# M01 - JavaScript Fundamentals

•••

## Variables

# Variables

1. Declaration and Assignment
2. Constants
3. Naming
4. Scope

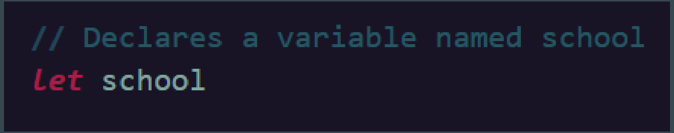
# Variables

## 1. Declaration and Assignment

# Variables

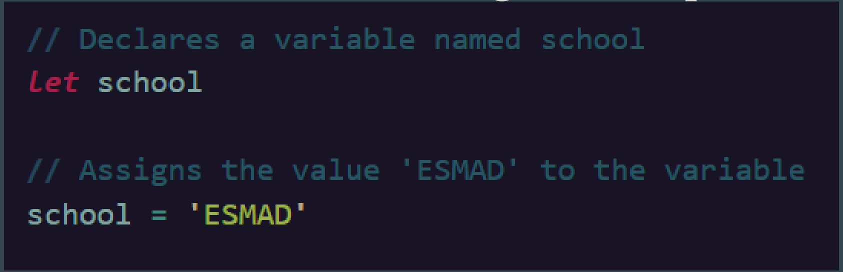
## 1. Declaration and Assignment

- Variables are named containers for storing data values.
- To declare a variable in JavaScript, use the keyword let



```
// Declares a variable named school
let school
```

- After the declaration, the variable has no value (has the value undefined)
- To assign a value to a variable, use the assignment operator =

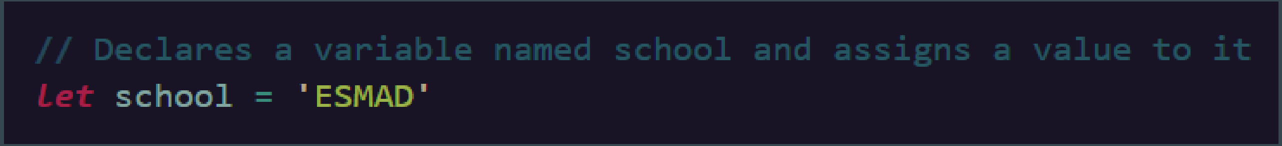


```
// Declares a variable named school
let school

// Assigns the value 'ESMAD' to the variable
school = 'ESMAD'
```

# Variables

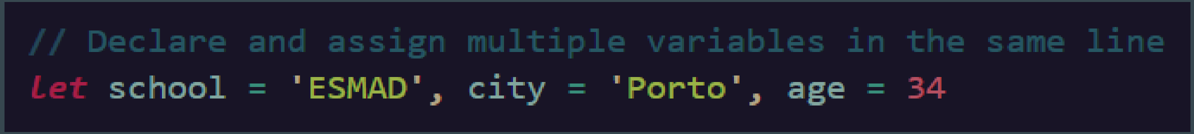
## 1. Declaration and Assignment

- To be concise, you can combine the declaration and assignment on a single line:



```
// Declares a variable named school and assigns a value to it
let school = 'ESMAD'
```

- You can also declare multiple variables on a single line:



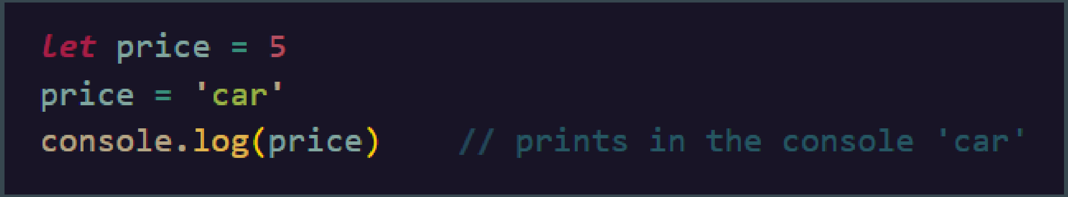
```
// Declare and assign multiple variables in the same line
let school = 'ESMAD', city = 'Porto', age = 34
```

# Variables

## 1. Declaration and Assignment

- A variable in JS can have a value of any type
- Unlike other languages, you do not need to tell JavaScript during variable declaration what type of value the variable will keep

  The value type of a variable can change during the execution of a program and JavaScript takes care of that automatically



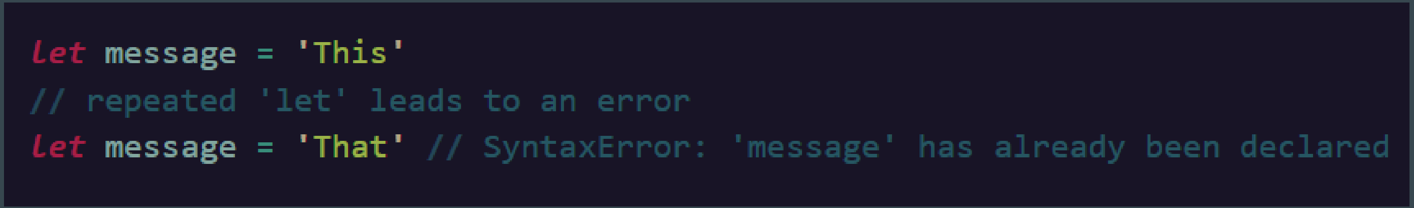
```
let price = 5
price = 'car'
console.log(price)    // prints in the console 'car'
```

- This feature is called dynamic typing

# Variables

## 1. Declaration and Assignment

- A variable should be declared only once
- A repeated declaration of the same variable is an error:



```
let message = 'This'
// repeated 'let' leads to an error
let message = 'That' // SyntaxError: 'message' has already been declared
```

- So, we should declare a variable once and then refer to it without let

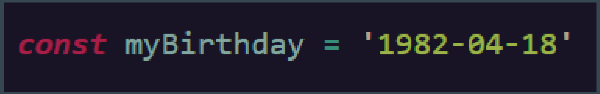
# Variables

## 2. Constants

# Variables

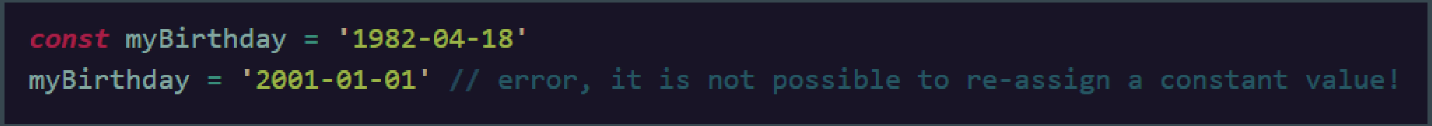
## 2. Constants

- To declare a constant (immutable) variable, use const instead of let:



```
const myBirthday = '1982-04-18'
```

- Variables declared using const are called "constants"
- They cannot be changed. An attempt to do so would cause an error:



```
const myBirthday = '1982-04-18'
myBirthday = '2001-01-01' // error, it is not possible to re-assign a constant value!
```

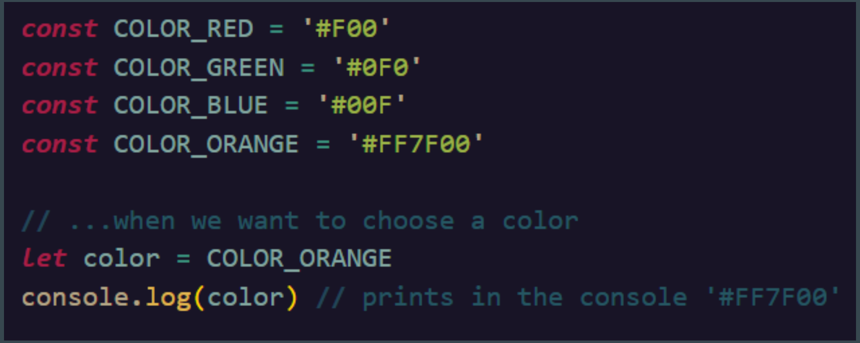
- When a programmer is sure that a variable will never change, he must declare it with const to ensure and clearly communicate that fact to everyone

# Variables

## 2. Constants

- There is a widespread practice of using constants as *aliases* for hard-to-remember values known before execution

- These constants are named using capital letters and *underlines*

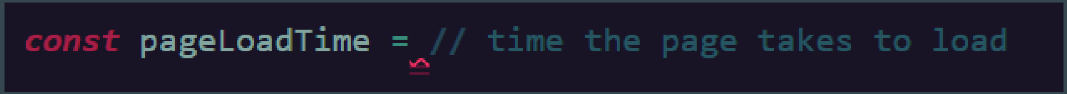


```
const COLOR_RED = '#F00'
const COLOR_GREEN = '#0F0'
const COLOR_BLUE = '#00F'
const COLOR_ORANGE = '#FF7F00'

// ...when we want to choose a color
let color = COLOR_ORANGE
console.log(color) // prints in the console '#FF7F00'
```

# Variables

## 2. Constants

- There are constants that
  - are known before execution (as a hexadecimal value for red)
  - are calculated at run time, but are not changed after the initial assignment



```
const pageLoadTime = // time the page takes to load
```

- The pageLoadTime value is not known before the page loads, so it is named normally. But it's a constant because it doesn't change after the assignment
- In other words, capitalized constants are used only as aliases for *hard-coded* values

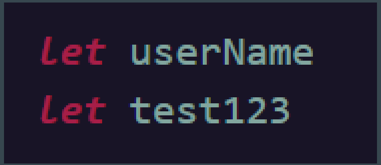
# Variables

## 3. Naming

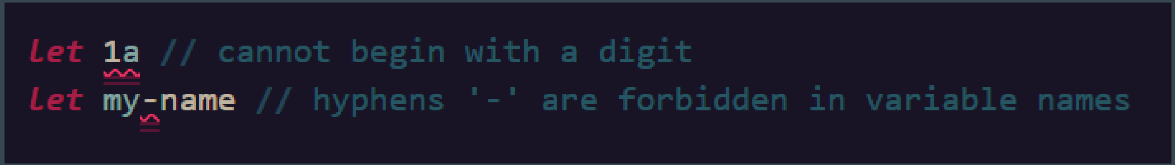
# Variables

## 3. Naming

- There are two limitations to variable names in JavaScript:
    - The name must contain only letters, digits or the symbols $ and _
    - The first character cannot be a digit
- Examples of valid and invalid names:



```
let userName
let test123
```



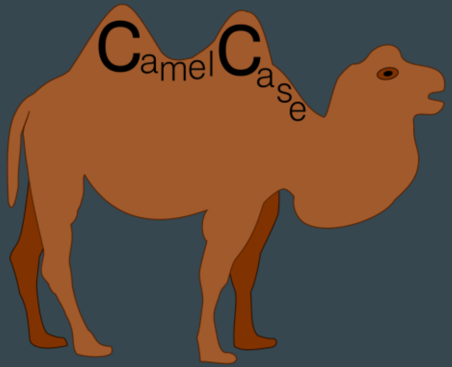
```
let 1a // cannot begin with a digit
let my-name // hyphens '-' are forbidden in variable names
```

# Variables

## 3. Naming

- Do not use names that are too short or names that are too long
- When the name contains multiple words, camelCase is commonly used
    - Words go one after the other, each beginning with a capital letter
    - Example: myVeryLongName

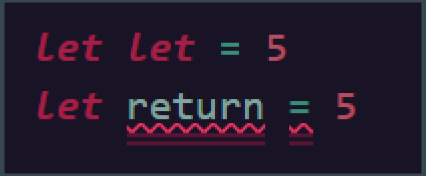
- Other naming conventions: snake_case



# Variables

## 3. Naming

- Upper and lowercase
  - Variable names are case sensitive
  - The apple and AppLE variable names are 2 different variables
- Reserved words
  - There is also a list of reserved words, which cannot be used as variable names because they are used by the language itself
  - For example: let, class, return and function are reserved



```
let let = 5
let return = 5
```

# Variables

## 3. Naming

- Good practices:
  - Use readable names like userName or shoppingCart
  - Don't use abbreviations or short names like a, b, c, unless you really know what you're doing
  - Create descriptive and concise names. Examples of invalid names are data and value. These names say nothing. There is no problem using them if the context of the code makes it exceptionally obvious what data or values the variable is referencing

# Variables

## 3. Naming

- Good practices:
  - Take into account the terms used by the development team. If a site visitor is called a user, we should name related variables currentUser or newUser instead of currentVisitor or newManInTown

  - Don’t mix natural languages (portuguese with english)
    - Use evenly only one
    - Suggestion: english (reserved words of the language already in english, world wide readable code, etc.)

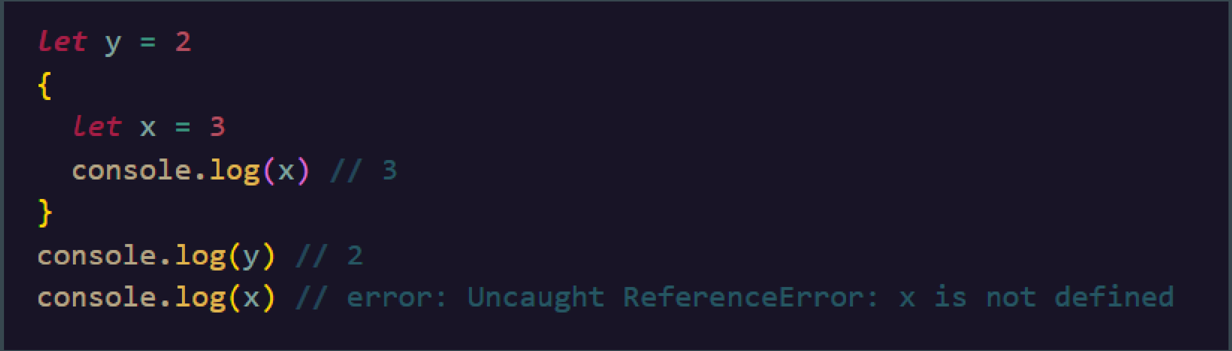
# Variables

## 4. Scope

# Variables

## 4. Scope

- The scope of a variable is the context where it was defined
- Naming variables via let or const restricts the variable's access to the nearest surrounding block
- A JavaScript block is represented by { ... }



```
let y = 2
{
  let x = 3
  console.log(x) // 3
}
console.log(y) // 2
console.log(x) // error: Uncaught ReferenceError: x is not defined
```

# M01 - JavaScript Fundamentals

•••

## Data Types

# Data Types

1. Definition
2. Number
3. BigInt
4. String
5. Boolean
6. Null and Undefined
7. Object and Symbol
8. Type Conversions

# Data Types

## 1. Definition

# Data Types

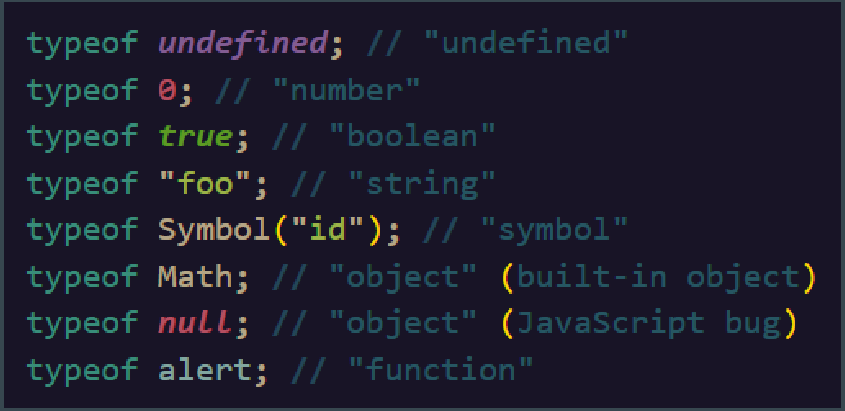
## 1. Definition

- A value in JavaScript is always of a certain type (e.g. a string, number)
- There are eight basic data types in JavaScript:
  - Number for numbers of any type: integer or floating point
  - BigInt to represent integers of arbitrary length
  - String for texts. Can have one or more characters, there is no separate single character type
  - Boolean for true/false
  - null for unknown values: an independent type that has a single null value
  - undefined for unassigned values: an independent type that has a single undefined value
  - Object for more complex data structures
  - Symbol for unique identifiers

# Data Types

## 1. Definition

- Type checking is done by the typeof operator
- Returns the type of the argument. It is useful when we want to process values of different types differently or just do a quick check.
- Supports two forms of syntax:
  - As an operator: typeof x
  - As a function: typeof(x)
- The return is a string with the type name



```
typeof undefined; // "undefined"
typeof 0; // "number"
typeof true; // "boolean"
typeof "foo"; // "string"
typeof Symbol("id"); // "symbol"
typeof Math; // "object" (built-in object)
typeof null; // "object" (JavaScript bug)
typeof alert; // "function"
```

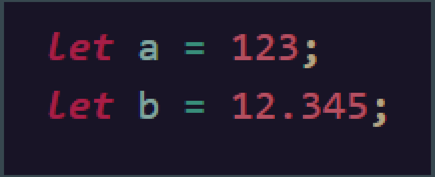
# Data Types

## 2. Number

# Data Types

## 2. Number

- The number type represents integers and floating point numbers



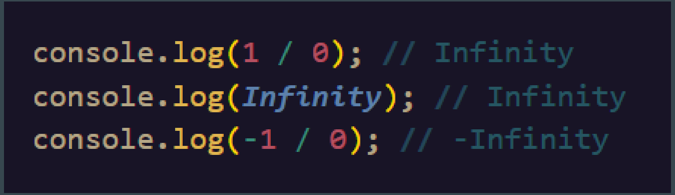
```
let a = 123;
let b = 12.345;
```

- In addition to regular numbers, there are so-called “special numeric values” that also belong to this type of data: Infinity, -Infinity and NaN

# Data Types

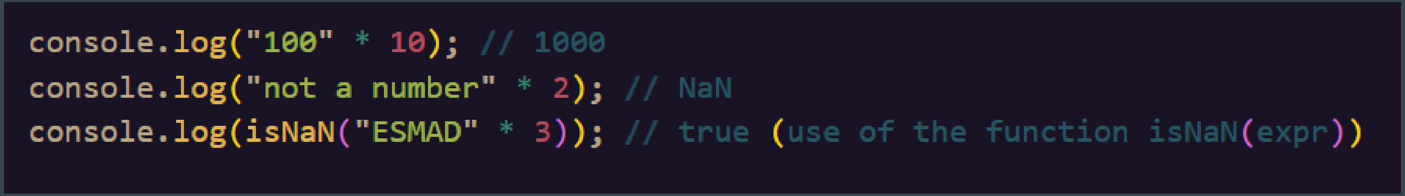
## 2. Number

- The Infinity value represents the mathematical infinity ∞. It is a special value that is greater than any number. We can get it as a result of a division by zero:



```
console.log(1 / 0); // Infinity
console.log(Infinity); // Infinity
console.log(-1 / 0); // -Infinity
```

- The NaN (Not a Number) value represents a computational error - incorrect or undefined mathematical operation:



```
console.log("100" * 10); // 1000
console.log("not a number" * 2); // NaN
console.log(isNaN("ESMAD" * 3)); // true (use of the function isNaN(expr))
```

# Data Types

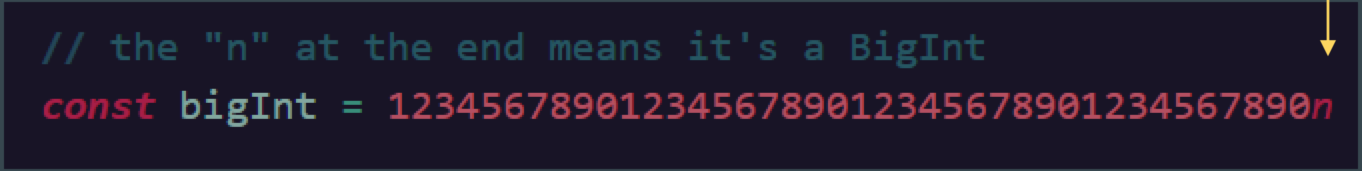
## 3. BigInt

# Data Types

## 3. BigInt

- In JavaScript, the “number” type cannot represent integer values larger than ($2^{53}$-1) (that’s 9007199254740991), or less than -($2^{53}$-1) for negatives
- For most purposes that’s quite enough, but sometimes we need really big numbers, e.g. for cryptography or microsecond-precision timestamps.

- BigInt type represents integers of arbitrary length.
- A BigInt value is created by appending n to the end of an integer:



```
// the "n" at the end means it's a BigInt
const bigInt = 1234567890123456789012345678901234567890n
```

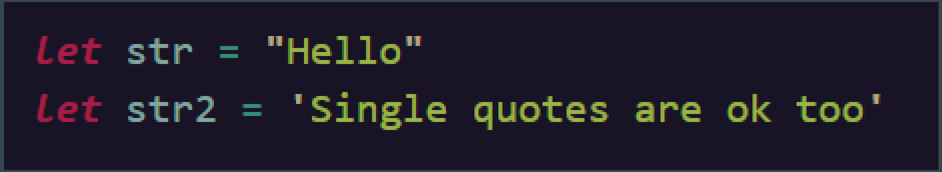
# Data Types

## 4. String

# Data Types

## 4. String

- Used to store and manipulate text (string of alphanumeric characters)
- A string value can be between:
  - Double quotes: "Olá"
  - Single quotes: 'Olá'
  - Backticks: `Olá`
- There is no difference between double quotes and single quotes in JavaScript

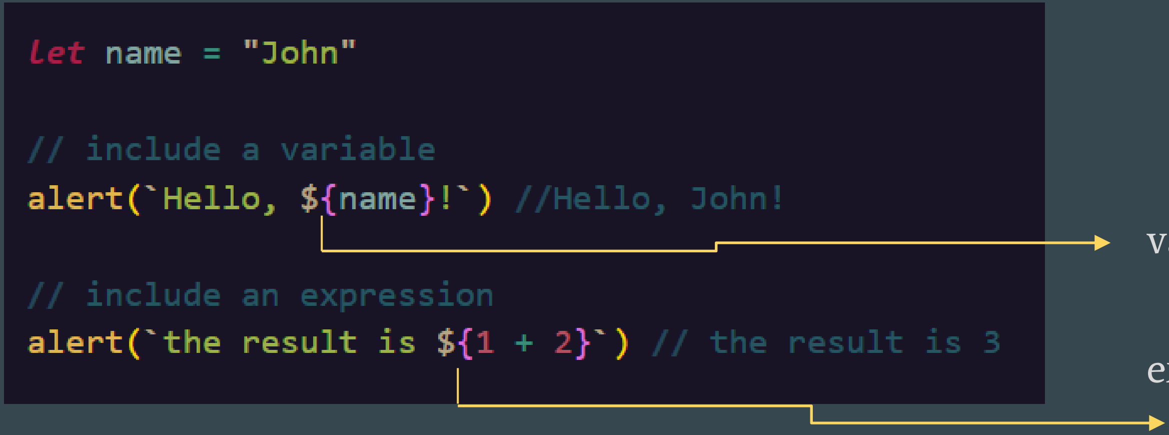


```
let str = "Hello"
let str2 = 'Single quotes are ok too'
```

# Data Types

## 4. String

- Backticks are “extended functionality” quotes because they allow you to incorporate variables and expressions in a string, wrapping them in ${...}



```
let name = "John"

// include a variable
alert(`Hello, ${name}!`) //Hello, John!

// include an expression
alert(`the result is ${1 + 2}`) // the result is 3
```

variabe content

expression content

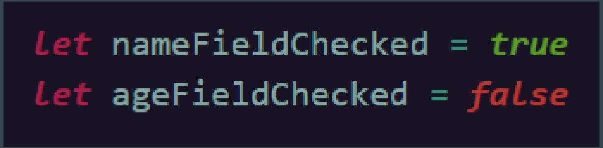
# Data Types

## 5. Boolean

# Data Types

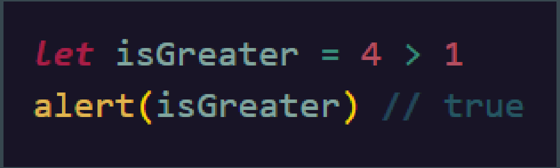
## 5. Boolean

- The boolean type has only two values: true and false
- This type is commonly used to store yes/no values: true means "yes, correct" and false means "no, incorrect"



```
let nameFieldChecked = true
let ageFieldChecked = false
```

- Boolean values also appear as a result of comparisons:



```
let isGreater = 4 > 1
alert(isGreater) // true
```

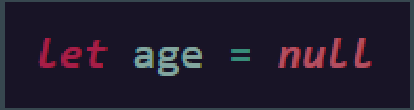
# Data Types

## 6. Null and Undefined

# Data Types

## 6. Null and Undefined

- Null
  - The null value does <u>not belong to any of the types </u>described previously
  - It forms a separate type that contains only the null value:



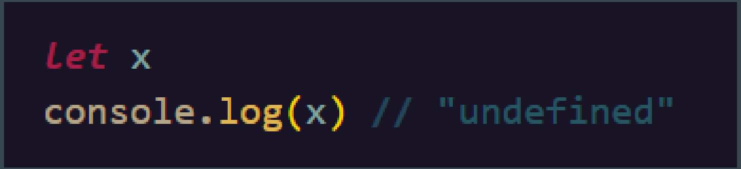
```
let age = null
```

  - It is not a “reference to a non-existent object” or “null pointer” as in other languages
  - It is just a special value that represents "nothing", "empty" or "unknown value"
  - The above code indicates that the age is unknown or is empty for some reason

# Data Types

## 6. Null and Undefined

- Undefined
  - Means “unassigned value”
  - If a variable is declared, but no value is assigned, its value is undefined:



```
let x
console.log(x) // "undefined"
```

  - Good practices:
    - use null to assign a value "empty" or "unknown" to a variable
    - use undefined to check if a value has been assigned to a variable

# Data Types

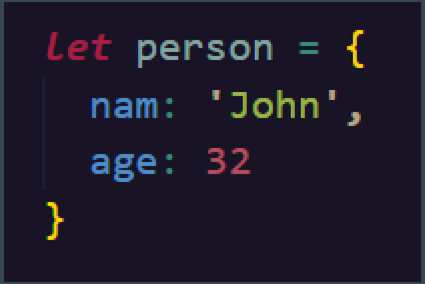
## 7. Object and Symbol

# Data Types

## 7. Object and Symbol

- Object
  - The object type is special
  - All other types are called "primitives" because their values can contain only one thing (be it a string or a number...)
  - Objects are used to store collections of data/complex entities
- Symbol
  - Special type used to create unique identifiers for objects

- Both types will be discussed later in these slides



```
let person = {
  nam: 'John',
  age: 32
}
```

# Data Types

## 8. Type Conversions

# Data Types

## 8. Type Conversions

- Most of the time, operators and functions automatically convert the values assigned to them to the correct type. This is called type conversion.

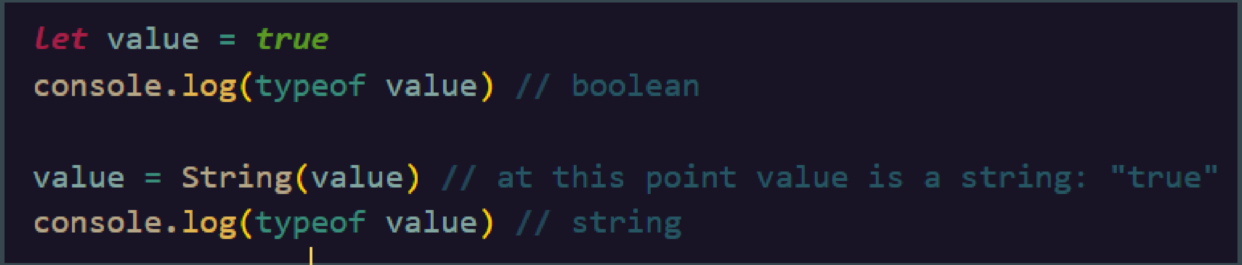
- For example, the alert automatically converts any value to a string in order to display it. Mathematical operations convert values into numbers.

- The three most used type conversions are:
  - conversion to string
  - conversion to number
  - conversion to boolean

# Data Types

## 8. Type Conversions

- Conversion to string
  - String conversion happens when we need the string form of a value
  - For example, alert(value) does this to show the value
  - We can also call the String(value) function to convert a value to a string:



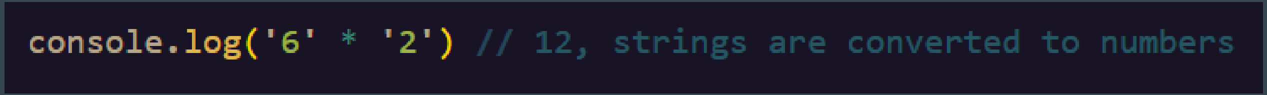
```
let value = true
console.log(typeof value) // boolean

value = String(value) // at this point value is a string: "true"
console.log(typeof value) // string
```

Typeof operator: returns the type of a variable or an expression

# Data Types

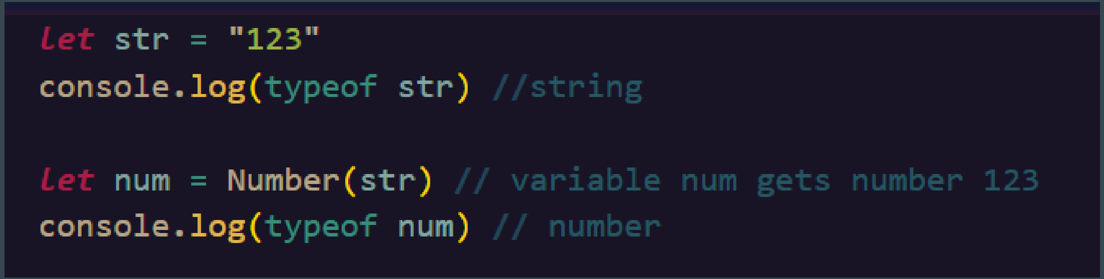
## 8. Type Conversions

- Conversion to number
  - Numeric conversion happens automatically in mathematical functions and expressions
  - For example, when multiplication * is applied to non-numbers:
  -



```
console.log('6' * '2') // 12, strings are converted to numbers
```

  - We can use the Number(value) function to explicitly convert a value to a number

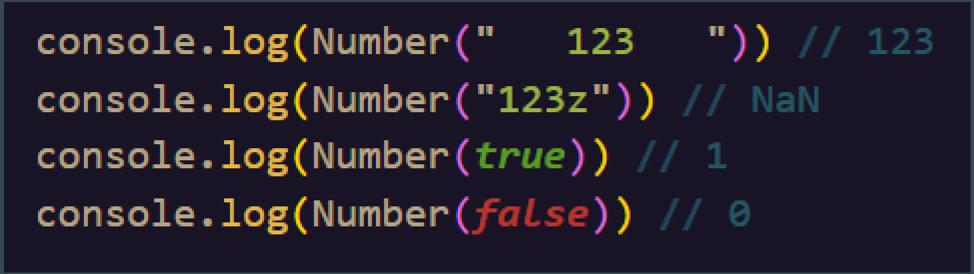


```
let str = "123"
console.log(typeof str) //string

let num = Number(str) // variable num gets number 123
console.log(typeof num) // number
```

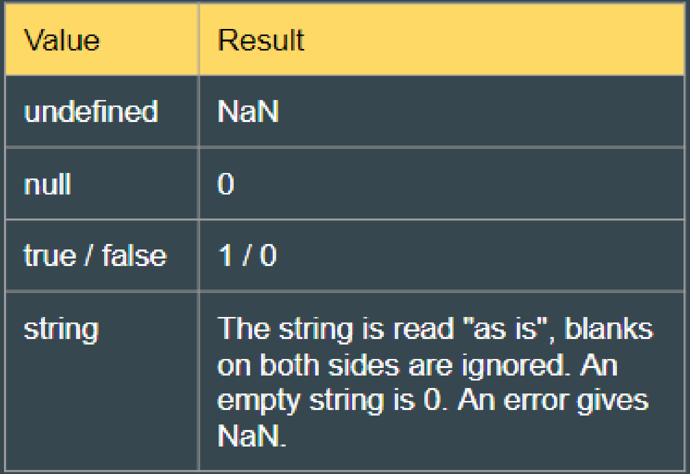
# Data Types

## 8. Type Conversions

- Conversion to number (rules)



```
console.log(Number("   123   ")) // 123
console.log(Number("123z")) // NaN
console.log(Number(true)) // 1
console.log(Number(false)) // 0
```

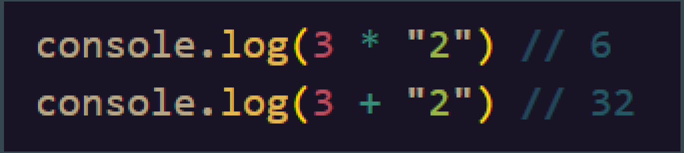


| Value | Result |
|---|---|
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| string | The string is read "as is", blanks on both sides are ignored. An empty string is 0. An error gives NaN. |

# Data Types

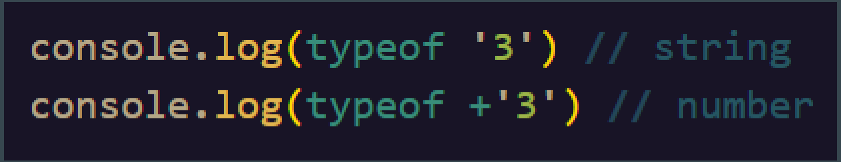
## 8. Type Conversions

- Conversion to number
  - Almost all mathematical operations convert values into numbers.
  - A notable exception is the addition +. If one of added values is a string, the other is also converted to a string by joining them:



```
console.log(3 * "2") // 6
console.log(3 + "2") // 32
```

  - Using the + sign immediately before a string it will try to convert to number



```
console.log(typeof '3') // string
console.log(typeof +'3') // number
```

# M01 - JavaScript Fundamentals

•••

## Operators

# Operators

1. Definition
2. Arithmetic
3. Concatenation
4. Assignment
5. Rest and Exponentiation
6. Increment/Decrement
7. Comparison
8. Logic

# Operators

## 1. Definition

# Operators

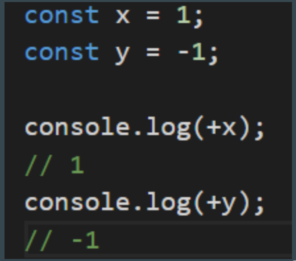
## 1. Definition

- There are several types of operators in JavaScript
- Types of operators:
  - Arithmetic (+, -, *, /)
  - Concatenation (+)
  - Assignment (=)
  - Rest (%)
  - Exponentiation (**)
  - Increment/Decrement (++, --)
  - Comparison (<, >, <=, >=, ==, !=, ===, !==)
  - Logic (||, && e !)

# Operators

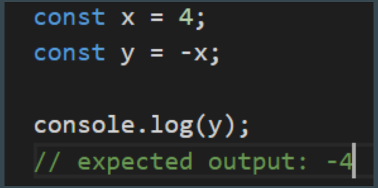
## 1. Definition

- Operator precedence



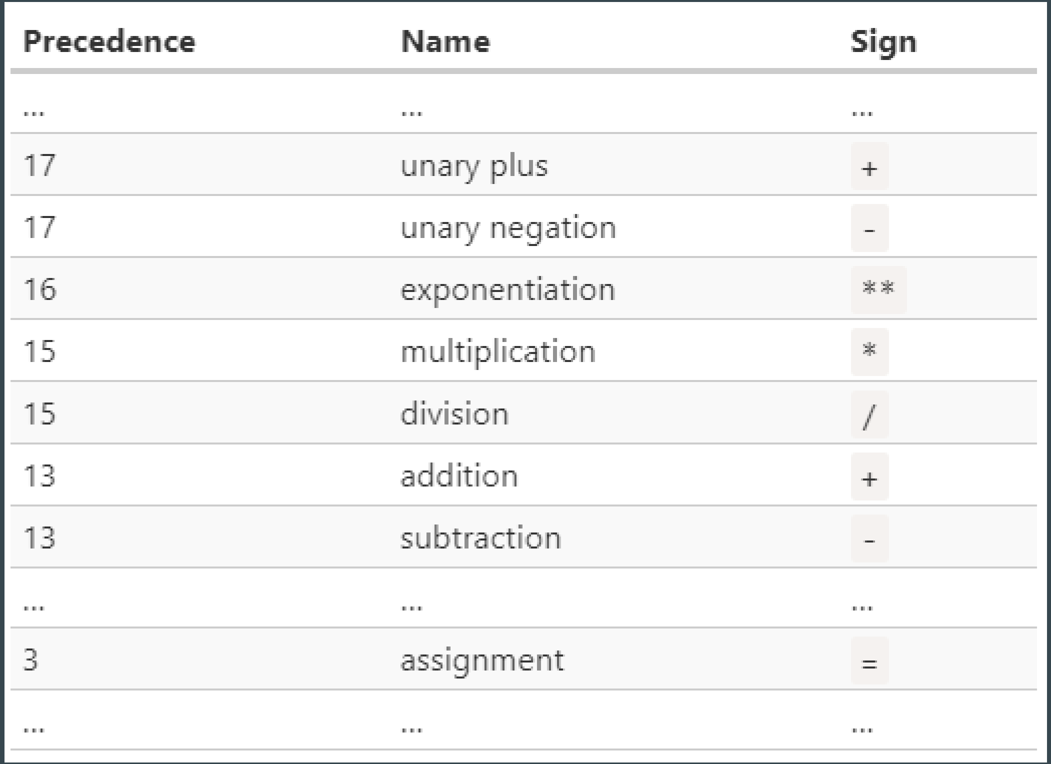
```
const x = 1;
const y = -1;

console.log(+x);
// 1
console.log(+y);
// -1
```

operator that precedes the operand



```
const x = 4;
const y = -x;

console.log(y);
// expected output: -4
```

precedes its operand and negates it



| Precedence | Name | Sign |
|---|---|---|
| ... | ... | ... |
| 17 | unary plus | + |
| 17 | unary negation | - |
| 16 | exponentiation | ** |
| 15 | multiplication | * |
| 15 | division | / |
| 13 | addition | + |
| 13 | subtraction | - |
| ... | ... | ... |
| 3 | assignment | = |
| ... | ... | ... |

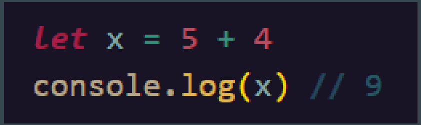
# Operators

## 2. Arithmetic

# Operators

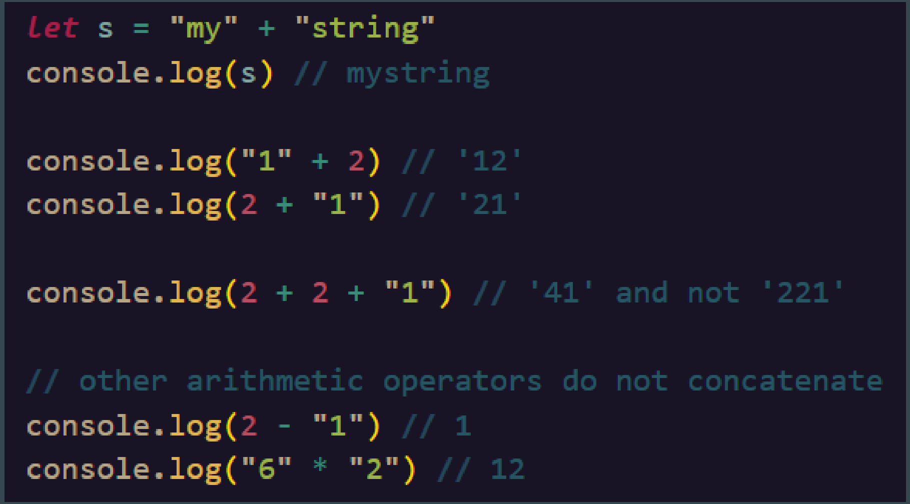
## 2. Arithmetic Operators (+, -, *, /)

- Used to perform arithmetic operations on numbers (literals or variables)
- Examples of basic operations: +, -, *, /



```
let x = 5 + 4
console.log(x) // 9
```

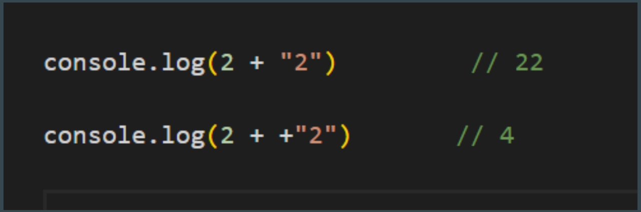
- The + operator if applied to strings it concatenates them
- If one of the operands is a string, the other will also be converted to a string



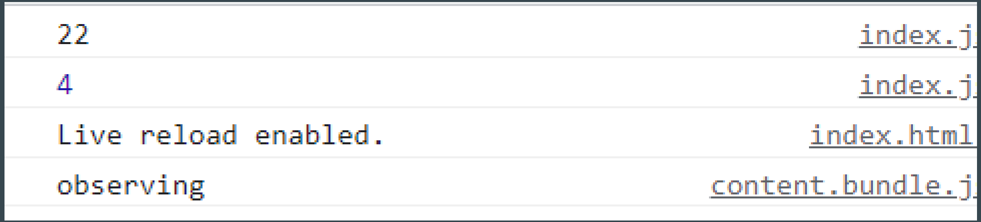
```
let s = "my" + "string"
console.log(s) // mystring

console.log("1" + 2) // '12'
console.log(2 + "1") // '21'

console.log(2 + 2 + "1") // '41' and not '221'

// other arithmetic operators do not concatenate
console.log(2 - "1") // 1
console.log("6" * "2") // 12
```

# Operators

## 2. Arithmetic Operators (+, -, *, /)



```
console.log(2 + "2")        // 22

console.log(2 + +"2")       // 4
```



```
22                                   index.j
4                                    index.j
Live reload enabled.                 index.html
observing                            content.bundle.j
```

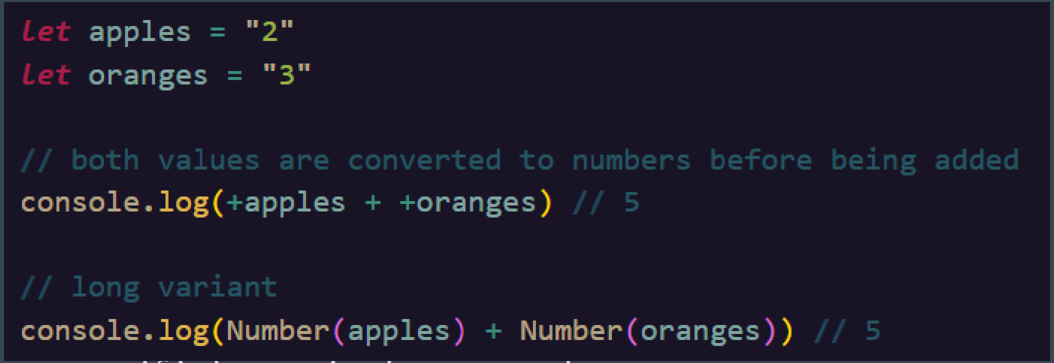
# Operators

## 3. Concatenation

# Operators

## 3. Concatenation Operators (+)

- The + operator if applied to a single value does nothing with numbers
- But if the operand is not a number, the unary + converts it to a number



```
let apples = "2"
let oranges = "3"

// both values are converted to numbers before being added
console.log(+apples + +oranges) // 5

// long variant
console.log(Number(apples) + Number(oranges)) // 5
```

- An operator is unary if it has a single operand
- For example, unary negation - inverses the sign of a number

# Operators

## 4. Assignment

# Operators

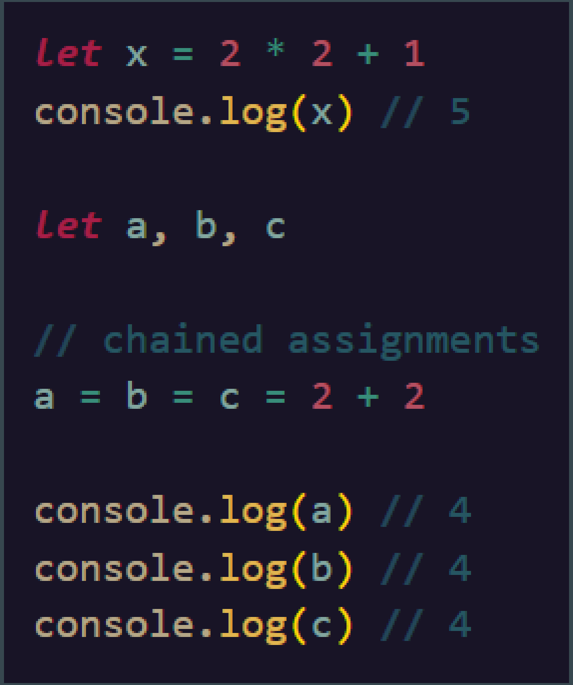
## 4. Assignment Operators (=)

- The assignment = is also an operator

- It is listed in the precedence table with a very low priority of 3

- That is why, when we assign a variable, such as x = 2 * 2 + 1, calculations are done first and then the = is evaluated, storing the result in x

- It is also possible to chain assignments



```
let x = 2 * 2 + 1
console.log(x) // 5

let a, b, c

// chained assignments
a = b = c = 2 + 2

console.log(a) // 4
console.log(b) // 4
console.log(c) // 4
```

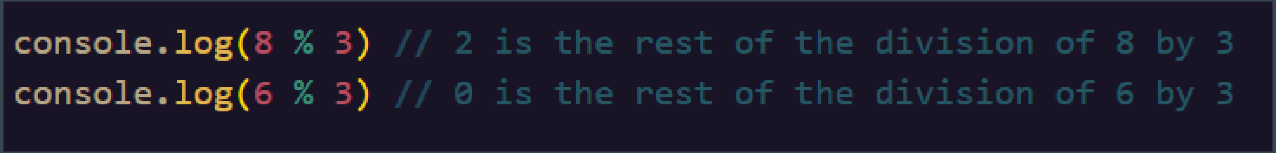
# Operators

## 5. Rest and Exponentiation

# Operators

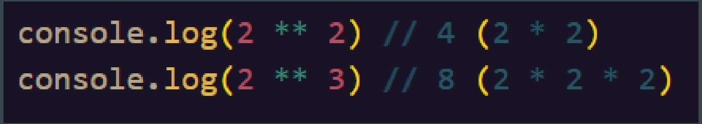
## 5. Rest (%) and Exponentiation (**) Operators

- Rest
  - The remainder operator (%), despite its appearance, is not related to percentages
  - The result of a%b is the remainder of the entire division of a by b



```
console.log(8 % 3) // 2 is the rest of the division of 8 by 3
console.log(6 % 3) // 0 is the rest of the division of 6 by 3
```

- Exponentiation
  - The exponentiation operator ** is a recent addition to the language
  - A natural number b, the result of a**b is a multiplied by itself b times



```
console.log(2 ** 2) // 4 (2 * 2)
console.log(2 ** 3) // 8 (2 * 2 * 2)
```

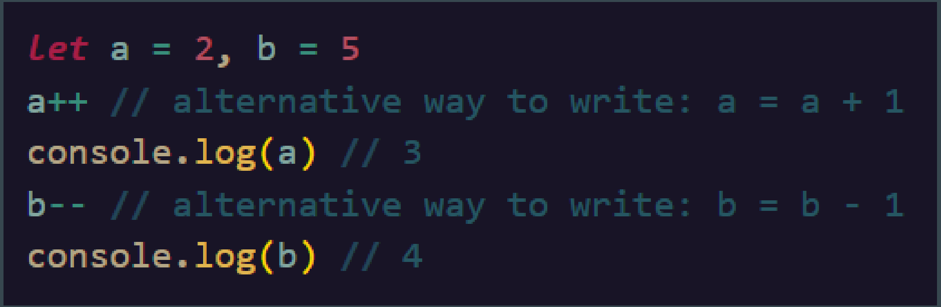
# Operators

## 6. Increment/Decrement

# Operators

## 6. Increment (++) and Decrement (--) Operators

- Increasing or decreasing a number by one is among the most common numerical operations
- There are special operators for this:
  - Increment ++ increases a variable by 1
  - Decrement -- decrease a variable by 1



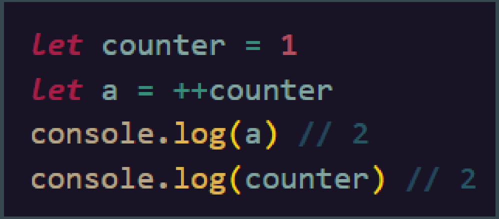
```
let a = 2, b = 5
a++ // alternative way to write: a = a + 1
console.log(a) // 3
b-- // alternative way to write: b = b - 1
console.log(b) // 4
```

# Operators

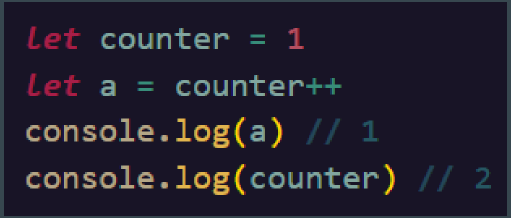
## 6. Increment (++) and Decrement (--) Operators

- ++ and -- operators can be placed before or after a variable
  - When the operator is after the variable, it is in the postfix form: counter ++
  - When the operator is before the variable, it is in the prefix form: ++counter
- Both statements do the same thing: increase the counter by 1

- So what's the difference?
  - The prefix returns the new value and postfix returns the old value (before the increment/decrement)



```
let counter = 1
let a = ++counter
console.log(a) // 2
console.log(counter) // 2
```



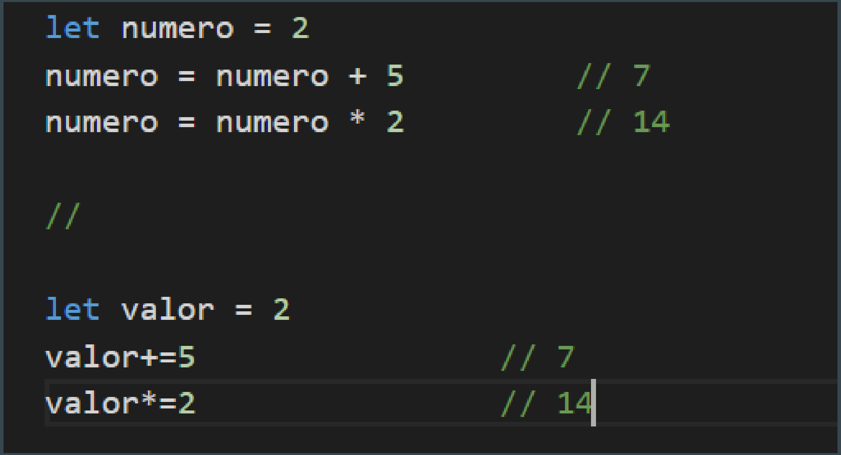
```
let counter = 1
let a = counter++
console.log(a) // 1
console.log(counter) // 2
```

# Operators

## 6. Increment (++) and Decrement (--) Operators

- Generally, we need to apply an operator to a variable and store the new result in that same variable

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



```
let numero = 2
numero = numero + 5         // 7
numero = numero * 2         // 14

//

let valor = 2
valor+=5                // 7
valor*=2                // 14
```

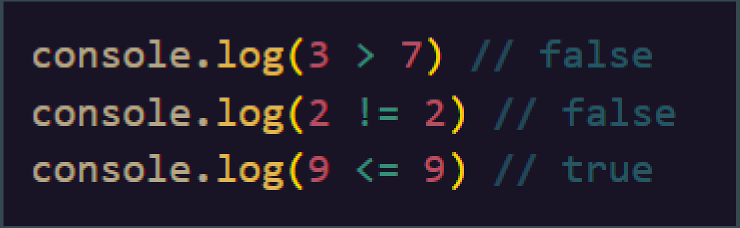
# Operators

## 7. Comparison

# Operators

## 7. Comparison Operators  (>, <, >=, <=, ==, !=, ===, !==)

- We know many math comparison operators:
  - Greater/less than: a > b, a < b
  - Greater/less than or equal to: a >= b, a <= b
  - Equal: a == b (note the double equal sign =. A single symbol a = b would mean an assignment)
  - Not equal. In mathematics, notation is ≠, but in JavaScript it is written as an assignment with an exclamation sign before it: a != b
- A comparison returns a Boolean value

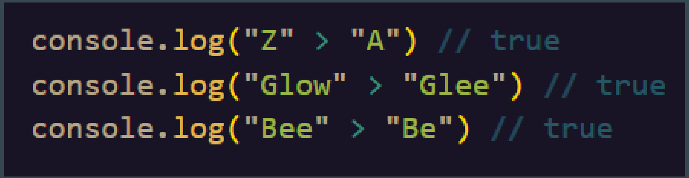


```
console.log(3 > 7) // false
console.log(2 != 2) // false
console.log(9 <= 9) // true
```

# Operators

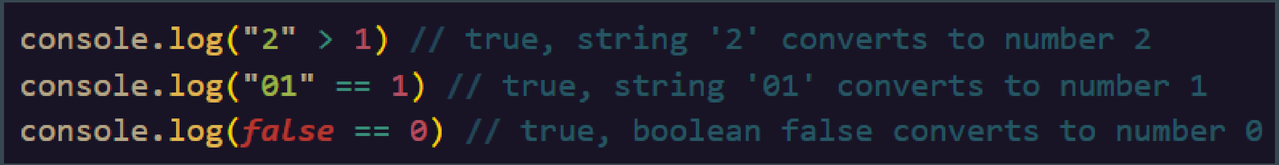
## 7. Comparison Operators (>, <, >=, <=, ==, !=, ===, !==)

- String comparison
  - To see if a string is longer than another, JS uses the so-called dictionary or lexicographic
  - In other words, strings are compared letter by letter



```
console.log("Z" > "A") // true
console.log("Glow" > "Glee") // true
console.log("Bee" > "Be") // true
```

- Comparison of different types
  - When comparing values of different types, JavaScript converts values into numbers

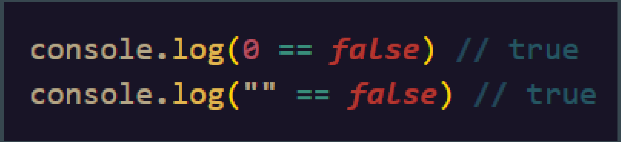


```
console.log("2" > 1) // true, string '2' converts to number 2
console.log("01" == 1) // true, string '01' converts to number 1
console.log(false == 0) // true, boolean false converts to number 0
```

# Operators

## 7. Comparison Operators (>, <, >=, <=, ==, !=, ===, !==)

- Strict equality
  - A regular equality check == has a problem
  - It is not possible to differentiate, for example, 0 from false:



```
console.log(0 == false) // true
console.log("" == false) // true
```

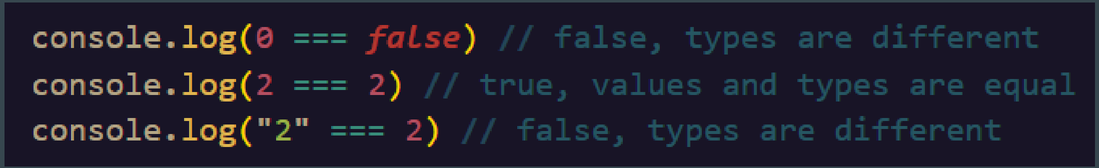
  - This is because operands of different types are converted to numbers by the equality operator ==. An empty string, as a false, becomes a zero.

  - Here comes the strict equality operator === which checks for equality <u>without type conversion</u>

# Operators

## 7. Comparison Operators (>, <, >=, <=, ==, !=, ===, !==)

- Strict equality
  - A strict equality operator === checks for 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.



```
console.log(0 === false) // false, types are different
console.log(2 === 2) // true, values and types are equal
console.log("2" === 2) // false, types are different
```

  - There is also a strict non-egalitarian operator !== analogous to !=

# Operators

## 8. Logic

# Operators

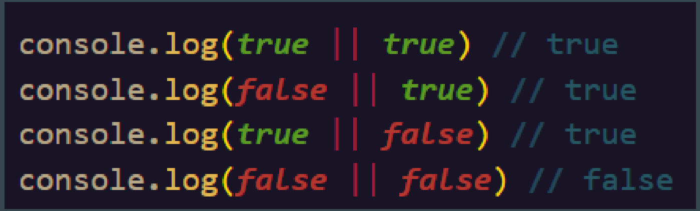
## 8. Logical Operators (||, && e !)

- There are three logical operators in JavaScript:
  - || (OR)
  - && (AND)
  - ! (NOT)
- Although they are called logical, they can be applied to values of any type, not just Booleans.
- Their result can also be of any kind

# Operators

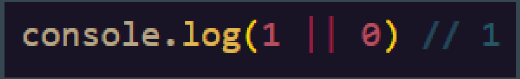
## 8. Logical Operators (||, && e !)

- Operator || (OR)
    - In classical programming, the logical OR is intended to manipulate only Boolean values
    - If any of its arguments are true, it will return true, otherwise it will return false



```
console.log(true || true) // true
console.log(false || true) // true
console.log(true || false) // true
console.log(false || false) // false
```

- If an operand is not Boolean, it will be converted to Boolean for evaluation
- For example, the number 1 is treated as true, the number 0 as false:

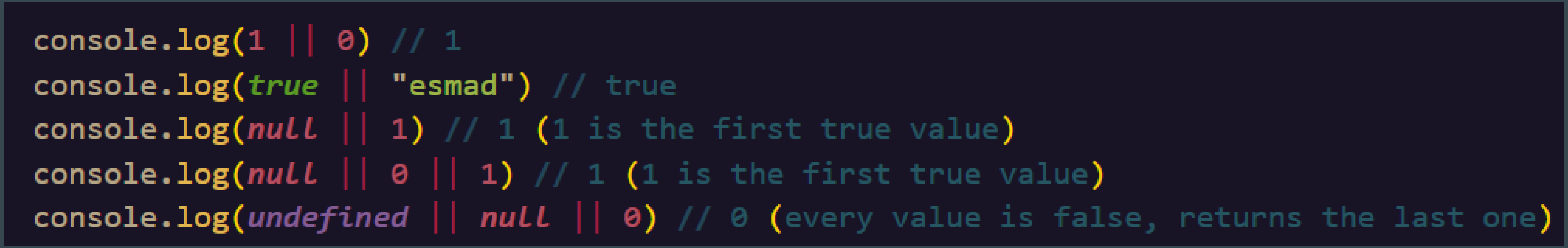


```
console.log(1 || 0) // 1
```

# Operators

## 8. Logical Operators (||, && e !)

- Operator || (OR)
    - Given various OR values: result = value1 || value2 || value3;
    - The operator || does the following:
        - Evaluates operands from left to right
        - For each operand, convert it to Boolean. If the result is true, it will stop and return the original value of that operand.
        - If all operands have been evaluated (all false), returns the last operand.
    - A value is returned in its original form, without conversion.

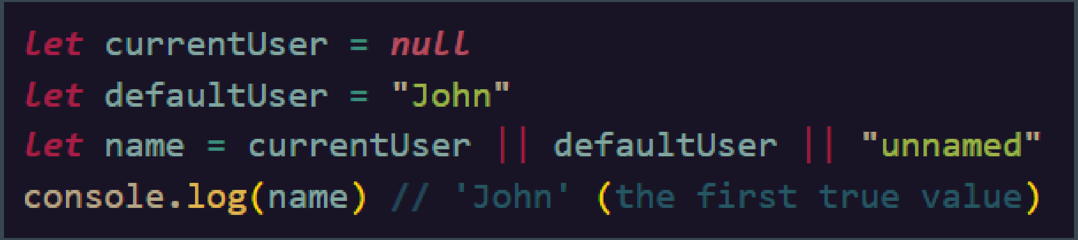


```
console.log(1 || 0) // 1
console.log(true || "esmad") // true
console.log(null || 1) // 1 (1 is the first true value)
console.log(null || 0 || 1) // 1 (1 is the first true value)
console.log(undefined || null || 0) // 0 (every value is false, returns the last one)
```

# Operators

## 8. Logical Operators (||, && e !)

- Operator || (OR)
  - Useful for obtaining the first true value from a list of variables or expressions
  - Imagine that we have several variables that can contain data or be null/undefined. How can we find the first one with data?

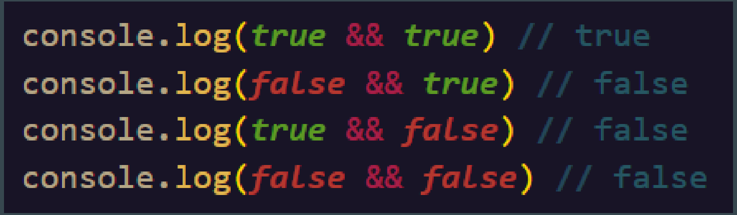


```
let currentUser = null
let defaultUser = "John"
let name = currentUser || defaultUser || "unnamed"
console.log(name) // 'John' (the first true value)
```

# Operators

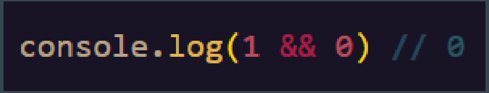
## 8. Logical Operators (||, && e !)

- Operator && (AND)
  - The AND operator returns true if both operands are true and false otherwise



```
console.log(true && true) // true
console.log(false && true) // false
console.log(true && false) // false
console.log(false && false) // false
```

  - If an operand is not Boolean, it will be converted to Boolean for evaluation
  - For example, the number 1 is treated as true, the number 0 as false:

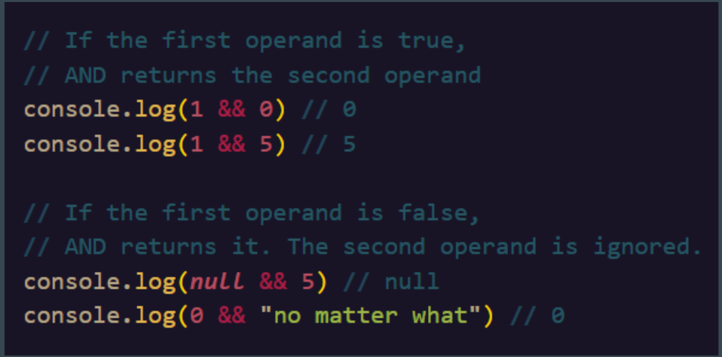


```
console.log(1 && 0) // 0
```

# Operators

## 8. Logical Operators (||, && e !)

- Operator && (AND)
    - Several values: result = value1 && value2 && value3;
    - The && operator does the following:
        - Evaluates operands from left to right
        - For each operand, convert it to a Boolean. If the result is false, it will stop and return the original value of that operand.
        - If all operands have been evaluated (that is, they were all true), the last operand will be returned.
    - In other words, the && operator returns the first false value or the last value, if they are all true.

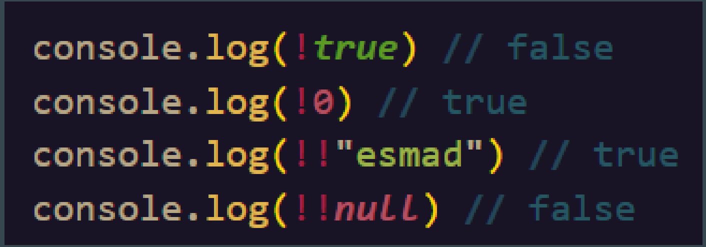


```
// If the first operand is true,
// AND returns the second operand
console.log(1 && 0) // 0
console.log(1 && 5) // 5

// If the first operand is false,
// AND returns it. The second operand is ignored.
console.log(null && 5) // null
console.log(0 && "no matter what") // 0
```

# Operators

## 8. Logical Operators (||, && e !)

- Operator ! (NOT)
  - The operator accepts a single argument and does the following:
    - Converts the operand to a Boolean type: true/false
    - Returns the inverse value
    - A double !! is used to convert a value to a Boolean type



```
console.log(!true) // false
console.log(!0) // true
console.log(!!"esmad") // true
console.log(!!null) // false
```

# M01 - JavaScript Fundamentals

•••

## Conditionals

# Conditionals

- Sometimes, we need to take different actions based on different conditions
- To do that, we use:
    - if - to specify a block of code to be executed, if certain condition is true
    - else - to specify a block of code to be executed, if the same condition is false
    - else...if - to specify a new condition to test, if the first condition is false
    - ? - ternary operator
    - switch - to specify alternative blocks of code to run

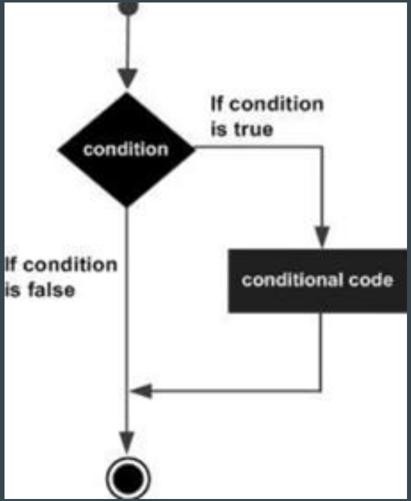
# Conditionals

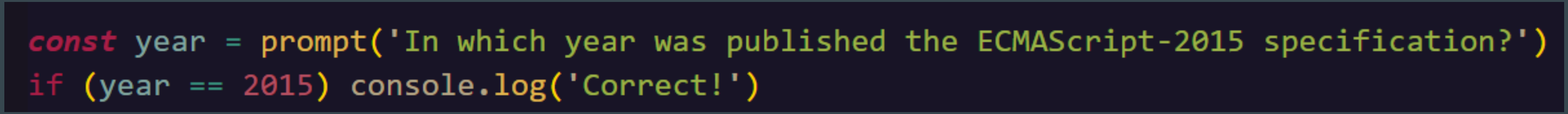
## 1. IF statement

# Conditionals

## 1. IF statement

- The if statement evaluates a condition and, if the result of the condition is true, executes a block of code



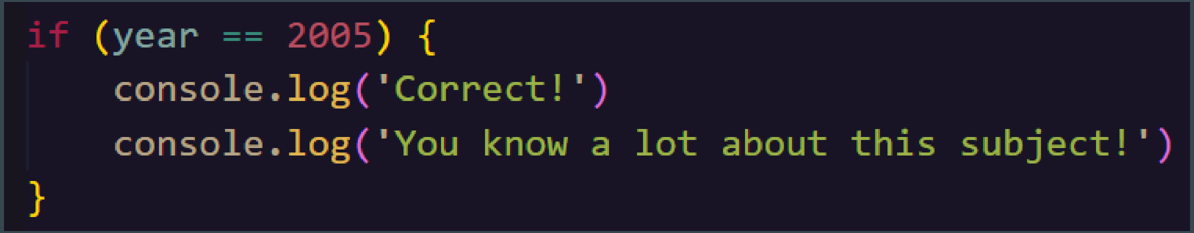


```
const year = prompt('In which year was published the ECMAScript-2015 specification?')
if (year == 2015) console.log('Correct!')
```

# Conditionals

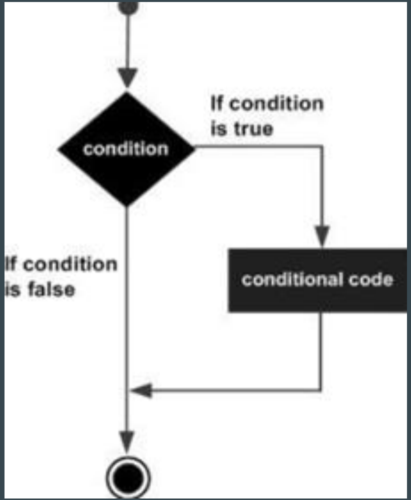
## 1. IF statement

- If we want to execute more than one declaration, we have to wrap our code block within curly braces:



```
if (year == 2005) {
    console.log('Correct!')
    console.log('You know a lot about this subject!')
}
```

- It is recommended that you always add the block of code with curly braces {}, even if there is only one instruction to be executed. This improves readability!



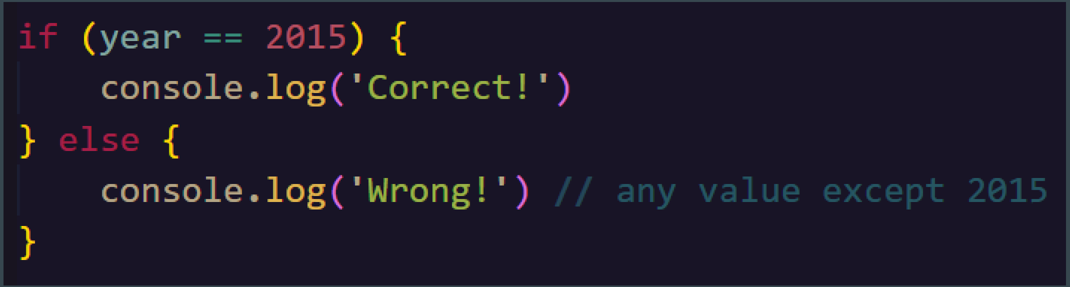
# Conditionals

## 2. ELSE statement

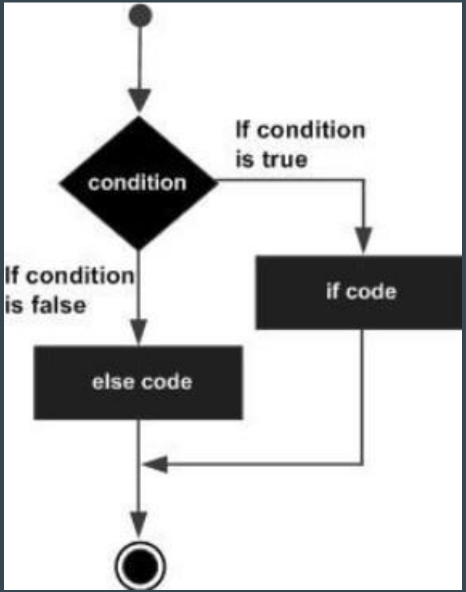
# Conditionals

## 2. ELSE statement

- The if statement can contain an optional else block
- It is executed when the condition is false



```
if (year == 2015) {
    console.log('Correct!')
} else {
    console.log('Wrong!') // any value except 2015
}
```



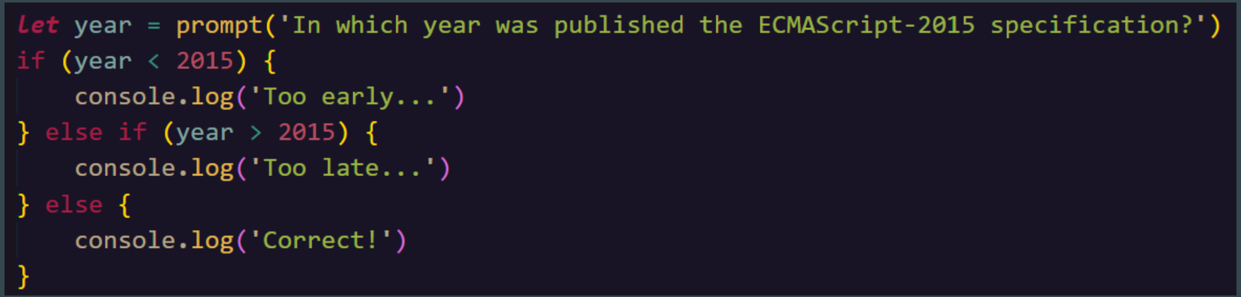
# Conditionals

## 3. ELSE...IF statement

# Conditionals

## 3. ELSE...IF statement

- Sometimes, we would like to test several variants of a condition
- The else if clause allows us to do that



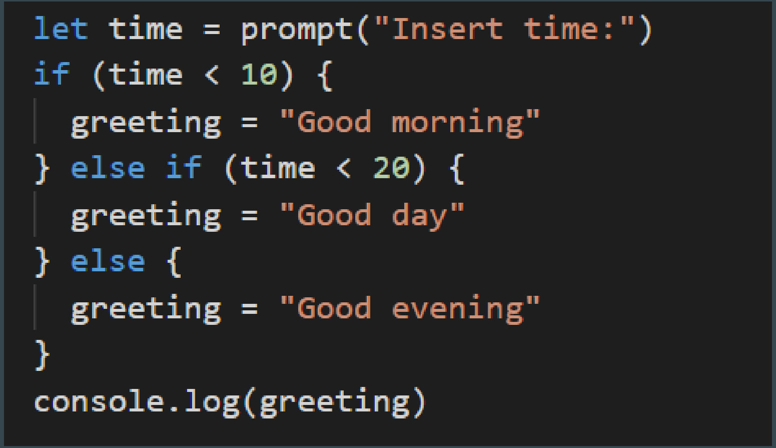
```
let year = prompt('In which year was published the ECMAScript-2015 specification?')
if (year < 2015) {
    console.log('Too early...')
} else if (year > 2015) {
    console.log('Too late...')
} else {
    console.log('Correct!')
}
```

- In the code above, JavaScript first checks the year < 2015. If it is false, it goes to the next year > 2015 condition. If it is also false, it shows the last alert.
- There may be more else if blocks. The final else is optional

# Conditionals

## 3. ELSE...IF statement

- Sometimes, we would like to test several variants of a condition
- The else if clause allows us to do that



```
let time = prompt("Insert time:")
if (time < 10) {
  greeting = "Good morning"
} else if (time < 20) {
  greeting = "Good day"
} else {
  greeting = "Good evening"
}
console.log(greeting)
```

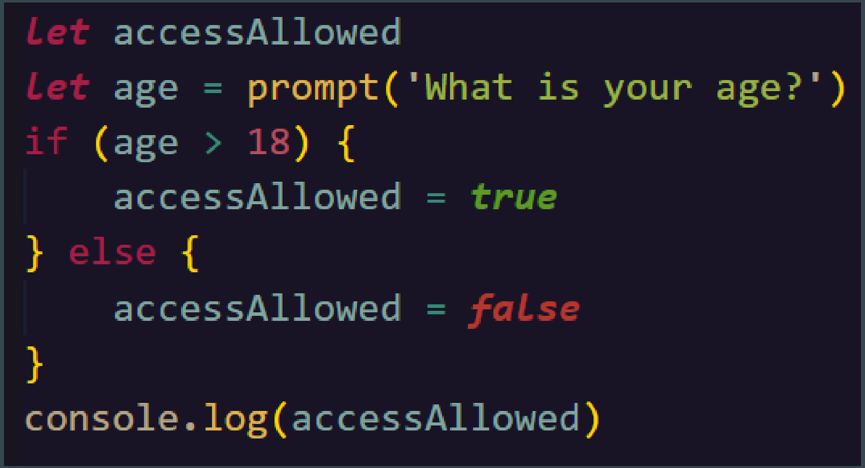
# Conditionals

## 4. ? Ternary Operator

# Conditionals

## 4. Ternary operator ?

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

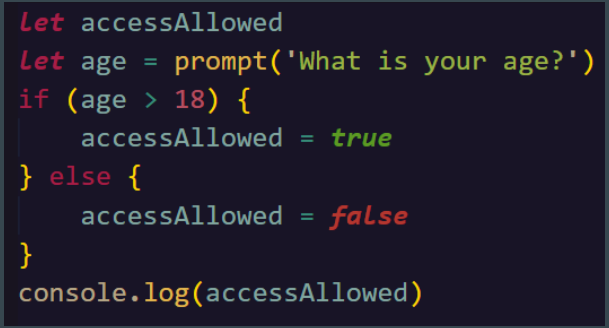


```
let accessAllowed
let age = prompt('What is your age?')
if (age > 18) {
    accessAllowed = true
} else {
    accessAllowed = false
}
console.log(accessAllowed)
```

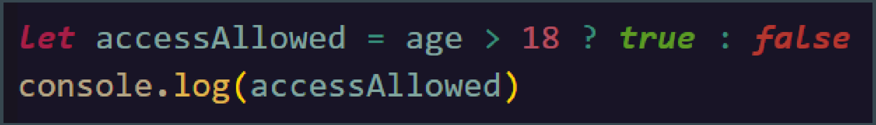
# Conditionals

## 4. Ternary operator ?

- The ternary operator (?) allows us to do this in a shorter and simpler way
- The formal term ternary means that the operator has three operands



```
let accessAllowed
let age = prompt('What is your age?')
if (age > 18) {
    accessAllowed = true
} else {
    accessAllowed = false
}
console.log(accessAllowed)
```



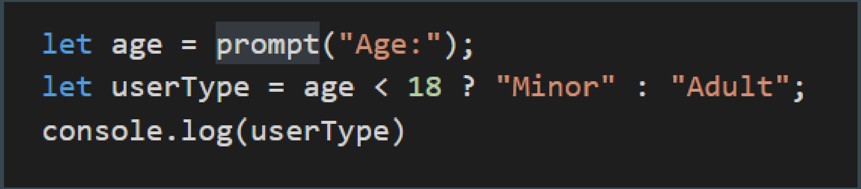
```
let accessAllowed = age > 18 ? true : false
console.log(accessAllowed)
```

- The condition (age>18) is evaluated: if true, the value true is returned and assigned to the variable accessAllowed. If false, the value false is assigned.

# Conditionals

## 4. Ternary operator ?

- The ternary operator (?) allows us to do this in a shorter and simpler way
- The formal term ternary means that the operator has three operands



```
let age = prompt("Age:");
let userType = age < 18 ? "Minor" : "Adult";
console.log(userType)
```

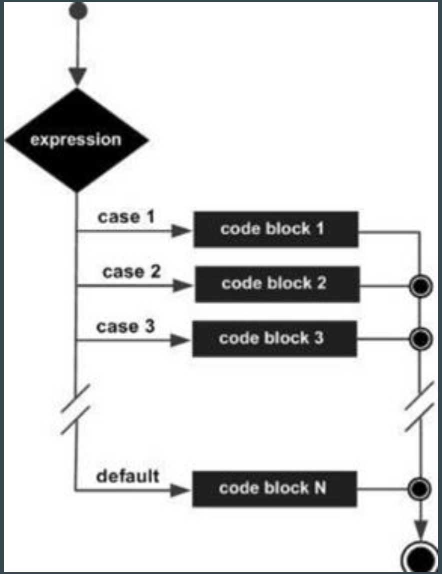
# Conditionals

## 5. SWITCH statement

# Conditionals

## 5. SWITCH statement

- The switch statement can replace multiple if checks
  - More descriptive way to compare a value with multiple variants
  - Switch has one or more case blocks and an optional default action



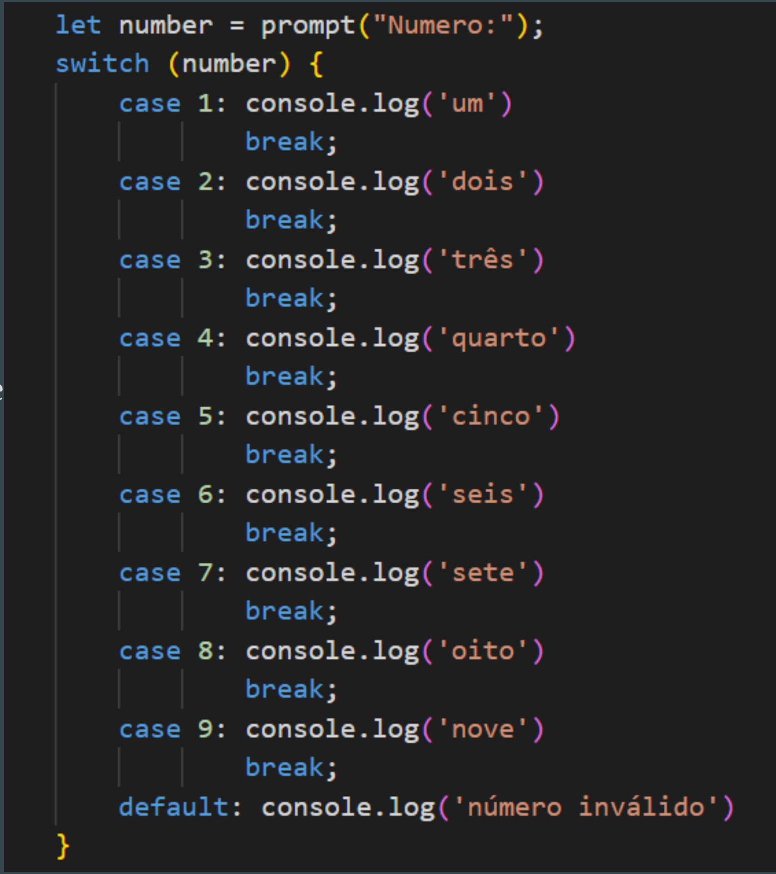
# Conditionals

## 5. SWITCH statement

- How does it work?

    - If equality is found, the switch starts executing the code from the corresponding case, until the nearest break (or until the end of the switch).

    - If no cases are matched, the default block is executed.



```
let number = prompt("Numero:");
switch (number) {
    case 1: console.log('um')
            break;
    case 2: console.log('dois')
            break;
    case 3: console.log('três')
            break;
    case 4: console.log('quarto')
            break;
    case 5: console.log('cinco')
            break;
    case 6: console.log('seis')
            break;
    case 7: console.log('sete')
            break;
    case 8: console.log('oito')
            break;
    case 9: console.log('nove')
            break;
    default: console.log('número inválido')
}
```