

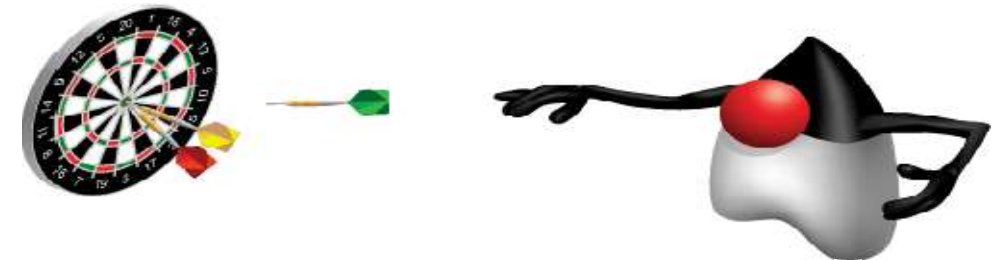


Datatypes And Variables

Objectives

After completing this lesson, you should be able to do the following:

- Typescript Basics
- Datatypes like Strings, Numbers, Boolean
- Null & Undefined
- Objects
- Special Types
- Advanced Types





Typescript Basics

Examining the Grammar of TypeScript

- Typescript is case-sensitive:

```
getElementById() != getElementById()
```

- Escape sequence: `\u`

Example:

```
var x = 15; // \u000A is allowed in comments
```

```
var a = "This line wouldn't be accepted in Java\u000A";
```

Line
feed



Examining the Grammar of TypeScript

➤ Comments:

```
// This is a single-line comment, until the end of line  
/* And here is a multi-line  
   comment, notice that this  
   type of comments cannot be nested */
```

➤ Literals:

```
null  
"True"  
'true'  
true  
15.5
```

Typescript Statements

- A Typescript statement is an instruction to perform a specific action. A Typical Typescript program consists of several such sequences of statements and they control the flow of the program.

```
1
2 //statement 1 create a variable
3 var message;
4
5 //statement 2 assigns "Hello World" to message variable
6 message="hello world";
7
8 //statmenent 3 print out console log
9 console.log(message);
10
```

- A single statement may span multiple lines.
- Or you can write multiple statements in a single line, provided you separate each statement by a semicolon.

Examining the Grammar of JavaScript

; the semicolon is used to indicate the end of a statement.

Automatic Semicolon Insertion

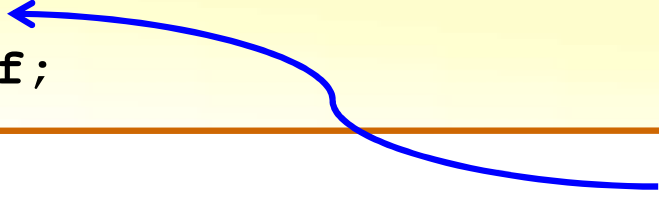
```
var x = 15.5  
var y = "Total";
```

semicolon is
inserted



```
a = b + c  
(d + e) = f;
```

semicolon is NOT
inserted



But, they are required if you have multiple statements in a line

```
var message ; message="hello world" ; console.log(message) ;
```

TypeScript Expressions

- Expressions are units of code that produces value. They can appear anywhere in the code. They can be part of the function arguments or right side of an assignment, etc

```
1
2 5+7      //This is an arithmetic expression that evaluates to 12
3 I++      //Arithmetic expression
4 'Something' //Primary Expressions
5 a && b    // Logical Expressions
6
```

- The expressions can be of various types Arithmetic, String, Primary, Array & object Initialisation, Logical, Left-Hand side. Property access, object Creation, Function definition, invocation, etc.

Whitespace and Line Breaks

- You can add spaces, tabs, and newline characters anywhere in the Typescript Program.
- The Compiler will ignore them.
- You can use them to indent your code so that it very easily readable.



Typescript Datatypes

Static Typing

- Typescript Types (or data types) bring static type checking into the dynamic world of Javascript.
- A very distinctive feature of TypeScript is the support of static typing.
- This means that you can declare the types of variables, and the compiler will make sure that they aren't assigned the wrong types of values.
- If type declarations are omitted, they will be inferred automatically from your code.

- A Small example. Any variable, function argument or return value can have its type defined on initialization:

```
var breakfastItem: string = ' Masala Dosa ', // String
    calories: number = 150, // Numeric
    tasty: boolean = true; // Boolean

// Alternatively, you can omit the type declaration:
// var breakfastItem = ' Masala Dosa ';
// The function expects a string and an integer.
// It doesn't return anything so the type of the function itself
// is void.

function speak(food: string, energy: number): void {
    console.log("Our " + food + " has " + energy + "calories.");
}

speak(breakfastItem, calories);
```

- Because TypeScript is compiled to JavaScript, and the latter has no idea what types are, they are completely removed:

```
// JavaScript code from the above TS example.  
  
    var breakfastItem = ' Masala Dosa ',  
        calories = 150,  
        tasty = true;  
  
function speak(food, energy)  
{  
    console.log("Our " + food + " has " + energy + "  
    calories.");  
}  
  
speak(breakfastItem, calories);
```

- However, if we try to do something illegal, on compilation tsc will warn us that there is an error in our code. For example:

```
// The given type is boolean, the provided value is a string.  
var tasty: boolean = "I haven't tried it yet";
```

```
main.ts(1,5): error TS2322: Type 'string' is not assignable  
to type 'boolean'.
```

- It will also warn us if we pass the wrong argument to a function:

```
function speak(food: string, energy: number): void
{
    console.log("Our " + food + " has " + energy + "
    calories.");
}
// Arguments don't match the function parameters.
speak("trippie cheesburger", "a ton of");
```

```
main.ts(5,30): error TS2345: Argument of type 'string' is not
    assignable to parameter of type 'number'.
```

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.
- The code declares two numbers and adds them using the + operator. The runtime correctly identifies the data type as a number and runs the arithmetic addition on them.

```
1  
2 let num1=10  
3 let num2=10  
4  
5 console.log(num1+num2) //20      Numbers are added  
6
```


- The another code is very similar to the previous code. Here instead of adding two numbers, we add two strings using the + operator. Here interpreter correctly identifies the data type as a string and hence joins them.

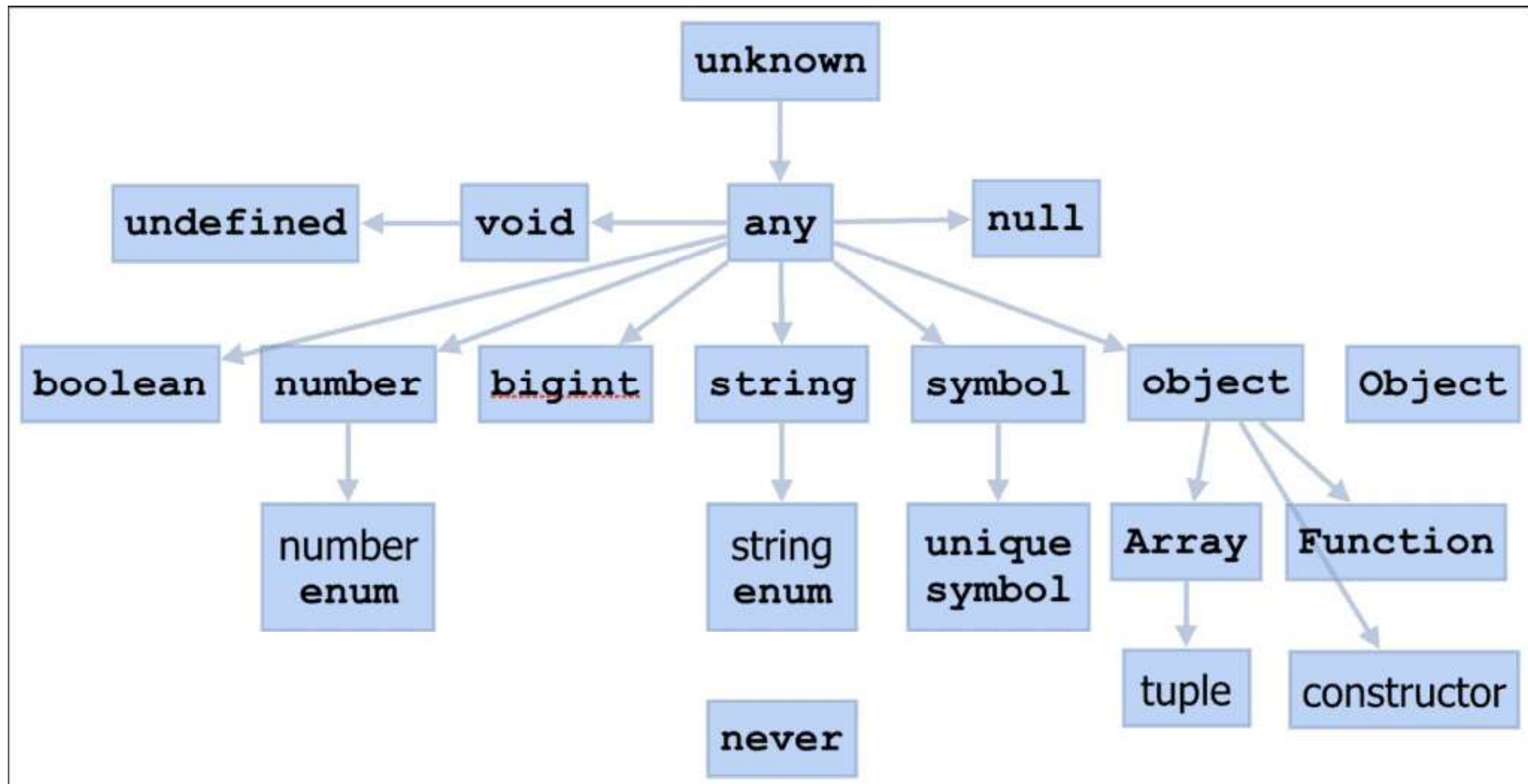
```
1  
2 let str1="Hello"  
3 let str2="world"  
4 console.log(str1+ str2) //HelloWorld    Strings are joined  
5
```

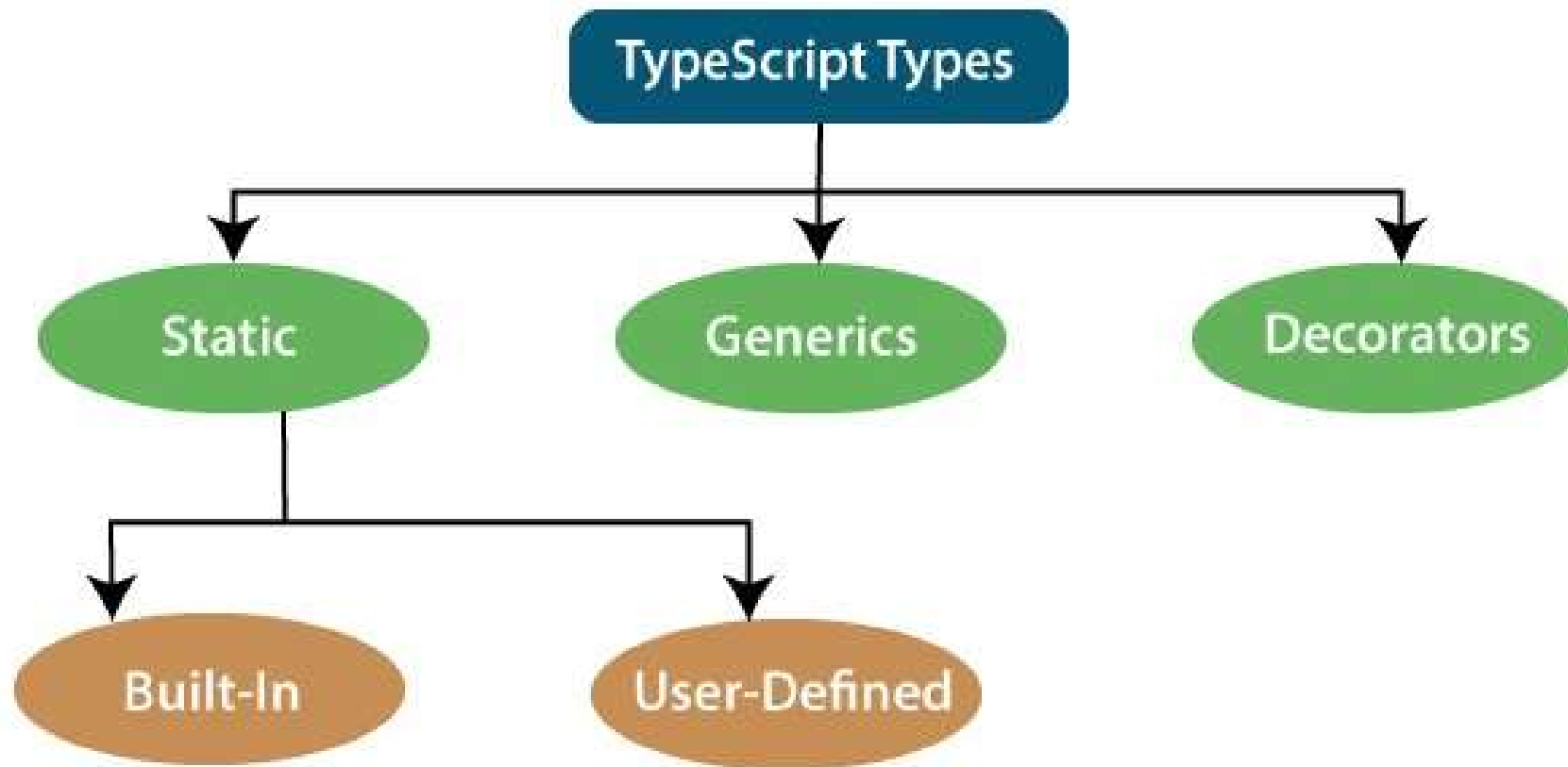
- How does the interpreter know when to perform the addition and when to join them?. By examining the data type. If both the variables are numbers then the interpreter adds them. If any one of them is a string, then it joins them.
- Hence it is very important for the compiler or interpreter to identify the data type of the variable. Otherwise, it may end up joining numbers and adding strings, etc.

Typescript Data Types

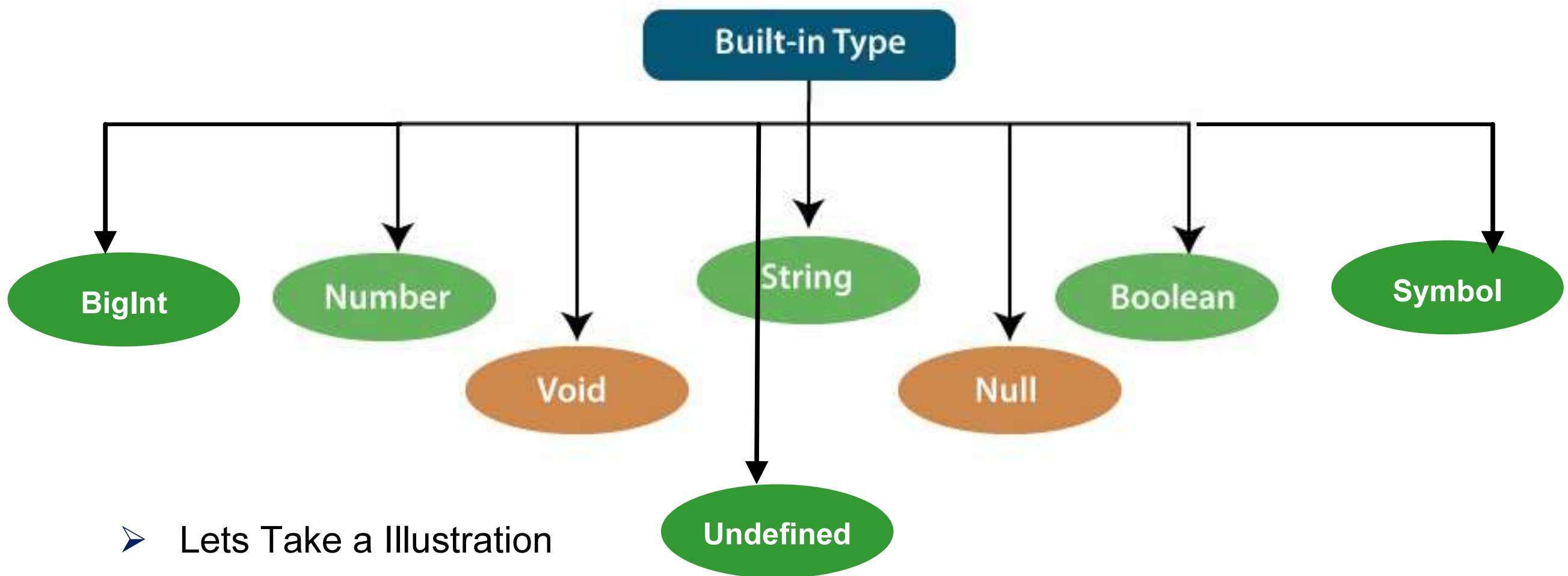
- **JavaScript** has eight data types. Seven primitive types and one object Data type. The primitive types are number, string, boolean, bigint, symbol, undefined, and null. Everything else is an object in JavaScript.
- The **TypeScript** Type System supports all of them and also brings its own special 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

Datatypes





Built-in or Primitive Type



➤ Lets Take a Illustration

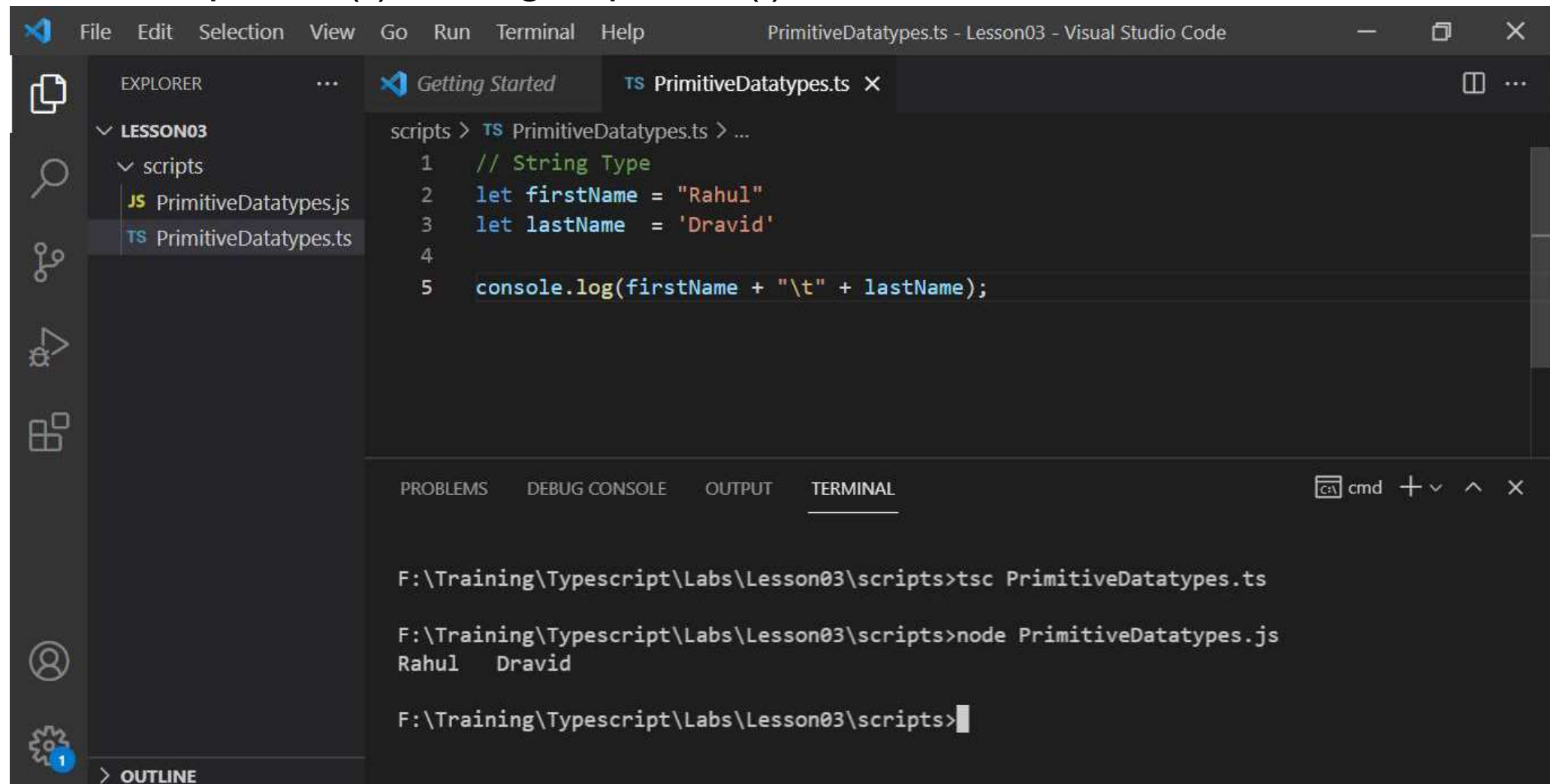
Typescript Type Checking

- TypeScript types introduce strong type checking to JavaScript. The use of types is optional, but if use the types, then they are analyzed by the compiler, and errors are thrown if they are used incorrectly.
- Lets take an Illustration

- TypeScript supports 7 primitive types number, string, boolean, bigint, symbol, undefined, and null.
- All other data types are objects in Typescript. A primitive data type is a data type that is not an object and has no methods. All primitives are immutable.

String

- We use the string data type to store textual data. The string value is enclosed in double-quotes (") or single quotes (').



The screenshot shows the Visual Studio Code interface. The Explorer panel on the left shows a project structure with 'LESSON03' containing a 'scripts' folder with 'PrimitiveDatatypes.js' and 'PrimitiveDatatypes.ts'. The main editor displays 'PrimitiveDatatypes.ts' with the following code:

```
1 // String Type
2 let firstName = "Rahul"
3 let lastName = 'Dravid'
4
5 console.log(firstName + "\t" + lastName);
```

The bottom panel shows the TERMINAL with the following commands and output:

```
F:\Training\Typescript\Labs\Lesson03\scripts>tsc PrimitiveDatatypes.ts
F:\Training\Typescript\Labs\Lesson03\scripts>node PrimitiveDatatypes.js
Rahul  Dravid
F:\Training\Typescript\Labs\Lesson03\scripts>
```

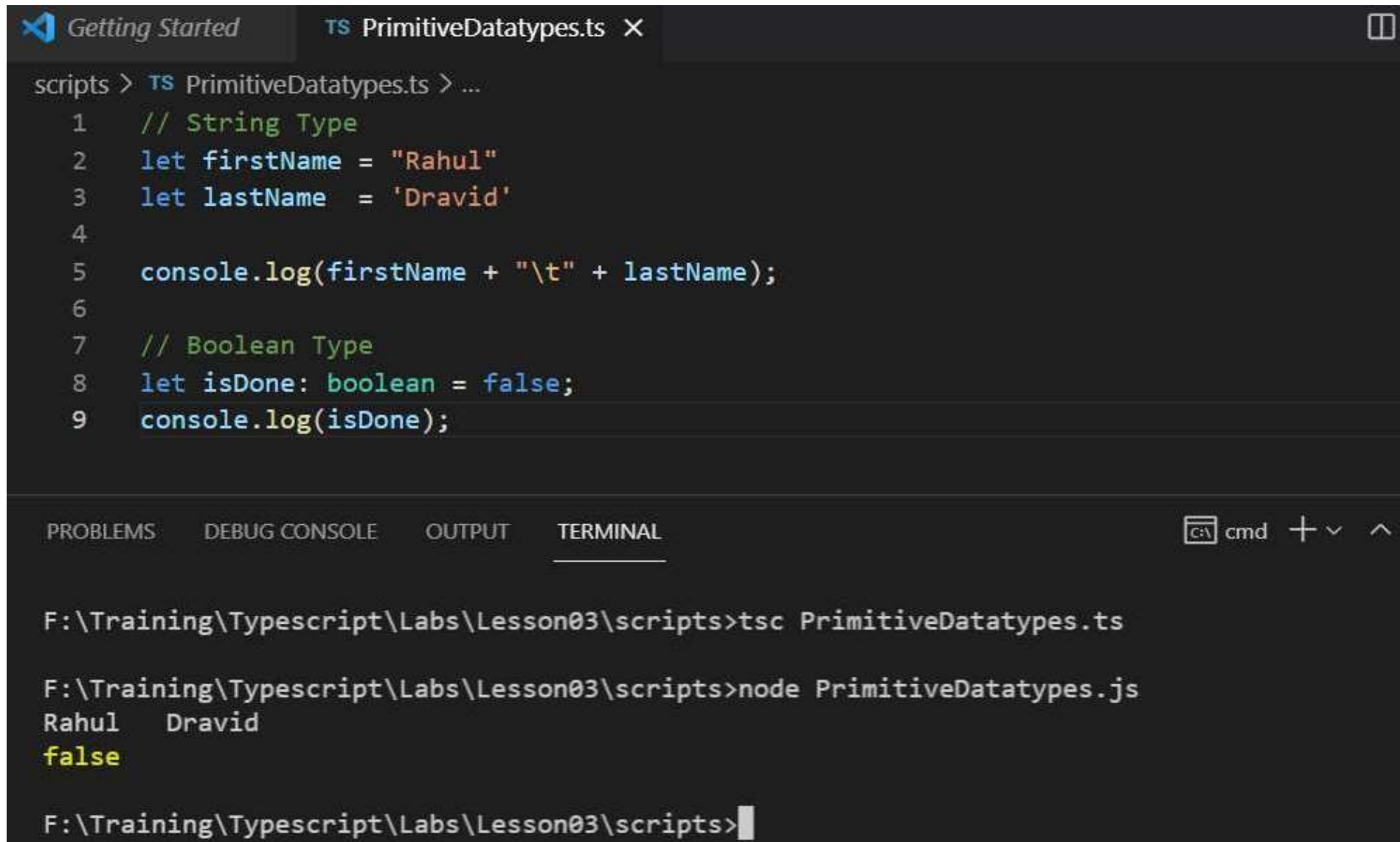

Multiline string

- The strings can span multiple lines in such cases the strings are surrounded by the backtick/backquote (`) character

```
let sentence: string = `Hello, welcome to the world of typescript,  
the typed super of javascript`;
```

Boolean

- The Boolean Data Type is a simple true/false value



The screenshot shows a VS Code editor window with a file named 'PrimitiveDatatypes.ts'. The code defines string and boolean variables and logs their values. Below the editor, the terminal shows the execution of the TypeScript compiler and Node.js, resulting in the output 'Rahul Dravid' and 'false'.

```
scripts > TS PrimitiveDatatypes.ts > ...
1  // String Type
2  let firstName = "Rahul"
3  let lastName  = 'Dravid'
4
5  console.log(firstName + "\t" + lastName);
6
7  // Boolean Type
8  let isDone: boolean = false;
9  console.log(isDone);
```

PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL

```
F:\Training\Typescript\Labs\Lesson03\scripts>tsc PrimitiveDatatypes.ts
F:\Training\Typescript\Labs\Lesson03\scripts>node PrimitiveDatatypes.js
Rahul   Dravid
false
F:\Training\Typescript\Labs\Lesson03\scripts>
```

Number

- The number data types in TypeScript are 64-bit floating-point values and are used to represent integers and fractions.
- Typescript also supports hexadecimal and decimal literals. It also supports the binary and octal literals introduced in ECMAScript 2015.

```
TS PrimitiveDatatypes.ts X
scripts > TS PrimitiveDatatypes.ts > ...
5  console.log(firstName + "\t" + lastName);
6
7  // Boolean Type
8  let isDone: boolean = false;
9  console.log(isDone);
10
11 // Number Type
12 let decimal: number = 10;
13 let hex: number = 0xa00d;      //hexadecimal number starts with 0x
14 let binary: number = 0b1010;  //binary number starts with 0b
15 let octal: number = 0o633;    //octal number starts with 0o
16
```

Bigint

- bigint is the new introduction in Typescript 3.2. This will provide a way to represent whole numbers larger than 253.
- You can get a bigint by calling the BigInt() function or by writing out a BigInt literal by adding an n to the end of any integer numeric literal

```
TS PrimitiveDatatypes.ts X
scripts > TS PrimitiveDatatypes.ts > ...
10
11 // Number Type
12 let decimal: number = 10;
13 let hex: number = 0xa00d; //hexadecimal number starts with 0x
14 let binary: number = 0b1010; //binary number starts with 0b
15 let octal: number = 0o633; //octal number starts with 0c
16
17 // Bigint
18 let big1: bigint = BigInt(100); // the BigInt function
19 let big2: bigint = 100n; // a BigInt literal. end with n
20 console.log(big1 + "\t" + big2);
21
```

Null and Undefined

- JavaScript has two ways to refer to the null. They are null and undefined and are two separate data types in Typescript as well.
- The null and undefined are subtypes of all other types. That means you can assign null and undefined to something like a number

```
22
23 // Null and Undefined
24 let u: undefined = undefined;
25 let n: null = null;
26 |
27
```

- undefined Denotes value is given to all uninitialized variables.
- null: Represents the intentional absence of object value.

Opting out of Type Checking [ANY]

- Typescript allows us to opt-out of type checking by assigning any type to a variable. The compiler will not perform type checking on variables whose type is any.
- any is a special data type that can hold any data. You can change the data type. We use this when we do not know the type of data. any is specific to typescript.
- When a variable's type is not given and typescript cannot infer its type from the initialization then it will be treated as an any type.
- Lets take an Illustration

Symbol

- The symbol is the new primitive type introduced in ES6 and represents the JavaScript symbol primitive type.
- It represents unique tokens that may be used as keys for object properties. it is created by the global Symbol() function.
- Each time the Symbol() function is called, a new unique symbol is returned.

- Everything that isn't a Primitive type in TypeScript is a subclass of the object type. Objects provide a way to group several values into a single value. You can group any values of other types like string, number, booleans, dates, arrays, etc. An object can also contain other objects. We can easily build a more complex structure using them
- A Typescript object is a collection of key-value pairs. Each key-value pair is known as a property, where the key is the name of the property and value its value.

Creating Objects

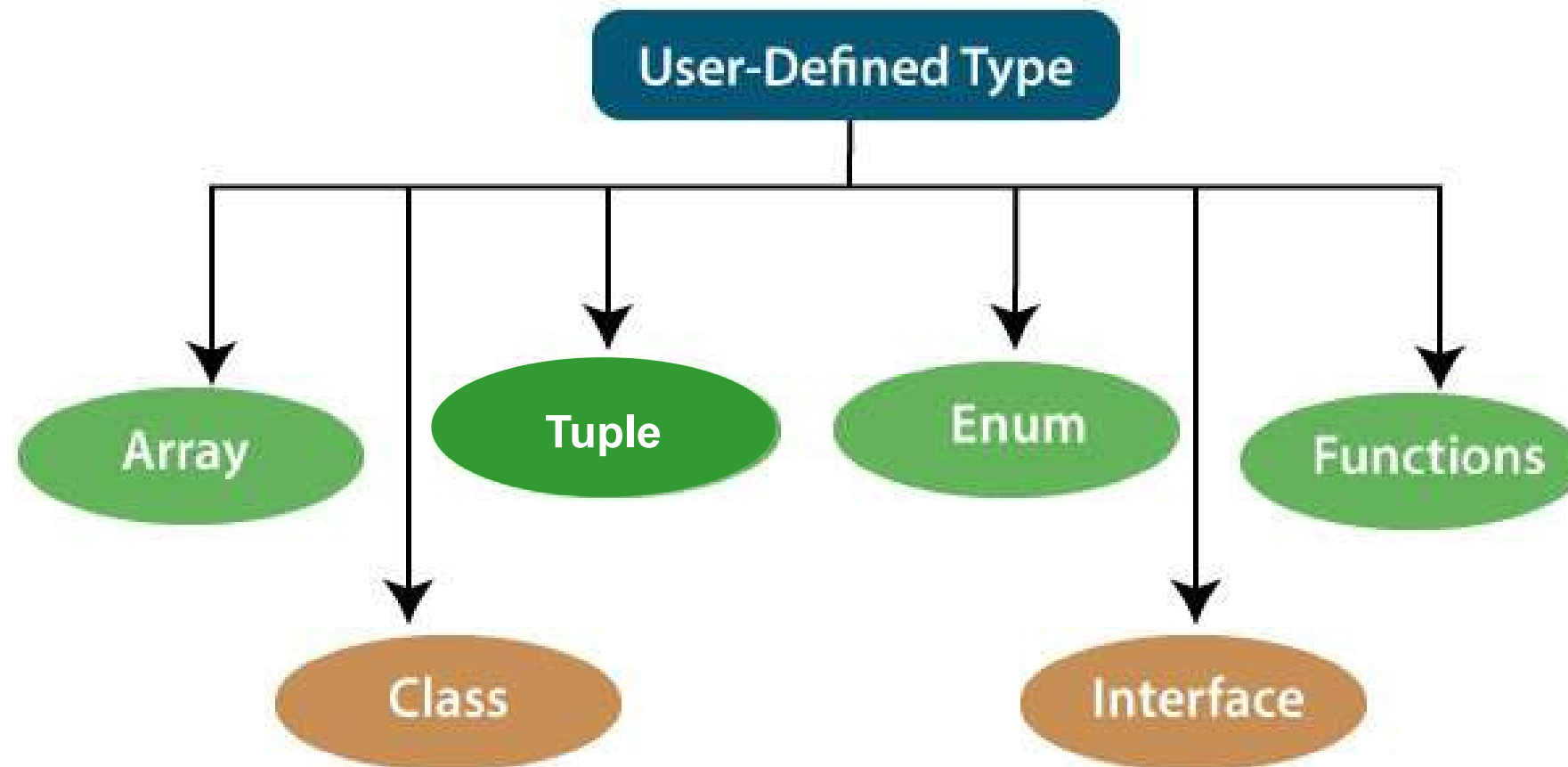
- Creating an empty object:

```
var obj_1 = {};  
var obj_2 = new Object();  
var obj_3 = Object.create(null);
```

- Creating an object:

```
var person = {  
  "full-name" : "John Doe",  
  age: 35,  
  address: {  
    address_line1: "Clear Trace, Glaslyn, Arkansas",  
    "postal code": "76588-89"  
  }  
};
```

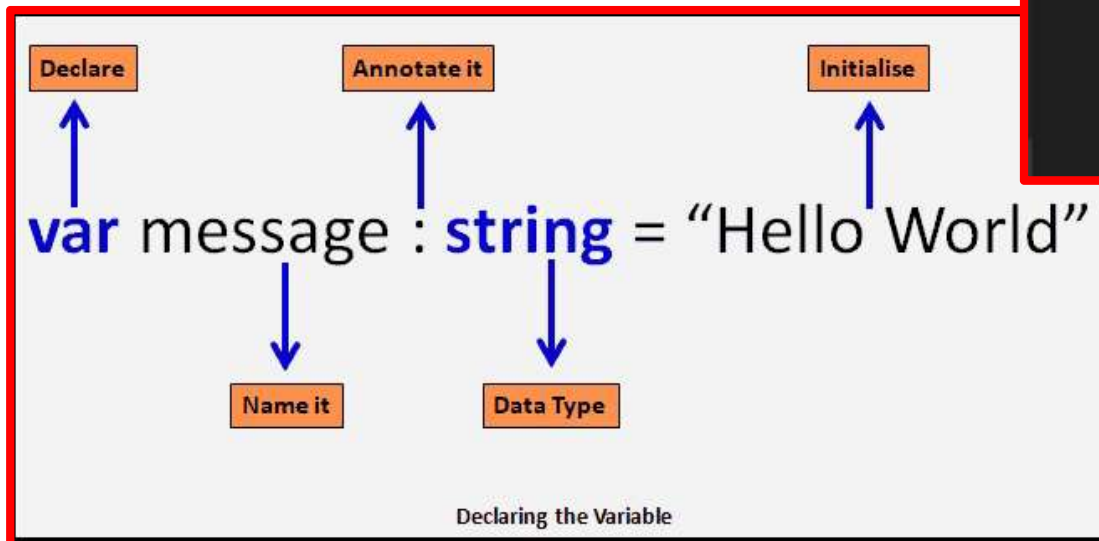
User-Defined Data Type





Type Annotations

- TypeScript is a typed language, where we can specify the type of the variables, function parameters and object properties.
- We can specify the type using :Type after the name of the variable, parameter or property. There can be a space after the colon.
- TypeScript includes all the primitive types of JavaScript- number, string and boolean.



```
TS PrimitiveDatatypes.ts  TS TypeAnnotations.ts X
scripts > TS TypeAnnotations.ts > ...
1 |
2 var age: number = 46; // number variable
3 var fullName: string = "Rahul Dravid"; // string variable
4 var isUpdated: boolean = true; // Boolean variable
```

- You cannot change the value using a different data type other than the declared data type of a variable.
- If you try to do so, TypeScript compiler will show an error. This helps in catching JavaScript errors.
- Type annotations are used to enforce type checking. It is not mandatory in TypeScript to use type annotations.
- However, type annotations help the compiler in checking types and helps avoid errors dealing with data types. It is also a good way of writing code for easier readability and maintenance by future developers working on your code.

TS TypeAnnotations.ts X

scripts > TS TypeAnnotations.ts > ...

```
5
6  //Example: Type Annotation of Parameters
7  function display(id:number, name:string)
8  {
9      console.log("Id = " + id + ", Name = " + name);
10 }
11
12 // Example: Type Annotation in Object
13 var employee : {
14     id: number;
15     name: string;
16 };
17
18 employee = {
19     id: 100,
20     name : "John"
21 }
```

Examples of Type Annotation

Arrays

- The arrays are annotated using the `string[]` or `Array`

```
TS TypeAnnotations.ts X
scripts > TS TypeAnnotations.ts > ...
22
23   var cities: string[] = ['Delhi', 'New York', 'London'];
24   //OR
25   var cities: Array<string> = ['Delhi', 'New York', 'London'];|
```

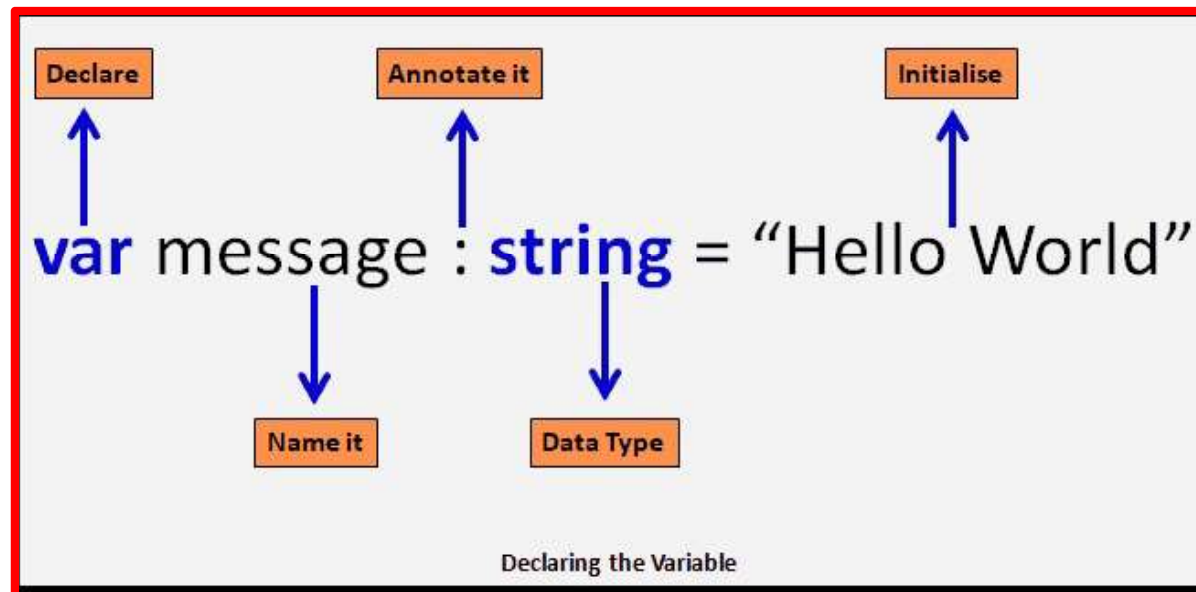
Anonymous Objects

- Here, we are creating an object with two properties. The properties are annotated with the type `number` & `string`.

```
TS TypeAnnotations.ts X
scripts > TS TypeAnnotations.ts > ...
26
27   var student: { id: number; name: string; };
28   student = { id: 100, name: "Rahul" }
```

Variables

- A Typescript variable is a storage for the data, where programs can store value or information. We must give a name to the variable. We can then refer the variable in another part of the program.
- We need to declare the variables before using them. We use let, var or const keyword to declare the variable.



Naming the Variable

We must follow these rules when naming the variable.

1. Variable name must be unique within the scope.
2. The first letter of a variable should be an upper case letter, Lower case letter, underscore or a dollar sign
3. Subsequent letters of a variable can have upper case letter, Lower case letter, underscore, dollar sign, or a numeric digit
4. They cannot be keywords.
5. Identifiers are case-sensitive.
6. They cannot contain spaces.

Valid and Invalid Identifiers

Example of Invalid Identifiers

identifier	Reason
break	Because it is a Reserved keyword
emp name	spaces are not allowed
emp-name	- not allowed
1result	cannot begin with a digit
emp@name	@ not allowed. only underscore and dollar allowed

Example of Valid Identifiers

empName	emp_name	_empName
result1	\$result	

Variable Declaration syntax

We can declare the variables in four different ways.

1. both type and initial value
2. only the type
3. only the initial value
4. without type and initial value

Keywords and Reserved Words

<code>abstract</code>	<code>do</code>	<code>if</code>	<code>private</code>	<code>void</code>
<code>as</code>	<code>delete</code>	<code>import</code>	<code>protected</code>	<code>while</code>
<code>boolean</code>	<code>else</code>	<code>implements</code>	<code>public</code>	<code>with</code>
<code>break</code>	<code>enum</code>	<code>in</code>	<code>return</code>	<code>yield</code>
<code>case</code>	<code>export</code>	<code>instanceOf</code>	<code>static</code>	
<code>catch</code>	<code>extends</code>	<code>interface</code>	<code>super</code>	
<code>class</code>	<code>false</code>	<code>let</code>	<code>switch</code>	
<code>const</code>	<code>finally</code>	<code>new</code>	<code>this</code>	
<code>continue</code>	<code>for</code>	<code>null</code>	<code>throw</code>	
<code>default</code>	<code>function</code>	<code>package</code>	<code>try</code>	

Contextual keywords

- The following keywords are restricted based on the context otherwise allowed.
- You can name the variable as the number.

TS VariableDeclarations.ts 1 X

scripts > TS VariableDeclarations.ts > ...

```
1  //The following is allowed.
2  var number:number=1000;
3  console.log(number);
4
5  //But, you cannot name the interface as the number. because the number itself is a type.
6  //This is not allowed.
7  interface number {
8      id:number
9      name:string
10 }
11
```

Variable Scope in TypeScript : Global, Local & function

- The variables have a scope or visibility.
- The scope of a variable determines which part of the program can access it.
- We can define a variable in three ways, so as to limit their visibility.
 - One is the local variable or block variable, which has the scope of a code block (block scope or local scope).
 - The second one is a function variable or class variable which has the function/class scope.
 - The last one is the global variable, which can be accessed anywhere in the program (global scope).

Difference between Let vs Var vs Const

The scope or visibility of the variable is the major difference between these keywords.

- var is function scoped

- ❖ The variables declared using var inside the function are available only within that function. If we declare them outside the function, then they are available everywhere i.e. they are a global variable.

- let & const is block scoped

- ❖ The variables declared using let or const are block-scoped. They are scoped to the block in which they are declared i.e. inside the if/try/catch/while/for or any code block (enclosed in curly parentheses).
- ❖ This means that we can only use it in the code block where we declare them. Outside the code block, they are invisible

Typescript Constants (Const)

- Typescript constants are variables, whose values cannot be modified.
- We declare them using the keyword `const`.
- They are block-scoped just like the `let` keyword. Their value cannot be changed neither they can be redeclared.

Declaring a const

We declare constants using the keyword `const` keyword

Example:

```
1  
2 const MaxAllowed=100 ;  
3
```


The initial value is a must

The initial value of the `const` must be specified along with the declaration.

```
1
2 const maxValue=100;    //Ok
3
4 const maxValue1;       //compile error
5 'const' declarations must be initialized.ts(1155)
6
```

Const is block scoped

The `const` is similar to the `let` keyword. They are local to the code block in which we declare them.

```
1
2 const Rate = 10;       //global scope
3
4 if (true) {
5   const Rate = 8;      //it is limited to this if block
6   console.log(Rate);   //Prints 8
7 }
8
9 console.log(Rate);     //Prints 10
10
```



Special Type

Typescript Never Type

- The Typescript Never type represents the type that never happens or the values that never occur.
- TypeScript introduced a new type never, which indicates the values that will never occur.
- The never type is used when you are sure that something is never going to occur. For example, you write a function which will not return to its end point or always throws an exception.

Example

```
function throwError(errorMsg: string): never {  
    throw new Error(errorMsg);  
}  
  
function keepProcessing(): never {  
    while (true) {  
        console.log('I always does something and never ends.')  
    }  
}
```

- The throwError() function throws an error and keepProcessing() function is always executing and never reaches an end point because the while loop never ends. Thus, never type is used to indicate the value that will never occur or return from a function.

Difference between never and void

- The void type can have undefined or null as a value whereas never cannot have any value.

```
let something: void = null;  
let nothing: never = null; // Error: Type 'null' is not assignable to type 'never'
```

TypeScript void type

- The void type denotes the absence of having any type at all. It is a little like the opposite of the any type.
- Typically, you use the void type as the return type of functions that do not return a value.

```
function log(message): void {  
    console.log(message);  
}
```

- It is a good practice to add the void type as the return type of a function or a method that doesn't return any value. By doing this, you can gain the following benefits:
 - Improve clarity of the code: you do not have to read the whole function body to see if it returns anything.
 - Ensure type-safe: you will never assign the function with the void return type to a variable.



Advanced Type

TypeScript union type

```
let result: number | string;
result = 10; // OK
result = 'Hi'; // also OK
result = false; // a boolean value, not OK
```

- A union type describes a value that can be one of several types, not just two. For example `number | string | boolean` is the type of a value that can be a number, a string, or a boolean.

```
function add(a: number | string, b: number | string) {
  if (typeof a === 'number' && typeof b === 'number') {
    return a + b;
  }
  if (typeof a === 'string' && typeof b === 'string') {
    return a.concat(b);
  }
  throw new Error('Parameters must be numbers or strings');
}
```


Intersection types

- We can create intersection type using the following syntax. Each type is separated by & sign.

`let a : type1 & type2 & .. & .. & typeN`

```
interface Person {  
  name: string;  
  age: number;  
}  
  
interface Student {  
  studentCode: string;  
  division: string;  
}  
  
let student: Student & Person = {  
  studentCode: "1",  
  division: "10",  
  name: "Rahul",  
  age: 20  
}
```

- Note that student has all the properties from the both the types. If you leave out of any one of the property as in name in the following code, the compiler will throw an error.

```
let student: Student & Person= {
    studentCode:"1",
    division:"10",
    name:"Rahul",
    age:20
}

let student1: Student & Person= {
    studentCode:"1",
    division:"10",
    age:20
}

//Type '{ studentCode: string; division: string; age: number; }' is not assignable to type
// 'Student & Person'.
// Property 'name' is missing in type '{ studentCode: string; division: string; age: number; }'
//but required in type 'Person'.
```

TypeScript type aliases

- Type aliases allow you to create a new name for an existing type. The following shows the syntax of the type alias:

```
type alias = existingType;
```

- The existing type can be any valid TypeScript type.

```
type chars = string;  
let message: chars; // same as string type
```

```
type alphanumeric = string | number;  
let input: alphanumeric;  
input = 100; // valid  
input = 'Hi'; // valid  
input = false; // Compiler error
```

Type inference

- Type inference describes where and how TypeScript infers types when you don't explicitly annotate them.
- When you declare a variable, you can use a type annotation to explicitly specify a type for it.

```
let counter: number;
```

- However, if you initialize the counter variable to a number, TypeScript will infer the type the counter to be number.

```
let counter = 0;
```

It is equivalent to the following statement:

```
let counter: number = 0;
```

Summary

In this lesson, you should have learned how to:

- Typescript Basics
- Datatypes like Strings, Numbers, Boolean
- Null & Undefined
- Objects
- Special Types
- Advanced Types

