

TYPESCRIPT

“TypeScript is a statically/strongly typed open source programming language that builds on JavaScript, giving you better tooling at any scale.”

Why should you use TypeScript?

- **Static typing.**
 - TypeScript is a superset of JavaScript that adds optional static typing and other features such as classes and modules.
 - TypeScript checks a program for errors before execution, and does so based on the kinds of values, it's a static type checker.
 - Once TypeScript's compiler is done with checking your code, it erases the types to produce the resulting "compiled" code. This means that once your code is compiled, the resulting plain JS code has no type information.
- **Access to latest features(ES6, ES7, etc...), before they become supported by major browsers.**
 - Use cutting edge ECMA features such as `Optional Chaining (?.)` operator without having to worry about browser support.

Why should you use TypeScript?

- Code completion and Intellisense
- Object Oriented Programming
 - Its helps programmers to write code in a object oriented manner if required. Thus helps users to jump into TS
- IDE Support
 - It is super well-supported by text editors including (VS Code, Atom, Sublime, Eclips, etc.)
- Large Community/Adoption
 - TypeScript is made for creating large, complex systems that the modern Web abounds with.

TypeScript's Popularity

TypeScript Is Fastest-Growing Programming Language

"TypeScript's share has almost tripled over the course of 6 years, increasing from 12 percent in 2017 to 34 percent in 2022," said the company's **State of Developer Ecosystem 2022**.

TypeScript passed four languages (C, PHP, C# and C++) over the past six years.

Setting up your environment

- There are many ways in which you can set up a coding environment. Such as:
 - Integrated Development Environment (IDE). Example: **VS Code**, Sublime Text, Atom, etc.
 - Web browser. Example: **Chrome**, Firefox, etc.
 - Online editor (optional). Example: StackBlitz, Replit, etc.

Install Typescript

<https://www.npmjs.com/package/typescript>

```
npm install -g typescript
```

```
tsc --init
```

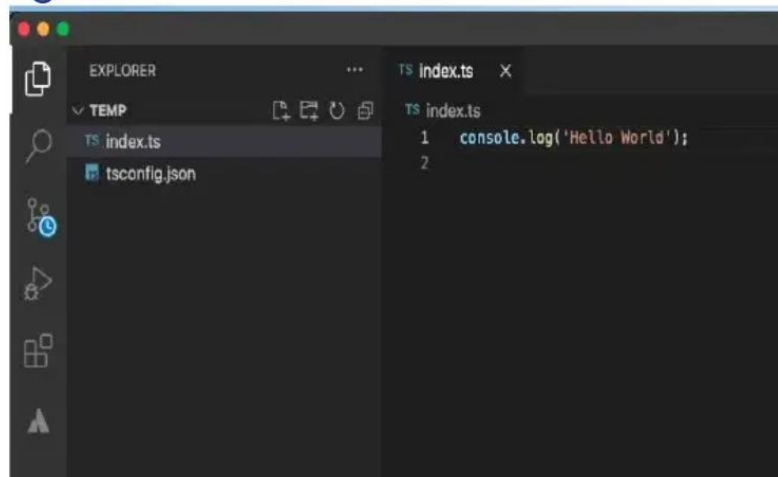
Create a new file and name it whatever you want or better name it `index.ts` just for convention.

Open your file in any text editor like vscode, notepad etc

First Typescript Program

Write the following code:

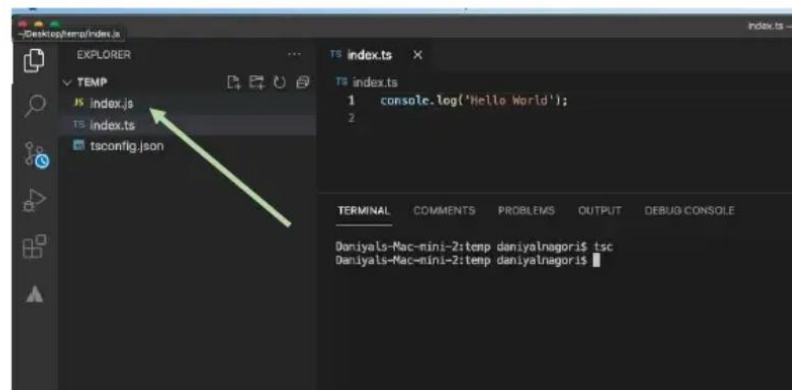
```
console.log("Hello World");
```



Run Typescript Program

```
tsc
```

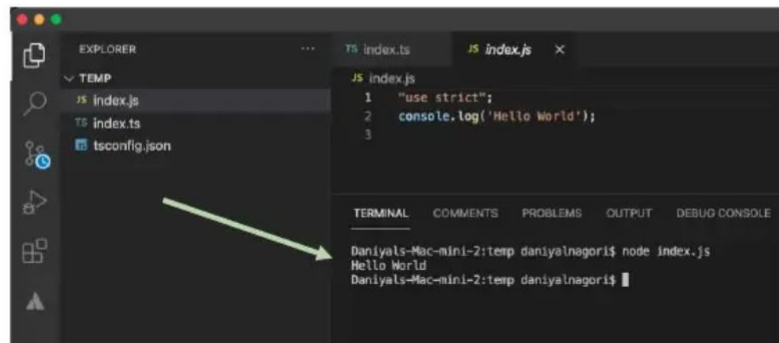
It will compile into Javascript.



Run Typescript Output

```
node index.js
```

You will see the output inside the terminal.



Variables

- Variable means anything that can vary.
- A TypeScript variable is simply **a name of storage location**.
- A variable must have a unique name.

Variables

- Variables are values in your code that can represent different values each time the code runs.
- The first time you create a variable, you declare it. And you need a special word for that: `let` , `var` , Or `const` .
Example: `let firstname = "Ali";`
- The commonly used naming conventions used for **variables** are camel-case.
Example: `let firstName = "Ali";`

Variables Names

- A variable name can't contain any spaces
- A variable name can contain only letters, numbers, dollar signs, and underscores.
- The first character must be a letter, or an underscore (-), or a dollar sign (\$).
- Subsequent characters may be letters, digits, underscores, or dollar signs.
- Numbers are not allowed as the first character of variable.

Type Annotations on Variables

When you declare a variable using `const`, `var`, or `let`, you can optionally add a type annotation to explicitly specify the type of the variable:

```
let myName: string = "Alice";
```

TypeScript doesn't use "types on the left"-style declarations like `int x = 0`; Type annotations will always go after the thing being typed.

In most cases, though, this isn't needed. Wherever possible, TypeScript tries to automatically infer the types in your code.

```
// No type annotation needed -- 'myName' inferred as type 'string'
```

```
let myName = "Alice";
```



Comments

- Single line TypeScript comments **start with two forward slashes (//)**.
- All text after the two forward slashes until the end of a line makes up a comment
- Even when there are forward slashes in the commented text.
- Multi-line Comments
- Multi-line comments start with `/*` and end with `*/`.
- Any text between `/*` and `*/` will be ignored by JavaScript.

Let, Var, Const

var and let are both used for variable declaration in TypeScript but the difference between them is that var is function scoped and let is block scoped. Variable declared by let cannot be redeclared and must be declared before use whereas variables declared with var keyword are hoisted.

const is an augmentation of let in that it prevents re-assignment to a variable.

Dont use var, use let and const

Primitive data types

- String
 - A string is used to store a text value.
Example: `let firstName = "Ali";`
- Number
 - A number is used to store a numeric value.
Example: `let score = 25;`
- Boolean
 - A boolean is used to store a value that is either `true` or `false`.
Example: `let isMarried = false;`
- Undefined
 - An undefined type is either when it has not been defined or it has not been assigned a value.
Example: `let unassigned;`
- Null
 - null is a special value for saying that a variable is empty or has an unknown value.
Example: `let empty = null;`

Template Literals

A new and fast way to deal with strings is **Template Literals or Template String**.

How we were dealing with strings before ?

```
var myName = "daniyal" ;  
var hello = "Hello "+ myName ;  
console.log(hello); //Hello daniyal
```

Template Literals

What is Template literals ?

As we mentioned before , it's a way to deal with strings and specially dynamic strings ; so you don't need to think more about what's the next quote to use single or double.

How to use Template literals

It uses a `backticks` to write string within it.

Analyzing and modifying data types

- You can check the type of a variable by entering **typeof**.

Example:

```
let testVariable = 1;
console.log(typeof testVariable);
```

- The variables in TypeScript cannot change types. Example:

```
let a = 2;
a = "2"; //Error
```

Operators

- Arithmetic operators:

- Addition

Example:

- `let n1 = 1;`
`let n2 = 2;`
`console.log(n1 + n2); // ?`
 - `let str1 = "1";`
`let str2 = "2";`
`console.log(str1 + str2); // ?`

Operators

- Arithmetic operators:

- Subtraction

Example:

- `let n1 = 5;`
`let n2 = 2;`
`console.log(n1 - n2); // ?`

- Multiplication

Example:

- `let n1 = 5;`
`let n2 = 2;`
`console.log(n1 * n2); // ?`

Operators

- Arithmetic operators:

- Division

Example:

- `let n1 = 4;`
`let n2 = 2;`
`console.log(n1 / n2); // ?`

- Exponentiation

Example:

- `let n1 = 2;`
`let n2 = 2;`
`console.log(n1 ** n2); // ?`

Operators

- Arithmetic operators:

- Modulus

Example:

- `let n1 = 10;`
`let n2 = 3;`
`console.log(n1 % n2); // ?`

Operators

- Assignment operators:
 - Assignment operators are used to assign values to variables.

Example:

- ```
let n = 5;
console.log(n); // 5
n += 5;
console.log(n); // 10
n -= 5;
console.log(n); // 5
```

## Operators

- Comparison operators:
  - Comparison operators are used to compare values of variables.

Example:

- ```
let n = 5;
console.log(n == 5); //
console.log(n === 5); //
console.log(n != 5); //
console.log(n > 8); //
console.log(n < 8); //
console.log(n >= 8); //
console.log(n <= 8); //
```

Operators

- Logical operators:

- Logical operators are used to combine multiple conditions in one.

Example:

- ```
let n = 5;
console.log(n >= 5 && n < 10); //
console.log(n > 5 && n < 10); //
console.log(n >= 5 || n < 10); //
console.log(n > 5 || n < 10); //
console.log(!(n < 10)); //
console.log(!(n > 10)); //
```

# Functions

- Functions are the primary means of passing data around in TypeScript. TypeScript allows you to specify the types of both the input and output values of functions.

// Parameter type annotation

```
function greet(name: string) {
 console.log("Hello, " + name.toUpperCase() + "!!");
}
```

# Return Type Annotations

You can also add return type annotations. Return type annotations appear after the parameter list:

```
function getFavoriteNumber(): number {
 return 26;
}
```

Much like variable type annotations, you usually don't need a return type annotation because TypeScript will infer the function's return type based on its return statements.

## Arrow Functions

```
let sum1 = (x: number, y: number): number => {
 return x + y;
}
```

```
sum1(10, 20); //returns 30
```

```
let sum2 = (x: number, y: number) => x + y; //can skip return
```

```
sum2(3, 4); //returns 7
```

## If, Else and Else If 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.

## If, Else and Else If Statements - Examples

- If Example:

- ```
let x = prompt("Where does the Pope live?");
let correctAnswer = "Pakistan";
if (x == correctAnswer) {
    alert("Correct!");
}
```

- else - Example

- ```
let x = prompt("Where does the Pope live?");
let correctAnswer = "Pakistan";
if (x == correctAnswer) {
 alert("Correct!");
} else {
 alert("Wrong!");
}
```

## If, Else and Else If Statements - Examples

- Else if - Example

- ```
let x = prompt("Where does the Pope live?");
let correctAnswer = "Pakistan";
if (x == correctAnswer) {
    alert("Correct!");
} else if (x=="Pakista") {
    alert("Close!");
} else {
    alert("Wrong!");
}
```

If Statement Nested

- JavaScript allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

If Statement Nested - Example

```
// Ticketing system
let country = prompt("Where do you live?")
// Number() function is used to convert the string to number
let age = Number(prompt("What's your age?"))

if (country === "pakistan") {
  if (age >= 18) {
    console.log("Here is your ticket")
  } else {
    console.error("Age restriction")
  }
} else {
  console.log("Invalid country")
}
```

Array

- **The Problem:** Suppose you have five fruits and you want to store them in the variable, But you have to create five variables to store the fruits which is not an efficient approach, what if you have thousands of fruits?
 - let fruit1 = "apple"
let fruit2 = "banana"
let fruit3 = "grapes"
let fruit4 = "strawberry"
let fruit5 = "orange"
- **The Solution:** Here the array comes into play which helps to store multiple data in a single variable.
 - let fruits = ["apple","banana", "orange", "grapes", "strawberry"]

Array - More About Array

- An array is a special variable, which can hold more than one value.
- An array can hold many values under a single name, and you can access the values by referring to an index number.
- In JavaScript, arrays always use numbered indexes.
- Array indexes start with 0.
- Examples:
 - ```
let fruits = ["apple","banana", "orange", "grapes", "strawberry"]
fruits[0] // apple
fruits[3] // grapes
```
  - ```
let x = [1, 2, "daniyal"] // Arrays can store multiple types of data
```

Arrays: Adding and removing elements

- When you work with arrays, it is easy to **remove elements** and **add new elements**. This is what **popping** and **pushing** is.
- The **pop()** method removes the last element from an array:
- The **pop()** method returns the value that was **"popped out"**
- The **push()** method adds a new element to an array (at the end).
- The **push()** method returns the new array length.


Arrays: Adding and removing elements - Examples

- ```
var pets = [];
pets[0] = "dog"; // adds "dog" to an array at 0 index
pets[1] = "cat"; // adds "cat" to an array at index 1

pets.pop(); // removes the last element of an array which is cat in our case
pets.push("parrot"); // adds a new element to an array
```



## Arrays: Removing, inserting, and extracting elements

- Shifting is equivalent to popping, but working on the **first element** instead of the **last**.
  - The **shift()** method removes the first array element and "**shifts**" all other elements to a lower index.
  - The **shift()** method returns the value that was "**shifted out**".
  - The **unshift()** method adds a new element to an array (at the beginning), and "**unshifts**" older elements:
  - The **unshift()** method returns the new array length.
- 

## Arrays: Removing, inserting, and extracting elements - Example

- ```
var pets = [];  
pets[0] = "dog"; // adds "dog" to an array at 0 index  
pets[1] = "cat"; // adds "cat" to an array at index 1  
  
pets.shift(); // removes the first element of an array which is cat in our case  
pets.unshift("parrot"); // adds a new element to an array (at the beginning)
```

Arrays: Removing, inserting, and extracting elements

Splicing and Slicing Arrays

- The **splice()** method adds new items to an array.
 - Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 0, "Lemon", "Kiwi");  
// adds elements to an array at 2nd index  
// deleted 0 elements
```
 - The **slice()** method slices out a piece of an array.
 - Example:

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
const citrus = fruits.slice(1); // [Orange,Lemon,Apple,Mango]
```
 - **Notes:**
The **slice()** method creates a new array.
- 

For Loops

- Loops are handy, if you want to run the same code over and over again, each time with a different value.

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

- From the example above, you can read
- Expression 1 sets a variable before the loop starts (let i = 0).
- Expression 2 defines the condition for the loop to run (i must be less than 5).
- Expression 3 increases a value (i++) each time the code block in the loop has been executed.

For Loops - Examples

- Example 1

```
for (let i = 0; i < 3; i++) {  
  console.log("Hello World")  
}
```

- Example 2

```
for (let i = 0; i < 3; i++) {  
  console.log("Hello World" + i)  
}
```

For Loops - Examples

- Example 3

```
var cleanestCities = ["Karachi", "Lahore", "Islamabad", "Peshawar"];

for (var i = 0; i <= 4; i++) {
  if ("Islamabad" === cleanestCities[i]) {
    console.log("It's one of the cleanest cities");
    break;
  }
}
```

For Loops - Examples

- Example 4

```
var cleanestCities = ["Karachi", "Lahore", "Islamabad", "Peshawar"];

var matchFound = "no";

for (var i = 0; i <= 4; i++){
  if ("Islambad" === cleanestCities[i]) {
    matchFound = "yes";
    alert("It's one of the cleanest cities");
  }
}

if (matchFound === "no") {
  alert("It's not on the list");
}
```

For Loops - Examples

- Example 5

```
var cleanestCities = ["Karachi", "Lahore", "Islamabad", "Peshawar"];

var numElements = cleanestCities.length;
var matchFound = false;

for (var i = 0; i < numElements; i++) {
  if ("Islamabad" === cleanestCities[i]) {
    matchFound = true;
    console.log("It's one of the cleanest cities");
    break;
  }
}
if (matchFound === false) {
  console.log("It's not on the list");
}
```

Nested For Loops - Example

- A nested loop is a loop within a loop.

```
var firstNames = ["BlueRay ", "Upchuck ", "Lojack ", "Gizmo ", "Do-Rag "];
var lastNames = ["Zzz", "Burp", "Dogbone", "Droop"];
var fullNames = [];

for (var i = 0; i < firstNames.length; i++) {
  for (var j = 0; j < lastNames.length; j++) {
    fullNames.push(firstNames[i] + lastNames[j]);
  }
}
```