Unit-4

TYPESCRIPT

Syllabus :

- 4.1 Compiler options
- 4.2 Classes
- 4.3 Interfaces
- 4.4 Generics
- 4.5 Decorators

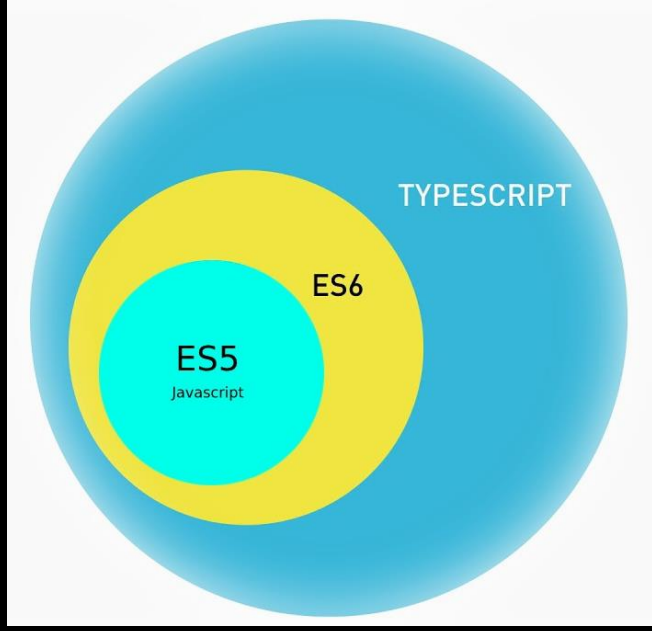
INTRODUCTION

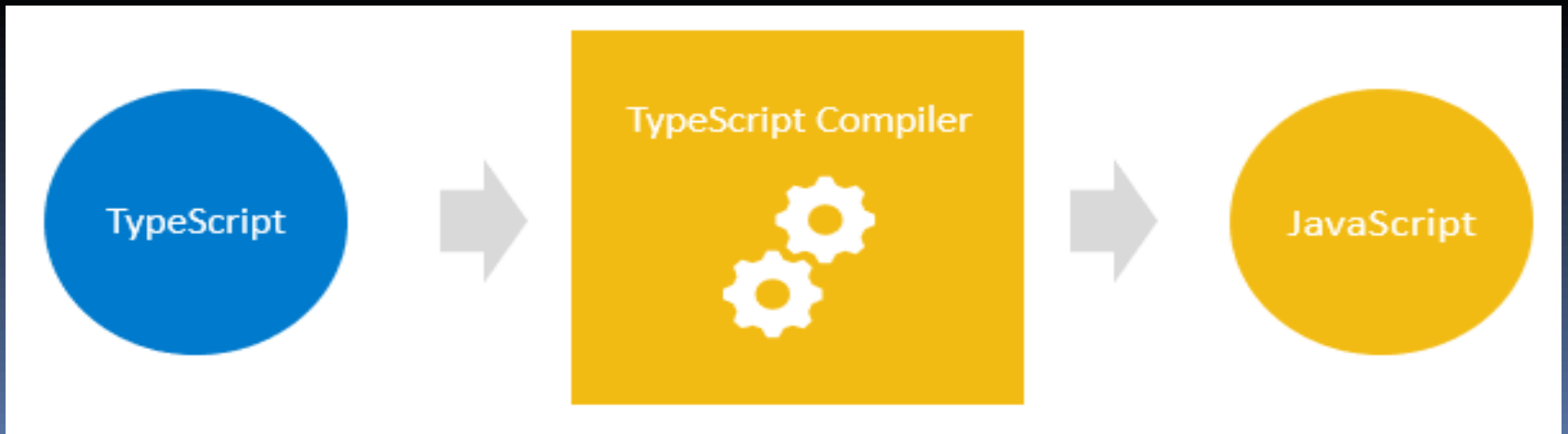
- TypeScript is a **programming language developed and maintained by Microsoft**.
- "**It is a strongly typed, pure Object oriented programming language that builds on JavaScript.**"
- It is a **strict syntactical superset of JavaScript** and **adds static typing** to the language.
- TypeScript offers **all of JavaScript's features**, and an additional layer on top of these: **TypeScript's type system**.
- Existing **JavaScript programs are also valid TypeScript programs**.
- **Can be used to develop JavaScript applications for both client-side and server-side execution**.
- **The TypeScript programs are compiled by a compiler is itself written in TypeScript and compiled to JavaScript.**

- The TypeScript source file is in **".ts"** extension.
- TypeScript **cannot run directly on the browser**.
- It **needs a compiler to compile the file** and generate its JavaScript file, which can run directly on the browser.
- Latest version of typescript: **Version 5.3**

Why Typescript?

- TypeScript is the **ES6 version of JavaScript** with **some additional features**.
- **Not all browsers supported ES6.**
- **TypeScript compiler converts TypeScript code to any chosen ECMAScript version.**
- TypeScript allows for **static typing**. This means that type mismatch errors are caught during compilation.
- TypeScript is an extension of the JavaScript language that uses **JavaScript's runtime** with a **compile-time type checker**.





Installing Typescript

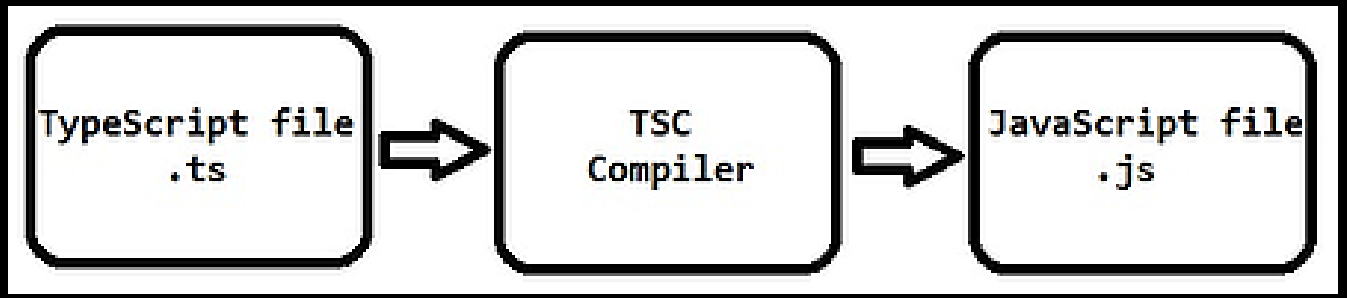
- For the latest stable version:
  - **npm install -g typescript** -> for global installation

Transpilation and Transpilers

- **Transpilation is defined as source-to-source compilation.**
- **Tools have been written to do this** and they are called **"transpilers".**
- Transpilers, or source-to-source compilers, are **tools that read the source code written in one programming language and produce the equivalent code in another programming language** with a similar level of abstraction.
- One of the most popular transpilers for JavaScript is **Babel**.
  - Babel is a tool that was created to aid in the transpilation between different versions of JavaScript.
- Typescript **transpiler converts Typescript code to JavaScript.**

tsc(the TypeScript compiler)

- The TSC (TypeScript Compiler) **is a source-to-source compiler (transcompiler / transpiler).**
- TypeScript uses TSC (TypeScript Compiler) compiler, which **convert TypeScript code (.ts file) to JavaScript (.js file).**



- **Syntax:**
  - tsc Code.ts - > output Code .js

Significant differences between Typescript and Javascript

Typescript	Javascript
Object Oriented Programming language and is superset of JavaScript	Object based Scripting language
Programs were compiled using TypeScript Compiler	Programs were interpreted
Supports Static Typing, Dynamic Typing	Supports Dynamic Typing
Developed at Microsoft	Developed by Netscape
File extension: .ts	File extension: .js
Supports generics, interfaces	Doesn't support generics, and interfaces
Specially used in Client Side along with Angular/React	Both client-side and server-side

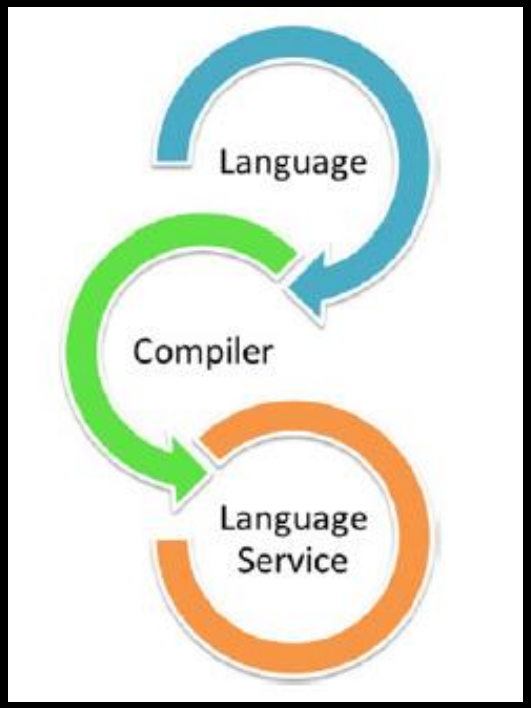
Key Benefits

- **Transpilation:**
  - Typescript transpiler(tsc) **converts typescript code to Javascript code** which can then be executed on any browser
  - The TypeScript transpiler **provides the error-checking feature**. It generates compilation errors, if it finds some sort of syntax errors.
  - It can highlight unexpected behavior in your code, lowering the chance of bugs.
- **Strong Static Typing** –
  - **Statically typed** - a variable is associated with a type which is known at compile time,
  - TypeScript comes with **type inference system** through the **TLS (TypeScript Language Service)**.
  - **The type of a variable, declared with no type, may be inferred by the TLS based on its value**.
- TypeScript **supports Object Oriented Programming** concepts like classes, interfaces, inheritance, etc.

Components of TypeScript

TypeScript has the following three components –

- **Language** – It comprises of the syntax, keywords, and type annotations/type signatures.
- **The TypeScript Compiler** – The TypeScript compiler (tsc) converts the instructions written in TypeScript to its JavaScript equivalent.
- **The TypeScript Language Service** – The language service supports the common set of a typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.



‘any’ Data Type in TypeScript

- In built-in data types, **any** is a special data-type, also the super data-type of all data types.
- If a variable is declared with any data type then we can assign any type value to that variable.

4.2 Classes

- TypeScript is object oriented JavaScript. it supports object-oriented programming features like classes, interfaces, inheritance, polymorphism, data-binding, generics etc.
- **A class in terms of OOP is a blueprint for creating objects.**
- **A class is a collection of objects having common properties.**
- Typescript gives built in support for class and got this feature from ES6.
- Use the **'class'** keyword to declare a class in TypeScript.
- **Syntax:**

```
class class_name{
   fields;
   methods;
}
```

- A class definition can include the following –
  - **Fields/Instance Variables**
  - **Functions**

  Together called as data members of the class.
- An object is an instance of class.
- To create an object/instance of the class, use with the **new** keyword followed by the class name.
- Syntax:
  - **let object_name = new class_name([arguments])**
- The **new keyword allocates memory** for objects created **at runtime**.
- All objects get memory in heap memory area.

- **Accessing Class Attributes and Functions:**
  - A class's attributes and functions **can be accessed by the object.**
  - With the help of **' . '** dot notation or bracket notation (**['']**) we access the data members of a class.

accessing an attribute :

obj.field_name or **obj['field_name']**

accessing a function :

obj.function_name()

- **Constructors:**
  - **A constructor is used to initialize an object**.
  - The constructor is a special type of method which is automatically invoked when creating an object.
  - In TypeScript, the constructor method is always defined with the name "**constructor**".
  - If you don't provide your own constructor, then **a default constructor will be supplied** for you.
  - Constructors can't have return type annotations.
  - A constructor is a function and hence can be parameterized.
  - There can be **only one** special method with the name **"constructor"** in a class. Having **more than one** occurrence of a constructor method in a class will throw an error.

Syntax:

- constructor() { … }
- constructor(argumento, argument1, … , argumentN) { … }

Class Inheritance:

- TypeScript supports the **class-based** inheritance.
- A class can reuse the properties and methods of another class. This is called inheritance in TypeScript.
- The TypeScript uses class inheritance through the **extends** keyword.
- TypeScript supports only **single** inheritance and **multilevel** inheritance.

```
class sub_class_name extends super_class_name
{
    // methods and fields
}
```

4.3 Interfaces

- An Interface is a structure which acts as a **contract** in our application.
- In TypeScript, an **interface** is an abstract type that tells the compiler which property names a given object can have.
- An interface describes the shape of an object in TypeScript.
- They can be used to provide information about object property names and the data types their values can hold to the TypeScript compiler.
- The TypeScript compiler uses interface for **type-checking** (also known as "duck typing" or "structural subtyping") whether the object has a specific structure or not.
- The interface contains only the **declaration** of the **methods** and **fields**, but **NOT the implementation**.

Declaring an interface

- **"interface"** is a keyword which is used to declare a TypeScript Interface.
- Syntax:

```
interface <Interface_Name>
{
        // variables' declaration
        // methods' declaration
}
```

  - We cannot instantiate the interface, but it can be referenced by the class object that implements it.

Class implementing an Interface

- In TypeScript, a class can implement interfaces to enforce particular contracts
- Class which implements an interface is bound to implement all its members.
- We can use an interface with a class using the keyword *implements.*
- **Syntax:**

```
class NameofClass implements InterfaceName
{

}
```

Extending Interfaces:

- Just like classes, an interface can **extend another interface**.
- We can use the "**extends**" keyword to implement inheritance among interfaces.
- **Syntax:**

```
interface child_interface extends parent_interface
{

}
```

4.4 Generics

- A major part of software engineering is building components that are reusable.
- In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is ***generics***.
- In Typescript, Generics offer a way to create reusable components.
- "***Generics provide a way to create a component that can work over a variety of types rather than a single one.***"
- To create generics, you will need to use ***Type*** parameters.
- Type parameters are defined by Type **or** **<Type>.**

- Syntax:

Generic Functions:

```
function functionName <Type>(args : Type) :    Type
 {
     //body
}
```

Generic functions with multiple types:

```
function functionName <U, V>(arg1: U, arg2: V) :  V
{
     //body
}
```

- **Generic Classes:**
  - Generic classes have a generic type parameter list in angle brackets (<>) following the name of the class.
  - **Syntax:**

```
class ClassName<Type>
{
  //fields
  //methods
}
```

4.1 Compiler Options

- Compiler options are **used to change the compiler's default operation.**
- The compiler options needed are defined by the **tsconfig.json** file.
- The tsconfig.json file's presence in a directory indicates that it is the **root of a TypeScript project**.
- The tsconfig.json file specifies the root files and the compiler options required to compile the project.

- **CLI COMMANDS**:
  - **tsc --version**
  - **To create tsconfig.json file:**
    - **Syntax: tsc --init**
  - **To set version of the emitted Javascript language version**
    - **Syntax: tsc --target version filename**
    - Example: tsc --target ES5 filename.ts
    - Target -> es3, es5, es6, es2015, es2016, es2017, es2018, es2019, es2020, es2021, or esnext
  - **To Enable experimental support for decorators.**
    - **Syntax:** tsc --experimentalDecorators

4.5 Decorators

- Decorators are an **experimental feature proposed for ES7**.
- In **use by some of the JavaScript frameworks including Angular 2**.
- They **provide a way to add both annotations and a meta-programming syntax(**writing programs that manipulate other programs or themselves based on metadata) for class declarations and members.
- ***A Decorator is a "special kind of declaration that can be attached to a classes, methods, accessor, property, or parameter".***
- Decorators use the form **@expression**, where expression **must evaluate to a function that will be called at runtime with information about the decorated declaration(target).**
- The target of a decorator depends on where you add them.
- Allows us to extend the functionality of classes and methods

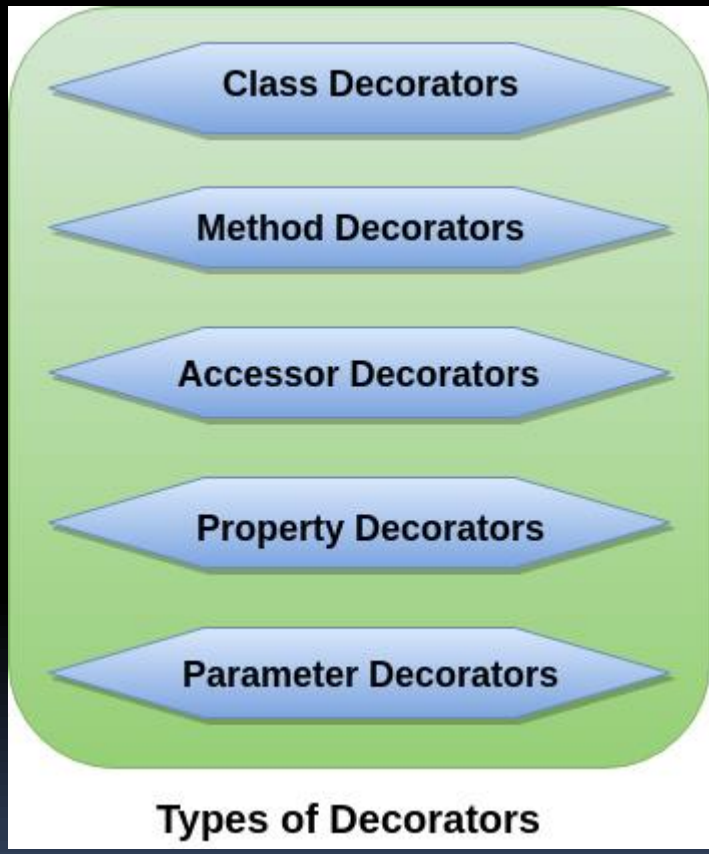
- A decorator ***is a function that is called with a specific set of parameters***.
- These **parameters are automatically populated by the JavaScript runtime**, and contain information about the class, method, or property to which the decorator has been applied.
- **The number of parameters, and their types, depends upon where a decorator was applied.**

- To enable experimental support for decorators, we must enable the **experimentalDecorators** compiler option either on the **command line** or in our **tsconfig.json**:
- **Command line:**
  - **tsc --experimentalDecorators**
- **tsconfig.json:**

```
{
    "compilerOptions":
    {
        "experimentalDecorators": true
    }
}
```

Currently, decorators can be added to the following components of a class:

- **Class Decorators:** can be applied to a class definition.
- **Property Decorators:** can be applied to a property within a class.
- **Accessor decorators :** An accessor is a getter and setter property of the class declaration. Decorators can be applied to either setter or getter method.
- **Method decorators:** can be applied to a method on a class.
- **Parameters:** can be applied to a parameter of a method within a class.



Types of Decorators

Example:

```
@classDecorator
class Person {
  @propertyDecorator
  public name: string;

  @accessorDecorator
  getName() {
    // ...
  }

  @methodDecorator
  printName(@parameterDecorator param: string) {
    // ...
  }
}
```

- To add multiple decorators, you add them together, one after the other:

```
@decoratorA
@decoratorB
class Person {
}
```

Creating Decorators:

- To create your own decorator, you have to **create a function with the same name** as your decorator.

@simple → **decorator**

```
class ClassName {
        //
}
function simple(target) {
// do something with 'target' ...
}
```

- The parameter(s) passed to the decorator will depend on where the decorator will be used.

Class Decorators:

- A *Class Decorator* is declared just before a class declaration
- functions that modify classes.
- The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition.
- They are called when the class is declared, NOT when a new instance is instantiated.

- @classdecorator

```
class Demo {

    }

function classdecorator(ctr : Function)
{   }
```

Decorator Factories:

- A decorator factory is a **function that returns a decorator.**

```
function factory(value: string)
{
    return function (target) {
    // this is the decorator
    };
}
```