conjunctions/disjunctions with
boolean literals */
return needsPermissions && (!needsPermissions || ((!shouldBeOpen ||
true) && (shouldBeOpen || (!hasBanner && (hasBanner || !hasCookie)))))

/* DeMorgan's Laws */
return needsPermissions && (!needsPermissions || ((!shouldBeOpen ||
true) && (shouldBeOpen || ((!hasBanner && hasBanner) || (hasBanner &&
!hasCookie))))

/* eliminate tautologies and contradictions, simplify */
return needsPermissions && (!needsPermissions || (shouldBeOpen ||
(hasBanner && !hasCookie)))

/* DeMorgan's Laws */
return (needsPermissions && !needsPermissions) || (needsPermissions
&& (shouldBeOpen || (hasBanner && !hasCookie)))

/* eliminate contradiction, simplify */
return needsPermissions && (shouldBeOpen || (hasBanner &&
!hasCookie))
```

I ended up with something that I think is more elegant and still readable:

```
const isGdprPreferencesModalOpen = ({
  needsPermissions,
  shouldBeOpen,
  hasBanner,
  hasCookie,
}) => (
  needsPermissions && (shouldBeOpen || (!hasBanner && !hasCookie))
);
```

Example 2

I found the following code (written by a coworker) while updating a component. Again, I felt the urge to eliminate the boolean literal returns, so I refactored it.

```
const isButtonDisabled = (isRequestInFlight, state) => {
  if (isRequestInFlight) {
    return true;
  }
  if (enabledStates.includes(state)) {
    return false;
  }
  return true;
};
```

Sometimes I do the following steps in my head or on scratch paper, but most often, I write each next step in the code and then delete the previous step.

```
// convert to if-else-if structure
let result;
if (isRequestInFlight) {
  result = true;
} else if (enabledStates.includes(state)) {
  result = false;
} else {
  result = true;
}
return result;
```

```
// convert to ternary
return isRequestInFlight
  ? true
  : enabledStates.includes(state)
    ? false
    : true;

/* convert from ternary to conjunction of disjunctions */
return (!isRequestInFlight || true) && (isRequestInFlight ||
((!enabledStates.includes(state) || false) &&
(enabledStates.includes(state) || true))

/* remove tautologies and contradictions, simplify */
return isRequestInFlight || !enabledStates.includes(state)
```

Then I end up with:

```
const isButtonDisabled = (isRequestInFlight, state) => (
  isRequestInFlight || !enabledStates.includes(state)
);
```

In this example, I didn't start with English phrases and I never bothered to interpret the code to English while doing the manipulations, but now, at the end, we can easily translate this: "the button is disabled if either the request is in flight or the state is not in the set of enabled states." That makes sense. If you ever translate your work back to English and it *doesn't* make sense, re-check your work. This happens to me often.

Example 3
While writing an A/B testing framework for my company, we had two master lists of Enabled and Disabled experiments and we wanted to check that *every* experiment (each a separate file in a folder) was recorded in one or the other list **but not both**. This means the enabled and disabled sets are *disjointed* and the set of all experiments is a subset of the conjunction of the two sets of experiments. The reason the set of all experiments must be a subset of the combination of the two lists is that there should not be a single experiment that exists *outside* the two lists.

```
const isDisjoint = !enabled.some(disabled.includes) &&
  !disabled.some(enabled.includes);
const isSubset = allExperiments.every(
  enabled.concat(disabled).includes
);
assert(isDisjoint && isSubset);
```

Conclusion

Hopefully this has all been helpful. Not only are the skills of translating English to code useful, but having the terminology to discuss different relationships (conjunction, implication, etc.) and the tools to evaluate them (truth tables) is handy.