# JS & TS Docs

Extensive docs on JS & TS

Kevin Comba

# Table of Contents

# JS

image.png

## Audience

This Tutorial is intended to help a newbie or beginner programmer get started with JavaScript.

## Introduction

It is one of the most famous & commonly used languages. Today Javascript powers millions of web pages and makes them interactive. It helps validate forms, run business logic on the client, create cookies, animate, and more. It runs on all devices, including Windows, Linux, Mac, Android, iOS, etc., and is supported by all browsers.

## What is Javascript

Javascript is the *scripting language*, we use to make web pages interactive. It is written in plain text on the HTML page and runs in the browser. Modern JavaScript evolved so much that, it can run on the server side today. We use it along with HTML & CSS and powers the entire web.

## History

- **Brendan Eich** of Netscape developed JavaScript for Netscape in 1995. It started as **Mocha**, then became **LiveScript**, and later **JavaScript**.
- Java was a very popular language at that time. Hence the marketing folks at Netscape decided to name it JavaScript to encash Java's popularity. They also designed it so that its syntax looks similar to that of Java.
- Microsoft included support for JavaScript (named JScript) from Internet Explorer 3.0. It also had its scripting language VBScript. Other browsers also incorporated JavaScript, while VBScript remained only with Internet Explorer. This eventually led to the demise of VBScript.
- The browser vendors started to come out with their implementation of JavaScript. Since there was no standard, each browser was free to implement. Hence the need for standardizing the syntax & rules of JavaScript.
- In 1997. Netscape presented JavaScript to ECMA International (https://ecma-international.org/), which standardized the specifications of the JavaScript under the name ECMAScript specifications (pronounced as "ek-ma-script"). The browsers now implement Javascript using **ECMAScript specifications**, which is why the JavaScript programs behave similarly across all browsers.

## JavaScript is an interpreted language

- JavaScript is an **interpreted language**. We **cannot compile a JavaScript program into an executable.** We distribute it as plain text.
- To run an interpreted language, the client machines require an Interpreter. The Interpreters read the program line by line and execute each command. The Interpreted languages were once significantly slower than compiled languages.
- Every browser comes with a built-in Interpreter in the form of a **Javascript virtual machine**

## Javascript virtual machine

The JavaScript virtual machine (JVM) is the component of the browser that reads our JavaScript code, optimizes, and executes it.

The job of JVM is to Interpret the JavaScript code and run it. Each browser comes with a version of JVM. The following are some of the popular JVM:

- V8 (https://en.wikipedia.org/wiki/V8_(JavaScript_engine)) — Developed by Google for chrome.
- Rhino (https://en.wikipedia.org/wiki/Rhino_(JavaScript_engine)) — Developed by Mozilla Foundation for Firefox
- SpiderMonkey (https://en.wikipedia.org/wiki/SpiderMonkey_(JavaScript_engine)) — The first JavaScript engine, used by Netscape Navigator, and today powers Firefox
- JavaScriptCore (https://developer.apple.com/documentation/javascriptcore) — Developed by Apple for Safari
- Chakra (https://en.wikipedia.org/wiki/Chakra_(JavaScript_engine)) — Developed by Microsoft and runs on Microsoft Edge
- JerryScript (https://en.wikipedia.org/wiki/JerryScript) — is a lightweight engine for the IoT devices

## Javascript Tools & Frameworks

Many tools, Libraries & Frameworks are available, making working with Javascript easier.

A JavaScript framework (https://en.wikipedia.org/wiki/JavaScript_framework) is an application framework written in JavaScript. It contains tools & libraries and defines the entire application design.

The following are some of the popular JavaScript libraries & frameworks :

1. Angular
2. React
3. jQuery
4. Vue.js
5. Ext.js
6. Ember.js
7. Meteor
8. Mithril
9. Node.js
10. Aurelia
11. Backbone.js

## Javascript Transpilers

A JavaScript Transpiler is a tool that reads source code written in a different programming language and produces an equivalent code in Javascript.

1. Typescript
2. Dart
3. Babel
4. CoffeeScript
5. Coco
6. LiveScript
7. UberScript

# Getting Started

This section shows how to start with JavaScript. We start with building a simple hello world example application. We will learn the basic syntax and rules and learn about identifiers and naming rules. etc. Later we look at what variables are and how to declare variables using let, var, and const.

## JavaScript Code Editors & IDE

We can write Javascript code in any editor. For example, you can use Notepad on Windows and Vim on Linux. But they are basic and provide only some unique features to make writing code faster.

## List of Code editors

When it comes to Javascript code editors, you have two choices: a `full-featured IDE` or `lightweight code editors`.

- Full Pledged IDE
  - Webstorm (https://www.jetbrains.com/webstorm/)
- Light weight editors
  - Visual Studio Code (https://code.visualstudio.com/)
  - Notepad++
  - Vim
- Online Code editors
  - JS Fiddle
  - StackBlitz
  - CodePen
  - JSbin
  - Gitpod

### Autocompletion

Autocompletion is a feature that predicts the rest of the words that the user is typing. It is a must-have feature for a code editor. It reduces the time required to write code and lowers the risk of errors.

### Code organization

We must create different files and folders to organize the code as the application grows. The code editor must help us organize code into files and folders, making it easier to manage and maintain large codebases.

### Debugging tools

Another essential feature to consider when selecting a code editor is debugging tools. A good debugging tool assists developers in identifying and correcting code errors.

## Runtime Environment

JavaScript codes cannot run on their own. They need a runtime environment to run. Runtime is the environment in which a programming language executes.

Javascript can be run in runtime environments.

1. Browser Environment

2. Node Environment

### Browser Environment

#### The <script> Tag

In HTML, JavaScript code is inserted between `<script>` and `</script>` tags.

```html
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript in Body</h2>

<p id="demo"></p>
```

```
<script>
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>

</body>
</html>
```

### Node Environment

In node environment, you have to store your code in a myfirst.js file and run this file with node ie

```
console.log("hello world")
```

and run the code as :

```
C:\Users\Your Name>node myfirst.js
```

## JavaScript Keywords & Reserved Words

`JavaScript Keywords` have a special meaning in the context of a language. They are part of the `syntax of JavaScript`. These are reserved words and we cannot use them as `JavaScript Identifier names`. Using them will result in a compile error.

`arguments` `await` `break` `case` `catch` `class` `const` `continue` `debugger` `default` `delete` `do` `else` `enum` `export` `extends` `false` `finally` `for` `function` `get` `if` `import` `in` `instanceof` `new` `null` `return` `set` `super` `switch` `this` `throw` `true` `try` `typeof` `var` `void` `while` `with` `yield`

### List of Strict Mode Reserved Words

You cannot use the following reserved keywords only if you enable strict mode.

For Example

The following works, although we used let as the variable name.

```
let let=2;
console.log(let)          //2
```

But enabling the **strict mode** will result in an error.

```
"use strict"

let let=2;                    //Unexpected strict mode reserved word
console.log(let)
```

The following is a list of **strict mode-only** reserve words.

`arguments` `eval` `implements` `interface` `let` `package` `private` `protected` `public` `static` `yield`

# JavaScript Variable

JavaScript variables are storage for data, where programs can store value. Basically, It's an area of memory where JavaScript stores the data.

We usually assign a name to the memory. The programs can then use the variable name to identify the memory location. It makes the coding easier.

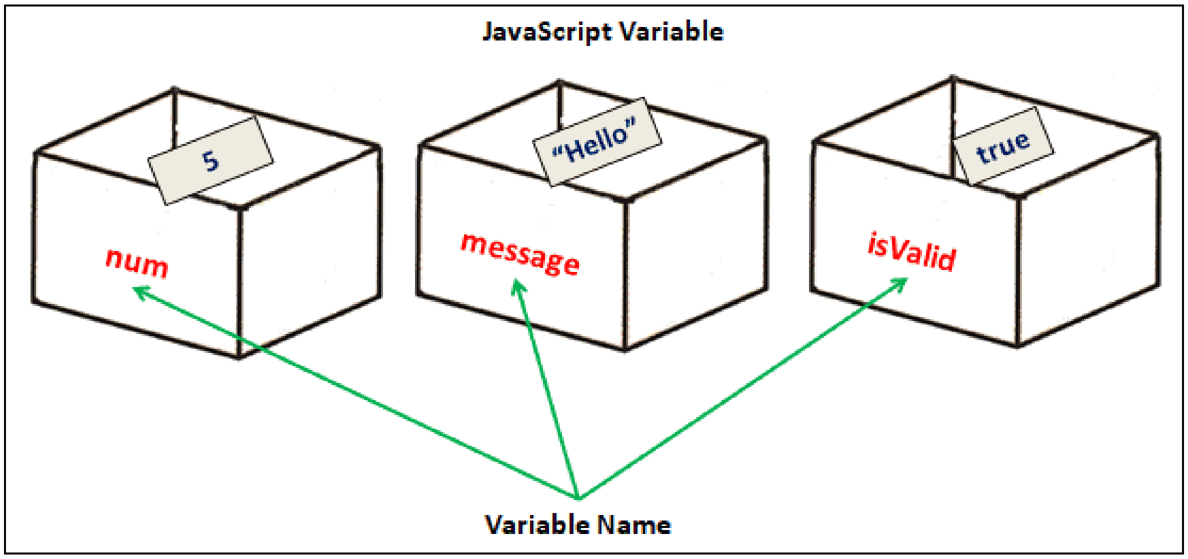


image_2.png

```
let num;
let messsage;
let isValid;

num=5
message="Hello"
isValid=true;
```

In **JavaScript**, a variable can store just about anything like numbers, strings, objects, arrays, etc. There are no restrictions on that. This is different from other languages like C# & Java, where a variable that stores numbers can only store numbers and nothing else.

## Declaring the variable

We need to create the variables before using them. In JavaScript, we call it declaring the variable. There are three keywords using which we can declare a variable.

- var
- let
- const

## Note

- The `var` keyword was used in all JavaScript code from `1995` to `2015`.
- The `let` and const keywords were added to JavaScript in `2015`.
- The `var` keyword should only be used in code written for `older browsers`.

Example using let:

```
let x = 5;
let y = 6;
let z = x + y;
```

Example using const:

```
const x = 5;
const y = 6;
const z = x + y;
```

Example using var:

```
var x = 5;
var y = 6;
var z = x + y;
```

## JavaScript Identifiers

All JavaScript `variables` must be `identified` with `unique names`.

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

1. Names can contain letters, digits, underscores, and dollar signs.
2. Names must begin with a letter.
3. Names can also begin with $ and _ (but we will not use it in this tutorial).
4. Names are case sensitive (y and Y are different variables).
5. Reserved words (like JavaScript keywords) cannot be used as names.

## Var vs Let

The main difference between var and let in JavaScript lies in their scope and behavior:

- Scope:
  - **var**: Has function scope, meaning it is accessible throughout the entire function where it is declared (even if declared inside a block like an if statement).
  - **let**: Has block scope, meaning it is only accessible within the block (like inside {}) where it is declared.

```
function testVarLet() {
    if (true) {
        var varVar = "I am a var";
        let letVar = "I am a let";
    }

    console.log(varVar); // Works fine, 'var' has function scope
    console.log(letVar); // Error: letVar is not defined, because 'let' 
has block scope
}
```

```
testVarLet();
```

- Hoisting:
    - **var**: Variables declared with var are hoisted to the top of their scope, but their initialization is not. This can lead to unexpected behavior if accessed before being assigned.
    - **let**: Variables declared with let are also hoisted, but they remain in a "temporal dead zone" until the declaration is encountered, which prevents accidental access before the declaration.

```
function hoistingExample() {
    console.log(varVar); // Undefined due to hoisting
    // console.log(letVar); // Error: Cannot access 'letVar' before
initialization

    var varVar = "I am a var";
    let letVar = "I am a let";
}

hoistingExample();
```

- Re-declaration:
    - **var**: You can re-declare the same variable multiple times within the same scope without any errors.
    - **let**: You cannot re-declare a variable in the same scope.

```
function redeclarationExample() {
    var x = 10;
    var x = 20; // No error, var allows re-declaration

    let y = 10;
    // let y = 20; // Error: Identifier 'y' has already been declared
}

redeclarationExample();
```

# Data Types

JavaScript has 8 Datatypes

1. `String`
2. `Number`
3. `Bigint`
4. `Boolean`
5. `Undefined`
6. `Null`
7. `Symbol`
8. `Object`

**The Object Datatype**: The object data type can contain both `built-in objects`, and `user defined objects`:

Built-in object types can be:

`objects`, `arrays`, `dates`, `maps`, `sets`, `intarrays`, `floatarrays`, `promises`, and more

```
// Numbers:
let length = 16;
let weight = 7.5;

// Strings:
let color = "Yellow";
let lastName = "Johnson";

// Booleans
let x = true;
let y = false;

// Object:
```

```javascript
const person = {firstName:"John", lastName:"Doe"};

// Array object:
const cars = ["Saab", "Volvo", "BMW"];

// Date object:
const date = new Date("2022-03-25");
```

# JavaScript Operators

Javascript operators are used to perform different types of mathematical and logical computations. Examples:

- The Assignment Operator `=` assigns values
- The Addition Operator `+` adds values
- The Multiplication Operator `*` multiplies values
- The Comparison Operator `>` compares values

## JavaScript Assignment

The **Assignment** Operator (=) assigns a value to a variable:

```
// Assign the value 5 to x
let x = 5;
// Assign the value 2 to y
let y = 2;
// Assign the value x + y to z:
let z = x + y;
```

## JavaScript Addition

The **Addition** Operator `(+)` adds numbers:

```
let x = 5;
let y = 2;
let z = x + y;
```

## JavaScript Multiplication

The Multiplication Operator `(*)` multiplies numbers:

```
let x = 5;
let y = 2;
let z = x * y;
```

## Types of JavaScript Operators

There are different types of JavaScript operators:

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. String Operators
5. Logical Operators
6. Bitwise Operators
7. Ternary Operators
8. Type Operators

### JavaScript Arithmetic Operators

Arithmetic Operators are used to perform arithmetic on numbers:

```
// Define two numbers
```

Explanation:

- Addition (+): Adds two numbers.
- Subtraction (-): Subtracts the second number from the first.
- Multiplication (*): Multiplies two numbers.
- Exponentiation (**): Raises the first number to the power of the second.
- Division (/): Divides the first number by the second.

- Modulus (%): Returns the remainder of the division.
- Increment (++): Increases a variable's value by 1.
- Decrement (--): Decreases a variable's value by 1.

| Operator | Description |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| ** | Exponentiation (ES2016) |
| / | Division |
| % | Modulus (Division Remainder) |
| ++ | Increment |
| -- | Decrement |

## JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

The Addition Assignment Operator (+=) adds a value to a variable

| Operator | Example | Same As |
|---|---|---|
| = | x = y | x = y |
| += | x += y | x = x + y |
| -= | x -= y | x = x - y |
| *= | x *= y | x = x * y |
| /= | x /= y | x = x / y |
| %= | x %= y | x = x % y |
| **= | x **= y | x = x ** y |

```
// Initial values
let x = 10;
let y = 5;

// Simple assignment
x = y;  // Assign the value of y to x
console.log(`x = y: ${x}`);  // Output: x = y: 5

// Add and assign
x += y;  // x = x + y
console.log(`x += y: ${x}`);  // Output: x += y: 10

// Subtract and assign
x -= y;  // x = x - y
console.log(`x -= y: ${x}`);  // Output: x -= y: 5

// Multiply and assign
x *= y;  // x = x * y
```

```
console.log(`x *= y: ${x}`);  // Output: x *= y: 25

// Divide and assign
x /= y;  // x = x / y
console.log(`x /= y: ${x}`);  // Output: x /= y: 5

// Modulus and assign
x %= y;  // x = x % y
console.log(`x %= y: ${x}`);  // Output: x %= y: 0

// Exponentiation and assign (ES2016)
x **= y;  // x = x ** y
console.log(`x **= y: ${x}`);  // Output: x **= y: 0
```

explanation:

- `=`: Basic assignment of the value from the right-hand side to the left-hand side.
- `+=`: Adds the right operand to the left operand and assigns the result to the left operand.
- `-=`: Subtracts the right operand from the left operand and assigns the result to the left operand.
- `*=`: Multiplies the left operand by the right operand and assigns the result to the left operand.
- `/=`: Divides the left operand by the right operand and assigns the result to the left operand.
- `%=`: Assigns the remainder when the left operand is divided by the right operand.
- `**=`: Performs exponentiation (raises the left operand to the power of the right operand) and assigns the result to the left operand.

### JavaScript Comparison Operators

| Operator | Description |
|---|---|
| `==` | equal to |
| `===` | equal value and equal type |
| `!=` | not equal |
| `!==` | not equal value or not equal type |
| `>` | greater than |
| `<` | less than |
| `>=` | greater than or equal to |
| `<=` | less than or equal to |
| `?` | ternary operator |

```
// Variables for comparison
let a = 10;
let b = 5;

// Using ternary operator to check equality (==)
let equalityCheck = (a == b) ? "a is equal to b" : "a is not equal to
b";
console.log(equalityCheck);  // Output: a is not equal to b

// Using ternary operator to check strict equality (===)
let strictEqualityCheck = (a === b) ? "a and b are of equal value and
type" : "a and b are not equal in value or type";
console.log(strictEqualityCheck);  // Output: a and b are not equal in
value or type
```

```
// Using ternary operator to check not equal (!=)
let notEqualCheck = (a != b) ? "a is not equal to b" : "a is equal to
b";
console.log(notEqualCheck);  // Output: a is not equal to b

// Using ternary operator to check strict not equal (!==)
let strictNotEqualCheck = (a !== b) ? "a and b are not equal in value or
type" : "a and b are equal in value and type";
console.log(strictNotEqualCheck);  // Output: a and b are not equal in
value or type

// Using ternary operator to check greater than (>)
let greaterThanCheck = (a > b) ? "a is greater than b" : "a is not
greater than b";
console.log(greaterThanCheck);  // Output: a is greater than b

// Using ternary operator to check less than (<)
let lessThanCheck = (a < b) ? "a is less than b" : "a is not less than
b";
console.log(lessThanCheck);  // Output: a is not less than b

// Using ternary operator to check greater than or equal to (>=)
let greaterThanOrEqualCheck = (a >= b) ? "a is greater than or equal to
b" : "a is less than b";
console.log(greaterThanOrEqualCheck);  // Output: a is greater than or
equal to b

// Using ternary operator to check less than or equal to (<=)
let lessThanOrEqualCheck = (a <= b) ? "a is less than or equal to b" :
"a is greater than b";
console.log(lessThanOrEqualCheck);  // Output: a is greater than b
```

Explanation:

- `Equality (==)`: Checks if the values of `a` and `b` are the same.

- `Strict equality (===)`: Checks if the values and types of `a` and `b` are the same.
- `Not equal (!=)`: Checks if the values of `a` and `b` are different.
- `Strict not equal (!==)`: Checks if either the values or types of `a` and `b` are different.
- `Greater than (>)`: Checks if `a` is greater than `b`.
- `Less than (<)`: Checks if `a` is less than `b`.
- `Greater than` or `equal to (>=)`: Checks if `a` is greater than or equal to `b`.
- `Less than` or `equal to (<=)`: Checks if `a` is less than or equal to `b`.

## JavaScript Logical Operators

| Operator | Description |
|---|---|
| `&&` | logical and |
| `\|\|` | logical or |
| `!` | logical not |

## JavaScript Type Operators

| Operator | Description |
|---|---|
| `typeof` | Returns the type of a variable |
| `instanceof` | Returns true if an object is an instance of an object type |

## JavaScript Bitwise Operators

Bit operators work on 32 bits numbers.

Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

| Operator | Description | Example | Same as | Result | Decimal |
|---|---|---|---|---|---|
| `&` | AND | `5 & 1` | `0101 & 0001` | 0001 | 1 |
| `\|` | OR | `5 \| 1` | `0101 \| 0001` | 0101 | 5 |
| `~` | NOT | `~ 5` | `~0101` | 1010 | 10 |
| `^` | XOR | `5 ^ 1` | `0101 ^ 0001` | 0100 | 4 |
| `<<` | left shift | `5 << 1` | `0101 << 1` | 1010 | 10 |
| `>>` | right shift | `5 >> 1` | `0101 >> 1` | 0010 | 2 |
| `>>>` | unsigned right shift | `5 >>> 1` | `0101 >>> 1` | 0010 | 2 |

# Control Flow Statements

## JavaScript if, else, and else if

Conditional statements are used to perform different actions based on different conditions.

## Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

### The if Statement

Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

syntax:

```
if (condition) {
  //  block of code to be executed if the condition is true
}
```

example:

```
if (hour < 18) {
  greeting = "Good day";
}
```

### The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is false.

syntax:

```
if (condition) {
  //  block of code to be executed if the condition is true
} else {
  //  block of code to be executed if the condition is false
}
```

example:

```
if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

### The else if Statement

Use the `else if` statement to specify a new condition if the first condition is false.

Syntax:

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and
condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and
condition2 is false
}
```

Example:

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

## JavaScript Switch Statement

The `switch` statement is used to perform different actions based on different conditions.

Syntax:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

The `getDay()` method returns the weekday as a number between `0` and `6`.

(Sunday=0, Monday=1, Tuesday=2 ..)

This example uses the weekday number to calculate the weekday name:

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

## The break Keyword

When JavaScript reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution inside the switch block.

It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

⚠ ***Note:*** *If you omit the break statement, the next case will be executed even if the evaluation does not match the case*

## The default Keyword

The `default` keyword specifies the code to run if there is no case match:

The getDay() method returns the weekday as a number between 0 and 6.

If today is neither Saturday (6) nor Sunday (0), write a default message:

```
switch (new Date().getDay()) {
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

The result of text will be: Today is Sunday

## JavaScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value. JavaScript supports different kinds of loops:

- for - loops through a block of code a number of times
- for/in - loops through the properties of an object
- for/of - loops through the values of an iterable object
- while - loops through a block of code while a specified condition is true
- do/while - also loops through a block of code while a specified condition is true

### The For Loop

The for statement creates a loop with 3 optional expressions: syntax:

```
for (expression 1; expression 2; expression 3) {
  // code block to be executed
}
```

- Expression 1 is executed (one time) before the execution of the code block.
- Expression 2 defines the condition for executing the code block.
- Expression 3 is executed (every time) after the code block has been executed.

```
for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

Right-Angled Triangle (ascending order):

```
// Define the height of the triangle
let height = 5;

// Using a for loop to print the triangle
for (let row = 1; row <= height; row++) {
    let stars = ''; // Empty string to build each row

    // Add '*' to the string for each column in the current row
    for (let column = 1; column <= row; column++) {
        stars += '*';  // Append '*' to the stars string
    }

    // Print the row
    console.log(stars);
}
```

Output for height = 5:

```
*
**
***
****
*****
```

Reversed Right-Angled Triangle (descending order) :

```
// Define the height of the triangle
let height = 5;

// Using a for loop to print the reversed triangle
for (let row = height; row >= 1; row--) {
    let stars = ''; // Empty string to build each row

    // Add '*' to the string for each column in the current row
    for (let column = 1; column <= row; column++) {
        stars += '*';  // Append '*' to the stars string
    }

    // Print the row
    console.log(stars);
}
```

Output for `height = 5`:

```
*****
****
***
**
*
```

**Explanation:**

- In both examples, we use two `for` loops:
    - The outer loop controls the number of rows.
    - The inner loop controls the number of `*` printed in each row.

For the right-angled triangle:

- The number of `*` increases as the rows go down (from 1 to `height`).

For the reversed right-angled triangle:

- The number of `*` decreases as the rows go down (from `height` to 1).

You can change the `height` variable to create larger or smaller triangles!

### The For In Loop

The JavaScript `for in` statement loops through the properties of `an Object`:

syntax:

```
for (key in object) {
  // code block to be executed
}
```

example:

```
const person = {fname:"John", lname:"Doe", age:25};

let text = "";
for (let x in person) {
  text += person[x];
}
```

- The `for in` loop iterates over a `person` object
- Each iteration returns a `key (x)`
- The key is used to access the `value` of the key
- The value of the key is `person[x]`

#### For In Over Arrays

The JavaScript `for in` statement can also loop over the properties of an Array:

Syntax:

```
for (variable in array) {
  code
}
```

Example:

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
    console.log(numbers[x])
    txt += numbers[x];
}

console.log(txt)
```

## The For Of Loop

The JavaScript for of statement loops through the values of an iterable object.

It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more:

Syntax:

```
for (variable of iterable) {
  // code block to be executed
}
```

### Looping over an Array

```
const cars = ["BMW", "Volvo", "Mini"];

let text = "";
for (let x of cars) {
    console.log(x)
  text += x;
}
```

### Looping over a String

```
let language = "JavaScript";
let text = "";
for (let x of language) {
    console.log(x)
    text += x;
}
```

### forEach

The forEach method in JavaScript is used to execute a provided function once for each element in an array (or array-like object). It is commonly used when you need to iterate through an array and perform an operation on each element.

Syntax:

```
array.forEach(function(element, index, array) {
  // Code to be executed for each element
});
```

- element: The current element in the array.
- index (optional): The index of the current element.
- array (optional): The array that forEach is being applied to.
- Iterating through an array of numbers

```
let numbers = [1, 2, 3, 4, 5];

numbers.forEach(function(number) {
console.log(number * 2); // Multiply each number by 2
});
```

Output:

```
2
4
6
```

```
8
10
```

- Using forEach with an index

```
let fruits = ["apple", "banana", "cherry"];

fruits.forEach(function(fruit, index) {
    console.log(`${index + 1}: ${fruit}`);
});
```

Output:

```
1: apple
2: banana
3: cherry
```

- Using forEach with arrow functions

```
let fruits = ["apple", "banana", "cherry"];

fruits.forEach((fruit, index) => {
    console.log(`${index + 1}: ${fruit}`);
});
```

Output:

```
1: apple
2: banana
3: cherry
```

- Iterating over an array of objects

```
let people = [
    { name: "Alice", age: 30 },
    { name: "Bob", age: 25 },
    { name: "Charlie", age: 35 }
];

people.forEach(person => {
    console.log(`${person.name} is ${person.age} years old.`);
});
```

Output:

```
Alice is 30 years old.
Bob is 25 years old.
Charlie is 35 years old.
```

## While loop

The `while` loop loops through a block of code as long as a specified condition is true.

Syntax:

```
while (condition) {
  // code block to be executed
}
```

To create a right-angled triangle using a while loop in JavaScript, you can build the pattern by printing asterisks (*) row by row, with each row having one more * than the previous one.

Here's an example of how to do it:

```
// Define the height of the triangle
let height = 5;

// Initialize the starting row
let row = 1;
```

```
while (row <= height) {
    let stars = ''; // Empty string to build each row

    // Add '*' to the string for each column in the current row
    let column = 1;
    while (column <= row) {
        stars += '*';  // Append '*' to the stars string
        column++;
    }

    // Print the row
    console.log(stars);

    // Move to the next row
    row++;
}
```

Explanation:

- The outer while loop controls the number of rows (height).
- The inner while loop controls the number of stars in each row. For row 1, it prints 1 star, for row 2, it prints 2 stars, and so on.
- The stars variable is built by concatenating * for each iteration in the inner loop.
- After constructing each row, it is printed to the console using console.log().

For height = 5, the output will look like this:

```
*
**
***
****
*****
```

To create a right-angled triangle in reverse (starting with the largest row of stars and decreasing by one on each line) using a while loop in JavaScript, you can do the following:

```
// Define the height of the triangle
let height = 5;

// Initialize the starting row (same as height)
let row = height;

while (row > 0) {
    let stars = ''; // Empty string to build each row

    // Add '*' to the string for each column in the current row
    let column = 1;
    while (column <= row) {
        stars += '*';  // Append '*' to the stars string
        column++;
    }

    // Print the row
    console.log(stars);

    // Move to the next row (decreasing the row count)
    row--;
}
```

## The Do While Loop

The `do while` loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax:

```
do {
  // code block to be executed
}
while (condition);
```

Example: The example below uses a do while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

# Arrays

An array is a special variable, which can hold more than one value:

```
const cars = ["Saab", "Volvo", "BMW"];
```

## Why Use Arrays?

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
let car1 = "Saab";
let car2 = "Volvo";
let car3 = "BMW";
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

## Creating an Array

Syntax:

```
const array_name = [item1, item2, ...];
```

⚠ *It is a common practice to declare arrays with the const keyword.*

```
const cars = ["Saab", "Volvo", "BMW"];
```

or

```
const cars = [
  "Saab",
  "Volvo",
  "BMW"
];
```

### Using the JavaScript Keyword new

```
const cars = new Array("Saab", "Volvo", "BMW");
```

⚠ The two examples above do exactly the same. There is no need to use `new Array()`. For simplicity, readability and execution speed, use the array literal method.

## Accessing First Array Elements

You access an array element by referring to the `index number`:

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
```

⚠ Note: Array indexes **start with 0.** `[0]` is the `first element`. `[1]` is the `second element`.

## Accessing the Last Array Element

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits[fruits.length - 1];
```

## Changing an Array Element

This statement changes the value of the first element in cars:

```
cars[0] = "Saab"
```

## Looping Array Elements

With JavaScript, the full array can be accessed by referring to the array name:

- **Array.forEach()**

```
cars.forEach((item, index)=> {
    console.log(`the index: ${index} and the value : ${item}`)
})
```

- **for**

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fLen = fruits.length;

let text = "<ul>";
for (let i = 0; i < fLen; i++) {
  text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
```

## Arrays are Objects

⚠

- Arrays are a special type of objects. The typeof operator in JavaScript returns "object" for arrays.
- But, JavaScript arrays are best described as arrays.
- Arrays use numbers to access its "elements". In this example, person[0] returns John:

```
const cars = ["Saab", "Volvo", "BMW"];

console.log(typeof cars) // "object"
```

## Array Methods

- `push()`: add an element at the end of an Array

```
let a = [10, 20, 30, 40, 50];
a.push(60);
a.push(70, 80, 90);
console.log(a);
```

Output :

```
[ 10, 20, 30, 40, 50,  60, 70, 80, 90 ]
```

- `unshift()`: is used to add elements to the front of an Array.

```
let a = [20, 30, 40];
a.unshift(10, 20);
console.log(a);
```

Output

```
[ 10, 20, 20, 30, 40 ]
```

- `pop()`: is used to remove elements from the end of an array.

```
let a = [20, 30, 40, 50];
a.pop();
console.log(a);
```

- shift(): is used to remove elements from the beginning of an array

```
let a = [20, 30, 40, 50];
a.shift();
console.log(a);
```

Output:

```
[ 30, 40, 50 ]
```

- The splice(): is used to Insert and Remove elements in between the Array.

```
let a = [20, 30, 40, 50];
a.splice(1, 3);
a.splice(1, 0, 3, 4, 5);
console.log(a);
```

Output:

```
[ 20, 3, 4, 5 ]
```

- The first a.splice(1, 3) removes 3 elements (30, 40, 50) starting at index 1. The array becomes [20].
- The second a.splice(1, 0, 3, 4, 5) inserts 3, 4, and 5 at index 1 without removing anything. The array becomes [20, 3, 4, 5].

- The slice(): returns a new array containing a portion of the original array, based on the start and end index provided as arguments

```
const a = [1, 2, 3, 4, 5];
const res = a.slice(1, 4);
console.log(res);
console.log(a)
```

output:

```
[ 2, 3, 4 ]
[ 1, 2, 3, 4, 5 ]
```

- The slice() method creates a new array by extracting elements from index 1 to 3 (exclusive of 4) from the original array.
- The original array remains unchanged, and the result is [2, 3, 4].

- map(): creates an array by calling a specific function on each element present in the parent array. It is a non-mutating method.

```
let a = [4, 9, 16, 25];
let sub = a.map(geeks);

function geeks() {
    return a.map(Math.sqrt);
}
console.log(sub);
```

Output:

```
[ [ 2, 3, 4, 5 ], [ 2, 3, 4, 5 ], [ 2, 3, 4, 5 ], [ 2, 3, 4, 5 ] ]
```

- The code defines a geeks function, but instead of operating on the individual array 'a' elements, it applies Math.sqrt() to the entire a array, resulting in a nested array of square roots.
- The output will be an array of the same length as a, but each element will be the result of applying arr.map(Math.sqrt).

- The reduce() method is used to reduce the array to a single value and executes a provided function for each value of the array (from left to right) and the return value of the function is stored in an accumulator.

```
let a = [88, 50, 25, 10];
let sub = a.reduce(geeks);

function geeks(tot, num) {
    return tot - num;
}
console.log(sub);
```

Output: 3

- The reduce() method iterates over the array ‘a’ and applies the geeks function, which subtracts each element (num) from the running total (tot).
- For the array [88, 50, 25, 10], the calculation proceeds as: 88 – 50 – 25 – 10.
- The reverse() method is used to reverse the order of elements in an array. It modifies the array in place and returns a reference to the same array with the reversed order.

```
let a = [1, 2, 3, 4, 5];
console.log(a);
a.reverse();
console.log(a);
```

Output:

```
[1,2,3,4,5]
[ 5, 4, 3, 2, 1 ]
```

# Functions

JavaScript functions are defined with the function keyword.

```
function functionName(parameters) {
  // code to be executed
}
```

example:

```
function myFunction(a, b) {
  return a * b;
}
```

## Function Hoisting

- Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope.
- Hoisting applies to variable declarations and to function declarations.
- Because of this, JavaScript functions can be called before they are declared:

```
myFunction(5);

function myFunction(y) {
  return y * y;
}
```

## Function Parameters and Arguments

- Function parameters are the names listed in the function definition.
- Function arguments are the real values passed to (and received by) the function. Syntax:

```
function functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

## Default Parameter Values

```
function myFunction(x, y = 10) {
  return x + y;
}
myFunction(5);
```

## Types Function

### Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the `function` keyword, the `return` keyword, and the `curly brackets`.

```
// ES5
var x = function(x, y) {
  return x * y;
}

// ES6
const x = (x, y) => x * y;
```

### Immediately-invoked Function Expressions (IIFE)

Function expressions can be made "self-invoking".

A self-invoking expression is invoked (started) automatically, without being called.

Syntax:

```
(function() {
    // statements
})()
```

Example:

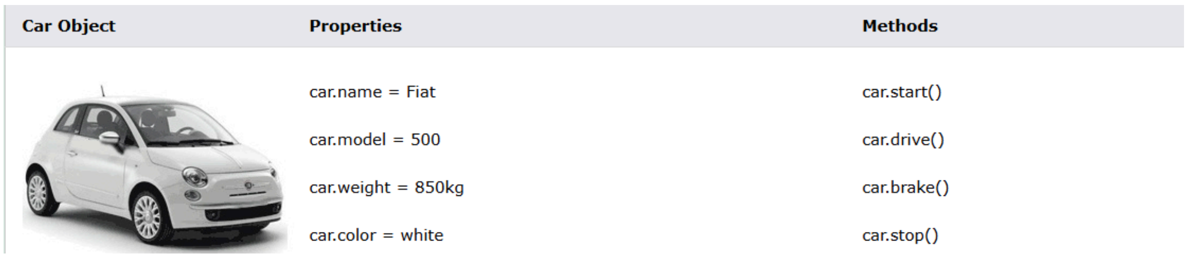
```
(function () {
  let x = "Hello!!";  // I will invoke myself
})();
```

# Objects

## Real Life Objects

In real life, objects are things like: houses, cars, people, animals, or any other subjects.

Here is a car object example:



| Car Object | Properties | Methods |
|---|---|---|
| | car.name = Fiat | car.start() |
| | car.model = 500 | car.drive() |
| | car.weight = 850kg | car.brake() |
| | car.color = white | car.stop() |

image_3.png

### Object Properties

A real life car has properties like weight and color:

car.name = Fiat, car.model = 500, car.weight = 850kg, car.color = white.

Car objects have the same properties, but the values differ from car to car.

### Object Methods

A real life car has methods like start and stop:

car.start(), car.drive(), car.brake(), car.stop().

Car objects have the same methods, but the methods are performed at different times.

## Creating a JavaScript Object

```
// Create an Object
const person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
```

Spaces and line breaks are not important. An object initializer can span multiple lines:

```
// Create an Object
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

This example creates an empty JavaScript object, and then adds 4 properties:

```
// Create an Object
const person = {};

// Add Properties
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

### Using the new Keyword

```
// Create an Object
const person = new Object();

// Add Properties
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

## Object Properties

The named values, in JavaScript objects, are called properties.

| Property | Value |
|---|---|
| firstName | John |
| lastName | Doe |
| age | 50 |
| eyeColor | blue |

Objects written as name value pairs are similar to:

- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

## Accessing Object Properties

You can access object properties in two ways:

```
objectName.propertyName
```

```
objectName["propertyName"]
```

## JavaScript Object Methods

Methods are actions that can be performed on objects.

Methods are function definitions stored as property values.

```
const person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

## In JavaScript, Objects are King.

If you Understand Objects, you Understand JavaScript. In JavaScript, almost "everything" is an object.

- Objects are objects
- Maths are objects
- Functions are objects
- Dates are objects
- Arrays are objects
- Maps are objects
- Sets are objects

## JavaScript Primitives

A `primitive value` is a value that has no properties or methods.

`3.14` is a primitive value

A `primitive data type` is data that has a primitive value.

JavaScript defines 7 types of primitive data types:

- string

- number
- boolean
- null
- undefined
- symbol
- bigint

⚠ Immutable : Primitive values are immutable (they are hardcoded and cannot be changed). if x = 3.14, you can change the value of x, but you cannot change the value of 3.14.

## JavaScript Objects are Mutable

Objects are mutable: They are addressed by reference, not by value.

If person is an object, the following statement will not create a copy of person:

- The object x is not a copy of person. The object x is person.
- The object x and the object person share the same memory address.
- Any changes to x will also change person:

```
//Create an Object
const person = {
  firstName:"John",
  lastName:"Doe",
  age:50, eyeColor:"blue"
}

// Try to create a copy
const x = person;
```

```
// This will change age in person:
x.age = 10;
```

# JSON

- JSON stands for JavaScript Object Notation
- JSON is a text format for storing and transporting data
- JSON is "self-describing" and easy to understand

## JSON Example

This example is a JSON string:

```
{"name" : "John", "age" : 30, "car" : null}
```

## JSON Data Types

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- null

⚠ JSON values cannot be one of the following data types:

- a function
- a date

- undefined

- Strings in JSON must be written in double quotes.

```
{"name" : "John"}
```

- Numbers in JSON must be an integer or a floating point.

```
{"age" : 30}
```

- Values in JSON can be objects.

```
{
    "employee" : {
      "name" : "John",
      "age" : 30,
      "city" : "New York"
    }
}
```

- Values in JSON can be arrays.

```
{
"employees" : ["John", "Anna", "Peter"]
}
```

- Values in JSON can be Boolean(true/false).

```
{"sale" : true}
```

- Values in JSON can be null.

```
{"middlename" : null}
```

## JSON Methods

### JSON.parse()

A common use of JSON is to exchange data to/from a web server. When receiving data from a web server, the data is always a string. Parse the data with `JSON.parse()`, and the data becomes a JavaScript object.

**Example**

Imagine we received this text from a web server:

```
'{"name":"John", "age":30, "city":"New York"}'
```

Use the JavaScript function `JSON.parse()` to convert text into a JavaScript object:

```
const obj = JSON.parse('{"name":"John", "age":30, "city":"New York"}');
```

### JSON.stringify()

A common use of JSON is to exchange data to/from a web server. When sending data to a web server, the data has to be a string. You can convert any JavaScript datatype into a string with `JSON.stringify()`.

Imagine we have this object in JavaScript:

```
const obj = {name: "John", age: 30, city: "New York"};
```

Use the JavaScript function `JSON.stringify()` to convert it into a string.

```
const myJSON = JSON.stringify(obj);
```

⚠ The result will be a string following the JSON notation. myJSON is now a string, and ready to be sent to a server:

# Asynchronous Js

In JavaScript, **asynchronous programming** allows you to perform tasks that take time (like fetching data from a server, reading files, or waiting for user input) without blocking the main thread. This means that while one task is being completed, JavaScript can continue running other code.

JavaScript provides several ways to work with asynchronous operations, including **callbacks**, **Promises**, and **async/await.**

## 1. Callbacks

A **callback** is a function passed into another function as an argument and is executed after a task completes. Callbacks were the original way to handle asynchronous operations in JavaScript.

### Example: Using Callbacks

```
function fetchData(callback) {
    setTimeout(() => {
        const data = "Fetched Data";
        callback(data);  // Callback function is called after 2 seconds
    }, 2000);
}

fetchData(function(result) {
    console.log(result);  // This will log "Fetched Data" after 2
seconds
});
```

### Explanation:

- The `fetchData` function simulates an asynchronous operation with `setTimeout`.
- After 2 seconds, the `callback` function is called with the data `"Fetched Data"`.
- This pattern is common for handling asynchronous tasks like reading files, making HTTP requests, etc.

However, callbacks can lead to **callback hell** when you have multiple nested callbacks, making the code harder to read and maintain.

## 2. Promises

A **Promise** is a more modern way to handle asynchronous operations. A promise represents the eventual completion (or failure) of an asynchronous operation. It allows you to attach `.then()` and `.catch()` methods to handle success or failure, respectively.

### Example: Using Promises

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const data = "Fetched Data";
            resolve(data);  // Resolve the promise with data
        }, 2000);
    });
}

fetchData().then(result => {
    console.log(result);  // This will log "Fetched Data" after 2
seconds
}).catch(error => {
    console.error(error);  // This will handle any errors
});
```

### Explanation:

- The `fetchData` function returns a promise that resolves after 2 seconds with the data `"Fetched Data"`.
- The `.then()` method is used to handle the result when the promise is resolved successfully.
- The `.catch()` method is used to handle any potential errors (though there are none in this case).

## 3. Async/Await

async/await is a more readable and cleaner way to work with asynchronous code. It is built on top of Promises, allowing you to write asynchronous code in a more synchronous-looking style.

- **async**: Used to declare an asynchronous function.
- **await**: Used inside an async function to pause execution until the Promise resolves.

### Example: Using Async/Await

```
async function fetchData() {
    const data = await new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Fetched Data");
        }, 2000);
    });
    console.log(data);  // This will log "Fetched Data" after 2 seconds
}

fetchData();
```

### Explanation:

- The fetchData function is marked as async, which allows us to use await inside it.
- await pauses the function execution until the promise resolves with the result "Fetched Data".
- This approach is more readable than chaining .then() and .catch(), especially when dealing with multiple asynchronous operations.

## Handling Errors with Async/Await

You can use try/catch blocks to handle errors with async/await.

```
async function fetchData() {
    try {
        const data = await new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve("Fetched Data");
            }, 2000);
        });
        console.log(data);  // This will log "Fetched Data" after 2
seconds
    } catch (error) {
        console.error("Error:", error);
    }
}

fetchData();
```

**Explanation:**

- In case of an error (like if the promise is rejected), the `catch` block will catch the error and log it.

## Example: Chaining Multiple Async Operations with Async/Await

If you have multiple asynchronous tasks that depend on each other, you can chain them together with `async/await`.

```
async function fetchData() {
    const data1 = await new Promise((resolve, reject) => {
        setTimeout(() => resolve("Data 1"), 1000);
    });

    const data2 = await new Promise((resolve, reject) => {
        setTimeout(() => resolve("Data 2"), 1000);
    });
    console.log(data1, data2);  // This will log "Data 1 Data 2" after 2
seconds
}

fetchData();
```

### Explanation:

- The await keyword ensures that data2 will not be fetched until data1 is successfully retrieved.
- The asynchronous operations are handled sequentially, making the code easier to read compared to using .then() with promises.

# Scope, Scope Chain, Lexical Scope & Closure

In JavaScript, understanding **scope**, **scope chain**, **lexical scope**, and **closure** is fundamental to writing efficient and bug-free code. Let's break down these concepts one by one, with detailed explanations and examples.

## 1. Scope

**Scope** refers to the context or environment in which variables and functions are accessible. It defines the visibility and lifetime of variables and functions in different parts of your code.

There are primarily two types of scopes in JavaScript:

- **Global Scope**: Variables declared outside any function are in the global scope. They can be accessed from anywhere in the code.
- **Local Scope**: Variables declared inside a function are in the local scope of that function. They can only be accessed within that function.

### Example:

```
let globalVar = "I am global"; // Global scope

function exampleFunction() {
    let localVar = "I am local"; // Local scope
    console.log(globalVar); // Can access global variable
    console.log(localVar); // Can access local variable
}

exampleFunction();

console.log(globalVar); // Can access global variable
console.log(localVar); // Error! localVar is not defined
```

**Explanation:**

- `globalVar` is defined in the **global scope** and can be accessed both inside and outside the function.
- `localVar` is defined inside the function `exampleFunction()` and can only be accessed within that function. Attempting to access `localVar` outside the function results in an error.

## 2. Scope Chain

The **scope chain** is the order in which the JavaScript engine looks for a variable. When you try to access a variable, JavaScript looks for it in the **local scope** first, then moves outward to the **parent scope** (if any), and finally looks in the **global scope.** The scope chain helps JavaScript determine the value of a variable.

**Example:**

```
let globalVar = "I am global"; // Global scope

function outerFunction() {
    let outerVar = "I am in outer function"; // Local scope of
outerFunction

    function innerFunction() {
        let innerVar = "I am in inner function"; // Local scope of
innerFunction
        console.log(innerVar); // Accessing local variable inside
innerFunction
        console.log(outerVar); // Accessing variable from outerFunction
scope
        console.log(globalVar); // Accessing global variable
    }

    innerFunction();
}
```

```
outerFunction();
```

### Explanation:

- The **scope chain** works as follows:

1. When `innerFunction()` is called, it first checks for `innerVar` in its own scope (the local scope of `innerFunction`).
2. If `innerVar` isn't found, it checks the **outer function's scope** (`outerFunction()`), and then the **global scope** if necessary.
3. The scope chain enables `innerFunction()` to access `outerVar` from its parent scope (`outerFunction`) and `globalVar` from the global scope.

## 3. Lexical Scope

**Lexical scope** (also called **static scope**) means that the **scope of a variable is determined by where the variable is declared in the code** (at the time of writing, not at runtime). In other words, functions are able to access variables from their outer (parent) scopes based on where they were defined, not where they are executed.

### Example:

```
function outer() {
    let outerVar = "I am in outer";

    function inner() {
        console.log(outerVar); // Accesses outerVar due to lexical
scoping
    }

    inner();
}

outer();
```

### Explanation:

- `inner()` has access to `outerVar` because of **lexical scoping.** It was **lexically** defined within `outer()`, so it can access `outerVar`, even though `inner()` is invoked within the function `outer()`.

In JavaScript, **lexical scoping** ensures that functions always remember where they were created, not where they are invoked.

## 4. Closure

A **closure** is a function that retains access to its lexical scope, even when the function is executed outside of that scope. In simpler terms, a closure is a function that "remembers" the environment in which it was created.

A closure happens when:

- A function is defined inside another function.
- The inner function references variables from the outer function.

### Example:

```
function outer() {
    let counter = 0; // Local variable in outer

    return function inner() {
        counter++; // The inner function has access to counter
        console.log(counter);
    };
}

const increment = outer(); // `increment` is now a closure
increment(); // Logs 1
increment(); // Logs 2
increment(); // Logs 3
```

### Explanation:

- `inner()` is a closure because it retains access to the variable `counter` from its **lexical scope** (the `outer()` function), even after `outer()` has finished executing.
- Every time `increment()` is called, it still has access to the `counter` variable from the **scope of `outer()`**, and the state of `counter` is preserved between calls.

## Key Points about Closures:

- Closures allow you to create functions with private data. The function retains access to variables that are "outside" its current scope.
- Closures are frequently used in scenarios such as **data encapsulation** (hiding private variables) and **callback functions** (where functions are passed and executed at a later time).

### Example: Closure for Data Encapsulation (Private Variables)

```
function createCounter() {
    let count = 0; // Private variable

    return {
        increment: function() {
            count++;
            console.log(count);
        },
        decrement: function() {
            count--;
            console.log(count);
        },
        getCount: function() {
            return count;
        }
    };
}
```

```
const counter = createCounter();
counter.increment(); // Logs 1
counter.increment(); // Logs 2
counter.decrement(); // Logs 1
console.log(counter.getCount()); // Logs 1
```

### Explanation:

- The `count` variable is **private** to the `createCounter` function, but is accessible through the `increment`, `decrement`, and `getCount` methods.
- These methods form a **closure** around `count` and maintain its state, even though the `createCounter` function has finished executing.

# Prototypes & Inheritance

## Prototypes and Inheritance in JavaScript

JavaScript is an object-oriented language, but it uses a unique inheritance model called **prototype-based inheritance.** Unlike class-based inheritance in languages like Java or C++, JavaScript uses prototypes to allow objects to inherit properties and methods from other objects.

Let's dive deep into **prototypes** and **inheritance** in JavaScript with detailed explanations and examples.

## 1. Prototypes in JavaScript

Every JavaScript object has an internal property called `[[Prototype]]`, which is a reference to another object. This object is called the **prototype.** Prototypes allow objects to inherit properties and methods from other objects.

In simpler terms:

- **Prototype** is like a blueprint for an object. Objects can inherit methods and properties from their prototype object.

### Example: Using Prototypes

```
// Create an object with some properties
let person = {
    firstName: "John",
    lastName: "Doe",
    greet: function() {
        console.log("Hello, " + this.firstName + " " + this.lastName);
    }
};

// Create another object using `Object.create()`, inheriting from
`person`
let anotherPerson = Object.create(person);
```

```
anotherPerson.firstName = "Jane";
anotherPerson.lastName = "Smith";

anotherPerson.greet(); // Outputs: "Hello, Jane Smith"
```

## Explanation:

- person is an object with properties firstName, lastName, and a method greet().
- anotherPerson is created with Object.create(person), which means anotherPerson will inherit properties and methods from person. The greet() method of person is available to anotherPerson, so when we call anotherPerson.greet(), it successfully accesses the inherited method.

The prototype of anotherPerson is person, so when anotherPerson doesn't have its own greet() method, it looks up to its prototype (person) to find it.

## 2. Prototype Chain

The **prototype chain** is a chain of objects connected through the [[Prototype]] property. If an object does not have a property or method, JavaScript looks for it in the object's prototype. If it's not found in the prototype, the search continues up the prototype chain.

The prototype chain looks like this:

- When you access a property or method of an object, JavaScript first checks the object itself.
- If the property or method isn't found in the object, it checks the prototype (the object from which the current object was created).
- If it's still not found, it checks the prototype of the prototype, and so on until it reaches null, which is the end of the prototype chain.

### Example: Prototype Chain

```
let animal = {
    eats: true
};

let dog = Object.create(animal); // dog inherits from animal
dog.barks = true;

let puppy = Object.create(dog); // puppy inherits from dog

console.log(puppy.eats);  // Inherited from animal - true
console.log(puppy.barks); // Inherited from dog - true
console.log(puppy.hasOwnProperty('eats')); // false, puppy doesn't have
'eats' directly
console.log(puppy.hasOwnProperty('barks')); // false, puppy doesn't have
'barks' directly
```

## Explanation:

- `animal` has a property `eats`.
- `dog` inherits from `animal` and has an additional property `barks`.
- `puppy` inherits from `dog` (and therefore from `animal` too).

When you access `puppy.eats`, JavaScript checks if `puppy` has the `eats` property. It doesn't, so it looks up the prototype chain and finds it in `animal`.

The **prototype chain** works recursively, so the `puppy` object has access to properties of both `dog` and `animal`.

## 3. Constructor Functions and Prototypes

In JavaScript, you can create multiple instances of objects using **constructor functions**. These functions are used to create objects, and each object created by the constructor function has access to the methods defined on the constructor's prototype.

### Example: Constructor Function and Prototypes

```
// Constructor function
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

// Add method to the prototype
Person.prototype.greet = function() {
    console.log("Hello, " + this.firstName + " " + this.lastName);
};

// Create an instance
let john = new Person("John", "Doe");
john.greet();  // Outputs: "Hello, John Doe"
```

## Explanation:

- `Person` is a **constructor function.** When you call `new Person()`, a new object is created, and `this` inside the constructor refers to that object.
- The `greet()` method is added to `Person.prototype`. This means that all instances of `Person` (like `john`) will have access to the `greet()` method via the prototype.

You can add methods to an object's prototype to ensure they are shared among all instances of that object, rather than each instance having its own copy of the method.

## 4. Inheritance in JavaScript

**Inheritance** in JavaScript refers to an object gaining access to properties and methods of another object. With prototype-based inheritance, JavaScript allows one object to inherit from another by linking their prototypes.

### Example: Inheritance Using Constructor Functions

```
// Parent constructor function
function Animal(name) {
    this.name = name;
}
```

```
// Parent method
Animal.prototype.makeSound = function() {
    console.log(this.name + " makes a sound");
};

// Child constructor function
function Dog(name, breed) {
    Animal.call(this, name);  // Call parent constructor with `this`
binding
    this.breed = breed;
}

// Inherit from Animal
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Child method
Dog.prototype.bark = function() {
    console.log(this.name + " barks");
};

// Create instances
let myDog = new Dog("Max", "Golden Retriever");

myDog.makeSound();  // Inherited from Animal - "Max makes a sound"
myDog.bark();       // Defined in Dog - "Max barks"
```

## Explanation:

- **Inheritance in JavaScript** can be achieved by setting the prototype of the child constructor (`Dog`) to be an instance of the parent constructor's prototype (`Animal.prototype`).
- `Animal.call(this, name)` ensures that the `name` property is inherited from the `Animal` constructor function.

- Dog.prototype is now linked to Animal.prototype, so instances of Dog have access to methods defined in Animal.

## 5. ES6 Classes and Inheritance

With the introduction of ES6, JavaScript introduced **classes** to provide a more familiar syntax for object creation and inheritance (similar to class-based inheritance in other languages like Java or Python). However, under the hood, JavaScript classes are still using prototype-based inheritance.

### Example: ES6 Classes and Inheritance

```
// Parent class
class Animal {
    constructor(name) {
        this.name = name;
    }

    makeSound() {
        console.log(this.name + " makes a sound");
    }
}

// Child class
class Dog extends Animal {
    constructor(name, breed) {
        super(name);  // Call the parent constructor
        this.breed = breed;
    }

    bark() {
        console.log(this.name + " barks");
    }
}

// Create an instance
let myDog = new Dog("Max", "Golden Retriever");
```

```
myDog.makeSound();  // Inherited from Animal - "Max makes a sound"
myDog.bark();       // Defined in Dog - "Max barks"
```

## Explanation:

- **ES6 `class` syntax** makes the code cleaner and more readable. The `extends` keyword is used to set up inheritance.
- `super(name)` calls the constructor of the parent class (`Animal`), and the child class (`Dog`) adds its own properties and methods.
- The prototype-based inheritance still applies under the hood, but the class syntax provides a more traditional object-oriented approach.

# ES6

ES6 (also known as ECMAScript 2015) is a major update to JavaScript that introduced many new features and improvements to the language. These new features make JavaScript more powerful, cleaner, and easier to write. Below is a detailed explanation of ES6 features, along with examples to help you understand how they work.

## 1. Let and Const

Before ES6, JavaScript only had `var` to declare variables. However, `var` has some issues like hoisting and scope problems. ES6 introduced `let` and `const` to provide more predictable behavior when declaring variables.

- **`let`**: Block-scoped variable declaration.
- **`const`**: Block-scoped, read-only variable declaration (once assigned, the value cannot be changed).

### Example:

```
// let example
let name = "John";  // Declare a variable with let
if (true) {
    let name = "Jane";  // This is a new variable, scoped inside the
block
    console.log(name);   // "Jane"
}
console.log(name);      // "John"

// const example
const pi = 3.14159;    // Declare a constant variable
// pi = 3.14;           // Error: Assignment to constant variable
console.log(pi);        // 3.14159
```

### Explanation:

- `let` is block-scoped, meaning it only exists within the block `{}` where it's defined.
- `const` ensures that the variable cannot be reassigned, making it useful for values that should not change.

## 2. Arrow Functions

Arrow functions provide a shorter syntax for writing functions and automatically bind the context (`this`) to the function's lexical scope, unlike regular functions.

### Example:

```
// Traditional function
function greet(name) {
    return "Hello, " + name;
}

// Arrow function
const greetArrow = (name) => "Hello, " + name;

console.log(greet("John"));         // "Hello, John"
console.log(greetArrow("Jane"));    // "Hello, Jane"
```

### Explanation:

- Arrow functions are more concise, especially for single-line functions.
- The `this` keyword inside an arrow function refers to the surrounding lexical context, unlike regular functions where `this` can change based on how the function is called.

## 3. Template Literals

Template literals provide an easier way to work with strings, especially when you need to embed expressions or variables. They use backticks (`` ` ``) instead of single or double quotes.

### Example:

```
let firstName = "John";
let lastName = "Doe";

// Traditional concatenation
let greeting = "Hello, " + firstName + " " + lastName;

// Template literals
let greetingTemplate = `Hello, ${firstName} ${lastName}`;

console.log(greeting);          // "Hello, John Doe"
console.log(greetingTemplate);  // "Hello, John Doe"
```

### Explanation:

- Template literals allow you to interpolate expressions using `${}` and can span multiple lines without the need for concatenation or escape characters.

## 4. Destructuring Assignment

Destructuring is a way to unpack values from arrays or objects into variables. It makes working with arrays and objects more convenient and readable.

### Example 1: Array Destructuring

```
let colors = ["red", "green", "blue"];

// Array destructuring
let [first, second, third] = colors;

console.log(first);  // "red"
console.log(second); // "green"
console.log(third);  // "blue"
```

### Example 2: Object Destructuring

```
let person = { name: "John", age: 30 };
```

```
// Object destructuring
let { name, age } = person;

console.log(name); // "John"
console.log(age);  // 30
```

### Explanation:

- **Array Destructuring** allows you to unpack elements from an array into variables.
- **Object Destructuring** allows you to extract properties from an object into variables with the same name as the property.

## 5. Spread Operator (...)

The spread operator (`...`) allows you to unpack elements from an array or object and spread them out into individual elements or properties. It's especially useful for copying or combining arrays and objects.

### Example 1: Spread with Arrays

```
let numbers = [1, 2, 3];

// Copy an array
let newNumbers = [...numbers];
newNumbers.push(4);

console.log(numbers);    // [1, 2, 3]
console.log(newNumbers); // [1, 2, 3, 4]
```

### Example 2: Spread with Objects

```
let person = { name: "John", age: 30 };

// Copy an object
let newPerson = { ...person, city: "New York" };
```

```
console.log(person);    // { name: "John", age: 30 }
console.log(newPerson); // { name: "John", age: 30, city: "New York" }
```

### Explanation:

- The spread operator allows you to easily copy arrays or objects and merge them with additional values.

## 6. Rest Parameter (...)

The rest parameter (`...`) is used to collect all remaining arguments passed to a function into an array. It provides an elegant way to handle a variable number of arguments.

### Example:

```
// Function with rest parameter
function sum(...numbers) {
    return numbers.reduce((acc, num) => acc + num, 0);
}

console.log(sum(1, 2, 3)); // 6
console.log(sum(4, 5, 6, 7, 8)); // 30
```

### Explanation:

- The `...numbers` collects all arguments passed to the function into an array, allowing the function to accept any number of arguments.

## 7. Classes

ES6 introduced **classes** as a syntactical sugar over the existing prototype-based inheritance. Classes allow you to define objects and their behaviors more clearly.

### Example:

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log(`Hello, my name is ${this.name} and I am ${this.age}
years old.`);
    }
}

const person = new Person("John", 30);
person.greet(); // "Hello, my name is John and I am 30 years old."
```

### Explanation:

- Classes in ES6 are used to create objects with specific properties and methods.
- The `constructor` method is a special method used to initialize an object when it is created.

## 8. Modules

ES6 introduced **modules**, which allow you to split your JavaScript code into smaller, reusable pieces. Each module can export variables, functions, or objects, and they can be imported into other files.

### Example:

**module.js** (Exporting)

```
export const name = "John";
export function greet() {
    console.log("Hello, " + name);
}
```

**app.js** (Importing)

```
import { name, greet } from './module.js';
```

```
console.log(name);  // "John"
greet();            // "Hello, John"
```

### Explanation:

- `export` is used to make variables, functions, or objects available to other files.
- `import` is used to bring the exported pieces of code into another file.

## 9. Promises

Promises provide a cleaner, more readable way to work with asynchronous operations, especially when handling callbacks and asynchronous tasks like API calls.

### Example:

```
// Creating a promise
let promise = new Promise((resolve, reject) => {
    let success = true;

    if (success) {
        resolve("The operation was successful");
    } else {
        reject("The operation failed");
    }
});

// Consuming the promise
promise
    .then(result => console.log(result))  // Success
    .catch(error => console.log(error));  // Error
```

### Explanation:

- A **promise** represents an operation that will eventually complete (or fail). You can use `.then()` to handle success and `.catch()` to handle errors.

## 10. Generators

Generators are a special type of function that can pause execution and resume it later. They are useful for handling asynchronous programming in a more readable way.

### Example:

```
function* numbers() {
    yield 1;
    yield 2;
    yield 3;
}

const numGenerator = numbers();

console.log(numGenerator.next()); // { value: 1, done: false }
console.log(numGenerator.next()); // { value: 2, done: false }
console.log(numGenerator.next()); // { value: 3, done: false }
console.log(numGenerator.next()); // { value: undefined, done: true }
```

### Explanation:

- The `yield` keyword is used to pause the function and return a value. The generator function can be resumed each time `.next()` is called.

# Object-Oriented Programing

# DOM

## The HTML DOM (Document Object Model)

When a web page is loaded, the browser creates a Document Object Model of the page.

The HTML DOM model is constructed as a tree of Objects:

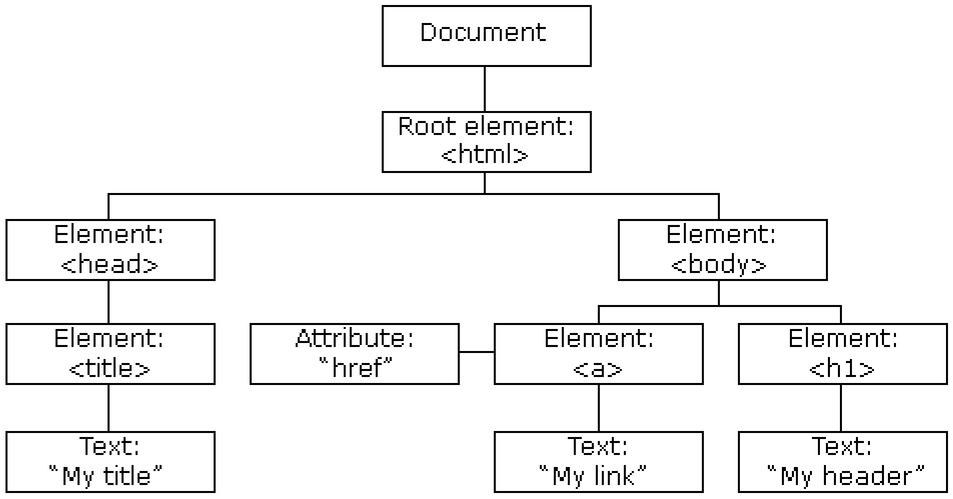


image_4.png

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

## What is the DOM?

The DOM is a W3C (World Wide Web Consortium) standard.

The DOM defines a standard for accessing documents:

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The W3C DOM standard is separated into 3 different parts:

- Core DOM - standard model for all document types
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

## What is the HTML DOM?

The HTML DOM is a standard `object` model and `programming interface` for HTML. It defines:

- The HTML elements as `objects`
- The `properties` of all HTML elements
- The `methods` to access all HTML elements
- The `events` for all HTML elements

## HTML DOM Methods

- HTML DOM `methods` are actions you can perform (on HTML Elements).
- HTML DOM `properties` are values (of HTML Elements) that you can set or change.

⚠ The document object represents your web page. If you want to access any element in an HTML page, you always start with accessing the document object. Below are some examples of how you can use the document object to access and manipulate HTML.

Here are tables and examples for the methods you've mentioned related to finding HTML elements, changing HTML elements, adding and deleting elements, and adding event handlers.

### 1. Finding HTML Elements

| Method | Description | Example |
|---|---|---|
| `document.getElementById(id)` | Find an element by element id | `document.getElementById("myId")` |
| `document.getElementsByTagName(name)` | Find elements by tag name | `document.getElementsByTagName("div")` |
| `document.getElementsByClassName(name)` | Find elements by class name | `document.getElementsByClassName("myClass")` |

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Finding Elements</title>
</head>
<body>
    <div id="myDiv">Hello, World!</div>
    <div class="myClass">Another div</div>

    <script>
        // Using getElementById
        let elementById = document.getElementById("myDiv");
        console.log(elementById.innerHTML);  // Outputs: Hello, World!
        // Using getElementsByTagName
        let divs = document.getElementsByTagName("div");
        console.log(divs[0].innerHTML);  // Outputs: Hello, World!

        // Using getElementsByClassName
        let elementsByClass =
document.getElementsByClassName("myClass");
        console.log(elementsByClass[0].innerHTML);  // Outputs: Another
div
    </script>
</body>
</html>
```

## 2. Changing HTML Elements

| Property/Method | Description | Example |
|---|---|---|
| `element.innerHTML = newContent` | Change the inner HTML of an element | `element.innerHTML = "<p>New content</p>"` |
| `element.attribute = newValue` | Change the attribute value of an HTML element | `element.src = "newImage.jpg"` |
| `element.style.property = newStyle` | Change the style of an HTML element | `element.style.backgroundColor = "blue"` |
| `element.setAttribute(attribute, value)` | Change the attribute value of an HTML element | `element.setAttribute("href", "newLink.html")` |

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Changing HTML Elements</title>
```

```
</head>
<body>
    <div id="content">Original content</div>
    <img id="image" src="oldImage.jpg" alt="Old Image">

    <script>
        // Change innerHTML
        let contentDiv = document.getElementById("content");
        contentDiv.innerHTML = "New content added!";

        // Change image src
        let image = document.getElementById("image");
        image.src = "newImage.jpg";

        // Change background color style
        contentDiv.style.backgroundColor = "yellow";

        // Change href attribute
        let link = document.createElement("a");
        link.setAttribute("href", "https://newlink.com");
        link.innerHTML = "Visit New Link";
        document.body.appendChild(link);
    </script>
</body>
</html>
```

## 3. Adding and Deleting Elements

| Method | Description | Example |
|---|---|---|
| document.createElement(element) | Create an HTML element | let newDiv = document.createElement("div") |
| document.removeChild(element) | Remove an HTML element from the DOM | parentElement.removeChild(childElement) |
| document.appendChild(element) | Add an HTML element as a child | parentElement.appendChild(newElement) |
| document.replaceChild(new, old) | Replace an HTML element with another | parent.replaceChild(newElement, oldElement) |
| document.write(text) | Write directly into the HTML output stream | document.write("<p>Hello World</p>") |

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Adding and Deleting Elements</title>
</head>
<body>
    <div id="parentDiv">
        <p>This is a parent div.</p>
    </div>

    <script>
        // Create a new element
        let newElement = document.createElement("p");
        newElement.innerHTML = "This is a new paragraph.";
        // Append new element to parent
        let parentDiv = document.getElementById("parentDiv");
        parentDiv.appendChild(newElement);

        // Remove an element
        let paragraphToRemove = parentDiv.getElementsByTagName("p")[0];
        parentDiv.removeChild(paragraphToRemove);

        // Replace an element
        let newElementToReplace = document.createElement("p");
        newElementToReplace.innerHTML = "This paragraph replaced the old
one.";
        parentDiv.replaceChild(newElementToReplace, newElement);

        // Write content directly into HTML
        document.write("<h1>Page Loaded</h1>");
    </script>
</body>
</html>
```

### 4. Adding Event Handlers

| Method | Description | Example |
|---|---|---|
| `document.getElementById(id).onclick` | Add event handler code to an onclick event | `document.getElementById("myBtn").onclick = function() { alert("Clicked!"); }` |

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Adding Event Handlers</title>
</head>
<body>
```

```
    <button id="myBtn">Click me!</button>

    <script>
        // Add an onclick event handler to a button
        document.getElementById("myBtn").onclick = function() {
            alert("Button clicked!");
        };
    </script>
</body>
</html>
```

**Explanation:**

- When the button with `id="myBtn"` is clicked, it triggers the `onclick` event handler, which displays an alert box.

## HTML DOM Elements

### Finding HTML Elements

Often, with JavaScript, you want to manipulate HTML elements.

To do so, you have to find the elements first. There are several ways to do this:

- Finding HTML elements by id
- Finding HTML elements by tag name
- Finding HTML elements by class name
- Finding HTML elements by CSS selectors
- Finding HTML elements by HTML object collections

### Finding HTML Elements in JavaScript

In JavaScript, when you want to manipulate HTML elements, the first step is to **find** them in the DOM (Document Object Model). There are several methods available for finding elements, and each has its specific use cases.

## 1. Finding HTML Elements by ID

The easiest way to find a specific HTML element is by using the **element id.** Each element in an HTML document can have a unique `id` attribute, and you can use `document.getElementById()` to find that element.

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Finding Elements by ID</title>
</head>
<body>
    <div id="intro">This is the introduction section.</div>

    <script>
        // Finding element by id
        const element = document.getElementById("intro");

        // Check if element exists
        if (element) {
            console.log(element.innerHTML);  // Outputs: This is the
introduction section.
        } else {
            console.log("Element not found.");
        }
    </script>
</body>
</html>
```

**Explanation:**

- The method `document.getElementById("intro")` finds the element with the `id="intro"`.
- If the element is found, it will be returned, and you can access its properties, such as `innerHTML`. If it's not found, it returns `null`.

## 2. Finding HTML Elements by Tag Name

You can use `document.getElementsByTagName()` to find all elements with a specific tag name. This method returns a **live HTMLCollection**, meaning it automatically updates if the DOM changes.

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Finding Elements by Tag Name</title>
</head>
<body>
    <p>This is a paragraph.</p>
    <p>This is another paragraph.</p>
    <div>Some content in a div.</div>

    <script>
        // Find all <p> elements
        const paragraphs = document.getElementsByTagName("p");

        // Loop through all paragraphs and log their content
        for (let i = 0; i < paragraphs.length; i++) {
            console.log(paragraphs[i].innerHTML);  // Outputs: This is a
paragraph. and This is another paragraph.
        }
    </script>
</body>
</html>
```

**Explanation:**

- The method `document.getElementsByTagName("p")` returns all `<p>` elements in the document.
- You can loop through the resulting HTMLCollection to access each element.

## 3. Finding HTML Elements by Class Name

If you want to find elements that share the same class, use `document.getElementsByClassName()`. It returns a **live HTMLCollection** of elements with the specified class name.

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Finding Elements by Class Name</title>
</head>
<body>
    <div class="intro">This is the first intro section.</div>
    <div class="intro">This is the second intro section.</div>

    <script>
        // Find all elements with class "intro"
        const introElements = document.getElementsByClassName("intro");

        // Loop through all elements and log their content
        for (let i = 0; i < introElements.length; i++) {
            console.log(introElements[i].innerHTML);  // Outputs the
content of both divs with class "intro"
        }
    </script>
</body>
</html>
```

**Explanation:**

- The method `document.getElementsByClassName("intro")` returns all elements that have the class name `intro`.
- It returns an HTMLCollection of elements, and you can loop through them to access each one.

## 4. Finding HTML Elements by CSS Selectors

The `document.querySelectorAll()` method allows you to find all elements that match a specified CSS selector. This gives you more flexibility, allowing you to find elements based on **id**, **class**, **attributes**, or **combinations** of those selectors.

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Finding Elements by CSS Selectors</title>
</head>
<body>
    <p class="intro">This is a paragraph with the class "intro".</p>
    <p class="intro">This is another paragraph with the class "intro".
</p>
    <p>This is a regular paragraph without the "intro" class.</p>

    <script>
        // Find all <p> elements with class "intro"
        const introParagraphs = document.querySelectorAll("p.intro");

        // Loop through and log each element's content
        introParagraphs.forEach(function(paragraph) {
            console.log(paragraph.innerHTML);  // Outputs the two intro
paragraphs
        });
    </script>
</body>
</html>
```

**Explanation:**

- The method `document.querySelectorAll("p.intro")` uses the CSS selector `p.intro` to find all `<p>` elements with the class `intro`.

- `querySelectorAll()` returns a **NodeList**, which can be iterated over using `forEach` or other array methods.

# JS Web APIs

What is Web API?

- API stands for Application Programming Interface.
- A Web API is an application programming interface for the Web.
- A Browser API can extend the functionality of a web browser.
- A Server API can extend the functionality of a web server.

## Browser APIs

All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data. For example, the Geolocation API can return the coordinates of where the browser is located.

### Web Storage API

The Web Storage API is a simple syntax for storing and retrieving data in the browser. It is very easy to use:

- **The localStorage Object**

The localStorage object provides access to a local storage for a particular Web Site. It allows you to store, read, add, modify, and delete data items for that domain.

#### The setItem() Method

The `localStorage.setItem()` method stores a data item in a storage.

```
localStorage.setItem("name", "John Doe");
```

#### The getItem() Method

The `localStorage.getItem()` method retrieves a data item from the storage.

```
localStorage.getItem("name");
```

**removeItem() method**

```
localStorage.removeItem("name");
```

- **The sessionStorage Object**

The sessionStorage object is identical to the localStorage object. The difference is that the sessionStorage object stores data for one session.

⚠ The data is deleted when the browser is closed.

**The setItem() Method**

The `sessionStorage.setItem()` method stores a data item in a storage. It takes a name and a value as parameters:

```
sessionStorage.setItem("name", "John Doe");
```

**The getItem() Method**

The `sessionStorage.getItem()` method retrieves a data item from the storage. It takes a name as parameter:

# TS

- Typescript is nothing but a superset of JavaScript. It is built on top of javascript and introduces syntax enhancements. It brings support for types and class-based object-oriented programming paradigm to the world of Javascript. When compiled (or transpiled) it produces Javascript.

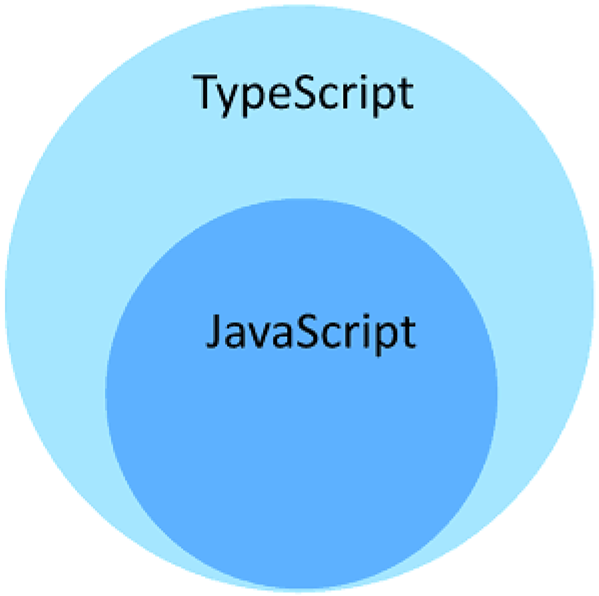


image_5.png

- Any valid Javascript code is also a valid Typescript code. It uses the same syntax and semantics as the javascript
- Typescript is open source and free to use. It is designed, developed and maintained by Microsoft.

## What TypeScript is not

Typescript is not a new programming language. It just a syntactic sugar added over Javascript. You can write pure javascript code and typescript still compiles it just as it is to Javascript.

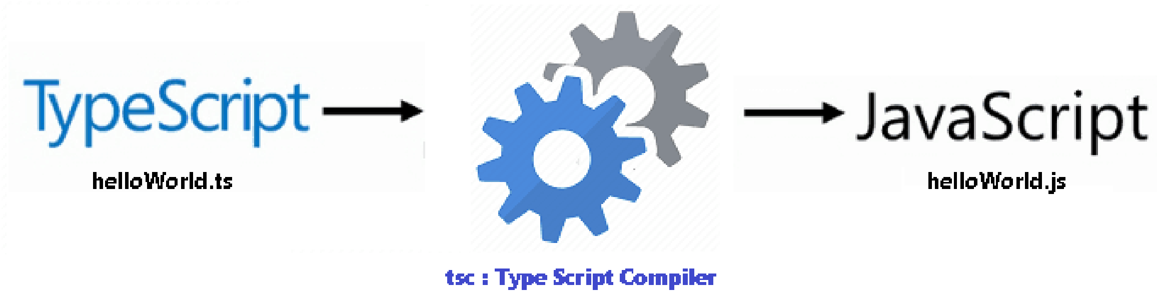


image_6.png

If you know Javascript, the learning curve is very lean. If you are new to javascript, but coming from the c# or, Java background, you will see a lot of similarities in concept.

## Benefits of TypeScript

- Optional Type System TypeScript provides the static type system which provides great help in catching programming errors at compile time.

> ⚠ Javascript is a dynamically typed system. The variables can hold any values. The type of variable is determined on the fly. The javascript implicitly converts types for example string to a number. This is ok for a small app, but large apps this can be a lot of headaches. It is difficult to test to see if the proper types are passed and errors always happen at runtime.

- Intellisense & syntax checking The static Type system helps in provide better tooling support in IDE. The intellisense, syntax checking & code completion are few of the major benefits you get with the tooling support. This speeds up the development time and also ensures that the programmers make fewer mistakes with typos. All the major editors like VSCode, atom, sublime text includes the tooling support for Typescript
- Maintainable code Typescript brings Types, Classes, interfaces & modules. it makes the code more maintainable and scalable. It much easier to organize the Typescript code, than a Javascript code
- Language Enhancement Typescript comes with several language features. It supports Encapsulation through classes and modules. Supports constructors, properties &

functions. It has support for Interfaces. You can make use of Arrow functions or lambdas or anonymous functions

## Typescript Components

The architecture of TypeScript is neatly organized in different layers as shown in the image below. The three major layers are

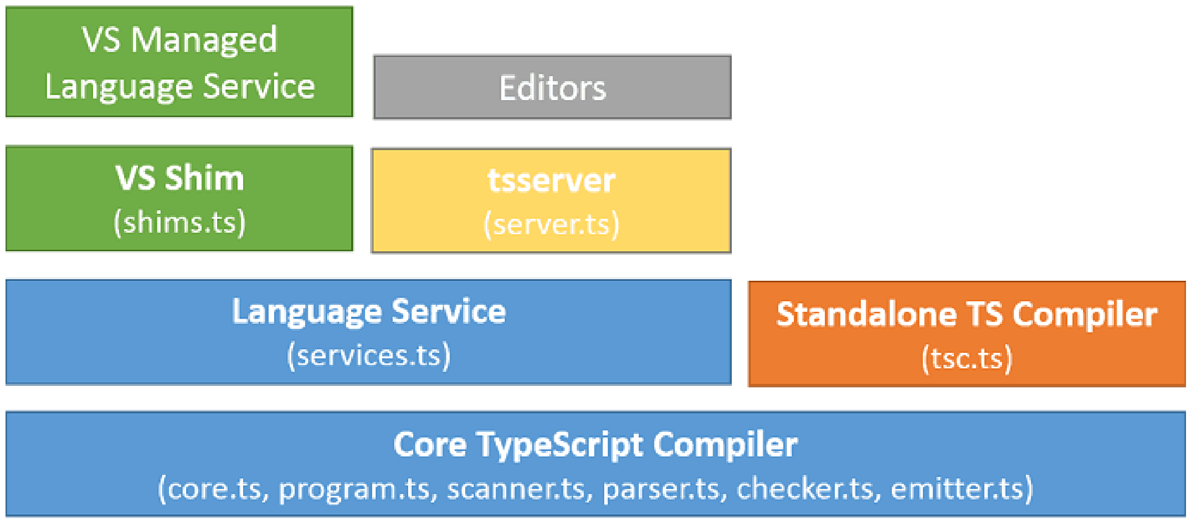

VS Managed Language Service
Editors
VS Shim (shims.ts)
tsserver (server.ts)
Language Service (services.ts)
Standalone TS Compiler (tsc.ts)
Core TypeScript Compiler (core.ts, program.ts, scanner.ts, parser.ts, checker.ts, emitter.ts)

image_7.png

- **Core TypeScript Compiler**: The TypeScript compiler manages the task of type-checking our code and converting it into valid JavaScript code. The compiler is made up of several different layers like core, program, scanner, parser, checker & emitter, etc
- **Typescript Standalone Compiler** (TSC) : The batch compilation CLI. Mainly handle reading and writing files for different supported engines (e.g. Node.js)
- **Typescript Language Services**: The Language Service supports the editors and other tools to provide better assistance in implementing features such as IntelliSense, code completion, formatting and outlining, colorization, code re-factoring like rename, Debugging interface helpers like validating breakpoints, etc.

# Setting-up a TS project

## Step 1: Install Node.js and npm

Before setting up TypeScript, make sure you have **Node.js** and **npm** (Node Package Manager) installed. You can verify their installation by running:

```
node -v
pnpm -v
```

If you haven't installed them, download and install Node.js from Node.js Download (https://nodejs.org/).

## Step 2: Initialize a Node.js Project

1. Create a new directory for your project:

   ```
   mkdir my-ts-project
   cd my-ts-project
   ```

2. Initialize the Node.js project:

   ```
   npm init -y
   ```

   or

   ```
   pnpm init -y
   ```

   This creates a `package.json` file with default values.

## Install TypeScript

You need to install **TypeScript** for compiling `.ts`.

1. Install TypeScript:

```
pnpm add -D typescript tsx
```

## Create tsconfig.json File

The `tsconfig.json` file is used to configure how TypeScript compiles your code. You can generate a `tsconfig.json` with default settings by running:

```
pnpx tsc --init
```

Alternatively, you can manually create a `tsconfig.json` file with the following configuration:

```
{
  "compilerOptions": {
    "target": "es5",                           // Target JavaScript version
(ES5 for wide compatibility)
    "module": "commonjs",                       // Module system (CommonJS
for Node.js)
    "strict": true,                             // Enable strict type-
checking options
    "esModuleInterop": true,                    // Allows default imports
from CommonJS modules
    "jsx": "react",                             // Enable JSX support for
React in .tsx files
    "skipLibCheck": true,                       // Skip type checking of
declaration files
    "outDir": "./dist",                         // Output directory for
compiled JavaScript
    "rootDir": "./src",                         // Root directory for source
files
    "forceConsistentCasingInFileNames": true // Ensure consistent casing
in file names
  },
  "include": [
    "src/**/*.ts",    // Include all .ts files
    "src/**/*.tsx"    // Include all .tsx files (for React)
  ],
  "exclude": [
    "node_modules"    // Exclude node_modules from the compilation
process
  ]
}
```

## Key Options in tsconfig.json:

- `jsx`: Set to `react` for projects using JSX (e.g., React). This ensures `.tsx` files are compiled correctly.
- `target`: JavaScript version to compile to, such as `es5` or `es6`.
- `outDir`: Directory to store compiled JavaScript files.
- `rootDir`: Directory where your TypeScript source files are located (`src`).
- `include`: Specifies that `.ts` and `.tsx` files in the `src` folder should be compiled.
- `strict`: Enables strict type-checking to help catch errors early.

## Compile TypeScript to JavaScript

To compile your TypeScript code, run the TypeScript compiler (`tsc`) in the terminal:

```
npx tsc
```

or

```
pnpx tsc
```

## Add Build and Start Scripts (Optional)

You can add custom scripts to your `package.json` file to make the compilation process easier.

1. Open `package.json` and add the following under `"scripts"`:

```
"scripts": {
   "dev": "tsx watch index.ts", // run the code in dev mode
  "build": "tsc",         // Compile TypeScript files
  "start": "node dist/index.js"  // Run compiled JavaScript for
Node.js projects
}
```

2. Now you can run the following commands:

- • **Run TypeScript Code in dev **:

  ```
  pnpm run dev
  ```

- **Compile TypeScript**:

  ```
  pnpm run build
  ```

- **Run the compiled JavaScript**:

  ```
  pnpm start
  ```

# Typescript Basics

Typescript Types (or data types) bring static type checking into the dynamic world of Javascript.

## What is a Data Type?

The Type or the Data Type is an attribute of data that tells us what kind of value the data has. Whether it is a `number`, `string`, `boolean`, etc. The type of data defines the operations that we can do on that data.

## Typescript Data Types

JavaScript has eight data types. `Seven` primitive types and one object Data type. The primitive types are `number`, `string`, `boolean`, `bigint`, `symbol`, `undefined`, and null. Everything else is an `object` in JavaScript.

> The TypeScript Type System supports all of them and also brings its own special types. They are `unknown`, `any`, `void` & `never`. TypeScript also provides both numeric and string-based `enums`. Enums allow a developer to define a set of named constants

The following diagram shows the relationship between various types.

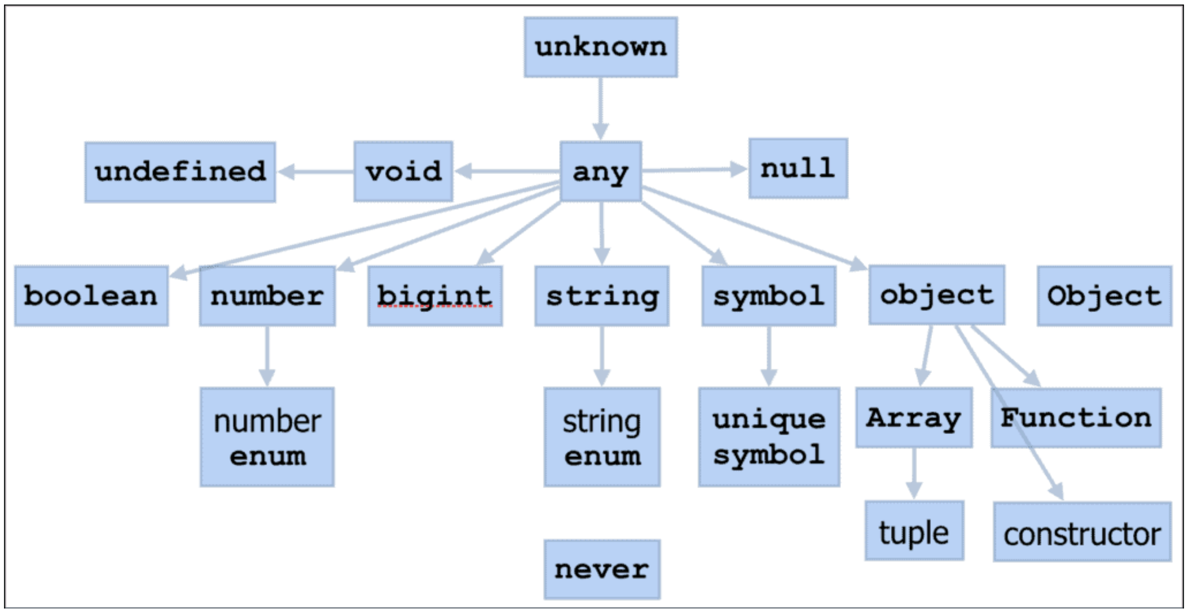


image_8.png

Sure! Here's a detailed set of notes on the topics you requested, along with examples to help you understand each concept:

## 1. TypeScript Data Types

In TypeScript, variables are assigned types to enforce type safety. These types can be basic types like numbers and strings, or more complex types like arrays and objects.

### Basic Data Types:

- **number**: Represents both integer and floating-point numbers.
- **string**: Represents text values.
- **boolean**: Represents `true` or `false`.
- **null**: Represents a null value (intentional absence of any object value).
- **undefined**: Represents an undefined value (a variable that has been declared but not assigned a value).
- **void**: Used when there is no return value for a function.

**Example:**

```
let age: number = 25;
let name: string = "Alice";
let isActive: boolean = true;
let nothing: null = null;
let undefinedValue: undefined = undefined;
```

## 2. Type Annotations

Type annotations in TypeScript are used to explicitly specify the type of a variable, function parameter, or return type. Type annotations help TypeScript enforce the correct type for variables and functions.

### Syntax:

```
let variableName: type = value;
```

**Example:**

```
let message: string = "Hello, TypeScript!";
let count: number = 10;
```

In this example, `message` is annotated as a `string`, and `count` is annotated as a `number`. This helps TypeScript enforce that only values of these types can be assigned to these variables.

## 3. Variable Declaration

In TypeScript, you can declare variables using the `let`, `const`, and `var` keywords.

### let:

- Used to declare a block-scoped variable.
- The value of `let`-declared variables can be changed.

### const:

- Used to declare a block-scoped, read-only constant.
- The value of a `const` variable cannot be changed after initialization.

### var:

- **Function-scoped** variable declaration (more common in JavaScript).
- It is less commonly used in TypeScript since `let` and `const` are more predictable.

**Example:**

```
let age: number = 30; // Variable with let
const birthYear: number = 1991; // Constant with const
var city: string = "New York"; // Function-scoped variable with var
```

## 4. Identifiers & Keywords

- **Identifiers** are the names given to variables, functions, classes, etc. In TypeScript, an identifier can contain letters, digits, underscores, and dollar signs, but it cannot begin with a digit.
- **Keywords** are reserved words that have a special meaning in TypeScript and cannot be used as identifiers. For example, `let`, `const`, `return`, `if`, `else`, `class`, `function`, etc.

### Example:

```
let personName: string = "John"; // "personName" is an identifier
const myClass: string = "Math"; // "myClass" is an identifier

// Keywords cannot be used as identifiers:
let if = 5; // Error: "if" is a keyword
```

## 5. Variable Scope

In TypeScript, the **scope** of a variable determines where the variable can be accessed. The scope is determined by where the variable is declared (within a function, a block, etc.).

### Types of Scopes:

- **Global Scope**: A variable declared outside any function or block is accessible anywhere in the program.
- **Function Scope**: Variables declared inside a function are only accessible within that function.
- **Block Scope**: Variables declared with `let` and `const` inside a block (`{}`) are only accessible within that block.

**Example:**

```
let globalVar: string = "I'm global";

function testScope() {
  let functionVar: string = "I'm local to the function";
  console.log(globalVar);  // Accessible
  console.log(functionVar); // Accessible
}

console.log(globalVar); // Accessible
// console.log(functionVar); // Error: functionVar is not defined
outside the function
```

## 6. Let, Var & Const

### let:

- Block-scoped.
- Can be reassigned.

### const:

- Block-scoped.

- Cannot be reassigned after initialization.

**var:**

- Function-scoped.
- Can be reassigned.
- Can lead to issues like hoisting and scope leakage, so it's best to avoid `var` in modern TypeScript code.

**Example:**

```
let number = 10;
number = 20;  // Valid

const constantValue = 30;
constantValue = 40; // Error: Assignment to constant variable.

var oldVariable = "Hello";
oldVariable = "World";  // Valid
```

## 7. Constants

Constants are variables whose value cannot be changed after initialization. In TypeScript, constants are declared with the `const` keyword. Constants must always be initialized at the time of declaration.

### Example:

```
const PI: number = 3.14;
PI = 3.14159; // Error: Cannot assign to 'PI' because it is a constant
```

## 8. Type Inference

TypeScript can automatically infer the type of a variable based on the assigned value. If you don't explicitly annotate a variable, TypeScript will infer the type for you.

### Example:

```
let number = 5;  // TypeScript infers 'number'
let message = "Hello";  // TypeScript infers 'string'
```

Even though the variables are not annotated with types, TypeScript knows the type because of the assigned value.

## 9. Any Type

The `any` type in TypeScript allows a variable to hold any value. It effectively disables type checking for that variable, making it behave like a plain JavaScript variable.

While `any` is useful in some cases (e.g., when dealing with dynamic content or external libraries), it should be used sparingly, as it undermines the benefits of type safety in TypeScript.

### Example:

```
let anything: any = 5;
anything = "Hello";  // Valid
anything = true;     // Valid
```

## 10. Strings

Strings in TypeScript work the same way as in JavaScript. A string can be enclosed in either single quotes, double quotes, or backticks (for template literals).

### Template Literals:

Template literals allow embedding expressions within string literals, using `${}` syntax.

#### Example:

```
let name: string = "Alice";
let greeting: string = `Hello, ${name}!`; // Template literal
console.log(greeting); // Outputs: Hello, Alice!
```

You can also perform string concatenation using the `+` operator:

```
let part1: string = "Hello, ";
let part2: string = "World!";
let combined: string = part1 + part2; // Concatenation
console.log(combined);  // Outputs: Hello, World!
```

# Strings

## String Data Type in TypeScript

In TypeScript, the **`string`** data type is used to represent textual data. Strings are sequences of characters and can be enclosed in single quotes (`'`), double quotes (`"`), or backticks (`` ` ``). In addition to basic string manipulation, TypeScript supports features like **template strings/literal strings**, **tagged templates**, and **string-to-number conversions.**

## 1. String Data Type

A `string` in TypeScript can hold any sequence of characters, including letters, numbers, symbols, and spaces. Strings are immutable, meaning once they are created, their content cannot be changed directly.

### Example:

```
let greeting: string = "Hello, TypeScript!";
let name: string = 'Alice';

console.log(greeting);  // Output: Hello, TypeScript!
console.log(name);      // Output: Alice
```

- **Concatenation**: You can join strings using the `+` operator.

```
let firstName: string = "John";
let lastName: string = "Doe";
let fullName: string = firstName + " " + lastName;
console.log(fullName);  // Output: John Doe
```

- **String length**: You can get the length of a string using `.length`.

```
let phrase: string = "TypeScript is awesome!";
console.log(phrase.length);  // Output: 22
```

## 2. Template Strings/Literal Strings

Template strings (also known as **template literals**) in TypeScript are a powerful feature that allows for easier string manipulation. They are enclosed in **backticks** (`` ` ``), not single or double quotes, and allow for **string interpolation**, i.e., embedding expressions within strings.

### Key Features:

- **Multiline strings**: Template literals allow you to write strings across multiple lines without the need for concatenation or escape sequences.
- **Expression interpolation**: You can embed expressions within `${}` syntax.

### Example:

```
let name: string = "Alice";
let age: number = 30;

// String interpolation (embedding expressions within the string)
let introduction: string = `Hello, my name is ${name} and I am ${age}
years old.`;
console.log(introduction); // Output: Hello, my name is Alice and I am
30 years old.
```

### Multiline string example:

```
let message: string = `This is a multiline
string using template literals.
It allows for easy formatting.`;
console.log(message);
// Output:
// This is a multiline
// string using template literals.
// It allows for easy formatting.
```

## 3. Tagged Templates

Tagged templates allow you to parse template literals with a function. The **tag function** gets the template literal as an argument, and you can manipulate or process the string and its expressions.

### Syntax:

```
tagFunction`template literal`
```

The **tag function** takes the literal parts of the string and the embedded expressions as its arguments.

### Example: Creating a simple tag function:

```
function highlight(strings: TemplateStringsArray, ...values: any[]) {
  let result = "";
  for (let i = 0; i < values.length; i++) {
    result += strings[i] + `<b>${values[i]}</b>`;  // Add bold tags
around expressions
  }
  result += strings[values.length];
  return result;
}

let name: string = "Alice";
let age: number = 30;

let message: string = highlight`Hello, my name is ${name} and I am
${age} years old.`;
console.log(message);  // Output: Hello, my name is <b>Alice</b> and I
am <b>30</b> years old.
```

In this example:

- The `highlight` function adds `<b>` tags around the dynamic values inside the template string (`${name}` and `${age}`).

- The strings array contains the static parts of the template, and values holds the dynamic expressions.

## 4. String to Number

In TypeScript (and JavaScript), you often need to convert a string that represents a number into an actual number type. This can be done using several methods, depending on the scenario.

### 1. Using parseInt()

The parseInt() function converts a string into an integer. It parses the string from left to right and returns the first integer it finds.

```
let str1: string = "42";
let num1: number = parseInt(str1);
console.log(num1);  // Output: 42 (as number)
```

- If the string cannot be converted into a number, parseInt() returns NaN.

### Example:

```
let str2: string = "hello";
let num2: number = parseInt(str2);
console.log(num2);  // Output: NaN (Not a Number)
```

### 2. Using parseFloat()

The parseFloat() function converts a string into a floating-point number. It works similarly to parseInt(), but it returns a decimal number.

```
let str3: string = "3.14";
let num3: number = parseFloat(str3);
console.log(num3);  // Output: 3.14
```

### 3. Using the Unary Plus (+) Operator

The unary plus operator (+) is a shorthand way to convert a string into a number. It tries to convert the value to a number automatically.

```
let str4: string = "100";
let num4: number = +str4;
console.log(num4);  // Output: 100 (as number)
```

### 4. Using Number() Constructor

The `Number()` function is a more explicit way to convert a string to a number. It works for both integers and floating-point numbers.

```
let str5: string = "123.45";
let num5: number = Number(str5);
console.log(num5);  // Output: 123.45
```

- If the string is not a valid number, `Number()` returns `NaN`.

### Example:

```
let str6: string = "Hello World";
let num6: number = Number(str6);
console.log(num6);  // Output: NaN
```

## Summary of Key Concepts

| Concept | Explanation | Example |
|---|---|---|
| **String Data Type** | Represents textual data. Can be enclosed in single quotes, double quotes, or backticks. | `let name: string = "Alice";` |
| **Template Strings** | Enclosed in backticks. Allows string interpolation and multi-line strings. | ``let greeting = `Hello, ${name}!`;`` |
| **Tagged Templates** | Allows processing of template literals through a tag function. | `highlight\`Hello, $!`` |
| **String to Number** | Converts strings to numbers using `parseInt()`, `parseFloat()`, `+`, or `Number()`. | `let num = + "123";` |

# Numbers

Sure! Here's a detailed explanation of **Number** data types in TypeScript, including concepts like **NaN**, **Min/Max values**, **EPSILON**, **Floating Point Precision**, and **Infinity**, with examples to help you understand each topic.

## 1. Number Data Type in TypeScript

The **`number`** data type in TypeScript is used to represent numeric values, including both integers and floating-point numbers. This is similar to JavaScript's `number` type, which can hold integers, floating-point numbers, or even special values like **`NaN`**, **`Infinity`**, and **`-Infinity`**.

TypeScript provides support for the following types of numbers:

- **Integers**: Whole numbers (e.g., `10`, `-5`, `200`).
- **Floating Point Numbers**: Decimal numbers (e.g., `3.14`, `-0.5`, `100.75`).

### Example:

```
let integer: number = 42;
let floatingPoint: number = 3.14159;
console.log(integer);       // Output: 42
console.log(floatingPoint); // Output: 3.14159
```

## 2. NaN in TypeScript

**`NaN`** stands for **Not-a-Number.** It is a special numeric value that indicates an operation that cannot produce a valid number. For example, dividing zero by zero or attempting to convert a non-numeric string to a number results in `NaN`.

### Characteristics:

- `NaN` is **not equal to any value**, including itself.
- It is a way to represent an undefined or unrepresentable value in numeric calculations.

### Example:

```
let result: number = 0 / 0;  // This will produce NaN
console.log(result);         // Output: NaN

let invalidNumber: number = Number("hello");
console.log(invalidNumber);  // Output: NaN
console.log(isNaN(invalidNumber)); // Output: true
```

In the example above:

- `0 / 0` results in `NaN`.
- `Number("hello")` results in `NaN` since "hello" is not a valid number.

### Checking NaN:

You can check if a value is `NaN` using the `isNaN()` function or the `Number.isNaN()` function.

```
let value: number = NaN;
console.log(isNaN(value));  // Output: true
console.log(Number.isNaN(value));  // Output: true
```

## 3. Min, Max & Safe Values in TypeScript

TypeScript uses the same numeric limits as JavaScript, which are defined in **`Number.MIN_VALUE`**, **`Number.MAX_VALUE`**, and **`Number.MAX_SAFE_INTEGER`**.

- **`Number.MIN_VALUE`**: The smallest positive number (greater than 0).
- **`Number.MAX_VALUE`**: The largest representable number.
- **`Number.MAX_SAFE_INTEGER`**: The largest integer that can be represented safely in JavaScript/TypeScript without precision loss.
- **`Number.MIN_SAFE_INTEGER`**: The smallest safe integer.

### Example:

```
console.log(Number.MIN_VALUE);        // Output: 5e-324 (smallest
positive number)
console.log(Number.MAX_VALUE);        // Output: 1.7976931348623157e+308
(largest number)
console.log(Number.MAX_SAFE_INTEGER); // Output: 9007199254740991
(largest safe integer)
console.log(Number.MIN_SAFE_INTEGER); // Output: -9007199254740991
(smallest safe integer)
```

These constants can be used to check if a number exceeds the representable limits in TypeScript.

## 4. EPSILON & Floating Point Precision

**`Number.EPSILON`** represents the smallest difference between two representable numbers in JavaScript/TypeScript. It is useful for handling precision issues when dealing with floating-point numbers.

Floating-point numbers in JavaScript and TypeScript can sometimes lead to precision issues due to the way they are stored in binary format.

### Example:

```
let a: number = 0.1 + 0.2;
let b: number = 0.3;

console.log(a === b);  // Output: false (due to floating-point precision
issues)
```

This happens because `0.1 + 0.2` does not result exactly in `0.3` due to floating-point precision issues.

To check if two floating-point numbers are effectively the same (considering precision), you can use `Number.EPSILON`:

```
let a: number = 0.1 + 0.2;
let b: number = 0.3;
```

```
console.log(Math.abs(a - b) < Number.EPSILON);  // Output: true
```

Here, we use `Math.abs(a - b) < Number.EPSILON` to check if the difference between `a` and `b` is smaller than the smallest possible difference.

## 5. Infinity in TypeScript

**`Infinity`** represents the mathematical infinity. It is a special numeric value that is larger than all finite numbers. **`-Infinity`** is the counterpart that represents negative infinity.

- **Positive Infinity**: `Infinity`
- **Negative Infinity**: `-Infinity`

You can get `Infinity` by dividing a positive number by zero and `-Infinity` by dividing a negative number by zero.

### Example:

```
let posInfinity: number = 1 / 0;
let negInfinity: number = -1 / 0;

console.log(posInfinity);  // Output: Infinity
console.log(negInfinity);  // Output: -Infinity

console.log(1 / Infinity);  // Output: 0 (smallest representable number)
console.log(-1 / Infinity); // Output: -0 (negative smallest
representable number)
```

## Summary of Key Concepts:

| Concept | Description | Example |
|---|---|---|
| **NaN (Not-a-Number)** | Represents an invalid number or a calculation that cannot produce a valid number (e.g., `0 / 0` or `Number("text")`). | `let result = 0 / 0;` `console.log(result);` |
| **Min & Max Values** | **`Number.MIN_VALUE`** is the smallest positive number. **`Number.MAX_VALUE`** is the largest possible number. | `console.log(Number.MAX_VALUE);` |
| **Safe Values** | **`Number.MAX_SAFE_INTEGER`** is the largest safe integer without precision loss. **`Number.MIN_SAFE_INTEGER`** is the smallest safe integer. | `console.log(Number.MAX_SAFE_INTEGER);` |
| **EPSILON & Precision** | **`Number.EPSILON`** is the smallest difference between two representable numbers. Helps manage floating-point precision errors. | `Math.abs(a - b) < Number.EPSILON` |
| **Infinity** | Represents positive and negative infinity. **`Infinity`** is larger than any number, and **`-Infinity`** is smaller than any number. | `let posInfinity = 1 / 0;` `console.log(posInfinity);` |

# BigInt

## BigInt Data Type in TypeScript

The **BigInt** data type was introduced to JavaScript and TypeScript to allow representation of **arbitrary-precision integers.** While the regular number type in JavaScript/TypeScript can represent integers up to **2^53 - 1** (Number.MAX_SAFE_INTEGER), BigInt can handle integers much larger than this, or even smaller than Number.MIN_SAFE_INTEGER.

## Key Characteristics of BigInt:

- **Arbitrary-Precision**: Unlike the number type which is limited to **double-precision floating point format**, BigInt can handle integers of arbitrary size, which means you can work with very large or very small integers beyond the safe limits of number.
- **Syntax**: A BigInt is created by appending an n to the end of an integer literal. For example, 1234567890123456789012345678901234567890n.

## 1. Creating BigInt

You can create a BigInt in two ways:

1. **Using the n suffix** (recommended):

```
let bigNum: BigInt = 1234567890123456789012345678901234567890n;
console.log(bigNum);  // Output:
1234567890123456789012345678901234567890n
```

2. **Using the BigInt() constructor**:

```
let bigNum: BigInt =
BigInt("1234567890123456789012345678901234567890");
console.log(bigNum);  // Output:
1234567890123456789012345678901234567890n
```

- The BigInt() constructor accepts a string that represents the integer.
- You cannot directly pass a floating-point number or a non-numeric string.

## 2. Operations with BigInt

You can perform basic arithmetic operations such as addition, subtraction, multiplication, division, and exponentiation with BigInt.

**Example:**

```
let num1: BigInt = 1234567890123456789012345678901234567890n;
let num2: BigInt = 9876543210987654321098765432109876543210n;

let sum: BigInt = num1 + num2;
let difference: BigInt = num2 - num1;
let product: BigInt = num1 * num2;
let quotient: BigInt = num2 / num1;

console.log(sum);        // Output:
11111111101111111110111111111011111111100n
console.log(difference); // Output:
8641975320864197532086429753208641975320n
console.log(product);    // Output:
121932631137021795226186647069939035392315418531801907245933536542406477
279031453736177535709258073310000n
console.log(quotient);   // Output: 8n
```

**Note**: Operations with BigInt return values of type BigInt.

## 3. BigInt and Comparison

You can compare BigInt values using comparison operators such as `<`, `>`, `<=`, `>=`, `===`, and `!==`.

**Example:**

```
let num1: BigInt = 1000n;
let num2: BigInt = 500n;
```

```
console.log(num1 > num2);   // Output: true
console.log(num1 === num2); // Output: false
```

## 4. BigInt and Type Safety

`BigInt` is not directly compatible with the `number` type, meaning that you cannot mix `BigInt` and `number` values in arithmetic operations.

For example:

```
let num: number = 10;
let bigNum: BigInt = 1000n;

console.log(num + bigNum); // Error: Cannot mix BigInt and other types
```

To resolve this, you would need to explicitly convert one type to the other, which is not always recommended unless you are sure about your logic.

## BigInt vs. Number

Although `BigInt` offers significant advantages in terms of handling large integers, there are key differences between `BigInt` and `number`:

| Feature | BigInt | Number |
|---|---|---|
| **Size** | Can represent arbitrarily large integers. | Limited to **2^53 - 1** for integers. |
| **Type** | Used specifically for large integers. | Can represent both integers and floating-point numbers. |
| **Precision** | Supports exact precision for large integers. | Limited precision due to floating-point representation (IEEE 754). |
| **Operations** | Supports arithmetic operations but only with other `BigInt`s. | Supports both integers and floating-point numbers, and operations with both. |
| **Syntax** | Created with `n` suffix or `BigInt()` constructor. | Normal numbers written directly (e.g., `123.45`, `100`). |
| **Compatibility** | Cannot be mixed with `number` in operations without explicit conversion. | Can easily be mixed with other `number` types in arithmetic. |
| **Use Case** | Use when working with large integers or arbitrary precision. | Use for typical numbers (integers and floating-point). |

### Example: BigInt vs. Number

```
let bigIntVal: BigInt = 1234567890123456789012345678901234567890n;
let numberVal: number = 1234567890123456789012345678901234567890;

console.log(typeof bigIntVal);  // Output: bigint
console.log(typeof numberVal);  // Output: number

// BigInt and number cannot be mixed directly:
```

```
console.log(bigIntVal + numberVal); // Error: Cannot mix BigInt and
other types.
```

To work with a `BigInt` and `number` together, you can convert one to the other explicitly, but it's generally not recommended because it might cause precision issues or other errors.

## 5. When to Use BigInt

- **Large Numbers**: Use `BigInt` when you need to work with numbers beyond the limit of `Number.MAX_SAFE_INTEGER` (i.e., larger than **2^53 - 1**).
- **Cryptography**: For cryptographic algorithms or applications involving very large prime numbers, `BigInt` is a suitable choice.
- **High Precision Calculations**: If your application requires high precision for very large integers, `BigInt` is ideal.

## 6. Limitations of BigInt

- **Performance**: `BigInt` operations might be slower than `number` operations, especially for smaller values.
- **No Floating-Point Support**: `BigInt` cannot be used with floating-point numbers. If you need to work with both integers and floating-point values, you will need to use `number` for floating-point operations.
- **No Implicit Type Conversion**: TypeScript will not automatically convert between `BigInt` and `number`, which means you'll need to handle the type conversion manually.

## Summary of Key Differences Between BigInt and Number

| **Aspect** | **BigInt** | **Number** |
|---|---|---|
| **Size Limit** | Arbitrary precision | Limited to 2^53 - 1 for integers |
| **Floating Point** | No floating-point support | Supports both integers and floating-point numbers |
| **Use Case** | Large integers, exact precision | General-purpose numbers |
| **Syntax** | `123456789n`, `BigInt()` constructor | `123`, `123.45` |
| **Arithmetic with Types** | Only with other `BigInt` types | Can work with both `number` and `BigInt` (after conversion) |
| **Mixing Types** | Cannot mix with `number` | Can mix with `number` in operations |

In conclusion, **`BigInt`** is an excellent choice for working with large integers or when exact precision is required. However, if you don't need to work with very large numbers and need to handle both integers and floating-point numbers, **`number`** might be more efficient and flexible.

# Boolean

## Boolean Data Type in TypeScript

The **Boolean** data type in TypeScript represents a logical entity that can have one of two possible values: `true` or `false`. These values are used to represent truth values in conditions, such as in control flow (if-else statements, loops, etc.).

### Characteristics of the Boolean Data Type:

- **Type:** Boolean is a primitive type, and it can only have two values: `true` or `false`.
- **Usage:** It is widely used in conditional statements, expressions, and as flags in applications to control behavior or represent binary state.

## 1. Creating Boolean Variables

You can declare and initialize a Boolean variable using either a boolean literal (`true` or `false`) or a Boolean constructor.

### Example 1: Boolean Literal

```
let isActive: boolean = true;
let isCompleted: boolean = false;

console.log(isActive);      // Output: true
console.log(isCompleted);   // Output: false
```

### Example 2: Using the Boolean Constructor

```
let isVerified: boolean = Boolean(1);  // Non-zero values are converted
to true
let isEmpty: boolean = Boolean(0);     // Zero is converted to false

console.log(isVerified); // Output: true
console.log(isEmpty);    // Output: false
```

In the above example:

- `Boolean(1)` converts to `true` because any non-zero number is truthy.
- `Boolean(0)` converts to `false` because `0` is falsy.

## 2. Boolean Expressions

Boolean values are often the result of expressions that compare two values or evaluate conditions. These expressions evaluate to either `true` or `false`.

### Comparison Operators:

- `==` (Equal to)
- `!=` (Not equal to)
- `>` (Greater than)
- `<` (Less than)
- `>=` (Greater than or equal to)
- `<=` (Less than or equal to)

### Example:

```
let a: number = 10;
let b: number = 20;

let result: boolean = a < b;  // Evaluates to true because 10 is less
than 20
console.log(result);          // Output: true
```

## 3. Boolean in Conditional Statements

Boolean values are most commonly used in control flow statements like **if**, **while**, or **for** loops to check whether a condition is true or false.

### Example 1: if Statement

```
let isLoggedIn: boolean = true;

if (isLoggedIn) {
    console.log("Welcome back!");
} else {
    console.log("Please log in.");
}
// Output: Welcome back!
```

### Example 2: while Loop

```
let count: number = 5;

while (count > 0) {
    console.log(count);
    count--;
}
// Output: 5, 4, 3, 2, 1
```

## 4. Boolean Coercion (Truthy and Falsy Values)

In JavaScript (and TypeScript), values other than `false`, `0`, `NaN`, `""` (empty string), `null`, and `undefined` are treated as **truthy.** These can be coerced into `true` when used in a Boolean context.

- **Truthy:** Non-zero numbers, non-empty strings, objects, arrays, etc.
- **Falsy:** `false`, `0`, `""`, `null`, `undefined`, `NaN`.

### Example:

```
let truthyValue: boolean = Boolean("Hello");  // Non-empty string is
truthy
let falsyValue: boolean = Boolean(0);         // Zero is falsy
```

```
console.log(truthyValue);  // Output: true
console.log(falsyValue);   // Output: false
```

## 5. Using Boolean in Logical Operations

You can use logical operators to combine Boolean values:

- `&&` (AND)
- `||` (OR)
- `!` (NOT)

### Example 1: Logical AND (&&)

```
let a: boolean = true;
let b: boolean = false;

let result: boolean = a && b; // Logical AND
console.log(result);  // Output: false
```

In this case, the result is `false` because both conditions need to be `true` for the `AND` operator to return `true`.

### Example 2: Logical OR (||)

```
let a: boolean = true;
let b: boolean = false;

let result: boolean = a || b; // Logical OR
console.log(result);  // Output: true
```

In this case, the result is `true` because at least one condition must be `true` for the `OR` operator to return `true`.

### Example 3: Logical NOT (!)

```
let a: boolean = true;
let b: boolean = false;

console.log(!a);  // Output: false (Negation of `true`)
console.log(!b);  // Output: true (Negation of `false`)
```

## 6. Boolean as Flags

A **Boolean flag** is a variable that is used to indicate the state of something, such as whether a process is complete or whether an action should be performed.

### Example:

```
let isUserAuthenticated: boolean = false;

// Simulate login process
function login(username: string, password: string) {
    if (username === "admin" && password === "password123") {
        isUserAuthenticated = true;  // User is authenticated
    }
}

login("admin", "password123");

if (isUserAuthenticated) {
    console.log("User has logged in successfully.");
} else {
    console.log("Invalid credentials.");
}
// Output: User has logged in successfully.
```

## 7. Boolean Data Type in TypeScript Type Annotations

In TypeScript, you can explicitly specify that a variable is of the `boolean` type using type annotations. This ensures that only `true` or `false` values can be assigned to the variable.

### Example:

```
let isActive: boolean = true;  // Boolean type annotation

isActive = false;  // This is valid
isActive = "yes";   // Error: Type 'string' is not assignable to type
'boolean'.
```

## 8. Boolean and TypeScript's Type Inference

TypeScript automatically infers the type of a variable based on its initial value, so you can omit the type annotation if you assign a `boolean` value.

### Example:

```
let isOpen = true;  // TypeScript infers `isOpen` as a boolean type

isOpen = false;     // Valid
isOpen = 1;         // Error: Type 'number' is not assignable to type
'boolean'.
```

## Summary of Boolean Data Type in TypeScript:

| Feature | Description | Example | | |
|---|---|---|---|---|
| **Boolean Values** | Can only have two values: `true` or `false`. | `let isCompleted: boolean = true;` | | |
| **Boolean Operations** | Logical operations with `&&`, ` | | , and !` operators. | `let result = a && b;` |
| **Conditional Statements** | Used in `if`, `while`, `for`, and other conditional statements to control flow. | `if (isActive) { console.log("Active"); }` | | |
| **Type Annotation** | Explicitly declaring a variable as `boolean`. | `let isLoggedIn: boolean = false;` | | |
| **Boolean Coercion** | Any non-falsy value (e.g., `1`, `"string"`, `[]`) is coerced to `true`, while falsy values (`0`, `""`) become `false`. | `Boolean(0); // false`, `Boolean(1); // true` | | |
| **Flags** | Commonly used as flags to indicate binary states. | `let isUserAuthenticated: boolean = true;` | | |

In conclusion, the **Boolean** data type is crucial for controlling logic in TypeScript, and it provides a simple but powerful mechanism for handling binary conditions and flags. Understanding how to use `true` and `false` effectively, along with logical operators and conditional statements, will help you write efficient and readable TypeScript code.

# Null & Undefined

## Null and Undefined in TypeScript

In TypeScript, `null` and `undefined` are both primitive values, but they represent different concepts. Understanding the difference between them is crucial to writing robust TypeScript code.

### 1. null

- **Definition:** `null` represents an intentional absence of any object value. It is explicitly assigned to a variable to indicate that the variable should not hold any valid data.
- **Usage:** It is typically used to indicate that a variable will eventually hold an object, but at the moment it is explicitly set to no value.

#### Example of null:

```
let person: { name: string, age: number } | null = null;  // person can
either hold an object or be null

if (person === null) {
    console.log("No person found.");
} else {
    console.log(`Name: ${person.name}, Age: ${person.age}`);
}
// Output: No person found.
```

In this example, the `person` variable is explicitly set to `null`, indicating that there is no person data available.

### 2. undefined

- **Definition:** `undefined` represents an uninitialized or absent value. It is the default value for uninitialized variables, function parameters, or object properties.

- **Usage:** It is used by JavaScript/TypeScript when a variable is declared but not assigned a value, or when a function does not return a value.

### Example of undefined:

```
let name: string;

console.log(name);  // Output: undefined
```

In the above example, the variable `name` is declared but not assigned any value, so it holds the value `undefined`.

## 3. Strict Null Checks in TypeScript

TypeScript's **Strict Null Checks** is a setting that provides stricter type checking regarding `null` and `undefined`. When **strict null checks** are enabled (in `tsconfig.json`), the types `null` and `undefined` are not assignable to other types (except `null` and `undefined` themselves, or `any`).

### Why Strict Null Checks?

In the default (non-strict) mode, `null` and `undefined` are treated as valid values for any type, which can lead to potential bugs if you unintentionally access properties or methods on `null` or `undefined` values.

For example:

```
let name: string;
name = null;  // Allowed by default
name = undefined;  // Allowed by default
```

With **strict null checks** enabled, TypeScript will enforce that `null` and `undefined` are treated separately from other types:

### Enabling Strict Null Checks:

In your `tsconfig.json`:

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

With **strict null checks** enabled:

```
let name: string;

name = null;        // Error: Type 'null' is not assignable to type
'string'.
name = undefined;   // Error: Type 'undefined' is not assignable to type
'string'.
```

Now, `null` and `undefined` are not implicitly assignable to variables of type `string`. This helps in catching errors early in the development process and avoids unexpected `null` or `undefined` values causing runtime errors.

## 4. Null vs Undefined

Both `null` and `undefined` represent absence or lack of value, but they are used in different contexts.

| Aspect | null | undefined |
|---|---|---|
| **Meaning** | Explicit absence of a value (empty or unknown). | A variable has been declared but not assigned a value. |
| **Type** | `null` is its own type. | `undefined` is also its own type. |
| **Assigned by default** | **No**, it must be explicitly assigned. | **Yes**, automatically assigned to uninitialized variables. |
| **Usage** | Used intentionally to represent no value. | Used when variables are declared but not initialized. |
| **Example** | `let person = null;` | `let person; console.log(person);` (outputs `undefined`) |

### Example of null and undefined:

```
let item1: string | null = null;      // null represents no value
let item2: string | undefined;        // undefined, uninitialized

console.log(item1);  // Output: null
console.log(item2);  // Output: undefined
```

In this example:

- `item1` is explicitly set to `null`, indicating that no value is assigned.
- `item2` is declared but not assigned any value, so it automatically holds `undefined`.

### Null vs Undefined in Functions:

```
function greet(name: string | null): string {
  if (name === null) {
    return "Hello, Guest!";
  }
  return `Hello, ${name}!`;
}

console.log(greet(null));   // Output: Hello, Guest!
console.log(greet(undefined));   // Output: Error in strict mode,
`undefined` is not assignable to `string | null`
```

In the above code, the function accepts `name` which can either be a string or `null`. However, passing `undefined` is not allowed in **strict mode** because `undefined` is not a valid value for the `string | null` type.

## 5. Difference Between null and undefined in TypeScript

| Aspect | null | undefined |
|---|---|---|
| **Intent** | Represents intentional absence of a value. | Represents uninitialized or absent value. |
| **Assigned Value** | Explicitly assigned `null` to a variable. | Automatically assigned by JavaScript to uninitialized variables. |
| **Default Behavior** | Never automatically assigned, must be set manually. | Automatically assigned to variables that are declared but not initialized. |
| **Type** | `null` is a primitive type. | `undefined` is also a primitive type. |
| **Equality** | `null == undefined` is true, but `null === undefined` is false. | `undefined` is a more general concept. |

## Conclusion:

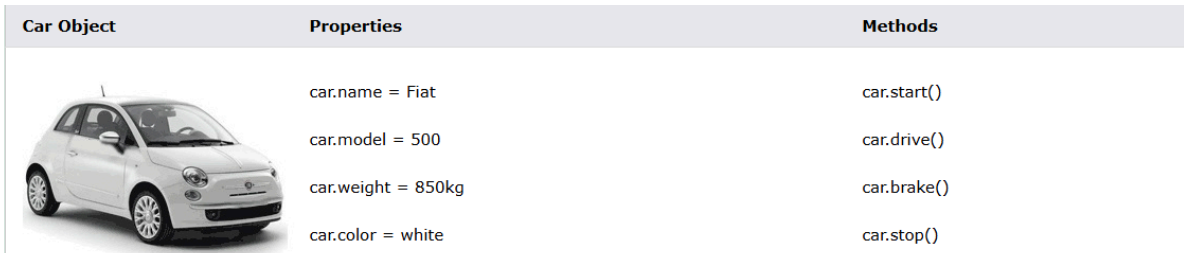
- **`null`** represents an intentional absence of any value and is explicitly assigned.
- **`undefined`** represents an uninitialized value or variable.

- **Strict Null Checks** in TypeScript enforce better handling of `null` and `undefined` by preventing accidental assignment of these values to non-nullable types.
- Enabling **strict null checks** increases code safety, making sure that `null` and `undefined` are not implicitly assigned to variables of other types, which prevents runtime errors due to null dereferencing.

By understanding the differences between `null` and `undefined` and leveraging TypeScript's `strictNullChecks`, developers can write cleaner and more robust code that is less prone to errors.

# Objects in Ts

## 1. Real Life Objects



| Car Object | Properties | Methods |
|---|---|---|
| | car.name = Fiat | car.start() |
| | car.model = 500 | car.drive() |
| | car.weight = 850kg | car.brake() |
| | car.color = white | car.stop() |

image_3.png

Objects in TypeScript are key-value pairs that represent real-world entities. For example, a "car" object might have properties like `make`, `model`, and `year`. These objects can represent almost anything in real life.

**Example:**

```
let car = {
  make: "Toyota",
  model: "Corolla",
  year: 2021
};
```

In this example, `car` is a real-life object with properties like `make`, `model`, and `year`. TypeScript provides type safety for objects by allowing us to specify types for properties.

## 2. Creating a JavaScript Object

In TypeScript, creating objects is similar to JavaScript, but we can also add type annotations for stricter type-checking.

**Example:**

```
// Object with inferred types
let person = {
  name: "John",
  age: 30,
  greet() {
    console.log("Hello " + this.name);
  }
};

// Object with explicit types
let employee: { name: string; age: number; jobTitle: string } = {
  name: "Jane",
  age: 35,
  jobTitle: "Developer"
};
```

Here, the `employee` object is explicitly typed using the `{ name: string; age: number; jobTitle: string }` syntax, which ensures that all properties must match the specified types.

## 3. Object Properties

Object properties can be accessed in two ways: dot notation or bracket notation. TypeScript will infer types based on the values provided unless explicitly declared.

**Example**:

```
let car = {
  make: "Toyota",
  model: "Corolla",
  year: 2021
};

console.log(car.make);  // Accessing property using dot notation
console.log(car["model"]);  // Accessing property using bracket notation
```

You can also create an object with **optional properties** or **read-only properties.**

**Optional Property Example**:

```
let person: { name: string; age: number; jobTitle?: string } = {
  name: "Jane",
  age: 35
};
```

In this example, the jobTitle is optional.

## 4. Accessing Object Properties

In TypeScript, properties can be accessed in the usual ways: dot notation or bracket notation. TypeScript ensures type safety while accessing object properties.

**Example**:

```
let user = {
  name: "Alice",
  age: 28
};

let userName: string = user.name;  // Access property using dot notation
let userAge: number = user["age"];  // Access property using bracket
notation
```

In this case, userName will be inferred as string, and userAge will be inferred as number.

## 5. JavaScript Object Methods

Objects in TypeScript can have methods (functions) as well. Methods can access object properties using this.

**Example**:

```
let calculator = {
  num1: 5,
  num2: 10,
  add() {
    return this.num1 + this.num2;
  },
  multiply() {
    return this.num1 * this.num2;
  }
};

console.log(calculator.add());       // Output: 15
console.log(calculator.multiply());  // Output: 50
```

In this case, the methods `add` and `multiply` are defined in the `calculator` object. TypeScript automatically infers the types of `num1` and `num2` based on the object properties.

# Arrays in TS

An array in TypeScript is a special variable that can hold more than one value. Arrays in TypeScript allow you to work with ordered collections of items of any type.

```
const cars: string[] = ["Saab", "Volvo", "BMW"];
```

## Why Use Arrays?

If you have a list of items (such as car names), you could store them in individual variables like this:

```
let car1: string = "Saab";
let car2: string = "Volvo";
let car3: string = "BMW";
```

However, if you have more items or if the number of items is dynamic (e.g., 300 cars), managing them in individual variables becomes cumbersome.

The solution is an array! An array can hold many values under a single name, and you can access the values by referring to an index number.

## Creating an Array

The general syntax for creating an array is:

```
const arrayName: type[] = [item1, item2, ...];
```

### Example:

```
const cars: string[] = ["Saab", "Volvo", "BMW"];
```

Alternatively, you can use the `Array` constructor:

```
const cars: Array<string> = new Array("Saab", "Volvo", "BMW");
```

⚠ **Note**: It is a common practice to declare arrays with the `const` keyword, as arrays are mutable by default in TypeScript.

## Accessing Array Elements

You access an array element by referring to the **index number** (starting from 0):

```
const cars: string[] = ["Saab", "Volvo", "BMW"];
let car: string = cars[0];  // Access the first element
console.log(car);  // Output: Saab
```

### Accessing the Last Array Element:

```
const fruits: string[] = ["Banana", "Orange", "Apple", "Mango"];
let fruit: string = fruits[fruits.length - 1];
console.log(fruit);  // Output: Mango
```

## Changing an Array Element

You can change an array element by referring to its index:

```
const cars: string[] = ["Saab", "Volvo", "BMW"];
cars[0] = "Tesla";  // Changes the first element
console.log(cars);  // Output: ["Tesla", "Volvo", "BMW"]
```

## Looping Through Array Elements

You can loop through arrays using various methods.

### Using forEach()

```
const cars: string[] = ["Saab", "Volvo", "BMW"];
cars.forEach((item, index) => {
    console.log(`The index: ${index} and the value: ${item}`);
});
```

### Using for loop

```
const fruits: string[] = ["Banana", "Orange", "Apple", "Mango"];
let text: string = "<ul>";
for (let i = 0; i < fruits.length; i++) {
    text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
console.log(text);  // Output: HTML list with fruit names
```

## Arrays are Objects

Arrays in TypeScript are a special type of object. The `typeof` operator will return `"object"` for arrays.

```
const cars: string[] = ["Saab", "Volvo", "BMW"];
console.log(typeof cars);  // Output: "object"
```

## Array Methods

TypeScript arrays come with several built-in methods for manipulating array elements:

### push(): Add an element to the end of an array

```
let a: number[] = [10, 20, 30, 40, 50];
a.push(60);
a.push(70, 80, 90);
console.log(a);  // Output: [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

### unshift(): Add elements to the front of an array

```
let a: number[] = [20, 30, 40];
a.unshift(10, 20);
console.log(a);  // Output: [10, 20, 20, 30, 40]
```

### pop(): Remove elements from the end of an array

```
let a: number[] = [20, 30, 40, 50];
a.pop();
console.log(a);  // Output: [20, 30, 40]
```

### shift(): Remove elements from the beginning of an array

```
let a: number[] = [20, 30, 40, 50];
a.shift();
console.log(a);  // Output: [30, 40, 50]
```

### splice(): Insert and remove elements from an array

```
let a: number[] = [20, 30, 40, 50];
a.splice(1, 3);  // Remove 3 elements starting at index 1
a.splice(1, 0, 3, 4, 5);  // Insert elements 3, 4, and 5 at index 1
console.log(a);  // Output: [20, 3, 4, 5]
```

### slice(): Extract a portion of an array and return it as a new array

```
const a: number[] = [1, 2, 3, 4, 5];
const res: number[] = a.slice(1, 4);  // Extract from index 1 to 3
console.log(res);  // Output: [2, 3, 4]
console.log(a);    // Output: [1, 2, 3, 4, 5] (original array remains
unchanged)
```

### map(): Create a new array by applying a function to each element

```
let a: number[] = [4, 9, 16, 25];
let sub: number[] = a.map(Math.sqrt);
console.log(sub);  // Output: [2, 3, 4, 5]
```

### reduce(): Reduce the array to a single value

```
let a: number[] = [88, 50, 25, 10];
let sub: number = a.reduce((tot, num) => tot - num);
```

```
console.log(sub);  // Output: 3
```

**reverse(): Reverse the order of elements in an array**

```
let a: number[] = [1, 2, 3, 4, 5];
console.log(a);  // Output: [1, 2, 3, 4, 5]
a.reverse();
console.log(a);  // Output: [5, 4, 3, 2, 1]
```

## Arrays with TypeScript Features

- **Typed Arrays**: TypeScript allows you to strongly type arrays to ensure that all elements are of the same type.

```
const cars: string[] = ["Saab", "Volvo", "BMW"];
const numbers: number[] = [1, 2, 3, 4, 5];
```

- **Array with Mixed Types**: If you need an array with elements of different types, you can use a tuple.

```
const carDetails: [string, number] = ["Saab", 2021];
```

- **Readonly Arrays**: If you want to ensure that an array cannot be modified, you can use the `ReadonlyArray<T>` type.

```
const readonlyCars: ReadonlyArray<string> = ["Saab", "Volvo", "BMW"];
// readonlyCars.push("Audi");  // Error: Property 'push' does not exist
on type 'readonly string[]'.
```

## Conclusion

Arrays in TypeScript work similarly to JavaScript arrays, but with strong typing. TypeScript ensures that arrays are more predictable and helps to catch errors at compile-time. You can use array methods such as `push()`, `pop()`, `shift()`, `unshift()`, `map()`, `filter()`, and `reduce()`, among others, while taking advantage of TypeScript's type checking to avoid runtime issues.

This comprehensive guide covers all the essential array operations and demonstrates how they are strongly typed in TypeScript.

# Custom Types

In TypeScript, both **Type Aliases** and **Interfaces** allow you to define custom types. While they can sometimes serve the same purpose, there are important differences in their usage and capabilities.

Let's go through each concept in detail with examples:

## 1. Type Aliases

A **type alias** in TypeScript is a way to define a custom name for any type, whether it's a primitive, union, tuple, or object type.

### Syntax:

```
type AliasName = Type;
```

### Example:

```
type Point = {
  x: number;
  y: number;
};

const point: Point = { x: 10, y: 20 };
```

Here, we created a `Point` type alias that represents an object with two properties: `x` and `y`, both of type `number`. This makes the code more readable.

### Type Aliases for Union and Intersection Types:

You can also use type aliases to create union and intersection types.

**Union Type Example**:

```
type StringOrNumber = string | number;

let value: StringOrNumber;
```

```
value = "Hello";  // Valid
value = 10;       // Valid
```

**Intersection Type Example**:

```
type Name = {
  firstName: string;
  lastName: string;
};

type Age = {
  age: number;
};

type Person = Name & Age;

const person: Person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
};
```

## 2. Interfaces in TypeScript

An **interface** in TypeScript is a way to define a contract for an object or function. It can be used to define the shape of an object, including its properties and their types.

### Syntax:

```
interface InterfaceName {
  propertyName: PropertyType;
}
```

### Example:

```
interface Person {
  name: string;
  age: number;
}

const john: Person = {
  name: "John Doe",
  age: 30,
};
```

In this example, the `Person` interface defines that a person object must have a `name` of type `string` and an `age` of type `number`.

## 3. Interface for Function Types

In TypeScript, you can use interfaces to define the signature of a function, which includes the types of parameters and the return type.

### Syntax:

```
interface FunctionName {
  (param1: Type1, param2: Type2): ReturnType;
}
```

### Example:

```
interface AddFunction {
  (a: number, b: number): number;
}

const add: AddFunction = (a, b) => {
  return a + b;
};

console.log(add(5, 10));  // Output: 15
```

In this example, the `AddFunction` interface defines a function type that accepts two `number` arguments and returns a `number`. The `add` function implements that interface.

## 4. Extending Interfaces

Interfaces in TypeScript can extend other interfaces, meaning you can inherit properties from one interface to create more complex types.

### Syntax:

```
interface ExtendedInterface extends BaseInterface {
  additionalProperty: Type;
}
```

### Example:

```
interface Animal {
  name: string;
  sound: string;
}

interface Dog extends Animal {
  breed: string;
}

const myDog: Dog = {
  name: "Buddy",
  sound: "Woof",
  breed: "Golden Retriever",
};
```

In this example:

- The `Dog` interface extends the `Animal` interface, which means it inherits the `name` and `sound` properties.
- The `Dog` interface also adds a new property, `breed`.

## 5. Interface vs Type Alias

While **interfaces** and **type aliases** can sometimes be used interchangeably, there are differences in their features and usage.

| Feature | Interface | Type Alias |
|---|---|---|
| **Declaration Merging** | Yes, interfaces can merge with other interfaces with the same name. | No, type aliases cannot merge. |
| **Extending Types** | Can extend other interfaces or classes. | Can extend other types using `&` (intersection). |
| **Use Cases** | Typically used to define object shapes and class implementations. | Used for more complex types like unions, tuples, or type compositions. |
| **Flexibility** | Less flexible, mainly used for object shapes and functions. | More flexible, can represent unions, intersections, and primitive types. |

### Example of Declaration Merging in Interface:

```
interface Car {
  make: string;
  model: string;
}

interface Car {
  year: number;
}

const myCar: Car = {
  make: "Toyota",
  model: "Corolla",
  year: 2021,
};
```

In this example, two `Car` interfaces are declared, and TypeScript automatically merges them, so the `Car` interface ends up with all three properties: `make`, `model`, and `year`.

## When to Use Type Alias vs Interface

- **Use Interfaces** when:
  - You want to create a contract for object shapes, especially when you need to use **declaration merging.**
  - You are designing objects or classes that need to adhere to a certain structure.
- **Use Type Aliases** when:
  - You need to define **unions**, **intersections**, or **tuples.**
  - You are working with more complex types (e.g., combining different types using `&`, defining a union type, etc.).

**Example** of union and intersection with type alias:

```
type Animal = {
  name: string;
  sound: string;
};

type Mammal = Animal & { hasFur: boolean };  // Intersection type

type Pet = Animal | { isFriendly: boolean }; // Union type

const dog: Mammal = { name: "Dog", sound: "Bark", hasFur: true };
const fish: Pet = { name: "Fish", sound: "Blub" };
```

## Summary

- **Type Aliases** are useful for defining complex types, such as unions and intersections, or when you want to work with primitive types, tuples, or function signatures.

- **Interfaces** are used to define object structures and class contracts, and they support declaration merging, making them more flexible when it comes to extending types.

Both `interface` and `type alias` are important and can often be used interchangeably, but their use cases and capabilities differ. TypeScript provides these two features to give you more control over your types, ensuring better code clarity and type safety.

# Special Types

In TypeScript, the `Never`, `Void`, and `Unknown` types are special types that help define the behavior and safety of your code. Here's a detailed explanation of each type with examples:

## 1. Never Type

The `never` type represents values that **never occur.** It's often used in situations where a function never completes or always throws an error, such as a function that never returns or an infinite loop. The `never` type is the **opposite** of `void`, because `void` indicates that a function does not return anything, but `never` indicates that the function **doesn't even return** at all.

### Characteristics of never:

- Functions that always throw an exception or have an infinite loop.
- TypeScript uses `never` for exhaustive checks (e.g., in `switch` statements or condition checks).

**Example**: Using `never` in functions that always throw errors:

```
function throwError(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {
    // Infinite loop
  }
}
```

Here:

- `throwError` always throws an error, so it **never** returns anything.

- infiniteLoop runs infinitely and also **never** returns anything.

**Use case**:

- never is used when you expect that a function should **never** complete normally (i.e., it never returns).

## 2. Void Type

The void type is typically used for functions that do not return any value. It's often used as the return type of functions that perform side-effects, like logging or updating UI elements, where no value is returned to the caller.

### Characteristics of void:

- Used for functions that don't return a value.
- Typically seen in function signatures.

**Example**: A function returning void:

```
function logMessage(message: string): void {
  console.log(message);
}

let result = logMessage("Hello, TypeScript!"); // The result will be
`undefined`
console.log(result);  // Output: undefined
```

Here:

- The logMessage function performs an action (logging a message) but **does not return a value.** Hence, its return type is void.
- The result is undefined, because void functions return nothing.

## 3. Unknown Type

The `unknown` type is a safer version of `any`. While `any` allows you to assign any type of value to a variable (thereby bypassing TypeScript's type safety), `unknown` forces you to perform **type-checking** before you can operate on the value.

## Characteristics of unknown:

- You can assign any value to an `unknown` type, but you cannot perform any operations on it until you assert or check its type.
- `unknown` is safer than `any` because it enforces type-checking before performing operations on the value.

**Example**: Using `unknown` type:

```
let value: unknown = 30;

value = "hello";  // Okay to assign a string

// You need to check the type before performing operations
if (typeof value === "string") {
  console.log(value.toUpperCase());  // Works because `value` is
confirmed to be a string
}

value = 100;
// value.toFixed();  // Error: Property 'toFixed' does not exist on type
'unknown'.
```

Here:

- `value` can hold any type, but before using it (e.g., calling `toUpperCase`), TypeScript requires you to check the type using a conditional (`typeof` in this case).
- With `unknown`, TypeScript ensures that you don't accidentally perform operations on a value of an inappropriate type.

**Use case**:

- `unknown` is useful when you want to accept a wide range of possible values but want to maintain safety by requiring the programmer to perform type-checking before using the value.

## Comparison of never, void, and unknown

| Type | Description | Use Case | Example |
|---|---|---|---|
| `never` | Represents a value that **never** occurs. | Functions that **never return** or run into an infinite loop. | `function throwError(message: string): never { throw new Error(message); }` |
| `void` | Represents the absence of a return value (like `undefined`). | Used for functions that perform an action but **do not return** anything. | `function logMessage(message: string): void { console.log(message); }` |
| `unknown` | A safer version of `any`. You must **check** the type first. | Used when you are not sure of the type, but want to ensure safety before performing operations on the value. | `let value: unknown = 30; if (typeof value === "string") { console.log(value.toUpperCase()); }` |

# Typescript Operators

Here's a more **strongly typed** approach to explaining **TypeScript Operators** with proper types and detailed examples for each:

## 1. Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations, such as addition, subtraction, multiplication, etc.

### Types:

- **Operand Types**: `number`
- **Return Type**: `number`

### Example:

```
let x: number = 5;
let y: number = 3;

let sum: number = x + y;         // Addition
let diff: number = x - y;        // Subtraction
let prod: number = x * y;        // Multiplication
let quotient: number = x / y;    // Division
let remainder: number = x % y;   // Modulus (Remainder)
let power: number = x ** y;      // Exponentiation

console.log(sum, diff, prod, quotient, remainder, power);
```

## 2. Unary Plus / Unary Minus Operators

These operators are used to convert a variable to a number and to negate a number.

### Types:

- **Unary Plus (`+`)**: Converts `string` or `boolean` to `number`.

- **Unary Minus (`-`)**: Negates a `number`.

### Example:

```
let str: string = "5";
let num: number = +str;         // Converts string "5" to number 5

let positive: number = 5;
let negative: number = -positive; // Negates value to -5

console.log(num, negative);
```

## 3. Increment/Decrement Operators

These operators are used to increase or decrease a variable's value by `1`.

### Types:

- **Operand Type**: `number`
- **Return Type**: `number`

### Example:

```
let count: number = 0;

count++;  // Post-increment, first returns count, then increments
console.log(count);  // Output: 1

++count;  // Pre-increment, increments count, then returns it
console.log(count);  // Output: 2

count--;  // Post-decrement, first returns count, then decrements
console.log(count);  // Output: 1

--count;  // Pre-decrement, decrements count, then returns it
console.log(count);  // Output: 0
```

## 4. Comparison / Relational Operators

These operators are used to compare two values.

### Types:

- **Operand Types**: `number`, `string`, `boolean`, `object`, etc.
- **Return Type**: `boolean`

### Example:

```
let a: number = 10;
let b: number = 5;

let isGreaterThan: boolean = a > b;   // Greater than
let isLessThan: boolean = a < b;      // Less than
let isEqualTo: boolean = a == b;      // Loose equality
let isNotEqual: boolean = a != b;     // Loose inequality

console.log(isGreaterThan, isLessThan, isEqualTo, isNotEqual);
```

## 5. Equality / Strict Equality Operators

These operators check for value and type equality.

### Types:

- **Operand Types**: `number`, `string`, `boolean`, `object`, etc.
- **Return Type**: `boolean`

### Example:

```
let num1: number = 5;
let num2: string = "5";

let looseEquality: boolean = num1 == num2;   // Loose equality
let strictEquality: boolean = num1 === num2;  // Strict equality

console.log(looseEquality, strictEquality);  // Output: true, false
```

## 6. Ternary Conditional Operator

The ternary operator provides a shorthand for an `if-else` statement.

### Types:

- **Operand Types**: `boolean`
- **Return Type**: Depends on the expressions

### Example:

```
let age: number = 18;

let result: string = age >= 18 ? "Adult" : "Minor";

console.log(result);  // Output: Adult
```

## 7. Logical Operators

Logical operators are used to perform logical operations on boolean values.

### Types:

- **Operand Types**: `boolean`
- **Return Type**: `boolean`

### Example:

```
let x: boolean = true;
let y: boolean = false;

let andResult: boolean = x && y;    // AND
let orResult: boolean = x || y;     // OR
let notResult: boolean = !x;        // NOT

console.log(andResult, orResult, notResult);  // Output: false, true,
false
```

## 8. Bitwise Operators

Bitwise operators operate on binary representations of numbers.

### Types:

- **Operand Types**: `number`
- **Return Type**: `number`

### Example:

```
let a: number = 5;  // Binary: 0101
let b: number = 3;  // Binary: 0011

let andResult: number = a & b;    // AND: 0101 & 0011 = 0001
let orResult: number = a | b;     // OR: 0101 | 0011 = 0111
let xorResult: number = a ^ b;    // XOR: 0101 ^ 0011 = 0110
let notResult: number = ~a;       // NOT: ~0101 = 1010

console.log(andResult, orResult, xorResult, notResult);
```

## 9. Assignment Operators

Assignment operators are used to assign values to variables.

### Types:

- **Operand Types**: `number`, `string`, `boolean`, etc.
- **Return Type**: `number` (or whatever type the left operand is)

### Example:

```
let x: number = 5;

x += 3;  // x = x + 3
x -= 2;  // x = x - 2
x *= 4;  // x = x * 4
x /= 2;  // x = x / 2
x %= 3;  // x = x % 3

console.log(x);  // Output: 2 (5 + 3 - 2 * 4 / 2 % 3)
```

## 10. Nullish Coalescing Operator (??)

The nullish coalescing operator returns the right operand when the left operand is `null` or `undefined`.

### Types:

- **Operand Types**: `null`, `undefined`, or any type.
- **Return Type**: The type of the right operand.

### Example:

```
let name: string | null = null;
let defaultName: string = "Guest";

let userName: string = name ?? defaultName;

console.log(userName);  // Output: Guest (because name is null)
```

## 11. Comma Operator (,)

The comma operator allows multiple expressions to be evaluated from left to right, and the result of the last expression is returned.

### Types:

- **Operand Types**: Any valid expression.
- **Return Type**: The type of the last expression evaluated.

### Example:

```
let a: number = 5;
let b: number = 10;

let result: number = (a++, b++, a + b);

console.log(result);  // Output: 21 (last expression: 6 + 10)
```

## 12. Operator Precedence

Operator precedence defines the order in which operators are evaluated in an expression.

### Example:

```
let result1: number = 2 + 3 * 4; // Multiplication has higher precedence
console.log(result1);  // Output: 14 (2 + (3 * 4))

let result2: number = (2 + 3) * 4; // Parentheses change precedence
console.log(result2);  // Output: 20 ((2 + 3) * 4)
```

## Conclusion

- **Arithmetic Operators**: Perform basic math operations (`+`, `-`, `*`, `/`, `%`).
- **Unary Operators**: Convert values to numbers or negate them (`+`, `-`).
- **Increment/Decrement Operators**: Increase or decrease by 1 (`++`, `--`).
- **Comparison Operators**: Compare values (`>`, `<`, `==`, `===`, etc.).
- **Ternary Operator**: Shortened `if-else` syntax.
- **Logical Operators**: Perform logical operations (`&&`, `||`, `!`).

- **Bitwise Operators**: Operate on binary numbers (`&`, `|`, `^`, `<<`, `>>`).
- **Assignment Operators**: Assign and perform operations (`=`, `+=`, `-=`, etc.).
- **Nullish Coalescing**: Returns the right operand if the left is `null` or `undefined`.
- **Comma Operator**: Evaluates multiple expressions and returns the result of the last one.
- **Operator Precedence**: Determines the order in which operations are evaluated.

Each operator is essential for performing operations on values, and understanding the types involved ensures type safety and proper use of these operators in TypeScript.

# Control Flow Statements in TS

## TypeScript if, else, and else if

Conditional statements are used to perform different actions based on different conditions.

## Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In TypeScript we have the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

### The if Statement

Use the if statement to specify a block of TypeScript code to be executed if a condition is true.

syntax:

```
if (condition) {
  //  block of code to be executed if the condition is true
}
```

example:

```
let greeting: string;
let hour: number = 15;
if (hour < 18) {
  greeting = "Good day";
}
```

### The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is false.

syntax:

```
if (condition) {
  //  block of code to be executed if the condition is true
} else {
  //  block of code to be executed if the condition is false
}
```

example:

```
let greeting: string;
let hour: number = 15;
if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

### The else if Statement

Use the `else if` statement to specify a new condition if the first condition is false.

Syntax:

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if condition1 is false and condition2
is true
} else {
  //  block of code to be executed if condition1 is false and condition2
is false
}
```

Example:

```
let greeting: string;
let time: number = 20;
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

### TypeScript Switch Statement

The `switch` statement is used to perform different actions based on different conditions.

Syntax:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.

- If there is no match, the default code block is executed.

The `getDay()` method returns the weekday as a number between `0` and `6` (Sunday=0, Monday=1, Tuesday=2...).

```
let day: string;
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

### The break Keyword

When TypeScript reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution inside the switch block.

It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

⚠ ***Note**: If you omit the break statement, the next case will be executed even if the evaluation does not match the case.*

### The default Keyword

The `default` keyword specifies the code to run if there is no case match:

The `getDay()` method returns the weekday as a number between 0 and 6.

```
let text: string;
switch (new Date().getDay()) {
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

The result of text will be: `"Today is Sunday"`

## TypeScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value. TypeScript supports different kinds of loops:

- `for` - loops through a block of code a number of times
- `for/in` - loops through the properties of an object
- `for/of` - loops through the values of an iterable object
- `while` - loops through a block of code while a specified condition is true
- `do/while` - also loops through a block of code while a specified condition is true

## The For Loop

The `for` statement creates a loop with 3 optional expressions:

Syntax:

```
for (expression1; expression2; expression3) {
  // code block to be executed
}
```

- `Expression 1` is executed (one time) before the execution of the code block.
- `Expression 2` defines the condition for executing the code block.
- `Expression 3` is executed (every time) after the code block has been executed.

```
let text: string = "";
for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

Right-Angled Triangle (ascending order):

```
let height: number = 5;

for (let row = 1; row <= height; row++) {
    let stars = '';
    for (let column = 1; column <= row; column++) {
        stars += '*';
    }
    console.log(stars);
}
```

Output for `height = 5`:

```
*
**
***
```

```
****
*****
```

Reversed Right-Angled Triangle (descending order):

```
let height: number = 5;

for (let row = height; row >= 1; row--) {
    let stars = '';
    for (let column = 1; column <= row; column++) {
        stars += '*';
    }
    console.log(stars);
}
```

Output for `height = 5`:

```
*****
****
***
**
*
```

## The For In Loop

The TypeScript `for in` statement loops through the properties of an Object:

Syntax:

```
for (key in object) {
  // code block to be executed
}
```

Example:

```
const person = { fname: "John", lname: "Doe", age: 25 };

let text: string = "";
```

```
for (let x in person) {
  text += person[x];
}
```

## For In Over Arrays

The TypeScript `for in` statement can also loop over the properties of an Array:

Syntax:

```
for (variable in array) {
  // code block to be executed
}
```

Example:

```
const numbers: number[] = [45, 4, 9, 16, 25];

let txt: string = "";
for (let x in numbers) {
  console.log(numbers[x]);
  txt += numbers[x];
}
```

## The For Of Loop

The TypeScript `for of` statement loops through the values of an iterable object.

It lets you loop over iterable data structures such as `Arrays`, `Strings`, `Maps`, `NodeLists`, and more:

Syntax:

```
for (variable of iterable) {
  // code block to be executed
}
```

Example:

```
const cars: string[] = ["BMW", "Volvo", "Mini"];

let text: string = "";
for (let x of cars) {
    console.log(x);
    text += x;
}
```

### forEach

The forEach method in TypeScript is used to execute a provided function once for each element in an array (or array-like object). It is commonly used when you need to iterate through an array and perform an operation on each element.

Syntax:

```
array.forEach(function(element, index, array) {
  // Code to be executed for each element
});
```

Example:

```
let numbers: number[] = [1, 2, 3, 4, 5];

numbers.forEach(function(number) {
  console.log(number * 2);
});
```

Output:

```
2
4
6
8
10
```

### While loop

The while loop loops through a block of code as long as a specified condition is true.

Syntax:

```
while (condition) {
  // code block to be executed
}
```

Example:

```
let height: number = 5;
let row: number = 1;

while (row <= height) {
    let stars: string = '';
    let column: number = 1;
    while (column <= row) {
        stars += '*';
        column++;
    }
    console.log(stars);
    row++;
}
```

Output for height = 5:

```
*
**
***
****
*****
```

## The Do While Loop

The do while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax:

```
do {
  // code block to be executed
}
while (condition);
```

Example:

```
let i: number = 0;
let text: string = "";
do {
  text += "The number is " + i;
  i++;
} while (i < 10);
```

# JSON

- JSON stands for JavaScript Object Notation
- JSON is a text format for storing and transporting data
- JSON is "self-describing" and easy to understand

## JSON Example

This example is a JSON string:

```
{"name" : "John", "age" : 30, "car" : null}
```

## JSON Data Types

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- null

⚠ JSON values cannot be one of the following data types:

- a function
- a date

- undefined

- Strings in JSON must be written in double quotes.

```
{"name" : "John"}
```

- Numbers in JSON must be an integer or a floating point.

```
{"age" : 30}
```

- Values in JSON can be objects.

```
{
    "employee" : {
      "name" : "John",
      "age" : 30,
      "city" : "New York"
    }
}
```

- Values in JSON can be arrays.

```
{
"employees" : ["John", "Anna", "Peter"]
}
```

- Values in JSON can be Boolean(true/false).

```
{"sale" : true}
```

- Values in JSON can be null.

```
{"middlename" : null}
```

## JSON Methods

### JSON.parse()

A common use of JSON is to exchange data to/from a web server. When receiving data from a web server, the data is always a string. Parse the data with `JSON.parse()`, and the data becomes a JavaScript object.

#### Example

Imagine we received this text from a web server:

```
'{"name":"John", "age":30, "city":"New York"}'
```

Use the JavaScript function `JSON.parse()` to convert text into a JavaScript object:

```
const obj = JSON.parse('{"name":"John", "age":30, "city":"New York"}');
```

### JSON.stringify()

A common use of JSON is to exchange data to/from a web server. When sending data to a web server, the data has to be a string. You can convert any JavaScript datatype into a string with `JSON.stringify()`.

Imagine we have this object in JavaScript:

```
const obj = {name: "John", age: 30, city: "New York"};
```

Use the JavaScript function `JSON.stringify()` to convert it into a string.

```
const myJSON = JSON.stringify(obj);
```

⚠ The result will be a string following the JSON notation. myJSON is now a string, and ready to be sent to a server:

# Asynchronous TS

Asynchronous programming in TypeScript allows you to perform tasks that take time (like fetching data from a server, reading files, or waiting for user input) without blocking the main thread. While one task is being completed, JavaScript can continue running other code. TypeScript ensures that asynchronous operations are properly typed, improving both safety and readability.

## 1. Callbacks

A **callback** is a function that is passed into another function as an argument and is executed after the task completes. While callbacks were the traditional way to handle asynchronous tasks in JavaScript, they can lead to "callback hell" if you have many nested callbacks.

### Example: Using Callbacks

```
function greetUser(myName: string, displayMessage: (message:string) =>
void): void {
    const greeting: string = `Hello, ${myName}! Welcome to our
application.`;
    displayMessage(greeting);
}

function displayMessage(message: string): void {
    console.log(message);
}

greetUser("Alice", displayMessage);
```

### Explanation:

`greetUser` is a function that takes two parameters:

- `myName`: a string.
- displayMessage as `callback`: a function that takes a string and returns nothing (void).
-

- `displayMessage` is a simple function that logs the message.

## When you call greetUser("Alice", displayMessage), it constructs the greeting and passes it to the callback.

### 2. Promises

A **Promise** is a modern way to handle asynchronous operations. A promise represents the eventual completion (or failure) of an asynchronous operation and allows for more readable code with `.then()` and `.catch()` methods for handling success or failure.

**Example: Using Promises**

```
interface Data {
    id: number;
    name: string;
}

// Sample data
const data: Data[] = [
    { id: 1, name: "John Doe" },
    { id: 2, name: "Jane Smith" },
    { id: 3, name: "Alice Johnson" },
];

// Simulates fetching data from a server
function fetchData(shouldFail: boolean): Promise<Data[]> {
    return new Promise((resolve, reject) => {
        console.log("Starting fetch...");

        setTimeout(() => {
            if (shouldFail) {
                reject(new Error("❌ Failed to fetch data"));  //
Simulate error
            } else {
                resolve(data);  // Return data successfully
            }
        }, 2000);  // Simulate network delay (2 seconds)
    });
}

// Call fetchData with true or false to simulate success/failure
fetchData(false)  // Change to `true` to simulate an error
    .then((result: Data[]) => {
        console.log("✅ Data fetched successfully:");
        result.forEach(item => {
            console.log(`- ID: ${item.id}, Name: ${item.name}`);
        });
    })
    .catch((error: Error) => {
        console.error(`🚫 Error occurred: ${error.message}`);
    })
    .finally(() => {
        console.log("🔚 Fetch operation completed.");
    });
```

**Key Learning Points:**

- fetchData() returns a Promise<Data[] >.
- .then() handles successful resolution.
- .catch() handles errors (rejections).
- .finally() runs no matter what —

**Try It Yourself:**

- Change fetchData(false) to fetchData(true) to test error handling.
- Add a loading message before the call and clear it after finally.

## 3. Async/Await

The async/await syntax allows you to write asynchronous code in a more synchronous-looking style. It is built on top of promises and provides a cleaner and more readable way to handle asynchronous operations.

- `async`: Declares an asynchronous function.
- `await`: Pauses execution of the `async` function until the promise resolves.

**Example: Using Async/Await**

```
// This works in environments that support fetch (e.g., browser or
Node.js with node-fetch or built-in fetch in Node 18+)

// Define an async function to fetch data
async function fetchUser(userId: number): Promise<void> {
    try {
        const response = await
fetch(`https://jsonplaceholder.typicode.com/users/${userId}`);

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const userData = await response.json();
        console.log("User data:", userData);
    } catch (error) {
        console.error("Failed to fetch user:", error);
    }
}

// Call the async function
fetchUser(1);
```

**Explanation:**

- `fetch` is used to make an HTTP request to the API.
- `await` pauses execution until the Promise resolves.
- `try/catch` handles any errors (e.g., network failure or invalid response).
- The function `fetchUser` is `async`, so you can `await` inside it.

### Handling Errors with Async/Await

With async/await, you can use `try/catch` blocks to handle errors more effectively.

```
async function fetchData(): Promise<void> {
    try {
        const data: string = await new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve("Fetched Data");
            }, 2000);
        });
        console.log(data);  // This will log "Fetched Data" after 2
seconds
    } catch (error) {
        console.error("Error:", error);
    }
}

fetchData();
```

**Explanation:**

- In case of an error (e.g., if the promise is rejected), the `catch` block catches the error and logs it.
- This makes error handling with `async/await` much cleaner than using `.catch()` in promises.

### Example: Chaining Multiple Async Operations with Async/Await

You can chain multiple asynchronous tasks using `async/await`. The operations are executed sequentially, and `await` ensures that each promise is resolved before continuing to the next one.

```
async function fetchData(): Promise<void> {
    const data1: string = await new Promise((resolve) => {
        setTimeout(() => resolve("Data 1"), 1000);
    });
    const data2: string = await new Promise((resolve) => {
        setTimeout(() => resolve("Data 2"), 1000);
    });

    console.log(data1, data2);  // This will log "Data 1 Data 2" after 2
seconds
}

fetchData();
```

**Explanation:**

- The first `await` ensures that `data1` is fetched before starting to fetch `data2`.
- The operations are sequential and readable, avoiding nested callbacks.

## TypeScript Types for Asynchronous Operations

In TypeScript, you can define types for asynchronous functions, including specifying the return type as `Promise<T>`.

**Example: Defining Types for Async Functions**

```
function fetchData(): Promise<string> {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Fetched Data");
        }, 2000);
    });
}

async function processData(): Promise<void> {
    const data: string = await fetchData();
    console.log(data);  // This will log "Fetched Data" after 2 seconds
}

processData();
```

In this example:

- The `fetchData()` function is typed to return a `Promise<string>`, meaning it will eventually resolve to a string value.
- The `processData()` function is an `async` function that returns `Promise<void>` and processes the result of `fetchData()`.

## Conclusion

Asynchronous programming in TypeScript allows you to handle operations that take time without blocking the main execution thread. TypeScript enhances JavaScript's async capabilities by adding strong typing, ensuring that your asynchronous operations are safe and predictable. Whether you're using callbacks, promises, or `async/await`, TypeScript ensures that your code is robust and type-safe, even in complex asynchronous scenarios.

- **Callbacks**: Functions passed as arguments and executed after a task completes.
- **Promises**: Represent the completion (or failure) of an asynchronous operation and allow chaining with `.then()` and `.catch()`.
- **Async/Await**: Provides a cleaner syntax for handling asynchronous operations and makes asynchronous code appear more synchronous.

By using **async/await** and **promises**, you can manage complex asynchronous tasks more easily while maintaining readability and type safety in TypeScript.

# Functions in Ts

In TypeScript, functions are defined using the `function` keyword. Type annotations are used to specify types for function parameters and return values, ensuring type safety.

## Basic Syntax

```
function functionName(parameters: type): returnType {
  // code to be executed
}
```

## Example

```
function myFunction(a: number, b: number): number {
  return a * b;
}
```

In this example, both `a` and `b` are explicitly typed as `number`, and the return type is also `number`.

## Function Hoisting

Hoisting is JavaScript's default behavior where declarations are moved to the top of their scope. This applies to both variable and function declarations.

In TypeScript, function declarations are hoisted, allowing you to call the function before its definition.

Example:

```
myFunction(5);

function myFunction(y: number): number {
  return y * y;
}
```

In this case, `myFunction` is called before its definition, but TypeScript ensures that the function is correctly typed and hoisted.

## Function Parameters and Arguments

In TypeScript, function `parameters` are the names listed in the function definition, and function `arguments` are the actual values passed to (and received by) the function.

Syntax:

```
function functionName(parameter1: type, parameter2: type, parameter3: 
type): returnType {
  // code to be executed
}
```

## Default Parameter Values

You can assign default values to function parameters in TypeScript. If a parameter is not passed, the default value will be used.

Example:

```
function myFunction(x: number, y: number = 10): number {
  return x + y;
}

myFunction(5); // y defaults to 10
```

In this example, if `y` is not provided, it will default to `10`.

## Types of Function

### Arrow Functions

Arrow functions provide a concise syntax for writing function expressions. In TypeScript, you can define types for the parameters and the return type of an arrow function.

```
// ES5
let x = function(x: number, y: number): number {
  return x * y;
}

// ES6 (Arrow function)
const x = (x: number, y: number): number => x * y;
```

In the ES6 version, the `function` keyword is omitted, and the return type is inferred or explicitly defined. The parameter types (`x` and `y`) are explicitly typed as `number`.

### Immediately Invoked Function Expressions (IIFE)

In TypeScript, Immediately Invoked Function Expressions (IIFE) can be defined as follows. An IIFE is a function that is invoked immediately after being defined.

Syntax:

```
(function() {
  // statements
})()
```

Example:

```
(function () {
  let x: string = "Hello!!"; // I will invoke myself
})();
```

In this example, the IIFE runs immediately after being defined. The variable `x` is typed as a `string` to maintain type safety.

This version uses TypeScript's strong typing to ensure that function parameters and return types are explicit, which provides better code validation and prevents type-related errors.

# Object Oriented Programing in TS

Object-Oriented Programming (OOP) is a paradigm that uses objects and classes to organize code. TypeScript is a superset of JavaScript that introduces strong typing and better tools for working with OOP concepts.

Let's break down the OOP concepts and how they work in TypeScript.

## 1. Objects and Properties

In TypeScript, an **object** is a collection of key-value pairs. You can define an object using interfaces or classes, and types can be specified for both keys and values.

### Example:

```
class Person {
    // Properties
    name: string;
    age: number;
    gender: string;
    // constructor
    constructor(_name: string, _age: number, _gender: string) {
        this.name = _name;
        this.age = _age;
        this.gender = _gender;
    }
    // methods
    greet():void {
        console.log(`hello , my name is ${this.name} and I'm ${this.age}
years old.`)
    }
}
// Accessing object properties
console.log(person.name); // John
console.log(person.age);  // 30
person.greet();           // Hello, John
```

### Explanation:

- We define an interface `Person` with properties and methods.
- Type annotations ensure `name` is a string, `age` is a number, and `greet` is a method.

## 2. Classes and Constructors

In TypeScript, we can define classes and use the `constructor` method to initialize properties of an object.

### Example: Creating Classes

```
class Person {
    name: string;
    age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    greet(): void {
        console.log(`Hello, my name is ${this.name} and I am ${this.age}
years old.`);
    }
}

// Creating an instance of the class
const john = new Person("John", 30);
john.greet();  // "Hello, my name is John and I am 30 years old."
```

**Explanation:**

- The `constructor` initializes the `name` and `age` properties.
- The `greet()` method can be called on instances of the class.

## 3. Inheritance in TypeScript

Inheritance allows a class (child class) to inherit properties and methods from another class (parent class). TypeScript uses the `extends` keyword for inheritance.

### Example: Inheritance with TypeScript Classes

```
// Parent class
class Animal {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    speak(): void {
        console.log(`${this.name} makes a sound`);
    }
}

// Child class (inherits from Animal)
class Dog extends Animal {
    breed: string;

    constructor(name: string, breed: string) {
        super(name);  // Call the parent class constructor
        this.breed = breed;
    }

    speak(): void {
        console.log(`${this.name} barks`);
    }
}

// Creating instances
const dog = new Dog("Max", "Golden Retriever");
dog.speak();  // "Max barks"
```

**Explanation:**

- Dog extends Animal, inheriting its properties and methods.
- super(name) calls the constructor of the parent class.
- The speak() method is overridden in Dog to provide specific functionality.

### Bank Account Scenario

```
// parent class
class BankAccount {
    constructor(public balance: number, public accountHolder: string) {
    }

    public deposit(amount: number): void {
        if (this.validateBalance(amount)) {
            this.balance += amount   // balance = balance + amount
        }
    }

    public getBalance(): number {
        return this.balance
    }

    public withDraw(amount: number): number | string {
        if (this.canWithDraw(amount)) {
            return this.balance -= amount   // balance = balance -
amount
        } else {
            return `your account balance : ${this.balance} is less than
the amount you want to withdraw : ${amount}`
        }
    }

    private validateBalance(amount: number): boolean {
        return amount > 0;
    }

    private canWithDraw(amountToWithDraw: number): boolean {
        return this.balance > amountToWithDraw
    }

}

// child class
class SavingAccount extends BankAccount {
    public interestRate: number;
    constructor(accountHolder: string, balance: number, _interestRate:
number) {
        super(balance, accountHolder)
        this.interestRate = _interestRate
    }

    public getBalance(): number {
        return this.balance
    }

    public calculateInterest(): void {
        const interest = this.getBalance() * this.interestRate;
        console.log(`interest earned : ${interest}`)
        this.balance += interest
    }
}


// const kevinAccount = new BankAccount(300, "Keren");
// console.log(kevinAccount.getBalance())
// kevinAccount.deposit(1000)
// console.log(kevinAccount.getBalance())
// console.log(kevinAccount.withDraw(1000))
// console.log(kevinAccount.getBalance())

const kerenAccount = new SavingAccount("keren", 300, 0.05)

kerenAccount.deposit(1000)

kerenAccount.calculateInterest()
```

```
console.log(kerenAccount.getBalance())
```

## 4. Access Modifiers: Public, Private, and Protected

Access modifiers are used to control the visibility of class members. In TypeScript, you have three types of access modifiers:

- **Public**: Members are accessible from anywhere.
- **Private**: Members are only accessible within the class.
- **Protected**: Members are accessible within the class and derived classes (subclasses).

### Example:

```
class Person {
    public name: string;
    private age: number;
    protected address: string;

    constructor(name: string, age: number, address: string) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    greet(): void {
        console.log(`Hello, my name is ${this.name}`);
    }

    getAge(): number {
        return this.age;  // Access private variable through a method
    }
}

const person = new Person("John", 30, "123 Main St");
console.log(person.name); // public, can be accessed outside
console.log(person.getAge()); // private, accessed via method

// console.log(person.age); // Error: age is private
```

**Explanation:**

- `name` is `public`, so it can be accessed directly.
- `age` is `private`, so it can only be accessed within the class.
- `address` is `protected`, so it can be accessed within the class and subclasses.

## 5. Getters and Setters

Getters and setters allow you to control access to private properties of a class. They allow you to execute code when accessing or modifying a property.

### Example:

```
class Person {
    private _age: number;

    constructor(age: number) {
        this._age = age;
    }

    get age(): number {
        return this._age;
    }

    set age(value: number) {
        if (value > 0) {
            this._age = value;
        } else {
            console.log("Age must be positive");
        }
    }
}
```

```
const person = new Person(30);
console.log(person.age); // 30 (via getter)
person.age = 35; // 35 (via setter)
console.log(person.age); // 35
person.age = -5; // Error: Age must be positive
```

**Explanation:**

- The `age` property is accessed using the getter and setter.
- The setter checks for valid input before modifying the private `_age` property.

## 6. Encapsulation

Encapsulation refers to bundling the data (properties) and methods (functions) that operate on the data into a single unit (class). It also refers to restricting access to some of an object's components to make the object more secure.

In TypeScript, we can achieve encapsulation with **private** and **protected** members.

### Example:

```
class Person {
    private _name: string;
    private _age: number;

    constructor(name: string, age: number) {
        this._name = name;
        this._age = age;
    }

    getName(): string {
        return this._name;
    }

    getAge(): number {
        return this._age;
    }
    setAge(age: number): void {
        if (age > 0) {
            this._age = age;
        }
    }
}

const person = new Person("John", 30);
console.log(person.getName()); // John
console.log(person.getAge());  // 30
person.setAge(35);
console.log(person.getAge());  // 35
```

**Explanation:**

- The `_name` and `_age` properties are private and not directly accessible from outside the class.
- The `getName()`, `getAge()`, and `setAge()` methods are used to control access to these private properties.

## 7. Polymorphism

Polymorphism is the ability of different classes to respond to the same method in different ways. In TypeScript, polymorphism is often achieved through method overriding.

### Example:

```
class Animal {
    speak(): void {
        console.log("The animal makes a sound");
    }
}

class Dog extends Animal {
    speak(): void {
        console.log("The dog barks");
    }
}

class Cat extends Animal {
    speak(): void {
        console.log("The cat meows");
    }
}

const dog = new Dog();
const cat = new Cat();

dog.speak(); // "The dog barks"
cat.speak(); // "The cat meows"
```

**Explanation:**

- `Dog` and `Cat` override the `speak()` method from `Animal`.
- Despite having the same method name, each class implements its own version of `speak()`.

## 8. Static Methods

Static methods are methods that belong to the class itself, not instances of the class. These methods are useful for utility functions that do not depend on instance data.

### Example:

```
class MathHelper {
    static add(a: number, b: number): number {
        return a + b;
    }

    static multiply(a: number, b: number): number {
        return a * b;
    }
}
```

```
console.log(MathHelper.add(5, 3));      // 8
console.log(MathHelper.multiply(5, 3)); // 15
```

**Explanation:**

- add() and multiply() are **static methods** that can be called directly on the class (MathHelper.add()), not on instances of the class.

# OOP Practise Scenarios

Great — let's elevate the **complexity** of each OOP scenario. Below are **enhanced versions** of the same ideas, introducing deeper interactions, more OOP principles (inheritance, encapsulation, polymorphism, abstraction), and even some design patterns where applicable.

## 🔷 1. Advanced Library Management System

**Description:** Design a system for managing a full-service library, including book lending, user accounts, overdue fines, and digital resources.

### Key Elements:

- **`LibraryItem` (abstract)** → base for `Book`, `DVD`, `EBook`
- **`UserAccount`** → handles login, borrowing history, fine calculation
- **`Borrowable` Interface** → enforces `checkOut()` and `returnItem()` for physical items only
- **`Librarian` vs `Member` (inherit from `User`)**
- **Feature:** Fine is auto-calculated based on return date

> ⚠ 🔹 *Polymorphism* through multiple item types 🔹 *Encapsulation* of fine logic in `UserAccount` 🔹 *Abstraction* using an abstract `LibraryItem` 🔹 *Interface* for borrowable vs digital-only items

## 🔷 2. Advanced E-Commerce System

**Description:** Create a multi-vendor e-commerce platform supporting different product types, user roles (admin, customer, seller), discounts, and order tracking.

### Key Elements:

- **`Product` (abstract)** → extended by `ElectronicProduct`, `ClothingProduct`, `FurnitureProduct`

- `User` → extended by `Admin`, `Seller`, `Customer`
- `Cart` → holds products, calculates total with tax and discount
- `Order` → links to `Shipping`, `Payment`, and `Customer`
- **Strategy Pattern:** Discount strategy applied at checkout

> ⚠ 🔹 *Inheritance* for product and user types 🔹 *Polymorphism* for payment (e.g. card, wallet, COD) 🔹 *Encapsulation* of cart/checkout logic 🔹 *Design Patterns:* Strategy (for discount), Factory (for product creation)

## 🔷 3. University Registration & Learning Management System

**Description:** Build a system that manages course registration, class schedules, grading, and content distribution. Include roles for students, professors, and administrators.

### Key Elements:

- `Person` → base for `Student`, `Professor`, `Admin`
- `Course` → includes lectures, assignments, enrolled students
- `Enrollment` **Class** → maps students to courses with grading logic
- `Assessment` **Interface** → implemented by `Quiz`, `Assignment`, `Project`
- **Notification System** for reminders and feedback

> ⚠ 🔹 *Abstraction* for people and content 🔹 *Polymorphism* in assessments 🔹 *Encapsulation* of grade calculations 🔹 *Composite Pattern* for course content (modules contain lectures, assignments)

## 🔷 4. Ride Sharing System with Dynamic Pricing & Ratings

**Description:** Create a full ride-sharing platform with user authentication, GPS location tracking, ride history, and driver/passenger matching logic.

### Key Elements:

- `User` → parent for `Driver`, `Passenger`
- `Ride` → includes pickup/dropoff, fare calculation, ratings
- **Dynamic Pricing Strategy** based on time of day, traffic
- **Vehicle Management** for drivers
- **Matching Algorithm** → nearest driver selection

> ⚠ 🔹 *Polymorphism* in pricing strategies 🔹 *Encapsulation* of rating and fare logic 🔹 *Strategy Pattern* for fare calculation 🔹 *Factory Pattern* for creating ride instances

## 🔷 5. Intelligent Zoo Management System

**Description:** Develop a smart zoo system with intelligent feeding schedules, sensor-driven habitat monitoring, and animal behavior tracking.

### Key Elements:

- `Animal` **(abstract)** → extended by `Bird`, `Mammal`, `Reptile`
- `Habitat` → includes temperature, feeding schedule, cleanliness
- **Observer Pattern:** Animal health alerts via sensors
- `FeedingStrategy` **Interface** → different feeding logic for herbivores, carnivores, omnivores
- **Zookeeper & Vet Roles** for monitoring

> ⚠ 🔹 *Polymorphism* in animal behavior and feeding 🔹 *Encapsulation* of habitat control 🔹 *Observer Pattern* for real-time alerts 🔹 *Interface Segregation* for

specific animal actions (fly, swim, hibernate)

# Generics

## Generics in TypeScript

Generics allow you to define functions, classes, and interfaces with placeholder types, providing greater flexibility and reusability while maintaining type safety. They enable you to work with any data type without sacrificing the type checking TypeScript offers.

## 1. What are Generics?

Generics allow you to define a function, class, or interface with a "type parameter" that can be replaced with different types. You define the type placeholder (usually with `T` or other identifiers), and TypeScript will infer the appropriate type based on the argument you pass into the generic function or class.

### Basic Syntax for Generics:

```
function functionName<T>(param: T): T {
  return param;
}
```

Here, `T` is a type parameter, which TypeScript replaces with an actual type when the function is called.

### Example - Generic Function:

```
function identity<T>(value: T): T {
  return value;
}

let numberIdentity = identity(10);  // Inferred type is number
let stringIdentity = identity("Hello");  // Inferred type is string

console.log(numberIdentity);  // Output: 10
console.log(stringIdentity);  // Output: Hello
```

In this example, identity is a function that returns the value passed to it, and it can work with any type (T), whether it's a string, number, or any other type.

## 2. Generic Interfaces

You can also use generics with interfaces, which makes it possible to define flexible, reusable types.

### Syntax:

```
interface GenericInterface<T> {
  value: T;
  getValue(): T;
}
```

### Example - Generic Interface:

```
interface Box<T> {
  value: T;
  getValue(): T;
}

const numberBox: Box<number> = {
  value: 42,
  getValue() {
    return this.value;
  },
};

const stringBox: Box<string> = {
  value: "Hello, World!",
  getValue() {
    return this.value;
  },
};
```

```
console.log(numberBox.getValue());  // Output: 42
console.log(stringBox.getValue());  // Output: Hello, World!
```

Here, the `Box` interface can accept a type `T` and be used with different types (e.g., `Box<number>` or `Box<string>`).

## 3. Generic Classes

You can also create **generic classes**, which allow you to work with different types of data while keeping everything strongly typed.

### Syntax:

```
class ClassName<T> {
  property: T;

  constructor(property: T) {
    this.property = property;
  }

  getProperty(): T {
    return this.property;
  }
}
```

### Example - Generic Class:

```
class Container<T> {
  private item: T;

  constructor(item: T) {
    this.item = item;
  }

  getItem(): T {
    return this.item;
  }
}
```

```
const numberContainer = new Container<number>(123);
console.log(numberContainer.getItem());  // Output: 123

const stringContainer = new Container<string>("TypeScript");
console.log(stringContainer.getItem());  // Output: TypeScript
```

In this example, `Container` is a generic class that can store any type of item (`T`). It can work with both numbers and strings, maintaining type safety for each.

## 4. Generic Constraints

While generics are flexible, sometimes you may want to restrict the types that can be passed to a generic function, class, or interface. You can use **generic constraints** to ensure that the generic type `T` extends or satisfies a specific type.

### Syntax:

```
function functionName<T extends SomeType>(param: T): T {
  // code
}
```

In this example, `T` is constrained to types that extend `SomeType`, meaning only objects of that type (or a subclass of that type) can be used with the function.

### Example - Generic Constraint:

```
interface Lengthy {
  length: number;
}

function logLength<T extends Lengthy>(item: T): void {
  console.log(item.length);
}

logLength("Hello");   // Output: 5 (string has a length property)
logLength([1, 2, 3]); // Output: 3 (array has a length property)
```

```
// logLength(10);  // Error: Argument of type 'number' is not assignable
to parameter of type 'Lengthy'.
```

Here, the logLength function is constrained to accept only objects that have a length property, such as strings or arrays.

## 5. Using Multiple Generics

You can also define functions, classes, or interfaces that work with multiple generic types.

### Syntax:

```
function functionName<T, U>(param1: T, param2: U): T {
  // code
}
```

### Example - Multiple Generics:

```
function combine<T, U>(value1: T, value2: U): string {
  return `${value1} and ${value2}`;
}

let combinedString = combine<string, number>("Age", 30);
console.log(combinedString);  // Output: "Age and 30"
```

In this example, combine is a function that takes two parameters, value1 and value2, with different types (T and U), and returns a string.

## 6. Using keyof with Generics

You can also use the keyof keyword with generics to ensure that the type passed as the argument is a valid key of the object type.

### Example - keyof with Generics:

```
interface Person {
  name: string;
  age: number;
}

function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const person = { name: "John", age: 30 };

let personName = getProperty(person, "name");  // type inferred as
string
let personAge = getProperty(person, "age");   // type inferred as number

console.log(personName);  // Output: John
console.log(personAge);   // Output: 30
```

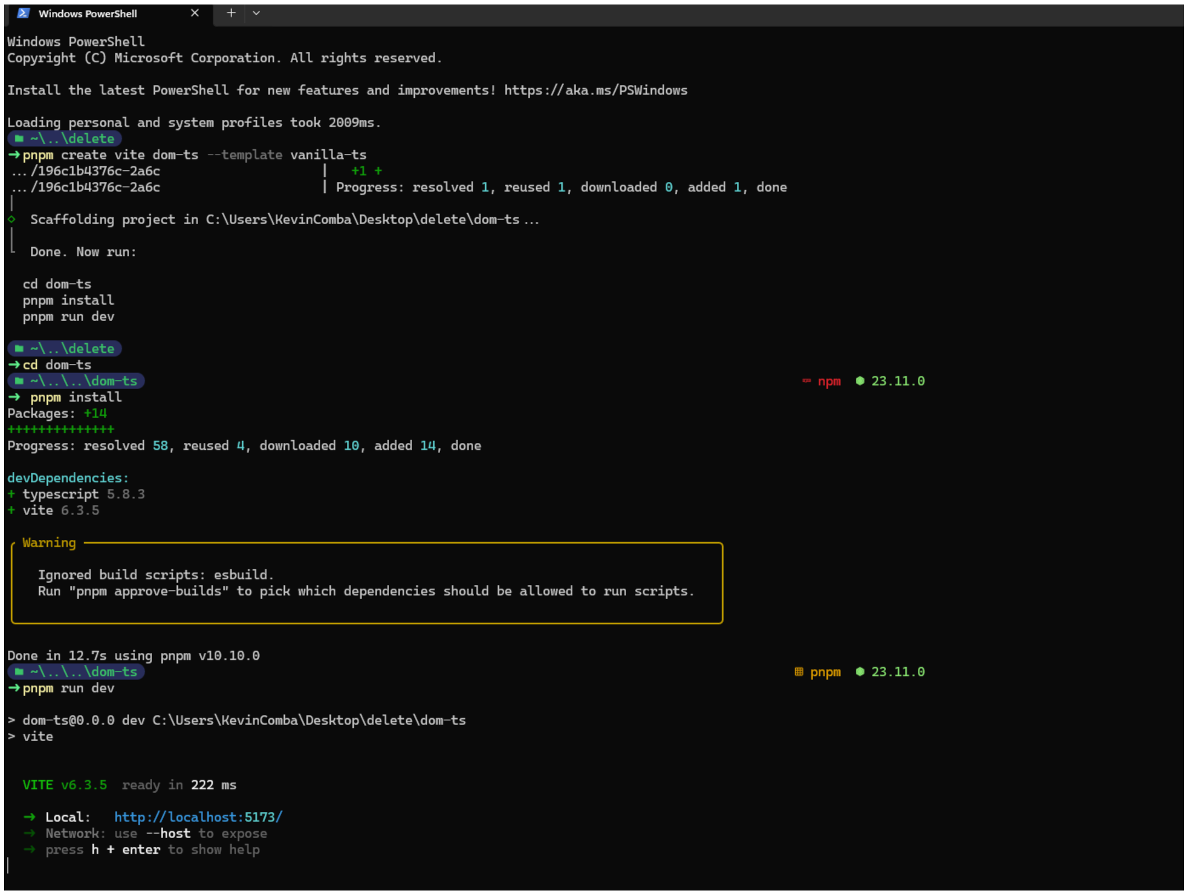
In this example, `K extends keyof T` ensures that `key` must be one of the keys of the object `T`. This ensures that you can't pass an invalid key.

## Summary of Generics and Constraints

| Concept | Description | Example |
|---|---|---|
| **Basic Generics** | Functions, classes, and interfaces that work with any type, providing flexibility and type safety. | `function identity<T>(value: T): T { return value; }` |
| **Generic Constraints** | Restrict the types that can be used in a generic function/class to types that extend a specific type. | `function logLength<T extends Lengthy>(item: T): void {}` |
| **Multiple Generics** | Functions or classes that take more than one generic type parameter. | `function combine<T, U>(value1: T, value2: U): string {}` |
| **`keyof` with Generics** | Ensures that a key passed as a generic parameter is a valid key of the type's properties. | `function getProperty<T, K extends keyof T>(obj: T, key: K)` |

# DOM in TS

## To set up a Ts-Dom project



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

Loading personal and system profiles took 2009ms.
 ~\..\delete
→pnpm create vite dom-ts --template vanilla-ts
.../196c1b4376c-2a6c                     |   +1 +
.../196c1b4376c-2a6c                     | Progress: resolved 1, reused 1, downloaded 0, added 1, done
|
◇  Scaffolding project in C:\Users\KevinComba\Desktop\delete\dom-ts...
|
└  Done. Now run:

  cd dom-ts
  pnpm install
  pnpm run dev

 ~\..\delete
→cd dom-ts
 ~\..\..\dom-ts                                                                                  npm  ● 23.11.0
→ pnpm install
Packages: +14
++++++++++++++
Progress: resolved 58, reused 4, downloaded 10, added 14, done

devDependencies:
+ typescript 5.8.3
+ vite 6.3.5

╭ Warning ──────────────────────────────────────────────────────────────────────────────╮
│                                                                                         │
│   Ignored build scripts: esbuild.                                                       │
│   Run "pnpm approve-builds" to pick which dependencies should be allowed to run scripts.│
│                                                                                         │
╰─────────────────────────────────────────────────────────────────────────────────────────╯

Done in 12.7s using pnpm v10.10.0
 ~\..\..\dom-ts                                                                                  pnpm  ● 23.11.0
→pnpm run dev

> dom-ts@0.0.0 dev C:\Users\KevinComba\Desktop\delete\dom-ts
> vite


  VITE v6.3.5  ready in 222 ms

  → Local:   http://localhost:5173/
  → Network: use --host to expose
  → press h + enter to show help
```

image_9.png

Certainly! Below is the full **TypeScript version** of the **HTML DOM (Document Object Model)** documentation, with both HTML and TypeScript code separated into respective `.html` and `.ts` files.

## The HTML DOM (Document Object Model) in TypeScript

The HTML DOM (Document Object Model) allows JavaScript to interact with web pages by representing them as a tree of objects. TypeScript provides type safety while manipulating these objects, ensuring that operations on HTML elements are correct and predictable.

### What is the DOM?

The **DOM** is a W3C (World Wide Web Consortium) standard that defines the structure of an HTML document as a tree of objects. JavaScript can access and manipulate these objects to modify the page dynamically.

The W3C DOM standard is split into three parts:

- **Core DOM**: A standard model for all document types (HTML, XML, etc.).
- **XML DOM**: A model for XML documents.
- **HTML DOM**: A model specifically for HTML documents.

### HTML DOM Methods in TypeScript

In TypeScript, you can manipulate HTML elements using various DOM methods. TypeScript helps by providing type annotations and ensuring the correct use of DOM methods.

### 1. Finding HTML Elements

| Method | Description | Example |
|---|---|---|
| `document.getElementById(id)` | Find an element by its `id` | `document.getElementById("myId")` |
| `document.getElementsByTagName(name)` | Find elements by tag name | `document.getElementsByTagName("div")` |
| `document.getElementsByClassName(name)` | Find elements by class name | `document.getElementsByClassName("myClass")` |

### Finding HTML Elements by ID, Tag, and Class

**HTML File: `index.html`**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```html
    <title>Finding Elements</title>
</head>
<body>
    <div id="myDiv">Hello, World!</div>
    <div class="myClass">Another div</div>

    <script src="app.js"></script>
</body>
</html>
```

**TypeScript File:** `app.ts`

```typescript
// Finding elements by ID, Tag Name, and Class Name

const elementById: HTMLElement | null =
document.getElementById("myDiv");
if (elementById) {
    console.log(elementById.innerHTML);  // Outputs: Hello, World!
}

const divs: HTMLCollectionOf<HTMLDivElement> =
document.getElementsByTagName("div");
console.log(divs[0].innerHTML);  // Outputs: Hello, World!

const elementsByClass: HTMLCollectionOf<HTMLElement> =
document.getElementsByClassName("myClass");
console.log(elementsByClass[0].innerHTML);  // Outputs: Another div
```

## 2. Changing HTML Elements

| Property/Method | Description | Example |
|---|---|---|
| `element.innerHTML = newContent` | Change the inner HTML of an element | `element.innerHTML = "<p>New content</p>"` |
| `element.attribute = newValue` | Change the value of an attribute of an element | `element.src = "newImage.jpg"` |
| `element.style.property = newStyle` | Change the style of an HTML element | `element.style.backgroundColor = "blue"` |
| `element.setAttribute(attribute, value)` | Change the value of an HTML element's attribute | `element.setAttribute("href", "newLink.html")` |

### Changing HTML Elements

**HTML File: `index.html`**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Changing HTML Elements</title>
</head>
<body>
    <div id="content">Original content</div>
    <img id="image" src="oldImage.jpg" alt="Old Image">

    <script src="app.js"></script>
</body>
</html>
```

**TypeScript File: `app.ts`**

```
// Change innerHTML
const contentDiv: HTMLElement | null =
document.getElementById("content");
if (contentDiv) {
    contentDiv.innerHTML = "New content added!";  // Changes the content
of the div
    contentDiv.style.backgroundColor = "yellow"; // Changes background
color
}

// Change image src
const image: HTMLImageElement | null = document.getElementById("image")
as HTMLImageElement;
if (image) {
    image.src = "newImage.jpg"; // Changes the image source
}

// Change href attribute for a link
const link: HTMLAnchorElement = document.createElement("a");
link.setAttribute("href", "https://newlink.com");
link.innerHTML = "Visit New Link";
document.body.appendChild(link); // Adds the link to the page
```

### 3. Adding and Deleting Elements

| Method | Description | Example |
|---|---|---|
| document.createElement(element) | Create a new HTML element | let newDiv = document.createElement("div") |
| document.removeChild(element) | Remove an HTML element from the DOM | parentElement.removeChild(childElement) |
| document.appendChild(element) | Append an HTML element as a child of another | parentElement.appendChild(newElement) |
| document.replaceChild(new, old) | Replace an HTML element with another | parent.replaceChild(newElement, oldElement) |
| document.write(text) | Write content directly to the HTML output stream | document.write("<p>Hello World</p>") |

### Adding and Deleting Elements

**HTML File: index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Adding and Deleting Elements</title>
</head>
<body>
    <div id="parentDiv">
        <p>This is a parent div.</p>
    </div>

    <script src="app.js"></script>
</body>
</html>
```

**TypeScript File: `app.ts`**

```
// Create a new element
const parentDiv: HTMLElement | null =
document.getElementById("parentDiv");
if (parentDiv) {
    const newElement: HTMLParagraphElement =
document.createElement("p");
    newElement.innerHTML = "This is a new paragraph.";
    parentDiv.appendChild(newElement); // Add the new paragraph to the
div

    // Remove an element
    const paragraphToRemove: HTMLParagraphElement =
parentDiv.getElementsByTagName("p")[0];
    parentDiv.removeChild(paragraphToRemove); // Remove the first
paragraph

    // Replace an element
    const newElementToReplace: HTMLParagraphElement =
document.createElement("p");
    newElementToReplace.innerHTML = "This paragraph replaced the old
one.";
    parentDiv.replaceChild(newElementToReplace, newElement); // Replace
the new paragraph
}

document.write("<h1>Page Loaded</h1>"); // Directly write content to the
page
```

## 4. Adding Event Handlers

| Method | Description | Example |
|---|---|---|
| `document.getElementById(id).onclick` | Add an event handler to an `onclick` event | `document.getElementById("myBtn").onclick = function() { alert("Clicked!"); }` |

### Adding Event Handlers

**HTML File: `index.html`**

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Adding Event Handlers</title>
</head>
<body>
    <button id="myBtn">Click me!</button>

    <script src="app.js"></script>
</body>
</html>
```

**TypeScript File: `app.ts`**

```typescript
// Add an onclick event handler to a button
const button: HTMLElement | null = document.getElementById("myBtn");
if (button) {
    button.onclick = function() {
        alert("Button clicked!");
    };
}
```

# DOM: Movie Project & IndexedDB

## Movie Management System with TypeScript and IndexedDB

This project demonstrates how to build a Movie Management System using TypeScript and the browser's native IndexedDB for persistent storage. The application allows users to add, edit, delete, and rate movies.

## Table of Contents

- Key Learning Points ("Key Learning Points" in "DOM: Movie Project & IndexedDB")
- Next Steps ("Next Steps" in "DOM: Movie Project & IndexedDB")

## Overview

This application showcases a movie management system where users can:

- Add new movies with title, director, and release year
- Rate movies on a scale of 0-5
- Edit existing movie entries
- Delete movies
- View all movies in a list format

The application uses TypeScript for type safety and IndexedDB for local browser storage, making it work offline and persist data between sessions.

## Prerequisites

- Node.js (v14 or higher)
- npm or pnpm (this project uses pnpm)
- Basic knowledge of TypeScript and HTML

## Project Setup

1. Create a new Vite project with TypeScript:

```
pnpm create vite ts-app --template vanilla-ts
cd ts-app
pnpm install
```

2. Install required dependencies:

```
pnpm add -D @types/node
```

## Step-by-Step Guide

### Step 1: Setting up the Project Structure

First, we need to set up the basic project structure:

```
src/
├── counter.ts           (Example file from Vite, can be ignored)
├── database.service.ts  (IndexedDB service)
├── main.ts              (Entry point)
├── movie.interface.ts   (TypeScript interface for Movie)
├── style.css            (Styling)
└── vite-env.d.ts        (Vite environment types)
```

### Step 2: Creating the Movie Interface

Let's define what a Movie object looks like in our system:

```
// movie.interface.ts
export interface Movie {
    id?: number;        // Optional as it's auto-generated
    title: string;
    director: string;
    year: number;
    rating: number;     // Rating from 0-5
}
```

This interface will ensure type safety throughout our application when working with movie data.

### Step 3: Building the Database Service

The `DatabaseService` class handles all interactions with IndexedDB:

```
// database.service.ts
import { Movie } from './movie.interface';

export class DatabaseService {
    private db: IDBDatabase | null = null;
    private readonly DB_NAME = 'MoviesDB';
    private readonly STORE_NAME = 'movies';

    constructor() {
        this.initDatabase();
    }

    public initDatabase(): Promise<void> {
        return new Promise((resolve, reject) => {
            const request = indexedDB.open(this.DB_NAME, 1);

            // Handle errors
            request.onerror = () => reject(request.error);

            // On successful opening
            request.onsuccess = () => {
                this.db = request.result;
                resolve();
            };

            // Create schema if needed (version upgrade)
            request.onupgradeneeded = (event) => {
                const db = (event.target as IDBOpenDBRequest).result;
                if (!db.objectStoreNames.contains(this.STORE_NAME)) {
                    const store = db.createObjectStore(this.STORE_NAME,
{
                        keyPath: 'id',
                        autoIncrement: true
                    });
                    store.createIndex('title', 'title', { unique: false
});
                }
            };
        });
    }

    // CRUD Operations for Movies
    async addMovie(title: string, director: string, year: number):
Promise<void> {
        const movie: Movie = { title, director, year, rating: 0 };
        return new Promise((resolve, reject) => {
            const transaction = this.db!.transaction([this.STORE_NAME],
'readwrite');
            const store = transaction.objectStore(this.STORE_NAME);
            const request = store.add(movie);

            request.onsuccess = () => resolve();
            request.onerror = () => reject(request.error);
        });
    }

    async updateMovie(id: number, title: string, director: string, year:
number): Promise<void> {
        return new Promise((resolve, reject) => {
            const transaction = this.db!.transaction([this.STORE_NAME],
'readwrite');
            const store = transaction.objectStore(this.STORE_NAME);
            const request = store.get(id);

            request.onsuccess = () => {
                const data = request.result;
                if (data) {
                    data.title = title;
                    data.director = director;
                    data.year = year;
                    store.put(data);
                    resolve();
                }
            };
            request.onerror = () => reject(request.error);
        });
    }

    async rateMovie(id: number, rating: number): Promise<void> {
        return new Promise((resolve, reject) => {
            const transaction = this.db!.transaction([this.STORE_NAME],
'readwrite');
            const store = transaction.objectStore(this.STORE_NAME);
            const request = store.get(id);

            request.onsuccess = () => {
                const data = request.result;
                if (data) {
                    data.rating = rating;
                    store.put(data);
                    resolve();
                }
            };
            request.onerror = () => reject(request.error);
        });
    }

    async deleteMovie(id: number): Promise<void> {
        return new Promise((resolve, reject) => {
            const transaction = this.db!.transaction([this.STORE_NAME],
'readwrite');
            const store = transaction.objectStore(this.STORE_NAME);
            const request = store.delete(id);

            request.onsuccess = () => resolve();
            request.onerror = () => reject(request.error);
        });
    }

    async getAllMovies(): Promise<Movie[]> {
        return new Promise((resolve, reject) => {
            const transaction = this.db!.transaction([this.STORE_NAME],
'readonly');
```

```
            const store = transaction.objectStore(this.STORE_NAME);
            const request = store.getAll();

            request.onsuccess = () => resolve(request.result);
            request.onerror = () => reject(request.error);
        });
    }
}
```

The service handles:

- Database initialization and schema creation
- Adding new movies
- Updating existing movies
- Rating movies
- Deleting movies
- Retrieving all movies

### Step 4: Setting up the UI

Now let's set up the main UI in our `main.ts` file:

```
// main.ts
import './style.css'
import { DatabaseService } from './database.service'

// Create async initialization function
async function initializeApp() {
  const db = new DatabaseService();
  // Wait for DB to initialize
  await db.initDatabase();

  document.querySelector<HTMLDivElement>('#app')!.innerHTML = `
    <div class="container">
      <h1>Movie Management System</h1>
      <div class="add-movie-form">
        <h2>Add New Movie</h2>
        <input type="text" id="title" placeholder="Movie Title" />
        <input type="text" id="director" placeholder="Director" />
        <input type="number" id="year" placeholder="Year" />
        <button id="addMovie">Add Movie</button>
      </div>

      <div class="movie-list">
        <h2>Movies</h2>
        <div id="moviesList"></div>
      </div>
    </div>
  `;

  // Display movies function definition and other UI logic follows...
}

// Start the application
initializeApp().catch(console.error);
```

## Step 5: Implementing CRUD Operations

Next, we'll add event listeners and functions to handle the CRUD operations:

```
// Inside initializeApp function in main.ts

// Update the displayMovies function
async function displayMovies() {
  const moviesList = document.querySelector('#moviesList')!;
  const movies = await db.getAllMovies();

  moviesList.innerHTML = movies.map(movie => `
      <div class="movie-card" id="movie-${movie.id}">
          <div class="movie-content">
              <h3>${movie.title}</h3>
              <p>Director: ${movie.director}</p>
              <p>Year: ${movie.year}</p>
              <p>Rating: ${movie.rating}/5</p>
              <div class="movie-actions">
                  <input type="number" min="0" max="5" placeholder="Rate
(0-5)" class="rating-input" data-id="${movie.id}">
                  <button onclick="editMovie(${movie.id})">Edit</button>
                  <button
onclick="deleteMovie(${movie.id})">Delete</button>
              </div>
          </div>
          <div class="edit-form" id="edit-${movie.id}" style="display:
none;">
              <input type="text" class="edit-title"
value="${movie.title}" placeholder="Movie Title">
              <input type="text" class="edit-director"
value="${movie.director}" placeholder="Director">
              <input type="number" class="edit-year"
value="${movie.year}" placeholder="Year">
              <button onclick="saveMovie(${movie.id})">Save</button>
              <button onclick="cancelEdit(${movie.id})">Cancel</button>
          </div>
      </div>
  `).join('');
}

// Add event listeners
document.querySelector('#addMovie')?.addEventListener('click', async ()
=> {
  const title = (document.querySelector('#title') as
HTMLInputElement).value;
  const director = (document.querySelector('#director') as
HTMLInputElement).value;
  const year = parseInt((document.querySelector('#year') as
HTMLInputElement).value);

  if (title && director && year) {
    await db.addMovie(title, director, year);
    await displayMovies();
    // Clear inputs
    (document.querySelector('#title') as HTMLInputElement).value = '';
    (document.querySelector('#director') as HTMLInputElement).value =
'';
    (document.querySelector('#year') as HTMLInputElement).value = '';
  }
});

// Add event listener for rating changes
document.addEventListener('change', async (e) => {
  const target = e.target as HTMLInputElement;
  if (target.classList.contains('rating-input')) {
    const movieId = parseInt(target.dataset.id!);
    const rating = parseFloat(target.value);
    if (rating >= 0 && rating <= 5) {
      await db.rateMovie(movieId, rating);
      await displayMovies();
    }
  }
});

// Add functions for editing, canceling edits, saving changes, and
deleting
window.deleteMovie = async (id: number) => {
  await db.deleteMovie(id);
  await displayMovies();
};

window.editMovie = (id: number) => {
  const movieCard = document.querySelector(`#movie-${id} .movie-
content`)! as HTMLElement;
  const editForm = document.querySelector(`#edit-${id}`)! as
HTMLElement;
  movieCard.style.display = 'none';
  editForm.style.display = 'block';
};

window.cancelEdit = (id: number) => {
  const movieCard = document.querySelector(`#movie-${id} .movie-
content`)! as HTMLElement;
  const editForm = document.querySelector(`#edit-${id}`)! as
HTMLElement;
  movieCard.style.display = 'block';
  editForm.style.display = 'none';
};

window.saveMovie = async (id: number) => {
  const editForm = document.querySelector(`#edit-${id}`)!;
  const title = (editForm.querySelector('.edit-title') as
HTMLInputElement).value;
  const director = (editForm.querySelector('.edit-director') as
HTMLInputElement).value;
  const year = parseInt((editForm.querySelector('.edit-year') as
HTMLInputElement).value);

  if (title && director && year) {
    await db.updateMovie(id, title, director, year);
    await displayMovies();
  }
};

// Display movies immediately after initialization
await displayMovies();

// Type declaration for global functions
declare global {
  interface Window {
    deleteMovie: (id: number) => Promise<void>;
    editMovie: (id: number) => void;
    cancelEdit: (id: number) => void;
    saveMovie: (id: number) => Promise<void>;
  }
}
```

## Step 6: Styling the Application

Add styling to make the application look good:

```css
/* style.css - key parts */
.container {
  max-width: 800px;
  margin: 0 auto;
  padding: 20px;
}

.add-movie-form {
  margin-bottom: 30px;
}

.add-movie-form input {
  margin: 5px;
  padding: 8px;
  border: 1px solid #ccc;
  border-radius: 4px;
}

.movie-card {
  border: 1px solid #ccc;
  padding: 15px;
  margin: 10px 0;
  border-radius: 8px;
}

.movie-actions {
  display: flex;
  gap: 10px;
  margin-top: 10px;
}

.rating-input {
  width: 60px;
  padding: 5px;
}
```

```
.edit-form {
  margin-top: 1rem;
  padding: 1rem;
  border-top: 1px solid #ccc;
}

.edit-form input {
  margin: 0.5rem;
  padding: 0.5rem;
  border: 1px solid #ccc;
  border-radius: 4px;
}
```

## Running the Application

To run the application:

```
pnpm run dev
```

This will start a development server, usually at http://localhost:5173/

## Key Learning Points

1. **TypeScript Integration**: Using interfaces for type safety
2. **IndexedDB**: Using a client-side database for persistent storage
3. **Promises**: Using async/await for asynchronous operations
4. **DOM Manipulation**: Safely working with DOM using TypeScript
5. **Event Handling**: Managing events with proper typing

## Next Steps

- Add search functionality

- Implement sorting (by rating, title, year)
- Add categories or tags for movies
- Implement user authentication
- Add movie poster uploads
- Implement data export/import functionality

Feel free to fork this project and extend it with your own features!

# JS Web APIs

What is Web API?

- API stands for Application Programming Interface.
- A Web API is an application programming interface for the Web.
- A Browser API can extend the functionality of a web browser.
- A Server API can extend the functionality of a web server.

## Browser APIs

All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data. For example, the Geolocation API can return the coordinates of where the browser is located.

### Web Storage API

The Web Storage API is a simple syntax for storing and retrieving data in the browser. It is very easy to use:

- **The localStorage Object**

The localStorage object provides access to a local storage for a particular Web Site. It allows you to store, read, add, modify, and delete data items for that domain.

#### The setItem() Method

The `localStorage.setItem()` method stores a data item in a storage.

```
localStorage.setItem("name", "John Doe");
```

#### The getItem() Method

The `localStorage.getItem()` method retrieves a data item from the storage.

```
localStorage.getItem("name");
```

**removeItem() method**

```
localStorage.removeItem("name");
```

- **The sessionStorage Object**

The sessionStorage object is identical to the localStorage object. The difference is that the sessionStorage object stores data for one session.

> ⚠ The data is deleted when the browser is closed.

**The setItem() Method**

The `sessionStorage.setItem()` method stores a data item in a storage. It takes a name and a value as parameters:

```
sessionStorage.setItem("name", "John Doe");
```

**The getItem() Method**

The `sessionStorage.getItem()` method retrieves a data item from the storage. It takes a name as parameter: