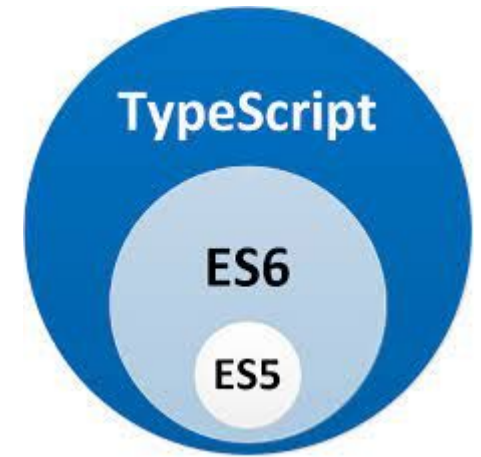




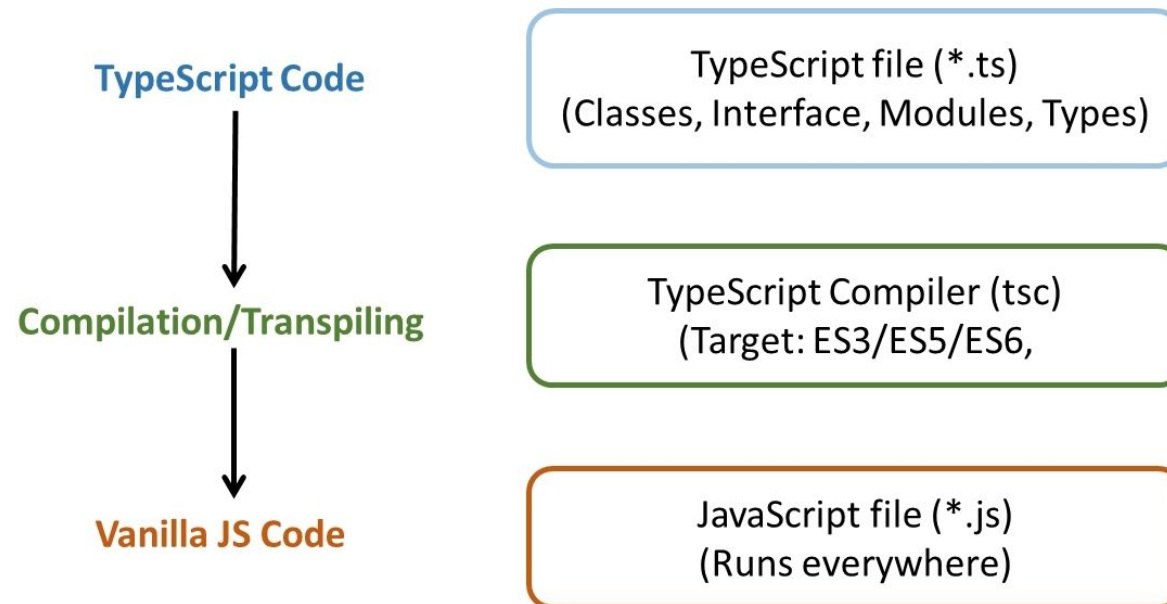
Module 2

Overview of TypeScript



- Open-source
- Object-oriented language
- Developed and maintained by Microsoft
- Licensed under Apache 2 license
- it's just JavaScript, with static typing ie. Typed Javascript
- Typed superset of JavaScript that compiles to plain JavaScript using Typescript Compiler which can run on any browser
 - TypeScript uses the JavaScript syntaxs and adds additional syntaxes for supporting Types
- TypeScript files use the .ts
- Official website: <https://www.typescriptlang.org>

- The TypeScript compiler is also implemented in TypeScript and can be used with any browser or JavaScript engines like Node.js.



Why TypeScript?

- Catches errors at compile-time, so that you can fix it before you run code.
- Supports object-oriented programming features like data types, classes, enums, etc., allowing JavaScript to be used at scale.
- TypeScript implements the future features of JavaScript a.k.a ES Next so that you can use them today.

Architecture

Design Goals of Typescript

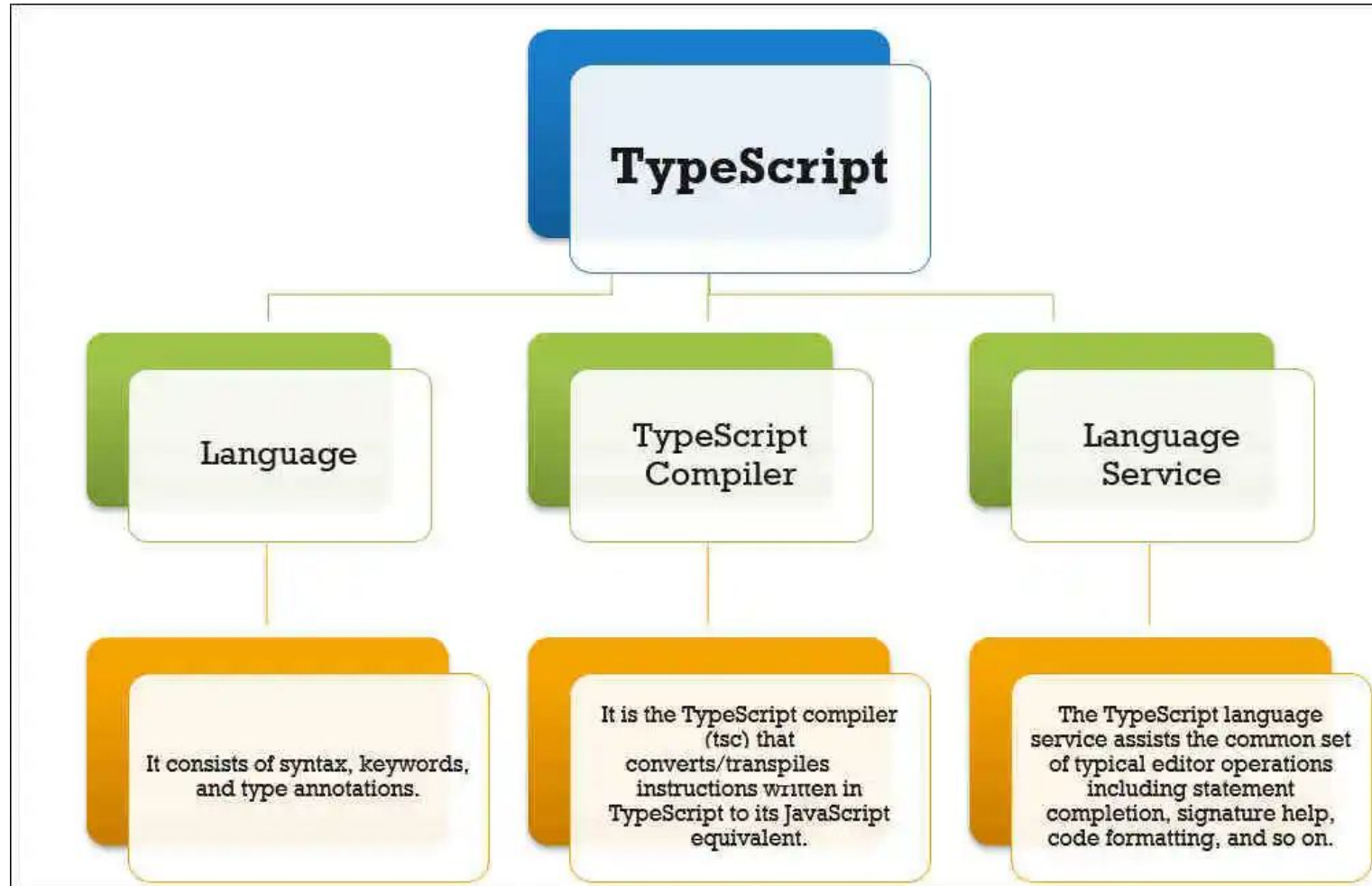
- **Statically identify JavaScript constructs that are likely to be errors**
 - strongly-typed programming language and perform static type checking at compile time.
- **Be a cross-platform development tool**
 - Microsoft released TypeScript under the open source Apache license and it can be installed and executed in all major operating systems.
- **Impose no runtime overhead on emitted programs**
 - the term design time or compile time to refer to the TypeScript code that is written while designing an application, whereas the term execution time or runtime to refer to the JavaScript code executed after compiling some TypeScript code.

- **High compatibility with existing JavaScript code**
 - any valid JavaScript program is also a valid TypeScript program (with a few small exceptions)
- **Provide a structuring mechanism for larger pieces of code**
 - Features like class-based object-orientation, interfaces, namespaces, and modules, help us to structure our code in a much better way.
 - reduce potential integration issues within the development team
 - Making code easier to maintain and scale

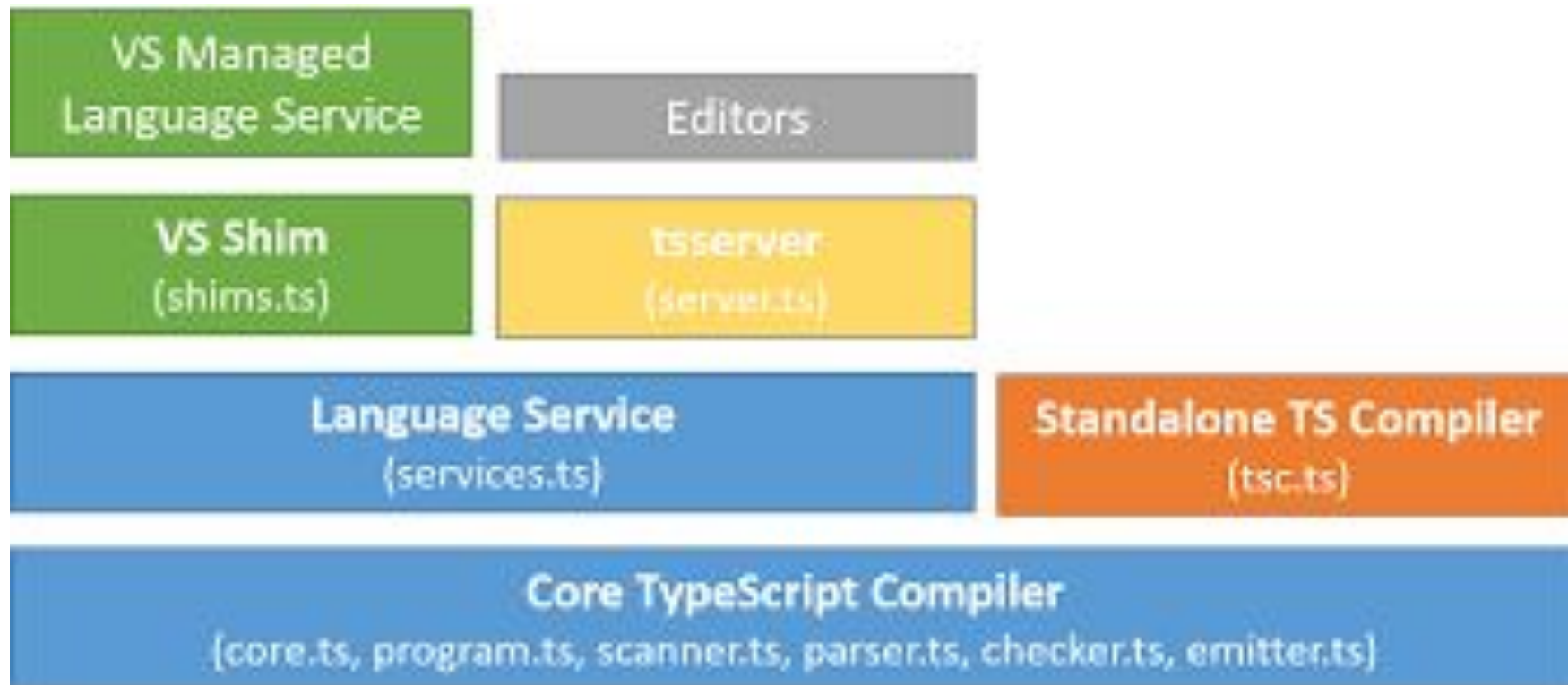
- **Impose no runtime overhead on emitted programs**
 - Some features of Typescript are only available at design time. For example, we can declare interfaces in TypeScript, but since JavaScript doesn't support interfaces, the TypeScript compiler will not declare or try to emulate this feature at runtime (in the output JavaScript code)

- **Align with current and future ECMAScript proposals**
 - TypeScript compiler with some mechanisms, such as code transformations (converting TypeScript features into plain JavaScript implementations) and type erasure (removing static type notation), to generate clean JavaScript code.

Components of TypeScript



- 3 major layers



- **Language:**

- Features the TypeScript language elements.
- It consists of syntax, keywords, and type annotations.

- **Compiler:**

- Changes the instructions written in TypeScript to its JavaScript equivalent.
- Performs the parsing, type checking, and transformation of your TypeScript code to JavaScript code.
- The TypeScript compiler configuration is given in tsconfig.json file.

- **Language Service:**

- Generates information that helps editors and other tools provide better assistance features, such as IntelliSense or automated refactoring.
- Assists the common set of typical editor operations like statement completion, signature help, code formatting and outlining, colorization, etc.

- **IDE integration (VS Shim):**

- The developers of the IDEs and text editors must perform some integration work to take advantage of the TypeScript features. TypeScript was designed to facilitate the development of tools that help to increase the productivity of JavaScript developers.

Install TypeScript using Node.js Package Manager (npm) on Windows

- Node is available here – <https://nodejs.org/en/download>
- Download and run the .msi installer for Node.
- Install Node.js and NPM from Browser
- Verify Installation
- Also to check the npm version, enter the command ***npm -v*** in the terminal window.

```
C:\Users>node -v
v4.2.3
C:\Users>_
```

LTS
Recommended For Most Users


Windows Installer
node-v16.13.1-x64.msi

Current
Latest Features


macOS Installer
node-v16.13.1.pkg


Source Code
node-v16.13.1.tar.gz

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binary (.tar.gz)

Linux Binaries (x64)

Linux Binaries (ARM)

Source Code

32-bit	64-bit
32-bit	64-bit
64-bit / ARM64	
64-bit	ARM64
64-bit	
ARMv7	ARMv8
node-v16.13.1.tar.gz	

How to use TypeScript? (in production)

- Install Node.js
- Install TypeScript compiler

```
npm install -g typescript
```

- Check the current version of the TypeScript compiler:

```
tsc --v
```

To Use

```
tsc <filename.ts>
```

```
node <filename.js>
```

OR in development

- Install Node.js and TypeScript compiler

```
npm install -g ts-node typescript
```

To Use

```
npx ts-node <filename.ts>
```

- ts-node is a TypeScript execution engine and REPL for Node.js.
- It JIT transforms TypeScript into JavaScript, enabling you to directly execute TypeScript on Node.js without precompiling.

Default tsconfig.json

```
{
  "compilerOptions":{
    "target":"es6",
    "moduleResolution":"node",
    "module":"commonjs",
    "declaration":false,
    "noLib":false,
    "emitDecoratorMetadata":true,
    "experimentalDecorators":true,
    "sourceMap":true,
    "pretty":true,
    "allowUnreachableCode":true,
    "allowUnusedLabels":true,
    "noImplicitAny":true,
    "noImplicitReturns":false,
    "noImplicitUseStrict":false,
    "outDir":"dist/",
    "baseUrl":"src/",
    "listFiles":false,
    "noEmitHelpers":true
  },
  "include":[
    "src/**/*"
  ],
  "exclude":[
    "node_modules"
  ],
  "compileOnSave":false
}
```


First Program

//FileName: Program1.ts

var message:string="HelloWorld";

console.log(message);

Output: HelloWorld

```
var message = "HelloWorld";  
console.log(message);
```

TypeScript Features

1. Cross-Platform/ Portable
2. Object-Oriented Language
3. Static type-checking

- checking, ensuring that the data flowing through the program is of the correct kind of data.
- Type checking cuts down on errors, sets the stage for better tooling, and allows developers to map their programs at a higher level.
- In static typing, compiler enforces that values use the same type

```
let value = 5;
```

```
value = "hello"; // error: Type '"hello"' is not assignable to type 'number'.
```

Javascript Code

```
function add(x, y){  
  return x + y;  
}  
let result = add(input1.value, input2.value);  
console.log(result);
```

// result of concatenating strings

Typescript Code

```
function add(x: number, y: number):number {  
  return x + y;  
}  
let result = add(input1.value, input2.value);
```

// throws error

- JavaScript is dynamically typed. It offers flexibility but also creates many problems.
- TypeScript adds an optional type system to JavaScript to solve these problems.

- **Optional Static Typing**

```
let value: any = 5;  
console.log(value);  
value = "hello";  
console.log(value);
```



```
function add(a: any, b: any): any {  
    if (typeof a === 'number' && typeof b === 'number') {  
        return a + b;  
    }  
    if (typeof a === 'string' && typeof b === 'string') {  
        return a.concat(b);  
    }  
}  
  
console.log(add(3,6));  
console.log(add("Hello","TypeScript"));
```

- **DOM Manipulation**

- TypeScript can be used to manipulate the DOM for adding or removing elements similar to JavaScript.

- **ES 6 Features**

JavaScript Vs TypeScript

Sr. No.	JavaScript	TypeScript
1.	It doesn't support strongly typed or static typing.	It supports strongly typed or static typing feature.
2.	Netscape developed it in 1995.	Anders Hejlsberg developed it in 2012.
3.	JavaScript source file is in ".js" extension.	TypeScript source file is in ".ts" extension.
4.	It is directly run on the browser.	It is not directly run on the browser.
5.	It is just a scripting language.	It supports object-oriented programming concept like classes, interfaces, inheritance, generics, etc.
6.	In this, number, string are the objects.	In this, number, string are the interface.
7.	Example: <script> function addNumbers(a, b) { return a + b; } var sum = addNumbers(15, 25); document.write('Sum of the numbers is: ' + sum); </script>	Example: function addNumbers(a:number, b:number) { return a + b; } var sum = addNumbers(15, 25); console.log('Sum of the numbers is: ' + sum);

TypeScript Variables

We can declare a variable in one of the four ways:

- **Declare type and value in a single statement**

[keyword] [identifier] : [type-annotation] = value;

- **Declare type without value. Then the variable will be set to undefined.**

[keyword] [identifier] : [type-annotation];

- **Declare its value without type. Then the variable will be set to any.**

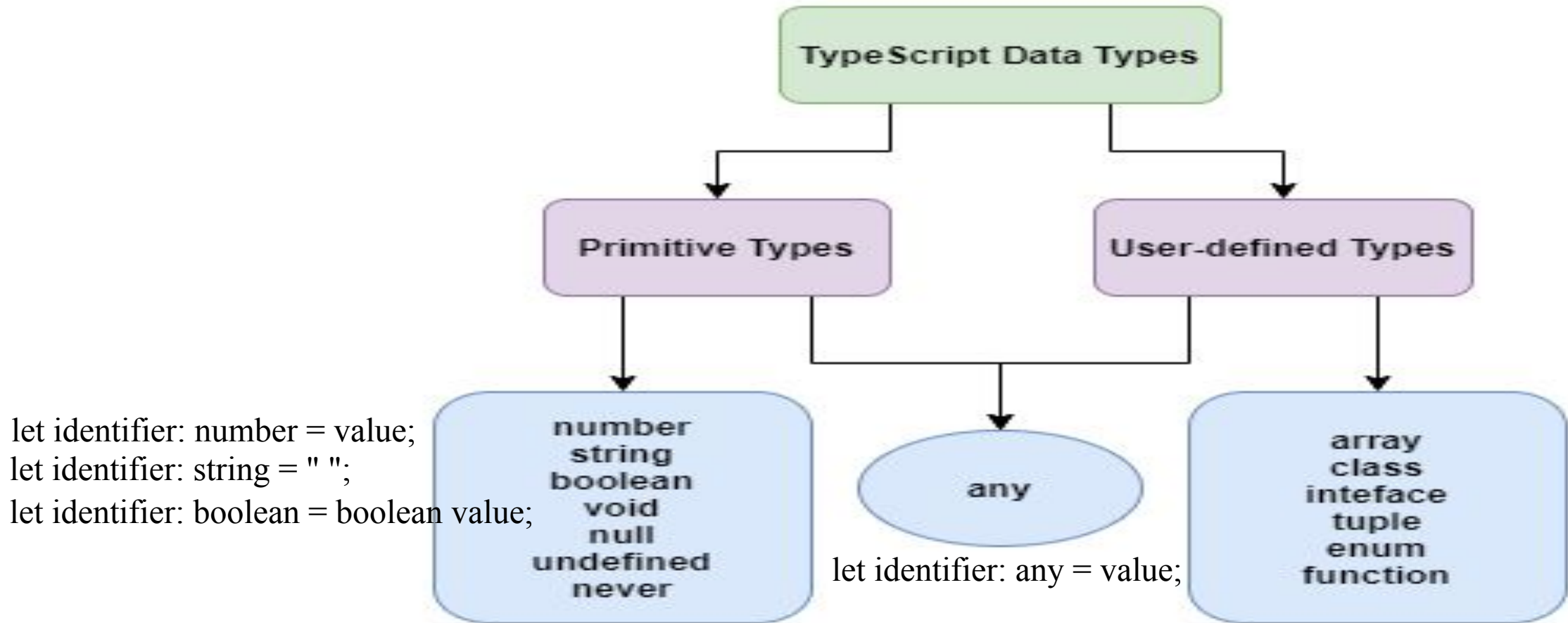
[keyword] [identifier] = value;

- **Declare without value and type.** Then the variable will be set to any and initialized with undefined.

[keyword] [identifier];

- **Note:**
- **Keyword]:** It's a keyword of the variable either **var, let or const** to define the scope and usage of variable.
- **[Variable Name]:** It's a name of the variable to hold the values in our application.
- **[Data Type]:** It's a type of data the variable can hold such as number, string, boolean, etc.
- **[Value]:** Assigning a required value to the variable.

TypeScript Data Types



Primitive types

Name	Description
<code>string</code>	represents text data
<code>number</code>	represents numeric values
<code>boolean</code>	has true and false values
<code>null</code>	has one value: null
<code>undefined</code>	has one value: <code>undefined</code> . It is a default value of an uninitialized variable
<code>symbol</code>	represents a unique constant value

Object types

- **Object types are functions, arrays, classes, etc.**
- **There are two main purposes of types in TypeScript:**
 - First, types are used by the TypeScript compiler to analyze your code for errors
 - Second, types allow you to understand what values are associated with variables.
 - Every value in TypeScript has a type.
 - A type is a label that describes the properties and methods that a value has.

TypeScript Types with Annotations

```
let uname: string = "Albert Einstein"; // string
let ulocation: string = "Germany"; // string
let age: number = 38; // number
let isScientetist: boolean = true; // boolean

console.log("Name:" + uname);
console.log("Location:" + ulocation);
console.log("Age:" + age);
console.log("Is Doctorate:" + isScientetist);
```

```
var uname = "Albert Einstein"; // string
var ulocation = "Germany"; // string
var age = 38; // number
var isScientetist = true; // boolean
console.log("Name:" + uname);
console.log("Location:" + ulocation);
console.log("Age:" + age);
console.log("Is a Scientetist:" +
isScientetist);
```

var Vs let keyword

Sr. No.	Var	let
1.	The var keyword was introduced with JavaScript.	The let keyword was added in ES6 (ES 2015) version of JavaScript.
2.	It has global scope.	It is limited to block scope.
3.	It can be declared globally and can be accessed globally.	It can be declared globally but cannot be accessed globally.
4.	<p>Variable declared with var keyword can be re-declared and updated in the same scope.</p> <p>Example:</p> <pre>function varGreeter(){ var a = 10; var a = 20; //a is replaced console.log(a); } varGreeter();</pre>	<p>Variable declared with let keyword can be updated but not re-declared.</p> <p>Example:</p> <pre>function varGreeter(){ let a = 10; let a = 20; //SyntaxError: //Identifier 'a' has already been declared console.log(a); } varGreeter();</pre>
5.	<p>It is hoisted.</p> <p>Example:</p> <pre>{ console.log(c); // undefined. //Due to hoisting var c = 2; }</pre>	<p>It is not hoisted.</p> <p>Example:</p> <pre>{ console.log(b); // ReferenceError: //b is not defined let b = 3; }</pre>

TypeScript Operators

- Arithmetic operators
- Comparison (Relational) operators
- Logical operators
- Bitwise operators
- Assignment operators
- Ternary/conditional operator
- Concatenation operator
- Advanced Type Operator

Arithmetic Operators

Operator	Operator_Name	Description	Example
+	Addition	It returns an addition of the values.	let a = 10; let b = 20; let c = a + b; console.log(c); // Output 30
-	Subtraction	It returns the difference of the values.	let a = 20; let b = 10; let c = a - b; console.log(c); // Output 10
*	Multiplication	It returns the product of the values.	let a = 30; let b = 20; let c = a * b; console.log(c); // Output 600
/	Division	It performs the division operation, and returns the quotient.	let a = 100; let b = 20; let c = a / b; console.log(c); // Output 5
%	Modulus	It performs the division operation and returns the remainder.	let a = 75; let b = 20; let c = a % b; console.log(c); // Output 15
++	Increment	It is used to increments the value of the variable by one.	let a = 15; a++; console.log(a); // Output 16
--	Decrement	It is used to decrements the value of the variable by one.	let a = 15; a--; console.log(a); // Output 14

Comparison (Relational) Operators

Operator	Operator Name	Description	Example
==	Is equal to	It checks whether the values of the two operands are equal or not.	let a = 10; let b = 20; console.log(a==b); //false console.log(a==10); //true console.log(10=='10'); //true
===	Identical(equal and of the same type)	It checks whether the type and values of the two operands are equal or not.	let a = 10; let b = 20; console.log(a===b); //false console.log(a===10); //true console.log(10==='10'); //false
!=	Not equal to	It checks whether the values of the two operands are equal or not.	let a = 10; let b = 20; console.log(a!=b); //true console.log(a!=10); //false console.log(10!='10'); //false
!==	Not identical	It checks whether the type and values of the two operands are equal or not.	let a = 10; let b = 20; console.log(a!==b); //true console.log(a!==10); //false console.log(10!=='10'); //true
>	Greater than	It checks whether the value of the left operands is greater than the value of the right operand or not.	let a = 30; let b = 20; console.log(a>b); //true console.log(a>30); //false console.log(20> 20'); //false
>=	Greater than or equal to	It checks whether the value of the left operands is greater than or equal to the value of the right operand or not.	let a = 20; let b = 20; console.log(a>=b); //true console.log(a>=30); //false console.log(20>='20'); //true
<	Less than	It checks whether the value of the left operands is less than the value of the right operand or not.	let a = 10; let b = 20; console.log(a<b); //true console.log(a<10); //false console.log(10<'10'); //false
<=	Less than or equal to	It checks whether the value of the left operands is less than or equal to the value of the right operand or not.	let a = 10; let b = 20; console.log(a<=b); //true console.log(a<=10); //true

Logical Operators

Operator	Operator_ Name	Description	Example
&&	Logical AND	It returns true if both the operands(expression) are true, otherwise returns false.	let a = false; let b = true; console.log(a&&b); //false console.log(b&&true); //true console.log(b&&10); //10 which is also 'true' console.log(a&&'10'); //false
	Logical OR	It returns true if any of the operands(expression) are true, otherwise returns false.	let a = false; let b = true; console.log(a b); //true console.log(b true); //true console.log(b 10); //true console.log(a '10'); // '10' which is also 'true'
!	Logical NOT	It returns the inverse result of an operand(expression).	let a = 20; let b = 30; console.log(!true); //false console.log(!false); //true console.log(!a); //false console.log(!b); //false console.log(!null); //true

Bitwise Operators

Operator	Operator_Name	Description	Example
&	Bitwise AND	It returns the result of a Boolean AND operation on each bit of its integer arguments.	let a = 2; let b = 3; let c = a & b; console.log(c); // Output 2
	Bitwise OR	It returns the result of a Boolean OR operation on each bit of its integer arguments.	let a = 2; let b = 3; let c = a b; console.log(c); // Output 3
^	Bitwise XOR	It returns the result of a Boolean Exclusive OR operation on each bit of its integer arguments.	let a = 2; let b = 3; let c = a ^ b; console.log(c); // Output 1
~	Bitwise NOT	It inverts each bit in the operands.	let a = 2; let c = ~ a; console.log(c); // Output -3
>>	Bitwise Right Shift	The left operand's value is moved to the right by the number of bits specified in the right operand.	let a = 2; let b = 3; let c = a >> b; console.log(c); // Output 0
<<	Bitwise Left Shift	The left operand's value is moved to the left by the number of bits specified in the right operand. New bits are filled with zeroes on the right side.	let a = 2; let b = 3; let c = a << b; console.log(c); // Output 16
>>>	Bitwise Right Shift with Zero	The left operand's value is moved to the right by the number of bits specified in the right operand and zeroes are added on the left side.	let a = 3; let b = 4; let c = a >>> b; console.log(c); // Output 0

Assignment Operators

Operator	Operator_ Name	Description	Example
=	Assign	It assigns values from right side to left side operand.	let a = 10; let b = 5; console.log("a=b:" +a); // Output 10
+=	Add and assign	It adds the left operand with the right operand and assigns the result to the left side operand.	let a = 10; let b = 5; let c = a += b; console.log(c); // Output 15
-=	Subtract and assign	It subtracts the right operand from the left operand and assigns the result to the left side operand.	let a = 10; let b = 5; let c = a -= b; console.log(c); // Output 5
*=	Multiply and assign	It multiplies the left operand with the right operand and assigns the result to the left side operand.	let a = 10; let b = 5; let c = a *= b; console.log(c); // Output 50
/=	Divide and assign	It divides the left operand with the right operand and assigns the result to the left side operand.	let a = 10; let b = 5; let c = a /= b; console.log(c); // Output 2
%=	Modulus and assign	It divides the left operand with the right operand and assigns the result to the left side operand.	let a = 16; let b = 5; let c = a %= b; console.log(c); // Output 1

Ternary Operator

expression ? expression-1 : expression-2;

Example:

```
let num = 10;
```

```
let result = (num > 0) ? "True":"False"
```

```
console.log(result);
```

```
//Output: True
```

Concatenation Operator

- **Example:**

```
let message = "Welcome to " + "WebX.0";  
console.log("Result of String Operator: " +message);
```

- **Output:**

Result of String Operator: Welcome to WebX.0

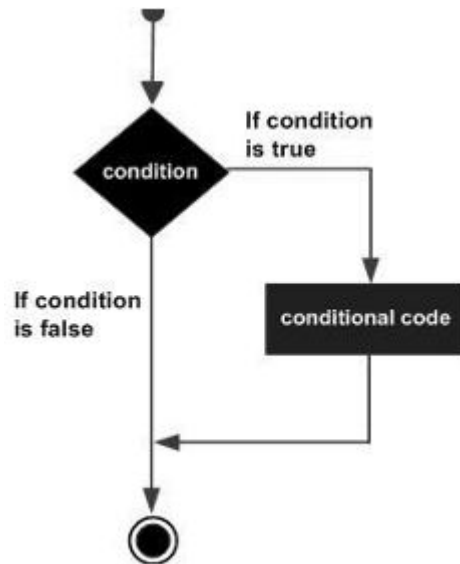
Advanced Type Operators

Operator_Name	Description	Example
in	It is used to check for the existence of a property on an object.	<pre>let Bike = {make: 'Honda', model: 'CLIQ', year: 2018}; console.log('make' in Bike); // Output: true</pre>
delete	It is used to delete the properties from the objects.	<pre>let Bike = { Company1: 'Honda', Company2: 'Hero', Company3: 'Royal Enfield' }; delete Bike.Company1; console.log(Bike); // Output: { Company2: 'Hero', Company3: 'Royal Enfield' }</pre>

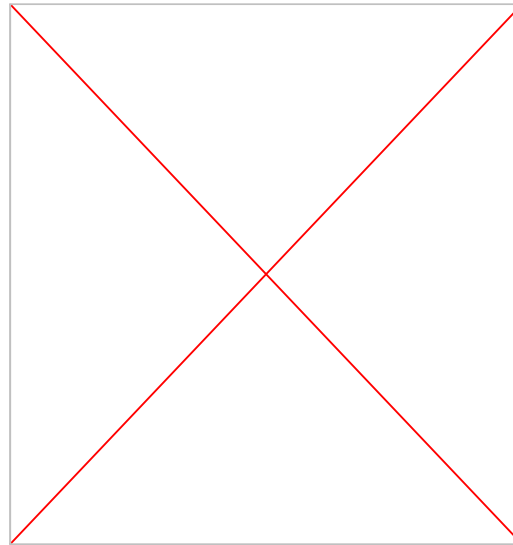
Operator_Name	Description	Example
typeof	It returns the data type of the operand.	let message = "Welcome to " + "Techneo"; console.log(typeof message); // Output: String
instanceof	It is used to check if the object is of a specified type or not.	let arr = [1, 2, 3]; console.log(arr instanceof Array); // true console.log(arr instanceof String); // false

Decision Making in TypeScript

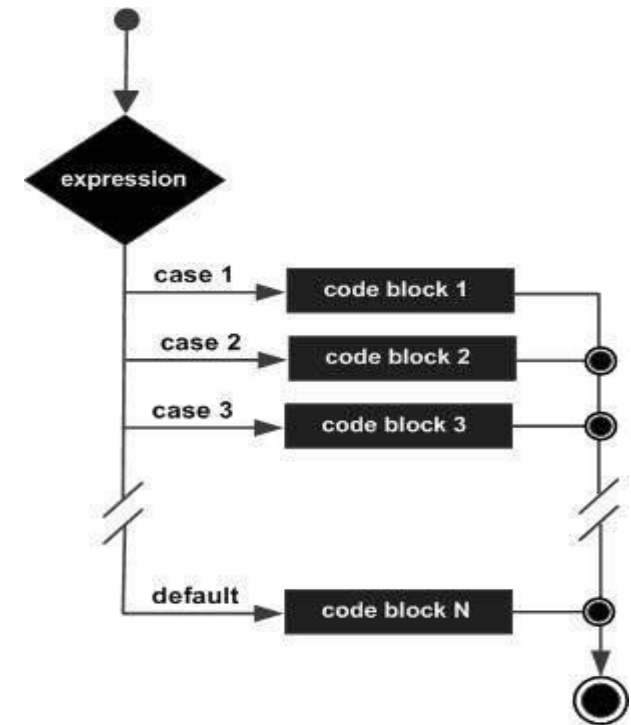
if statement



if...else statement

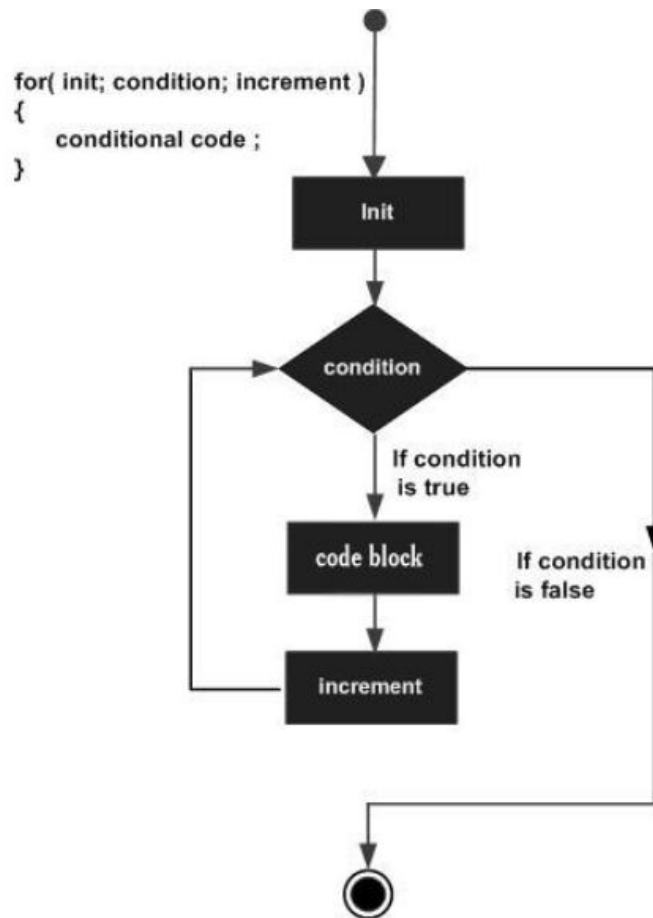


switch statement

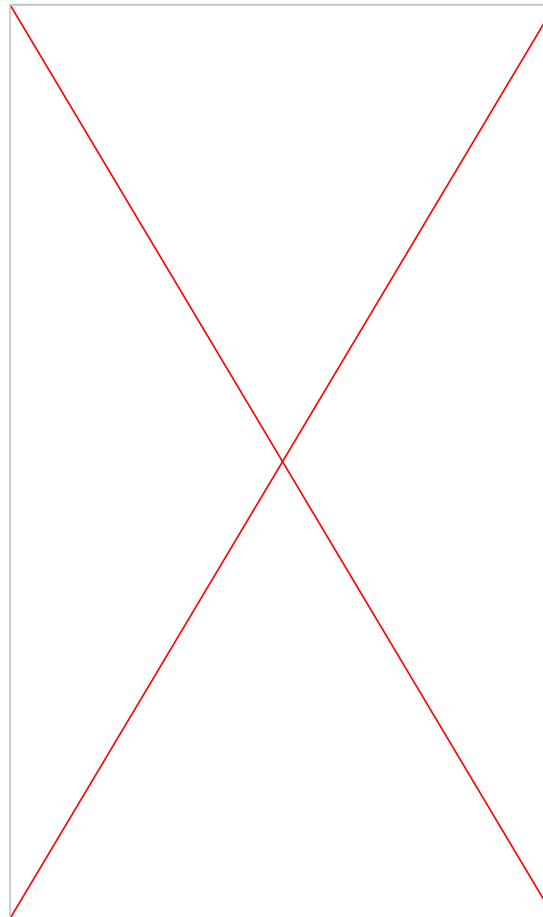


Loops in TypeScript

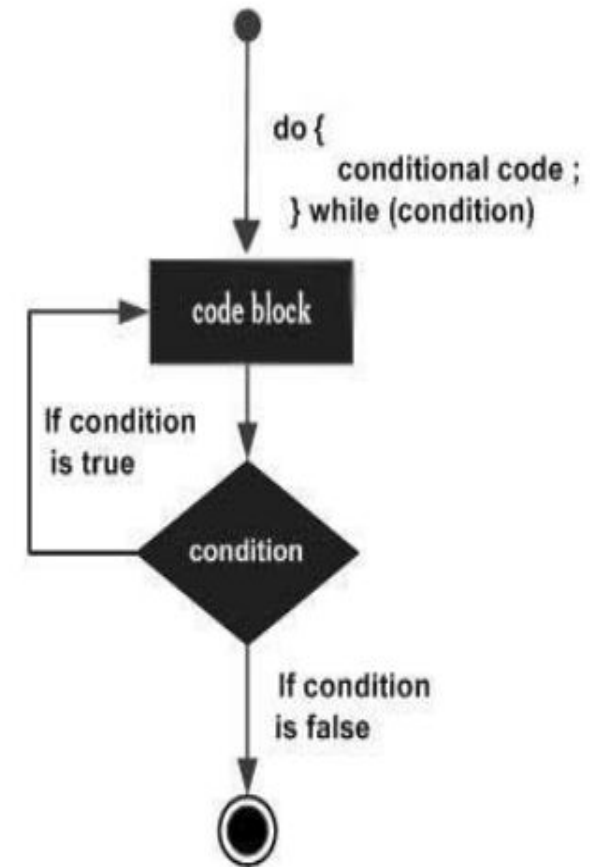
Definite Loop: for loop



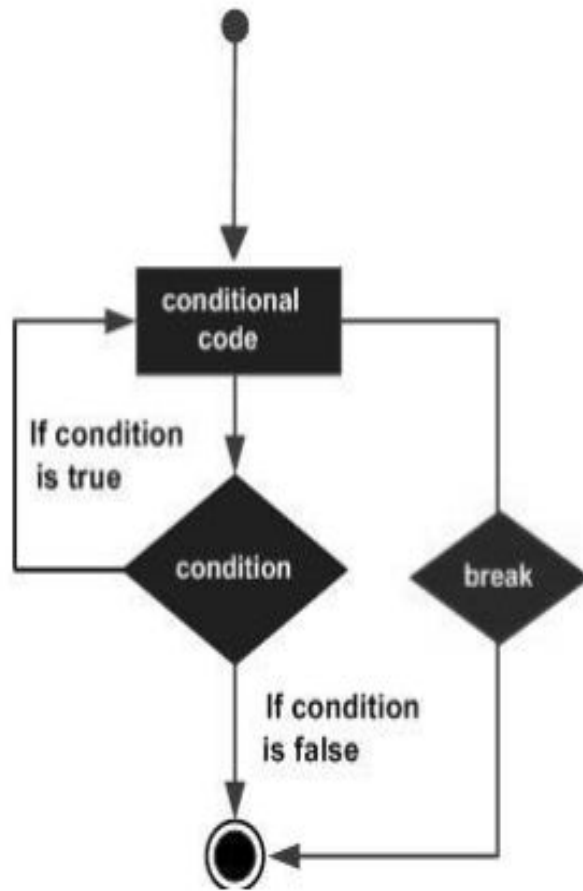
Indefinite Loop: while loop



do...while loop

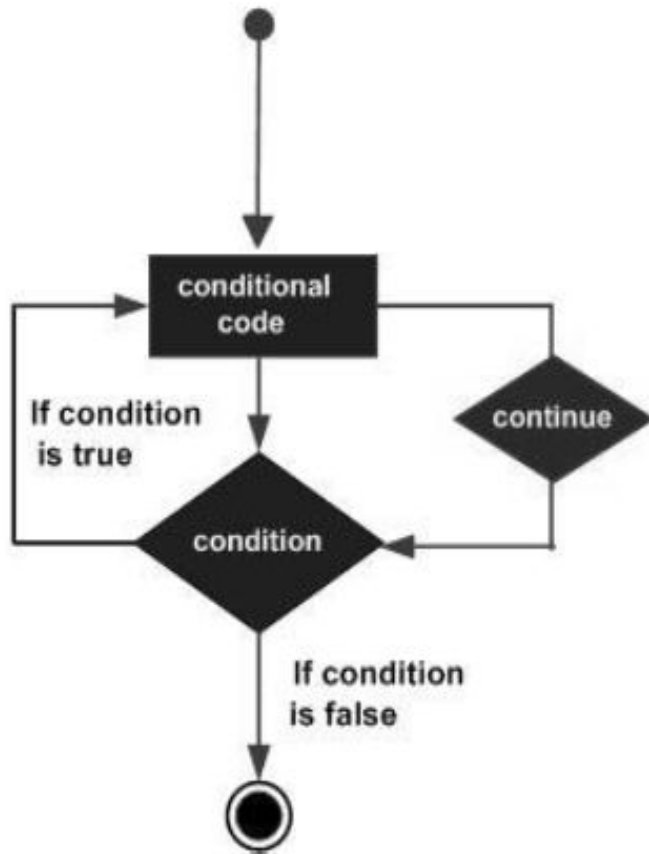


break Statement



```
var i:number = 1
while(i<=10) {
  if (i % 5 == 0) {
    console.log ("The first multiple of 5 between 1 and 10
is : "+i)
    break    //exit the loop if the first multiple is found
  }
  i++
} //outputs 5 and exits the loop
```

continue Statement



```
var num:number = 0  
var count:number = 0;
```

```
for(num=0;num<=20;num++) {  
    if (num % 2==0) {  
        continue  
    }  
    count++  
}
```

```
console.log (" The count of odd values between 0 and 20 is: "+count)
```

Output: The count of odd values between 0 and 20 is: 10

Functions in TypeScript

- Functions are the fundamental building block of any applications in JavaScript.
- It makes the code readable, maintainable, and reusable.
- We can use it to build up layers of abstraction, mimicking classes, information hiding, and modules.
- In TypeScript you are likely to find that most functions are actually written as methods that belong to a class.

Function Aspects

- **Function declaration**

```
function functionName( [arg1, arg2, ...argN] );
```

- **Function definition**

```
function functionName( [arg1, arg2, ...argN] ){  
    //code to be executed  
}
```

- **Function call**

```
FunctionName();
```

Function Creation

- We can create a function in two ways. These are:

1. **Named Function**

Syntax: `functionName([arguments]) { }`

Example:

`//Function Definition`

```
function display() {  
    console.log("Hello WebX.0!");  
}
```

`//Function Call`

```
display();// Hello WebX.0!
```

2. Anonymous Function

Syntax: `let res = function([arguments]) { }`

Example:

`// Anonymous function`

```
let myAdd = function (x: number, y: number) : number {  
    return x + y;  
};
```

`// Anonymous function call`

```
console.log(myAdd(2,3)); // 5
```


- Each parameter can be given a type annotation.
- When the function is called, the type of each argument passed to the function is checked.
- There is an additional type annotation outside of the parentheses that indicates the return type.

Optional Parameters

- In JavaScript, it is possible to call a function without supplying any arguments, even where the function specifies parameters. It is even possible in JavaScript to pass more arguments than the function requires.
- In TypeScript, the compiler checks each call and warns you if the arguments fail to match the required parameters in number or type.
- Because arguments are thoroughly checked, you need to annotate optional parameters to inform the compiler that it is acceptable for an argument to be omitted by calling code.

```
function getAverage(a: number, b: number, c ? : number): string {  
  var total = a;  
  var count = 1;  
  total += b;  
  Count++;  
  if (typeof c !== 'undefined') {  
    total += c; count++; }  
  var average = total / count;  
  return 'The average is ' +average;  
}
```

Default Parameters

- When you specify a default parameter, it allows the argument to be omitted by calling code and in cases where the argument is not passed the default value will be used instead.
- Default parameters are complementary to optional parameters.

```
function concatenate(items: string[], separator = ',' , beginAt = 0
, endAt = items.length ) {
    var result = '';
    for (var i = beginAt; i < endAt; i++) {
        result += items[i];
        if (i < (endAt - 1)) {
            result += separator; }
    }
    return result;
}
```

```
var items = ['A', 'B', 'C']; // 'A,B,C'
```

```
var result = concatenate(items); // 'B-C'
```

```
var partialResult = concatenate(items, '-', 1):
```

Rest Parameters

- Rest parameters allow calling code to specify zero or more arguments of the specified type.
- For the arguments to be correctly passed, rest parameters must
- follow these rules
 - Only one rest parameter is allowed.
 - The rest parameter must appear last in the parameter list.
 - The type of a rest parameter must be an array type.

```
function getAverage( ... a: number[]): string {  
  var total = 0; var count = 0; for (var i = 0; i < a.length; i++) {  
    total += a[i]; count++;  
  }  
  var average = total / count; return 'The average is ' +average;  
}  
var result = getAverage(2, 4, 6, 8, 10); // 'The average is 6'
```

Overloads

- In many languages, each overload has its own implementation but in TypeScript the overloads all decorate a **single implementation**.
- When you call a function that has overloads defined, the compiler constructs a list of signatures and attempts to determine the signature that matches the function call.
- If there are no matching signatures the call results in an error.


```
// Function overload for string parameters
```

```
function getAverage(a: string, b: string, c: string): string;
```

```
// Function overload for number parameters
```

```
function getAverage(a: number, b: number, c: number): string;
```

```
// Function implementation
```

```
function getAverage(a: string | number, b: string | number, c: string | number): string {  
    var total = parseInt(a as string, 10) + parseInt(b as string, 10) + parseInt(c as string, 10);  
    var average = total / 3;  
    return 'The average is ' + average;  
}
```

```
var result = getAverage(4, 3, 8); // Result: 'The average is 5'
```

```
console.log(result);
```

Arrow Function (Lambda Function)

- It omits the function keyword.
- uses fat arrow (\Rightarrow)
- It is also called a Lambda function.
- The arrow function has lexical scoping of "this" keyword

Syntax: (parameter1, parameter2, ..., parameterN) \Rightarrow expression;

- **Arrow function with Parameter**

```
let sum = (x: number, y: number): number => {  
    return x + y;  
}  
console.log(sum(10, 20)); //returns 30
```

- **Arrow function without a parameter**

```
let Print = () => console.log("Hello TypeScript");  
Print(); //Output: Hello TypeScript
```

TypeScript Classes and Objects

- In object-oriented programming languages like Java and C#, classes are the fundamental entities used to create **reusable** components.
- TypeScript introduced classes to avail the benefit of **object-oriented** techniques like encapsulation and abstraction.
- The class in TypeScript is compiled to plain JavaScript functions by the TypeScript compiler to work across platforms and browsers.
- A **class definition** can contain the following properties:
 1. **Fields:** It is a variable declared in a class.
 2. **Methods:** It represents an action for the object.
 3. **Constructors:** It is responsible for initializing the object in memory.
 4. **Nested class and interface:** It means a class can contain another class.

- **Syntax to declare a class**

- A class keyword is used to declare a class in TypeScript. We can create a class with the following syntax:

```
class <class_name>{  
    field;  
    method;  
}
```

- **Object Creation**

Syntax: let object_name = new class_name(parameter)

Object Initialization

- Object initialization means storing of data into the object.
- There are three ways to initialize an object. These are:

1. **By Reference Variable**

2. **By Method**

3. **By Constructor**

TypeScript Inheritance

- Inheritance is the ability of a program to create a new class from an existing class.
- It is a mechanism which acquires the **properties** and **behaviors** of a class from another class.
- The class whose members are inherited is called the **base class**, and the class that inherits those members is called the **derived/child/subclass**.

- In child class, we can override or modify the behaviors of its parent class.
- The TypeScript uses class inheritance through the **extends** keyword.
- TypeScript supports only **single** inheritance and **multilevel** inheritance. It doesn't support multiple and hybrid inheritance.
- We can use inheritance for **Method Overriding** (so runtime polymorphism can be achieved) and **code reusability**.

Before ES 6

Feature	JavaScript	TypeScript
Type Annotations	Dynamic typing; no type annotations	Statically typed; introduces type annotations
Constructor Inheritance	Uses <code>`call`</code> or <code>`apply`</code> to call base constructor	Uses <code>`super`</code> to call base class constructor
Access Modifiers	No access modifiers (public by default)	Supports access modifiers (public, private, etc.)
Class Syntax	Prototype-based inheritance	Class-based inheritance
Inheritance Syntax	<code>`Child.prototype = Object.create(Parent.prototype); Child.prototype.constructor = Child;`</code>	<code>`class Child extends Parent {}`</code>
Example	<code>`javascript function Dog(name, breed) { Animal.call(this, name); this.breed = breed; }`</code>	<code>`typescript class Dog extends Animal { constructor(name: string, public breed: string) { super(name); } }`</code>

After ES 6

Feature	JavaScript (ES6 and later)	TypeScript
Type Annotations	Dynamic typing; no type annotations	Statically typed; introduces type annotations
Constructor Inheritance	Uses <code>`super`</code> to call base class constructor	Uses <code>`super`</code> to call base class constructor
Access Modifiers	No access modifiers (public by default)	Supports access modifiers (public, private, etc.)
Class Syntax	Class-based syntax introduced in ES6	Class-based syntax with TypeScript enhancements
Inheritance Syntax	<code>`class Child extends Parent {}`</code>	<code>`class Child extends Parent {}`</code>
Example	<code>`javascript class Dog extends Animal { constructor(name, breed) { super(name); this.breed = breed; } }`</code>	<code>`typescript class Dog extends Animal { constructor(name: string, public breed: string) { super(name); } }`</code>

TypeScript Interfaces

- An Interface is a structure which acts as a **contract** in our application.
- used to enforce the implementation of specified properties or methods on an object.
- JavaScript doesn't support interfaces, but TypeScript does.
- It defines the syntax for classes to follow.
- We cannot instantiate the interface, but it can be referenced by the class object that implements it.
- The interface contains only the **declaration** of the **methods** and **fields**, but not the **implementation**.

- An interface can declare not only properties but also methods (no implementations though).
- A class declaration can then include the implements keyword followed by the name of the interface. In other words, while an interface just contains method signatures, a class can contain their implementations.

- **Syntax:**

```
interface interface_name {  
    // variables' declaration  
    // methods' declaration  
}
```

```
interface MotorVehicle {  
    startEngine(): boolean;  
    stopEngine(): boolean;  
    brake(): boolean;  
    accelerate(speed: number): void;  
    honk(howLong: number): void;  
}
```

Declares a method signature that should be implemented by a class

```
class Car implements MotorVehicle {  
    startEngine(): boolean {  
        return true;  
    }  
    stopEngine(): boolean {  
        return true;  
    }  
    brake(): boolean {  
        return true;  
    }  
    accelerate(speed: number): void {  
        console.log(`Driving faster`);  
    }  
  
    honk(howLong: number): void {  
        console.log(`Beep beep yeah!`);  
    }  
}
```

Implements the methods from the `interface`

```
const car = new Car();  
car.startEngine();
```

Instantiates the Car class

Uses the Car's API to start the engine

Using interfaces as custom types

- ts

```
interface Person {  
    firstName: string;  
    lastName: string;  
    age: number;  
}  
  
function savePerson (person: Person): void {  
    console.log('Saving ', person);  
}  
  
const p: Person = {  
    firstName: "John",  
    lastName: "Smith",  
    age: 25 };  
  
savePerson(p);
```

.js

```
1  "use strict";  
2  function savePerson(person) {  
3      console.log('Saving ', person);  
4  }  
5  const p = {  
6      firstName: "John",  
7      lastName: "Smith",  
8      age: 25  
9  };  
10 savePerson(p);  
11
```

- combining classes and interfaces brings flexibility to code design.
- option—extending an interface.


```
interface Flyable extends MotorVehicle{  
    fly(howHigh: number);  
    land();  
}  
  
class SecretServiceCar implements Flyable, Swimmable {  
  
    startEngine(): boolean {  
        return true;  
    };  
    stopEngine(): boolean{  
        return true;  
    };  
    brake(): boolean {  
        return true;  
    };  
    accelerate(speed: number) {  
        console.log(`Driving faster`);  
    }  
  
    honk(howLong: number): void {  
        console.log(`Beep beep yeah!`);  
    }  
}
```

One interface extends another


Declares a method signature to be implemented in a class

Implements the method from MotorVehicle

```
fly(howHigh: number) {  
  console.log(`Flying ${howHigh} feet high`);  
}  
  
land() { 2((CO16-7))  
  console.log(`Landing. Fasten your belts.`);  
}  
  
swim(howFar: number) {  
  console.log(`Swimming ${howFar} feet`);  
}  
}
```



Implements the
method from Flyable



Implements the method
from Swimmable

Generics

TypeScript Modules

- A module is a way to create a group of related variables, functions, classes, and interfaces, etc.
- It executes in the **local scope**, not in the **global scope**.
- We can create a module by using the **export** keyword and can use in other modules by using the **import** keyword.
- Modules import another module by using a **module loader**.
- The most common modules loaders which are used in JavaScript are the **CommonJS** module loader for **Node.js** and **require.js** for Web applications.

Steps:

1. Module Creation
2. Accessing the module in another file by using the **import** keyword.
3. Compiling and Execution of Modules

- **Example:** Let us understand the module with the following example.

1. Module Creation

//FileName: **addition.ts**

```
export class Addition{  
    constructor(private x: number, private y: number){  
    }  
    Sum(){  
        console.log("SUM: " +(this.x + this.y));  
    }  
}
```

2. Accessing the module in another file by using the import keyword.

//FileName: app.ts

```
import {Addition} from './addition';  
let addObject = new Addition(10, 20);  
addObject.Sum();
```

3. Compiling and Execution of Modules

Open the **terminal** and go to the location where you stored your **project**. Now, type the following command in the terminal window.

```
--module <target> <file path>
```

Use of the above targets depend on the application and module loader you are using.

```
$ tsc --module commonjs app.ts
```

```
$ node ./app.js
```

4. Output:

```
SUM: 30
```

References

1. Yakov Fain and Anton Moiseev, “TypeScript Quickly”, Manning Publications.
https://drive.google.com/file/d/164stbHHyQIWB_y8s90oTnS-SoFrxc6S_/view?usp=share_link
2. Steve Fenton, “Pro TypeScript: Application - Scale Javascript Development”, Apress
https://drive.google.com/file/d/1ma-ju9yrbBD_DrqSdJCtDkQZHB2czk85/view?usp=share_link
3. Boris Cherny, “Programming TypeScript- Making Your Javascript Application Scale”, O’Reilly Media Inc.
https://drive.google.com/file/d/1mf49NTqLRjhHZDET_bT1WYgk1B5VaEBa/view?usp=sharing
4. <https://www.javatpoint.com/typescript-tutorial>
5. <https://www.tutorialsteacher.com/typescript>
6. Typescript in 50 Lessons
https://drive.google.com/file/d/1mhUttBaMxiP7XUArNV3QUxLSFtM0CaWu/view?usp=share_link