

Lesson - 14

Operators
Conditional Operators

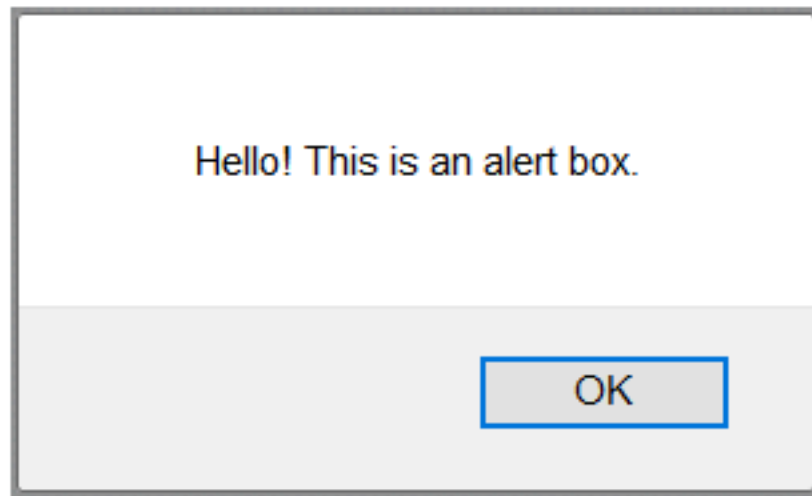


Lesson Plan

- Review of HW
- Interaction with alert, prompt, confirm
- Type Conversion
- Operators
- Comparisons
- Conditional Operators: if, ?

Interaction with alert, prompt, confirm

Alert box

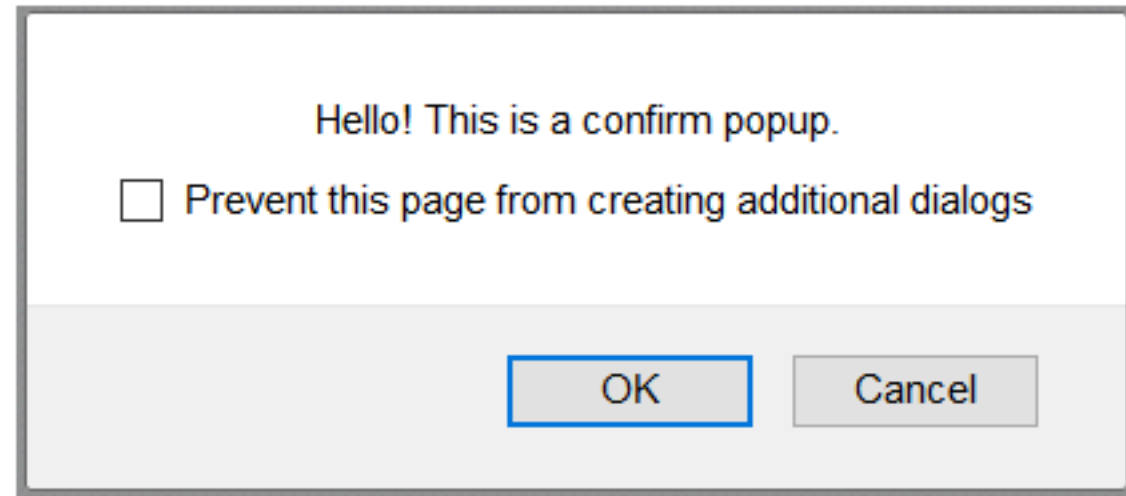


Hello! This is an alert box.

OK

This is a screenshot of a standard JavaScript alert box. It has a white background with a light gray border. The text "Hello! This is an alert box." is centered in the main area. At the bottom, there is a single button labeled "OK".

Confirm popup



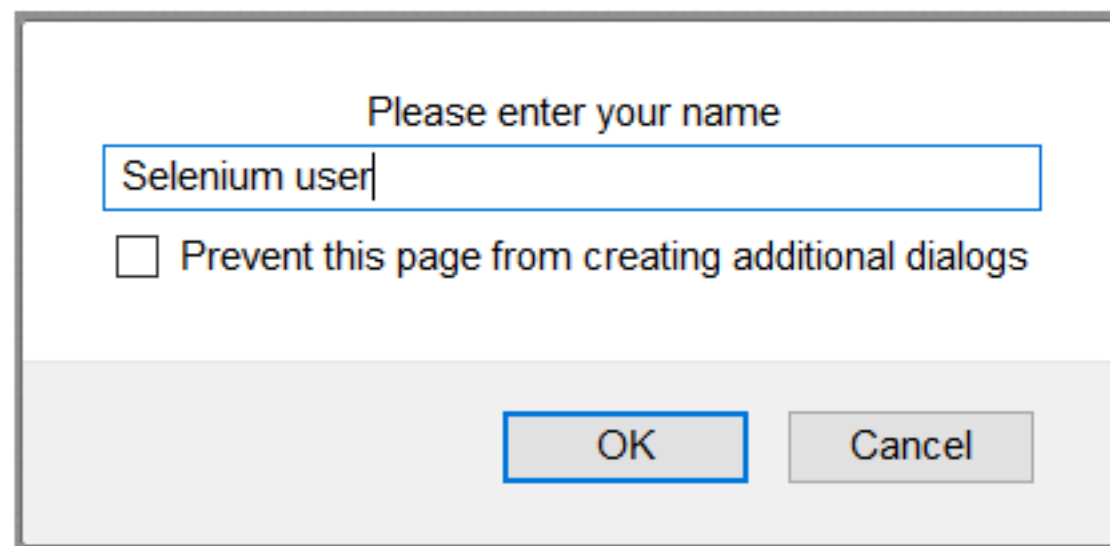
Hello! This is a confirm popup.

☐ Prevent this page from creating additional dialogs

OK Cancel

This is a screenshot of a JavaScript confirm dialog. It has a white background with a light gray border. The text "Hello! This is a confirm popup." is centered. Below it is a checkbox with the label "Prevent this page from creating additional dialogs". At the bottom, there are two buttons: "OK" and "Cancel".

Prompt dialogue box



Please enter your name

Selenium user

☐ Prevent this page from creating additional dialogs

OK Cancel

This is a screenshot of a JavaScript prompt dialog. It has a white background with a light gray border. The text "Please enter your name" is centered. Below it is a text input field containing the text "Selenium user". Below the input field is a checkbox with the label "Prevent this page from creating additional dialogs". At the bottom, there are two buttons: "OK" and "Cancel".

Alert

The mini-window with the message is called a *modal window*. The word “modal” means that the visitor can’t interact with the rest of the page, press other buttons, etc. until they have dealt with the window. In this case – until they press “OK”.



```
// Syntax:  
alert(message)  
  
// Example:  
alert('Hello! This is an alert box')
```

Hello! This is an alert box.

OK

Prompt

It shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel.

The function `prompt` accepts two arguments:

title

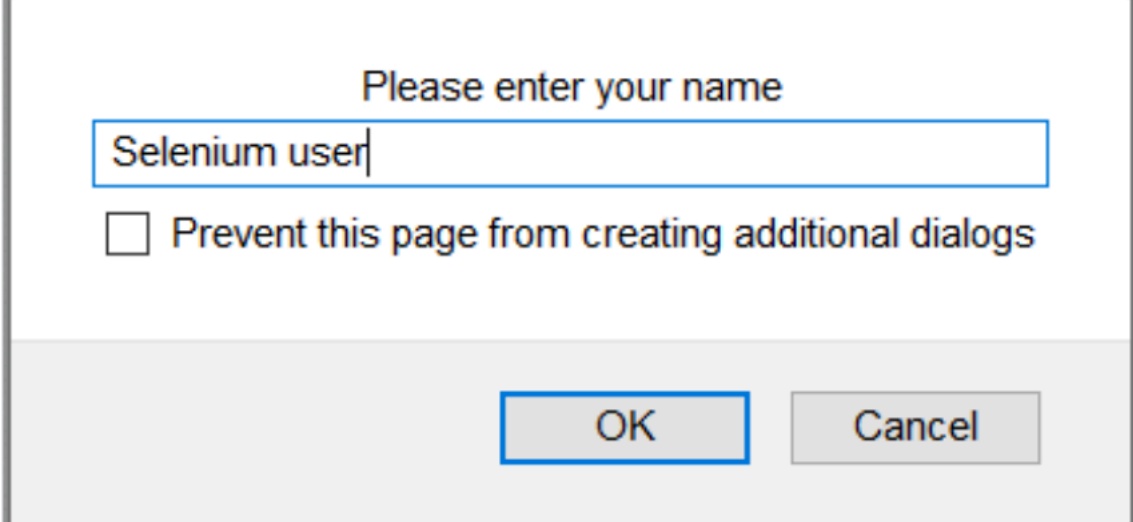
The text to show the visitor.

default

An optional second parameter, the initial value for the input field.

The visitor may type something in the prompt input field and press OK. Or they can cancel the input by pressing Cancel or hitting the Esc key.

The call to `prompt` returns the text from the input field or `null` if the input was canceled.



// Syntax

```
let result = prompt(title, [default])
```

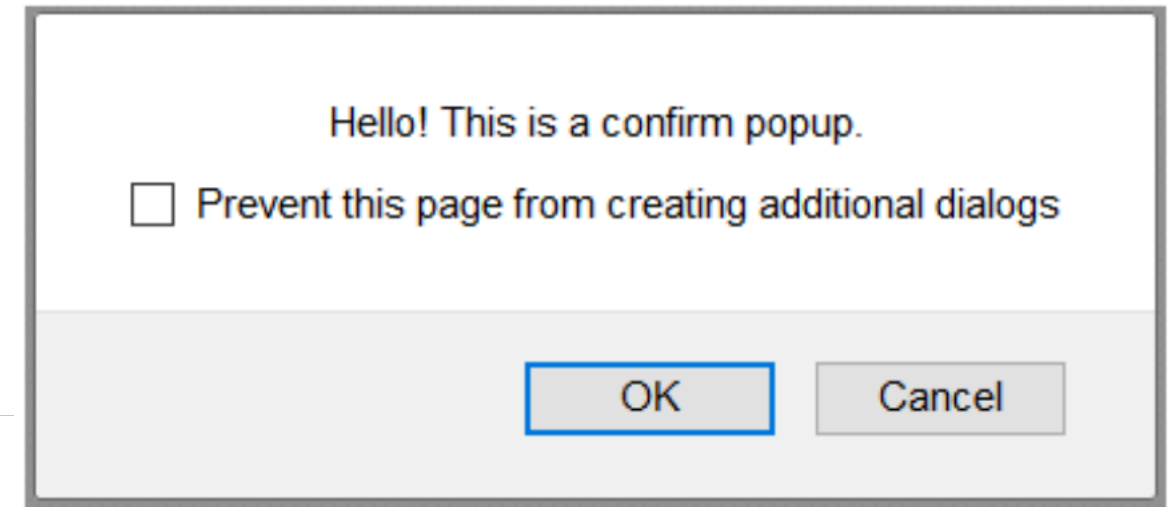
// Example

```
let result = prompt('How old are you?', 100);
```

Confirm

The function `confirm` shows a modal window with a `question` and two buttons: OK and Cancel.

The result is `true` if OK is pressed and `false` otherwise.



```
// Syntax
let result = confirm(question)

// Example
let result = prompt('By pressing OK, you confirm legal agreements');

// Usage
alert(result) // true if OK was pressed
```

Interaction summary

We covered 3 browser-specific functions to interact with visitors:

alert

shows a message.

prompt

shows a message asking the user to input text. It returns the text or, if Cancel button or Esc is clicked, `null`.

confirm

shows a message and waits for the user to press “OK” or “Cancel”. It returns `true` for OK and `false` for Cancel/Esc.

All these methods are modal: they pause script execution and don’t allow the visitor to interact with the rest of the page until the window has been dismissed.

There are two limitations shared by all the methods above:

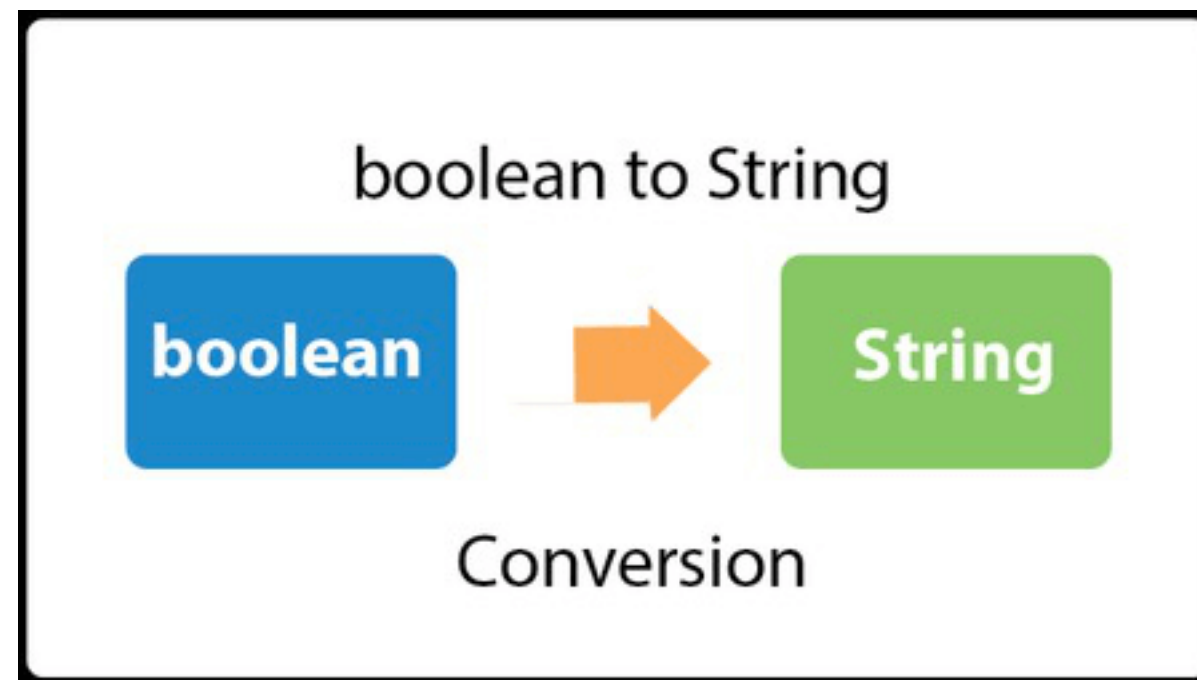
1. The exact location of the modal window is determined by the browser. Usually, it’s in the center.
2. The exact look of the window also depends on the browser. We can’t modify it.

Type Conversions

Most of the time, operators and functions automatically convert the values given to them to the right type.

For example, `alert` automatically converts any value to a string to show it. Mathematical operations convert values to numbers.

There are also cases when we need to explicitly convert a value to the expected type.



String conversion

String conversion happens when we need the string form of a value.

For example, `alert(value)` does it to show the value.

We can also call the `String(value)` function to convert a value to a string: _____



```
let value = true;  
alert(typeof value); // boolean
```

```
value = String(value); // now value is a string "true"  
alert(typeof value); // string
```

_____ String conversion is mostly obvious. A `false` becomes `"false"`, `null` becomes `"null"`, etc. _____

Number Conversion

Number type conversion can be met in the math expressions and comparisons. Here is where usually a lot of confusion comes from.

For example, when division `/` is applied to non-numbers:

The `+` operator actually works a bit different. **If one of the operands is a string, then all other operands are converted to string too** and it works like string concatenation, not like the math expression:

```
2 * "3" // 6
6 / "2" // 3
3 - "1" // 2


3 + "3" // "33"
```



```
// One of the operands is string "2"
// JavaScript will convert every other operand to string too
1 + "2" + true // The result is "12true"
```

Other number conversion rules

Other than that, the number conversion follows these rules:




```
parseInt(true)      // 1
parseInt(false)     // 0
parseInt(null)      // 0
parseInt(undefined) // NaN

parseInt(" 32 ")    // 32. Whitespaces from the start and end are removed
parseInt(" ")       // 0. If the remaining string is empty, the result is 0
parseInt("hi")      // NaN
```

Boolean conversion

This type conversion happens in the logical operations. It follows strict rules too yet they are mostly obvious:

- 0, NaN, undefined, null, "" are converting to false
- everything else, including objects, to true



```
if ("hello") // true
if (0)       // false
if ({}))     // true
```

Operators

We know many operators from school. They are things like addition `+`, multiplication `*`, subtraction `-`, and so on.

Before we move on, let's grasp some common terminology.

- An *operand* – is what operators are applied to. For instance, in the multiplication of `5 * 2` there are two operands: the left operand is `5` and the right operand is `2`. Sometimes, people call these “arguments” instead of “operands”.
- An operator is *unary* if it has a single operand. For example, the unary negation `-`

```
let x = 1;
x = -x;
alert( x ); // -1, unary negation was applied
```

- An operator is *binary* if it has two operands. The same minus exists in binary form as well:

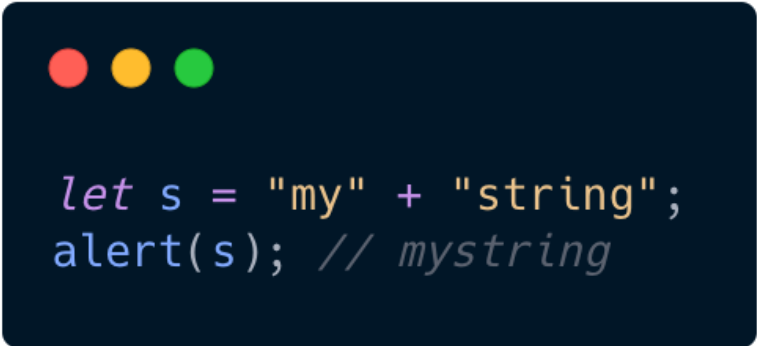
```
let x = 1, y = 3;
alert( y - x ); // 2, binary minus subtracts values
```

String concatenation, binary +

Now, let's see special features of JavaScript operators that are beyond school arithmetics.

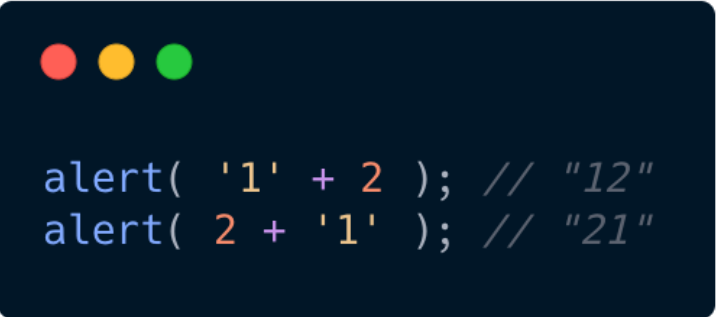
Usually, the plus operator `+` sums numbers.

But, if the binary `+` is applied to strings, it merges (concatenates) them:



```
let s = "my" + "string";  
alert(s); // mystring
```

Note that if one of the operands is a string, the other one is converted to a string too.



```
alert( '1' + 2 ); // "12"  
alert( 2 + '1' ); // "21"
```

String concatenation and conversion is a special feature of the binary plus `+`. Other arithmetic operators work only with numbers and always convert their operands to numbers.

What will be the result?



```
alert(2 + 2 + '1' ); // What is the result?
```

Numeric conversion.

Unary +

The plus `+` exists in two forms: the binary form that we used above and the unary form.

The unary plus or, in other words, the plus operator `+` applied to a single value, doesn't do anything to numbers. But if the operand is not a number, the unary plus converts it into a number.

```
// No effect on numbers
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// Converts non-numbers
alert( +true ); // 1
alert( +"" ); // 0
```

It actually does the same thing as `Number(...)`, but is shorter.

Why we need this?

The need to convert strings to numbers arises very often. For example, if we are getting values from HTML form fields, they are usually strings. What if we want to sum them?

The binary plus would add them as strings:

```
let apples = "2";  
let oranges = "3";  
  
alert( apples + oranges ); // "23", the binary plus concatenates strings
```

If we want to treat them as numbers, we need to convert and then sum them:

```
let apples = "2";  
let oranges = "3";  
  
// both values converted to numbers before the binary plus  
alert( +apples + +oranges ); // 5  
  
// the longer variant  
// alert( Number(apples) + Number(oranges) ); // 5
```

Why unary are applied before binary?

From a mathematician's standpoint, the abundance of pluses may seem strange. But from a programmer's standpoint, there's nothing special: unary pluses are applied first, they convert strings to numbers, and then the binary plus sums them up.

Why are unary pluses applied to values before the binary ones? As we're going to see, that's because of their *higher precedence*.

Operator precedence

If an expression has more than one operator, the execution order is defined by their *precedence*, or, in other words, the default priority order of operators.

From school, we all know that the multiplication in the expression `1 + 2 * 2` should be calculated before the addition. That's exactly the precedence thing. The multiplication is said to have *a higher precedence* than the addition.

Parentheses override any precedence, so if we're not satisfied with the default order, we can use them to change it. For example, write `(1 + 2) * 2`.

There are many operators in JavaScript. Every operator has a corresponding precedence number. The one with the larger number executes first. If the precedence is the same, the execution order is from left to right.

Operator precedence

Here's an extract from the [precedence table](#) (you don't need to remember this, but note that unary operators are higher than corresponding binary ones):

Precedence	Name	Sign
...
17	unary plus	+
17	unary negation	-
15	multiplication	*
15	division	/
13	addition	+
13	subtraction	-
...
3	assignment	=
...

As we can see, the “unary plus” has a priority of 17 which is higher than the 13 of “addition” (binary plus). That's why, in the expression `+apples + +oranges`, unary pluses work before the addition.

Assignment

Let's note that an assignment `=` is also an operator. It is listed in the precedence table with the very low priority of 3.

That's why, when we assign a variable, like `x = 2 * 2 + 1`, the calculations are done first and then the `=` is evaluated, storing the result in `x`.



```
let x = 2 * 2 + 1;  
alert( x ); // 5
```

Remainder %

The remainder operator %, despite its appearance, is not related to percents.

The result of `a % b` is the remainder of the integer division of `a` by `b`.

For instance:



```
alert( 5 % 2 ); // 1 is a remainder of 5 divided by 2  
alert( 8 % 3 ); // 2 is a remainder of 8 divided by 3  
alert( 6 % 3 ); // 0 is a remainder of 6 divided by 3
```

Exponentiation

The exponentiation operator `**` is a recent addition to the language.

For a natural number `b`, the result of `a ** b` is `a` multiplied by itself `b` times.

For instance:



```
alert( 2 ** 2 ); // 4  (2 * 2)
alert( 2 ** 3 ); // 8  (2 * 2 * 2)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2)
```

Increment / Decrement

Increasing or decreasing a number by one is among the most common numerical operations.

So, there are special operators for it:

- **Increment** `++` increases a variable by 1



```
let counter = 2;  
counter++;      // works the same as counter = counter + 1, but is shorter  
alert( counter ); // 3
```

- **Decrement** `--` decreases a variable by 1:



```
let counter = 2;  
counter--;     // works the same as counter = counter - 1, but is shorter  
alert( counter ); // 1
```

Important:

Increment/decrement can only be applied to variables. Trying to use it on a value like `5++` will give an error.

Postfix / prefix

The operators `++` and `--` can be placed either before or after a variable.

- When the operator goes after the variable, it is in “postfix form”: `counter++`.
- The “prefix form” is when the operator goes before the variable: `++counter`.

Both of these statements do the same thing: increase `counter` by 1.

Is there any difference? Yes, but we can only see it if we use the returned value of `++/--`.

Let’s clarify. As we know, all operators return a value. Increment/decrement is no exception. The prefix form returns the new value while the postfix form returns the old value (prior to increment/decrement).

To see the difference, here’s an example:

```
let counter = 1;
let a = ++counter; // will increment and return new value 2
alert(a); // 2
```

If the result of increment/decrement is not used, there is no difference in which form to use:

If we’d like to increase a value *and* immediately use the result of the operator, we need the prefix form:

If we’d like to increment a value but use its previous value, we need the postfix form:

Now, let’s use the postfix form:

```
let counter = 1;
let a = counter++; // will return counter and then increment
alert(a); // 1
```

Bitwise operators

Bitwise operators treat arguments as 32-bit integer numbers and work on the level of their binary representation.

These operators are not JavaScript-specific. They are supported in most programming languages.

The list of operators:

- AND (`&`)
- OR (`|`)
- XOR (`^`)
- NOT (`~`)
- LEFT SHIFT (`<<`)
- RIGHT SHIFT (`>>`)
- ZERO-FILL RIGHT SHIFT (`>>>`)

These operators are used very rarely. To understand them, we need to delve into low-level number representation and it would not be optimal to do that right now, especially since we won't need them any time soon. If you're curious, you can read the [Bitwise Operators](#) article on MDN. It would be more practical to do that when a real need arises.

Modify-in-place

We often need to apply an operator to a variable and store the new result in that same variable.

For example:

```
let n = 2;  
n = n + 5;  
n = n * 2;
```

This notation can be shortened using the operators `+=` and `*=`:

```
let n = 2;  
n += 5; // now n = 7 (same as n = n + 5)  
n *= 2; // now n = 14 (same as n = n * 2)  
  
alert( n ); // 14
```

Short “modify-and-assign” operators exist for all arithmetical and bitwise operators: `/=`, `-=`, etc.

Comparisons

We know many comparison operators from maths:

- Greater/less than: `a > b`, `a < b`.
- Greater/less than or equals: `a >= b`, `a <= b`.
- Equals: `a == b` (please note the double equals sign `=`. A single symbol `a = b` would mean an assignment).
- Not equals. In maths the notation is \neq , but in JavaScript it's written as an assignment with an exclamation sign before it: `a != b`.

Boolean is the result

Like all other operators, a comparison returns a value. In this case, the value is a boolean.

- `true` – means “yes”, “correct” or “the truth”.
- `false` – means “no”, “wrong” or “not the truth”.




```
alert( 2 > 1 ); // true (correct)
alert( 2 == 1 ); // false (wrong)
alert( 2 != 1 ); // true (correct)
```

String comparison

To see whether a string is greater than another, JavaScript uses the so-called “dictionary” or “lexicographical” order.

In other words, strings are compared letter-by-letter.



```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

The algorithm to compare two strings is simple:

1. Compare the first character of both strings.
2. If the first character from the first string is greater (or less) than the other string's, then the first string is greater (or less) than the second. We're done.
3. Otherwise, if both strings' first characters are the same, compare the second characters the same way.
4. Repeat until the end of either string.
5. If both strings end at the same length, then they are equal. Otherwise, the longer string is greater.

In the examples above, the comparison `'Z' > 'A'` gets to a result at the first step while the strings `"Glow"` and `"Glee"` are compared character-by-character:

1. G is the same as G.
2. l is the same as l.
3. o is greater than e. Stop here. The first string is greater.

Strict equality

A regular equality check `==` has a problem. It cannot differentiate `0` from `false`:

```
alert( 0 == false ); // true

// Same for empty strings
alert( '' == false ); // true
```

This happens because operands of different types are converted to numbers by the equality operator `==`. An empty string, just like `false`, becomes a zero.

What to do if we'd like to differentiate `0` from `false`?

A strict equality operator `===` checks the equality without type conversion.

In other words, if `a` and `b` are of different types, then `a === b` immediately returns `false` without an attempt to convert them.

```
alert( 0 === false ); // false, because the types are different
```

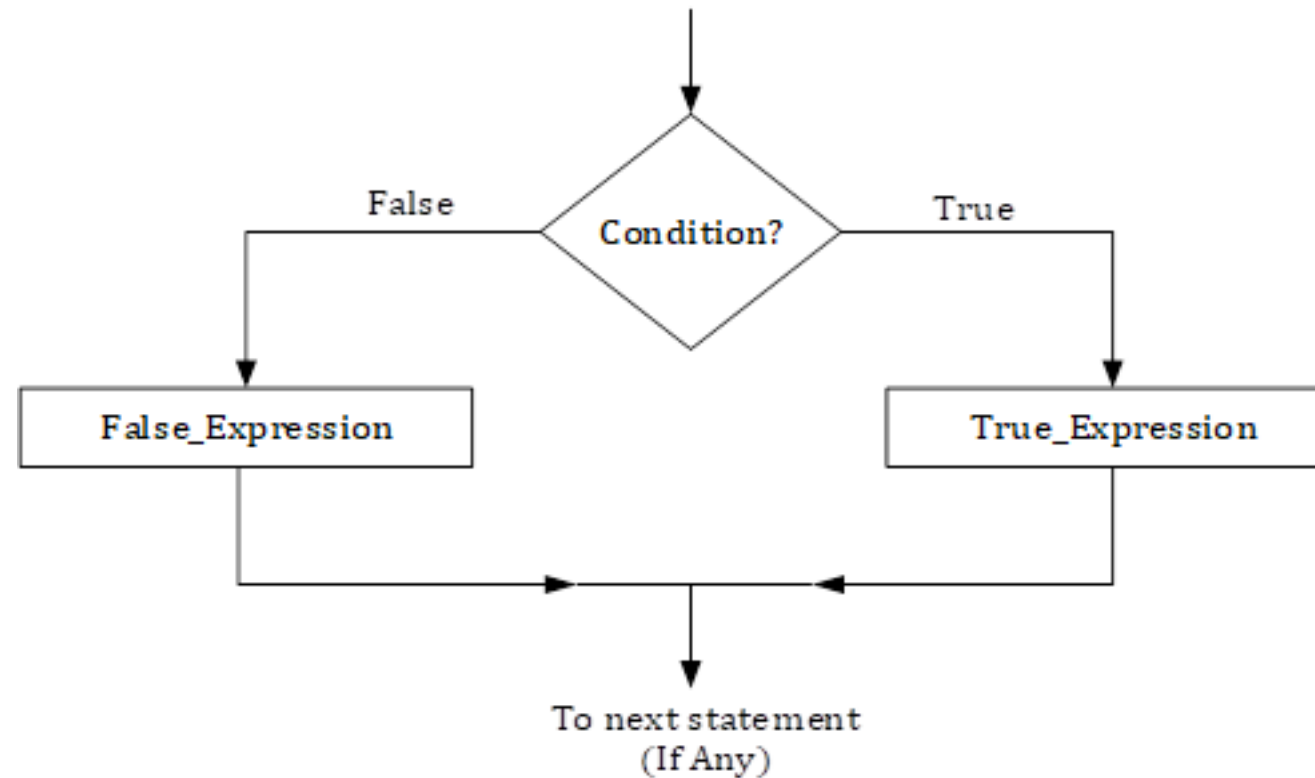
There is also a “strict non-equality” operator `!==` analogous to `!=`.

The strict equality operator is a bit longer to write, but makes it obvious what's going on and leaves less room for errors.

Conditional Operators if and ?

Sometimes, we need to perform different actions based on different conditions.


To do that, we can use the `if` statement and the conditional operator `?`, that's also called a “question mark” operator.



The 'if' statement

The `if(...)` statement evaluates a condition in parentheses and, if the result is `true`, executes a block of code.


For example:



```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');  
if (year == 2015) alert( 'You are right!' );
```

In the example above, the condition is a simple equality check (`year == 2015`), but it can be much more complex.

If we want to execute more than one statement, we have to wrap our code block inside curly braces:



```
if (year == 2015) {  
    alert( "That's correct!" );  
    alert( "You're so smart!" );  
}
```

We recommend wrapping your code block with curly braces `{}` every time you use an `if` statement, even if there is only one statement to execute. Doing so improves readability.

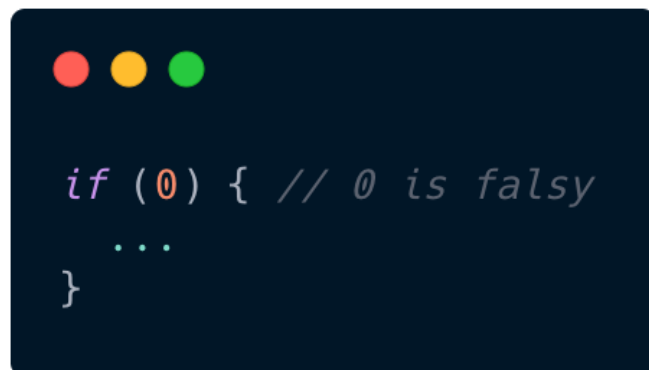
Boolean conversion

The `if (...)` statement evaluates the expression in its parentheses and converts the result to a boolean.

Let's recall the conversion rules from the chapter [Type Conversions](#):

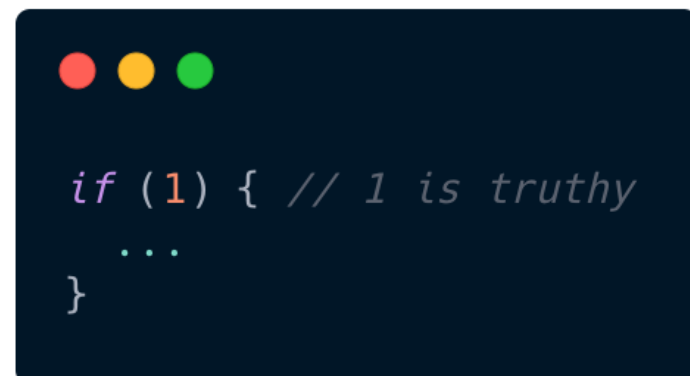
- A number `0`, an empty string `""`, `null`, `undefined`, and `NaN` all become `false`. Because of that they are called “falsy” values.
- Other values become `true`, so they are called “truthy”.

So, the code under this condition would never execute:

A code editor window with a dark blue background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following JavaScript code:

```
if (0) { // 0 is falsy
  ...
}
```

...and inside this condition – it always will:

A code editor window with a dark blue background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following JavaScript code:

```
if (1) { // 1 is truthy
  ...
}
```

The 'else' clause

The `if` statement may contain an optional “else” block. It executes when the condition is false.

For example:

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year == 2015) {
  alert( 'You guessed it right!' );
} else {
  alert( 'How can you be so wrong?' ); // any value except 2015
}
```

Several conditions: else if

Sometimes, we'd like to test several variants of a condition. The `else if` clause lets us do that.

For example:

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year < 2015) {
  alert( 'Too early...' );
} else if (year > 2015) {
  alert( 'Too late' );
} else {
  alert( 'Exactly!' );
}
```

In the code above, JavaScript first checks `year < 2015`. If that is falsy, it goes to the next condition `year > 2015`. If that is also falsy, it shows the last `alert`.

There can be more `else if` blocks. The final `else` is optional.

Conditional ?

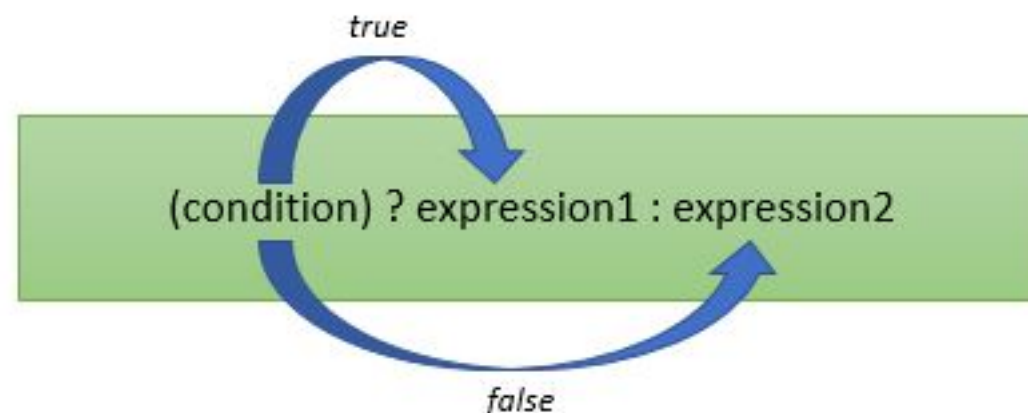
Sometimes, we need to assign a variable depending on a condition

```
let accessAllowed;  
let age = prompt('How old are you?', '');  
  
if (age > 18) {  
  accessAllowed = true;  
} else {  
  accessAllowed = false;  
}  
  
alert(accessAllowed);
```

```
let age = prompt('How old are you?', '');  
  
let accessAllowed = age > 18 ? true : false;  
alert(accessAllowed);
```

The so-called “conditional” or “question mark” operator lets us do that in a shorter and simpler way.

The operator is represented by a question mark `?`. Sometimes it’s called “ternary”, because the operator has three operands. It is actually the one and only operator in JavaScript which has that many.



Learning Resources

1. Conversions:

1. <https://javascript.info/type-conversions>
2. <https://dev.to/pixelgoo/understanding-javascript-type-conversions-43n>
3. https://www.w3schools.com/js/js_type_conversion.asp

2. Operators:

1. <https://javascript.info/operators>

3. Conditional operator:

1. https://www.tutorialspoint.com/javascript/javascript_ifelse.htm
2. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else>
3. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator
4. <https://learn.javascript.ru/ifelse>

Home Work

1. Create a new project in GitHub lesson-14-hw
2. Create a index.html file
3. Create a script.js file and attach it to index.html
4. Create a variable named availableQuantity and set it a numeric value(ex: 15).
5. Ask user to input a value that represent orderedQuantity. Assign value to a variable.
6. Check if values is greater than availableQuantity, then console log 'Sorry, we don't have enough in stock', if values is less than availableQuantity, then console log 'Great, we will soon ship your order'.
7. Complete this task both possible ways using 'ternary' and if conditional operator.