Variables:

1. **Variable Definition:**
   - A variable refers to something that can change or hold varying values.
   - In TypeScript, a variable is a symbolic name or identifier for a memory location that stores data.
2. **Uniqueness of Variable Names:**
   - Each variable must possess a distinct and unique name within its scope.
   - This name allows the programmer to refer to the stored value within the code.
3. **Dynamic Values:**
   - Variables serve as placeholders for different values during program execution.
   - They enable the code to work with various values at different times.
4. **Variable Declaration:**
   - When you create a variable for the first time, you declare it using keywords like **let**, **var**, or **const**.
   - For example: **let firstName = "Ali";**
5. **Naming Conventions:**
   - Commonly used naming conventions, like camel-case, help improve readability and consistency in variable names.
   - Example: **let firstName = "Ali";**

Variable Names:

1. **Valid Characters in Variable Names:**
   - Variable names cannot contain spaces.
   - They can consist of letters, numbers, dollar signs, and underscores.
2. **Naming Rules:**
   - The initial character of a variable name must be a letter, underscore, or dollar sign.
   - Subsequent characters can include letters, digits, underscores, or dollar signs.
3. **Restrictions on Initial Characters:**
   - Numbers are not permitted as the first character of a variable name.

   Variable names cannot contain spaces within them.

In TypeScript, naming conventions suggest using camelCase, like writing **camelCase**, where the first letter of each word after the first is capitalized.

Type Annotations on Variables:

1. **Defining Variable Types:**
    - When you create a variable using **const**, **var**, or **let**, you have the option to specify the type of data it will hold explicitly.
    - For instance: **let myName: string = "Alice";**
2. **Placement of Type Annotations:**
    - TypeScript doesn't use a style where types are declared on the left side like **int x = 0;**.
    - Type annotations are always written after specifying the variable.
    - Example: **let myName: string = "Alice";**
3. **Type Inference by TypeScript:**
    - TypeScript often infers variable types without explicit annotations.
    - In situations where the type is obvious from the assigned value, specifying the type is unnecessary.
    - For instance: **let myName = "Alice";** automatically infers **myName** as a **string** type.

Primitive data types

in programming are like the different labeled jars you use to organize things. Each jar is designed to hold a specific type of thing. For example:

1. **String**: This jar is for holding words or sentences.

let firstName = "Ali";

2. **Number**: This jar is for holding numbers, like counting marbles or measuring things.

let score = 25;

3. **Boolean**: This jar is like a switch that can be either on or off, representing true or false.

let isMarried = false;

4. **Undefined**: This jar is for things you haven't put anything in yet; it's empty for now.

let unassigned;

5. **Null**: This jar is for when you want to say there's nothing inside intentionally.

let empty = null;

Template literals

Before the introduction of template literals, working with strings involved using single or double quotes and concatenation. For instance:

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

Template literals offer a more dynamic and efficient way to handle strings. They allow for the embedding of expressions directly within the string without relying on concatenation.

So, what are template literals?

They're a modern way to work with strings, especially when dealing with dynamic content. They use backticks (`) to enclose the string. This means you can easily include variables or expressions directly within the string without worrying about using single or double quotes or having to concatenate strings together.

Here's an example of using template literals:

const myName = "daniyal"; const hello = `Hello ${myName}`; console.log(hello); // Hello daniyal

The **${}** syntax inside the backticks allows you to insert variables or even perform operations or calculations directly within the string, making it more flexible and readable.

Analyzing and modifying data types:

1. **Checking Variable Type:** You can figure out what type of information is stored in a variable using the **typeof** operator. It's like asking, "What kind of thing are you?"

let testVariable = 1;

console.log(typeof testVariable);

In this example, **typeof testVariable** will tell you that **testVariable** contains a number.

2. **Type Fixity in TypeScript:** TypeScript is a strict language when it comes to types. Once you declare a variable with a specific type, you can't suddenly change it to a different type. It's like saying, "I'm a number, and I'll always be a number."

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

In this TypeScript example, you initially say **a** is a number. Trying to assign a string to it (**a = "2"**) will cause an error because TypeScript doesn't allow switching types like that. This helps catch potential bugs early in the development process.

Arithmetic operators

Arithmetic operators are the tools in programming that perform mathematical calculations on values. They work much like the standard mathematical operations you'd use in math class.

Here are the main arithmetic operators:

Addition (+): It adds two values together.

let n1 = 1; let n2 = 2; console.log(n1 + n2); // 3 let str1 = "1"; let str2 = "2"; console.log(str1 + str2); // "12" (Note: When used with strings, + concatenates them)

Subtraction (-): It subtracts the second value from the first.

let n1 = 5; let n2 = 2; console.log(n1 - n2); // 3

Multiplication (*): It multiplies two values.

let n1 = 5; let n2 = 2; console.log(n1 * n2); // 10

Division (/): It divides the first value by the second.

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

Exponentiation (**): It raises the first value to the power of the second.

let n1 = 2; let n2 = 2; console.log(n1 ** n2); // 4 (2 raised to the power of 2 is 4)

Modulus (%): It gives the remainder when the first value is divided by the second.

let n1 = 10; let n2 = 3; console.log(n1 % n2); // 1 (10 divided by 3 leaves a remainder of 1)

Assignment operators

Assignment operators are used to both initialize and modify the values stored in variables. They're a shortcut way of performing an operation and assigning the result back to the variable.

+= (Addition assignment): It adds the right operand to the left operand and assigns the result to the left operand.

let n = 5; console.log(n); // 5 n += 5; // Equivalent to n = n + 5; console.log(n); // 10

-= (Subtraction assignment): It subtracts the right operand from the left operand and assigns the result to the left operand.

let n = 10; console.log(n); // 10 n -= 5; // Equivalent to n = n - 5; console.log(n); // 5

There are other assignment operators like *=, /=, and %= which perform multiplication, division, and modulus operations respectively in the same manner.

These operators are handy as they allow you to both perform an operation and update the value of a variable in a concise way.

Comparison operators:

Comparison operators are essential in programming to compare values and determine relationships between them. They return a boolean value (true or false) based on the comparison result.

Here's a breakdown of comparison operators:

== (Equal to): Checks if two values are equal, regardless of their data types (performs type coercion if necessary).

let n = 5; console.log(n == 5); // true

=== (Strict equal to): Compares two values to check if they are both equal in value and data type.

let n = 5; console.log(n === 5); // true

!= (Not equal to): Checks if two values are not equal.

let n = 5; console.log(n != 5); // false

> (Greater than), < (Less than), >= (Greater than or equal to), <= (Less than or equal to): These operators compare the values to check if one is greater, less, greater than or equal to, or less than or equal to the other.

let n = 5; console.log(n > 8); // false console.log(n < 8); // true console.log(n >= 8); // false console.log(n <= 8); // true

These operators are fundamental for implementing logic and decision-making within programs. They help determine conditions and control the flow of the program based on different comparisons.

Logical operators:

are used to combine or modify conditions and produce a single boolean result (either true or false). They're useful for building more complex conditions by combining simpler ones.

Here are explanations for various logical operators:

&& (Logical AND): Returns true if both conditions on either side of the && operator are true.

let n = 5; console.log(n >= 5 && n < 10); // true (Both conditions are true: n is greater than or equal to 5 and less than 10) console.log(n > 5 && n < 10); // false (One condition is false: n is not greater than 5)

|| (Logical OR): Returns true if at least one of the conditions on either side of the || operator is true.

let n = 5; console.log(n >= 5 || n < 10); // true (One condition is true: n is greater than or equal to 5) console.log(n > 5 || n < 10); // true (One condition is true: n is less than 10)

! (Logical NOT): Returns the opposite boolean value of the condition. If the condition is true, ! makes it false, and vice versa.

let n = 5; console.log(!(n < 10)); // false (n is less than 10, ! flips it to false) console.log(!(n > 10)); // true (n is not greater than 10, ! flips it to true)

These operators are incredibly powerful for controlling program flow based on multiple conditions and for creating more intricate decision-making processes within your code.