Agenda:

- Basic Data Types
- String and Templated String
- Arrays
- Tuples
- Enum
- any and void
- null and undefined
- Type Inference
- Type Casting
- Difference between let and var
- Const declaration
- for…of vs for…in statement

Basic Data Types

Basically, like every programming language provides its data types, TypeScript also helps us in defining various data types for JavaScript. Where in JavaScript you don't need to specify the type of the variable because JavaScript is a **dynamic type language.**

- Definition for a variable is a "**named space in the memory**" which stores the value.
- TypeScript follows following rules of JavaScript.
    a) Variable name can be **alphanumeric,** but should not start with a number.
    b) Variable cannot contains white spaces and special characters except underscore **'_'** and **'$'**.

These types can be classified as

1. **Primitive** – boolean, number, string, etc..
2. **Non-Primitive (Object types) –** modules, classes and interfaces.

Syntax for declaring data types in TypeScript is:

```
var <variableName>: <dataType> = <value>;
```

(Or)

```
let <variableName>: <dataType> = <value>;
```

From the above syntax **var** and **let** are keywords for declaring variable, and terms in the angular braces are the placeholder to keep their respective values.

Boolean – true or false, **using 0 and 1 will cause a compilation error.**

```
let isDone: boolean = false;
```

Number – All numeric values are represented by the number type, there aren't separate definitions for integers, floats or others.

```
let decimal: number = 6;
```

String: The text type, just like in vanilla JS strings can be surrounded by 'single quotes' or "double quotes".

```
let fullName: string = 'Sandeep Soni';
let age: number = 37;
```

Templated Strings: Syntactically these are strings that use backticks (i.e. `) instead of single (') or double (") quotes.

```
let sentence: string = `Hello, my name is ${ fullName }. I will be ${ age + 1 } years old next month.`
```

Multiline Strings: You would have needed to *escape the literal newline* using our favorite escape character \, and then put a new line into the string manually \n at the next line

```
let str = "This is Line1 \
        \nThis is Line2 \
        \nThis is Line3"
```

Arrays: Two ways to declare an array:

1. Using [ ]

```
let lst : number[] = [1,2,3,4,5]
```

2. Using Generic Array Type: Array<elemType>

```
let lst : Array<number> = [1,2,3,4,5]
```

TypeScript comes with a **ReadonlyArray**<T> type that is the same as Array<T> with all mutating methods removed, so you can make sure you don't change your arrays after creation:

```
let rolst: ReadonlyArray<number> = lst;
rolst[0] = 12; // error!
rolst.push(5); // error!
rolst.length = 100; // error!
lst = rolst; // error!
```

Tuple:

Tuple types allow you to express an array where the type of a fixed number of elements is known, but **need not be the same**.

For example, you may want to represent a value as a pair of a string and a number:

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error because string should be before number.
```

Enum

```
enum Color {Red = 1, Green, Blue};
let c: Color = Color.Green;
```

any – A variable with this type can have it's value set to a string, number, or ***any***thing else.

It's like "dynamic" in .net.

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

null and undefined:

- By default null and undefined are **subtypes of all other types**. That means you can assign null and undefined to something like number.
- In TypeScript 2.0, null and undefined have their **own types** which allows developers to explicitly express when null/undefined values are acceptable. Now, when something can be either a number or null, you can describe it with the union type number | null (which reads as "number or null").

void – Used on function that don't return anything.

```
function warnUser(): void {
   alert("This is my warning message");
}
```

Type Inference:

If datatype of a variable is not provided, its type will be inferred based on RHS expression type.

```
var msg = 10;
alert(typeof msg); //msg is Number
var msg1 = "A";
alert(typeof msg1); //msg is string
msg = "A" //Compilation Error
```

Type Casting:

a) Angle–bracket syntax.

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

b) as-syntax:

```
let strLength: number = (someValue as string).length;
```

Difference between let and var:

1) **Let** variables in JavaScript are **function scoped** which is not similar to other language variables which are block scoped, whereas **var** variables are global scoped.
2) **Let** variables will not be available once function/ block ends, whereas JavaScript variables are same as they are inside the block or outside the block.

Ex:

```
let message: string = "hello";
if (true) {
    let message: string = "world";
}
console.log(message);
```

output: **"hello"**

```
var message: string = "hello";
if (true) {
    var message: string = "world";
}
console.log(message);
```

output: "**world**"

```
function foo(input: boolean) {
  a++ // error - illegal to use a before it's declared
  let a = 100;
  let a = 100; // error - When let is used, variable cannot be re-declared at same level
  if (input)
  {
    let b1 = a + 1;
    var b2 = a + 1;
    return b;
  }
  a = b1 + a;  // Error: 'b1' doesn't exist here as its not in scope
  a = b2 + a;  // Correct: 'b2' does exist here.
  return a;
}
```

const declaration:

- Const is an addition offered by ES6/TypeScript, this allows variable to be immutable (can't be modified once initialized).
- **const** declaration must be **initialized.**

Ex: Const value = 10;

```
const numLivesForCat = 9;
const kitty = {
  name: "Aurora",
  numLives: numLivesForCat,
}
```

for..of vs. for..in statements:

- Both **for..of** and **for..in** statements iterate over lists; the values iterated on are different though**, for..in** returns a **list of *keys*** on the object being iterated, whereas **for..of** returns a **list of *values*** of the numeric properties of the object being iterated.
- In TypeScript/JavaScript objects have internal properties with themselves. One of the property is **[[Enumerable]].** If **[[Enumerable]]** is set to true it will iterate through object properties only rather through the items or elements.
  - **For..of** is a new way for iterating collections. It's introduced in **ES6**. Earlier you had to use **for or while loop** to iterate through elements of a collection. For **for..of** to work on an collection, the collection must have an **[Symbol.iterator]** property.

Here is an example that demonstrates this distinction:

```
let list = ["A", "B", "C"];
for (let i in list) {
  console.log(i + " "); // 0 1 2
}
for (let i of list) {
  console.log(i + " "); // A B C
}
```