Advanced Software Paradigms

Computer Programming Language

By

Jainee Gohil,

Kalyan Kollepara,

Mahesh Kumar Gudumala,

Prashanth Kumar Manji

Contents

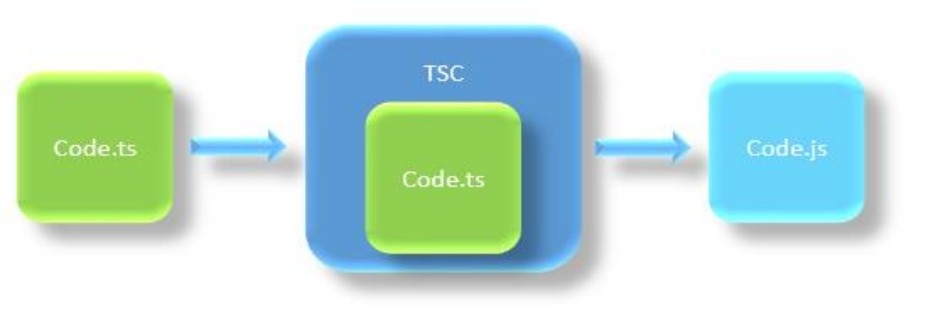
- About TypeScript
- What is Typescript? Why to use it over JavaScript?
- Set up TypeScript environment
- Variables and its scope
- Data Types
- Assignment Statements
- Support to OO Programming
- Components of TypeScript
- Concurrency
- Exception and Event Handling
- Functional Programming

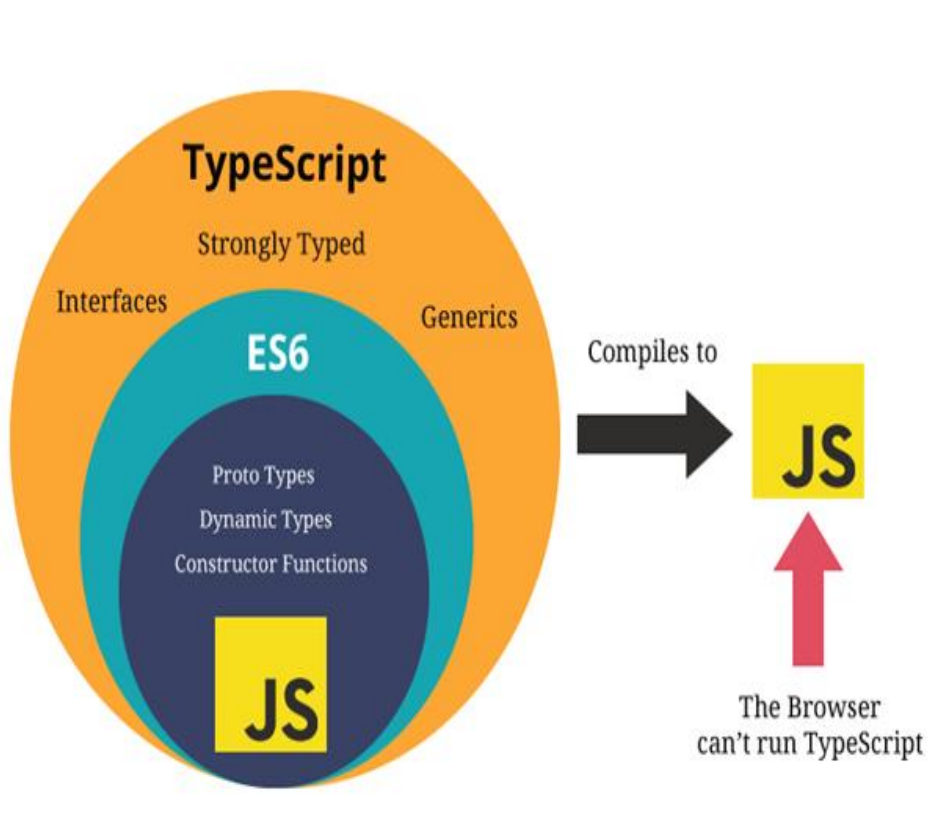
About TypeScript

- Open-Source Object-Oriented Language
- Developed and Maintained by: **Microsoft**
- Introduced by: **Anders Hejlsberg**, lead architect of C# and creator of Delphi and Turbo Pascal, has worked on the development of TypeScript
- First Released: October 2012

What is TypeScript?

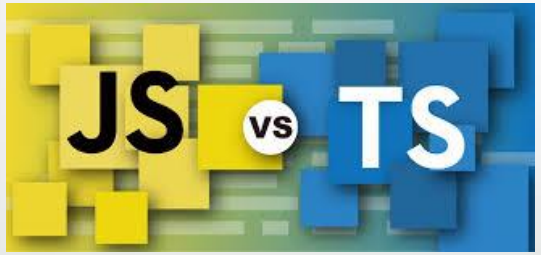
- Strongly typed superset of JavaScript
- Which complies the code into plain JavaScript
- Designed for: large-scale JavaScript application development
- File extension: .ts





Why need TypeScript over JavaScript?

JavaScript



TypeScript

Warns us about possible bugs that can happen from simple edge cases, such as forgetting null checks, type errors

Pre-requisite for TypeScript

- Any IDE (Most preferred Visual Code)
- Install Node.js Package Manager (npm)
- Set up package.json

  Prepresent your project configuration
- Install Typescript

Variable in TypeScript

Rules:

- Variable name must be an **alphabet** or **numeric digits**.
- Variable name cannot start with digits.
- Variable name cannot contain spaces and special character, except the **underscore(_)** and the **dollar($)** sign

Example:

Var name1 

Var 1name 

Var name_1 

Var name-1 

Variable Declaration in TypeScript

- Variables can be declared using:
  - Var
  - Let
  - Const
- Each of the variable declaration keyword has similar declaration and initialization.
- Those vary with each other in terms of their scope and usage.

  Scope essentially means where these variables are available for use

Example:

- Keyword name: type = value;
  - Var age: number = 50;
- Keyword name: type;
  - Var age:number;
- Keyword name = value;
  - Var age = 50;
- Keyword name;
  - Var age;

Scope for "Var"

- Global:
  - If variable is declared outside a function.
  - This means that any variable that is declared outside a function block is available for use in the whole window.
- Function:
  - If variable is declared within a function.
  - This means that it is available and can be accessed only within that function.
- **var** variables can be **updated** and **re-declared** within its scope

Example:

```
var course= "Computer Science";

    function Main() {
        var subject= "Software Paradigm";

    Console.log(course);
    Console.log(subject);
    }

Console.log(course);
Console.log(subject); //Error
```

course: Global scoped

subject: Function scoped

Scope for “Let”

- Block:
  - A block lives in curly braces “{}”. Anything within curly braces is a block.
  - **let** has scoped to the nearest enclosing block which can be smaller than a function block.
  - So a variable declared in a block with let is only available for use within that block.
  - **let** variables can be **updated** but not re-declared.

Example:

```
let course= "Computer Science";
let point = 4;

function Main() {
    If(point > 9){
        let subject= "Software Paradigm";
    Console.log(subject);
    }
Console.log(subject); //Error
}
```

Scope for “Const”

- Block:
  - Like **“**Let”, Const also has a block scope.
  - But difference between **“let”** and **“const”** is that:
    - const variable **cannot be** updated or re-declared.
    - const variable **must** be **initialized** at the time of **declaration**.

Example:

Not possible:

```
const course= “Computer Science";
course= “Computer Engineering";
```

Not possible:

```
const course= “Computer Science";
const course= “Computer Engineering ";
```

Data Types in TypeScript

- Boolean:
  - Basic datatype is the simple true/false call a Boolean.
- Number:
  - TypeScript are either floating point values or BigIntegers. These floating point numbers get the type number, while BigIntegers get the type bigint

Example:

Boolean:

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

Number:

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: bigint = 100n;
```

Data Types in TypeScript

- String:
  - Fundamental part of creating programs in TypeScript is working with textual data. As in other languages, we use the type string to refer to these textual datatypes. TypeScript also uses double quotes (") or single quotes (') to surround string data.

Example:

```
let fullName: string = `Bob Bobbington`;
let sentence: string = `Hello, my name is ${fullName}`
```

Data Types in TypeScript

- Array:
  - TypeScript allows you to work with arrays of values. Array types can be written in one of two ways.
  - In the first, you use the type of the elements followed by [] to denote an array of that element type:
  - The second way uses a generic array type, Array<elemType>:

Example:

```
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
```

Data Types in TypeScript

- Tuple:
  - Tuple types allow you to express an array with a fixed number of elements whose types are 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:

Example:

```
// Declare a tuple type
let x: [string, number];

// Initialize it
x = ["hello", 10]; // OK

// Initialize it incorrectly
x = [10, "hello"]; // Error
```

Data Types in TypeScript

- Enum:
  - A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.
  - Even manually set all the values in the enum:

Example:

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

enum Color {Red = 1,Green = 2,Blue = 4,}
let c: Color = Color.Green;
```

Data Types in TypeScript

- Unknown:
  - To describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content.
  - e.g. from the user – or we may want to intentionally accept all values in our API.
  - In these cases, we want to provide a type that tells the compiler and future readers that this variable could be anything, so we give it the unknown type.

Example:

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

Data Types in TypeScript

Any:

- In some situations, not all type information is available or its declaration would take an inappropriate amount of effort. These may occur for values from code that has been written without TypeScript or a 3rd party library. In these cases, we might want to opt-out of type checking. To do so, we label these values with the any type.
- The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type checking during compilation.
- Unlike unknown, variables of type any allow you to access arbitrary properties, even ones that don't exist. These properties include functions and TypeScript will not check their existence or type:

Example:

```
declare function getValue(key: string): any;
// OK, return value of 'getValue' is not checked
const str: string = getValue("myString");
let looselyTyped: any = 4;
// OK, ifItExists might exist at runtime
looselyTyped.ifItExists();
// OK, toFixed exists (but the compiler doesn't check)
looselyTyped.toFixed();
```

Data Types in TypeScript

- Void:
  - void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value.
  - Declaring variables of type void is not useful because you can only assign null (only if --strictNullChecks is not specified, see next section) or undefined to them:

Example:

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

let unusable: void = undefined;
// OK if `--strictNullChecks` is not given
unusable = null;
```

Data Types in TypeScript

- Null and Undefined
- Never
- Object

Example:

```
let u: undefined = undefined;
let n: null = null;
function error(message: string): never {
throw new Error(message);
}
declare function create(o: object | null): void;
```

Assignment Statements in TypeScript

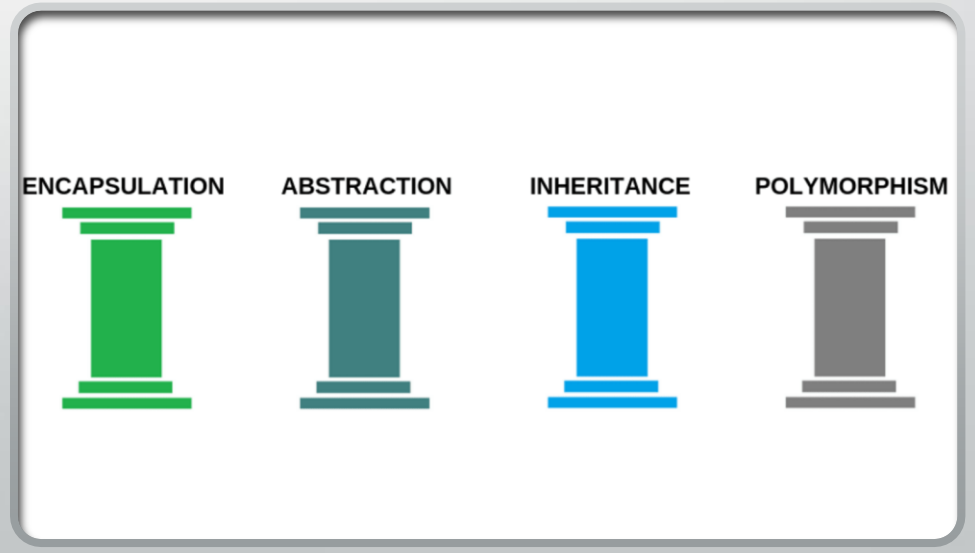
- Assignment operators are used to assign values to variables.
- This type of statement consists of a variable name, an assignment operator, and an expression.
- When appropriate, you can declare a variable and assign a value to it in a single statement.
- In assignment expressions, the right-hand expression is contextually typed by the type of the left-hand expression.

Assignment Statements in TypeScript

Operator	Same AS	Description
x = y	x = y	Simple assignment operator; assigns the value from the right side operand to the left side operand
x += y	x =x + y	Add AND assignment operator; it adds the right operand to the left operand and assigns the result to the left operand
x -= y	x =x - y	Subtract AND assignment operator; it subtracts the right operand from the left operand and assigns the result to the left operand
x *= y	x =x * y	Multiply AND assignment operator; it multiplies the right operand with the left operand and assigns the result to the left operand
x /= y	x =x / y	Divide AND assignment operator; it divides the left operand by the right operand and assigns the result to the left operand
x %= y	x =x % y	Modulus AND assignment operator; it divides the left operand by the right operand and assigns the remainder to the left operand

Support to OO Programming

- TypeScript enables you to code using object-oriented principles and techniques more efficiently and easily.
- The four main pillars of object-oriented Programming are



Encapsulation

- In object-oriented computer programming languages, the notion of encapsulation refers to the bundling of data, along with the methods that operate on that data, into a single unit.
- In TypeScript, we enforce encapsulation with methods and properties that only allow access to data that we control.

Example:

```
Withdraw(amount: number): boolean
{
    if (this._balance > amount)
    {
        this._balance -= amount
        return true;
    }
    return false;
}

private _balance: number;
get Balance(): number {
    return this._balance;
```

Inheritance

- Inheritance is a mechanism in which one class acquires the property of another class.
- It's quite easy to create an object model and inheritance chain with TypeScript. The '**extends**' keyword causes the child class to inherit from the denoted base class.

Example:

```
module Bank {
    class BankAccount { }

    class SavingsAccount extends B
ankAccount { }

    class CheckingAccount extends BankAccount { }
}
```

Abstraction

- Abstraction is the concept of object-oriented programming that "shows" only essential attributes and 'hides' unnecessary information.
- You can use the implements keyword to implement an interface in TypeScript, it's syntactically like the extends keyword.

Example:

```
export module Bank
{
    export interface Fee { ChargeFee(amount: number); }

    export class BankAccount implements Fee {
        ChargeFee(amount: number) { }
    }
}
```

Polymorphism

- Polymorphism is a feature of object-oriented programming languages that allows a specific routine to use variables of different types at different times.
- TypeScript enables polymorphism via method overrides.

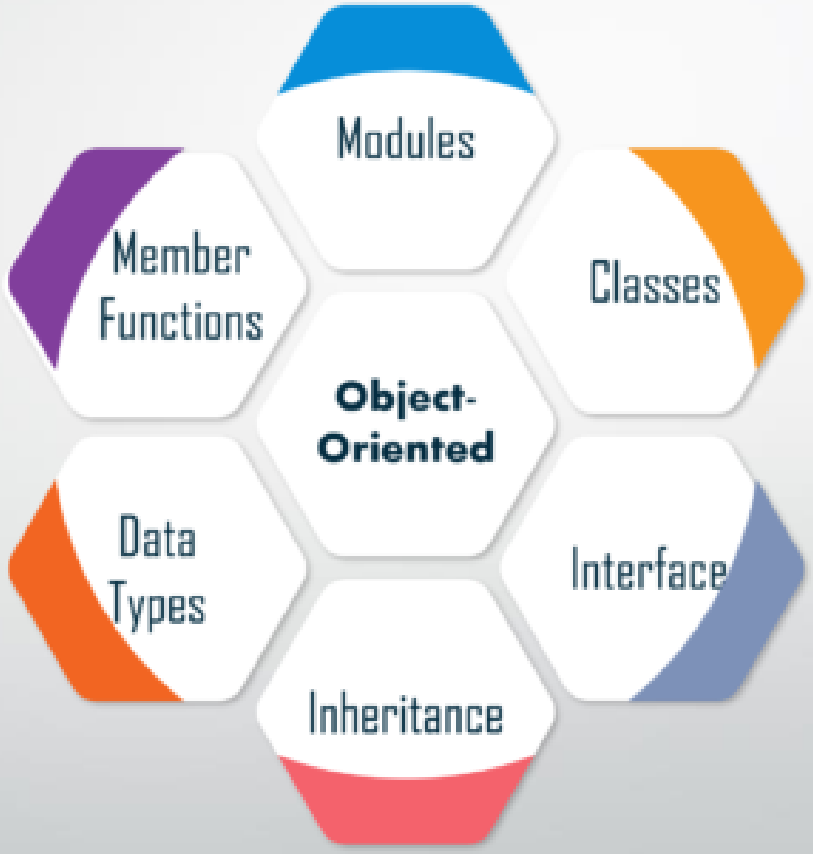
Example:

```
class SavingsAccount extends BankAccount
{
    ChargeFee(amount: number)
    {
        if (this.Balance > 1000) { amount = 0; }
        else { amount += 1.00; }

        this.Balance =+ amount;
    }
}
```

Components of TypeScript

Object-Oriented Terms



Concurrency

- Concurrency is the execution of the multiple instruction sequences at the same time.
- It is defined as one which uses the concept of simultaneously executing processes or threads of execution as a means of structuring a program.
- Concurrency in software execution can occur at four different levels:
  - Instruction level
  - Statement level
  - Unit level
  - Program level

Concurrency

- Running of Multiple Applications
- Better Resource Utilization
- Better Average Response Time
- Better Performance

Exception Handling

- The **Exception Handling** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.
- TypeScript has an Error class that you can use for exceptions. You throw an error with the throw keyword. You can catch it with a try / catch block pair

Example:

```
try {
    throw new Error('Something bad  happened');
    }
catch(e) {
    console.log(e);
}
```

Error Sub Types

- Beyond the built in Error class there are a few additional built-in error classes that inherit from Error that the TypeScript runtime can throw:
  - RangeError
  - ReferenceError
  - SyntaxError
  - TypeError
  - URIError

Event Handling in TypeScript

- Events provide a channel of communication between different parts of an application. There are several techniques for creating and handling events, each with its own advantages and disadvantages.
- Event handling reduces coupling between components to increase maintainability, flexibility and decoupling

Event Handling in TypeScript

- Below are some techniques for creating events and event handlers in TypeScript:
  1. Event Property Handlers
  2. Event Listeners with EventTarget
  3. Event Listeners with EventEmitter

Functional Programming in TypeScript

- **Functional programming** (often abbreviated FP) is the process of building software by composing **pure functions**, avoiding **shared state, mutable data,** and **side-effects**.
- Functional programming is **declarative** rather than **imperative**, and application state flows through pure functions.
- TypeScript is not a purely functional language but offers a lot of concepts which are in line with functional languages

Functional Programming in TypeScript

- **Some of the important concepts of functional programming are:**
  - First-class and higher-order functions.
  - Pure functions
  - Recursion
  - Lazy evaluation

Functional Programming in TypeScript

- **First-class and higher-order functions**

  First-class functions(function as a first-class citizen) means you can assign functions to variables, pass a function as an argument to another function or return a function from another.

- **Pure functions**

  a pure function should return values only based on the arguments passed and should not affect or depend on global state.

- **Recursion**

  Functional programming favors recursion over looping. The downside of the recursive approach is that it will be slower compared to an iterative approach most of the times.

- **Lazy evaluation**

  Lazy evaluation or non-strict evaluation is the process of delaying evaluation of an expression until it is needed.