Basic of Angular: TypeScript

What is TypeScript?

- TypeScript is a strongly typed, object-oriented, compiled programming language that builds on JavaScript.
- It is a superset of the JavaScript language.
- It allows us to use strict types.
- Supports modern features like arrow functions, let, const etc.
- Extra features like generics, interfaces, tuples etc.
- You can install the TypeScript compiler into your system by using ***npm***.

```
npm install -g typescript
```

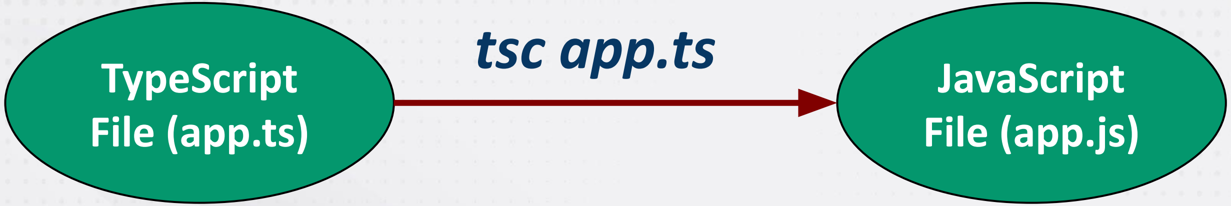
- TypeScript compiles into simple JavaScript.
- The TypeScript compiler is also implemented in TypeScript and can be used with any browser or JavaScript engines like Node.js.

How to use TypeScript?

- TypeScript code is written in a file with ***.ts*** extension and then compiled into JavaScript using the TypeScript compiler.
- A TypeScript file can not be executed directly. First it is converted into JavaScript and then the js file get executed.
- The command ***tsc <filename>.ts*** compiles the TypeScript code into a plain JavaScript file.

```
tsc app.ts
```

- JavaScript files can then be included in the HTML and run on any browser.



Features of TypeScript?

- **Object Oriented Programming Language**: TypeScript provides support for OOPs concepts like Classes, Interfaces, Inheritance, etc.
- **Compilation:** TypeScript compiler provides the feature of error checking. If there is a syntax error in the code, then TypeScript will generate a compilation error so that the error could get highlighted before runtime.
- **Strong Static Typing:** TypeScript has the feature of strong static typing which comes through TLS(TypeScript language service).
- **Supports JavaScript Libraries:** As it is a superset of JavaScript, all the libraries and existing JavaScript code are valid TypeScript code as well.

Features of TypeScript?

□ **Portable:** TypeScript is portable as the code written in TypeScript can be run on any browser or operating system. It can run in any environment in which JavaScript can run.

□ **JavaScript is TypeScript:** Code written in JavaScript with a ***.js*** extension can be converted to ***TypeScript*** by changing the extension from ***.js*** to ***.ts***.

□ **DOM Manipulation:** We can make use of TypeScript to handle DOM for adding or removing elements.

□ **Code Readability:** Its code is written using classes and interfaces. They provide organization to the program and therefore it is easy to maintain and debug the code.

Components of TypeScript

- TypeScript has ***three*** main components:
  - **Language:** The syntax, keywords, and type annotations.
  - **The TypeScript Compiler (TSC):** Converts the instructions written in TypeScript to its JavaScript equivalent.
  - **The TypeScript Language Service:** An additional layer of editor-like applications, such as statement completion, signature help, code formatting, converting of variable to specific type.

Compiling TypeScript

- Open the starter resource in visual studio code resource
- Create a TypeScript file inside the starter folder and write some ts code.

app.ts file

```
const title = "This is ts
session";
const inputs =
document.querySelectorAll
("input");
inputs.forEach((el) => {
  console.log(el);
});
```

tsc app.ts

app.js file

```
var title = "This is ts
session";
var inputs =
document.querySelectorAll("i
nput");
inputs.forEach(function (el)
{
    console.log(el);
});
```

We can also use -w with command for automatic compilation on file save tsc app.ts -w

Get-ExecutionPolicy -List
Set-ExecutionPolicy RemoteSigned
Set-ExecutionPolicy Restricted

Data Types in TypeScript

- Primitive types available in TypeScript are **boolean**, **bigint**, **null**, **number**, **string**, **symbol**, **undefined**, **any** (allow anything), **unknown** (ensure someone using this type declares what the type is), **never** (it's not possible that this type could happen), and **void** (a function which returns undefined or has no return value).
- We can check the type of a variable by using **typeof.**
- Strings in ts can be in single quotes and double quotes.
- ***TypeScript use strict types i.e if we define a variable as string then we can not change the type of that variable.***

TypeScript Array

- In TypeScript we can create an array as follow:

```
let data = ["Abhinav", "Manoj", "Vinay", "Aman"];
```

- Now we can not add any other type value into this array. TypeScript will treat it as a string array.

```
data.push(100);
```

```
Argument of type number' is not
assignable to parameter of type
```

- If you want to add number type value to this array then assign one integer value to this array.

```
let data = ["Abhinav", "Manoj", "Vinay", "Aman", 1];
data.push(100);
```

- Now it will work fine.

TypeScript Objects

- In TypeScript we can create an object as follow:

```
let student = {
  name: "Abhi",
  age: 19,
  branch: "CSE",
  marks: 87,
};
```

- We can update the property of an object as:

```
student.age=20;
```

- We can not assign different type value to any property.

```
student.name = 10;
```

```
Type 'number' is not assignable
to type 'string'
```

```
let student = {
  name: "Abhi",
  age: 19,
  branch: "CSE",
  marks: 87,
};
// Assign different object
to this variable
student = { name: "Aman",
age: 21, branch: "IT",
marks: 85 };
```

TypeScript Explicit Types

```
// Basic types
let studentName: string;
let age: number;
let isActive: boolean;
// Assign value to these variables
studentName = "Aman";
age = 20;

// Arrays
let students: string[]; // String array
//students.push("Ajay"); // We can not do
this as the array is not initialized
let studentArr: string[] = []; // String
array with initialization
studentArr.push("Ajay"); // Now We can do
this
```

□ We can create a mixed type array using union type.

```
// Union type
let mixedArr: (string | number |
boolean)[] = [];
// This array can contain string,
number and boolean value
mixedArr.push(10);
mixedArr.push(true);
// We can also use union type with
normal variables

let input: string | number | boolean;
```

TypeScript Explicit Types

```
// Object
let studentOne: object;
studentOne = { name: "Akash", age: 20, gender: "Male" };
// We can assign array to this object because array is an object
studentOne = [];
// If you do not want to assign array to this object then you can use
let studentTwo: {
  name: string;
  age: number;
  gender: string;
};
studentTwo = { name: "Aman", age: 20, gender: "Male" };
// We can not add any extra property to this variable
studentTwo = { name: "Abhay", age: 31, gender: "Male", marks: 20 };
// The above statement will not work
```

Dynamic *any* Type

□ We use any type to define a variable that can contain any type value in future.

□ If you do not know the type of variable in advance then you can use any.

```
let input: any;
input = 20; //number type
console.log(typeof input);
input = "This is string"; //string type
console.log(typeof input);
input = [10, "name", 30, "text"]; //array object type
console.log(input);
input.push(true);
console.log(input);
// Object
input = {name: "Abhi", age: 15, uid: "5477567567558",
  city: "Noida",
};
// any type array
let input: any[] = [];
input.push(10);
// any type object
let student: { name: any; age: any; gender: any };
```

TypeScript Functions

- TypeScript functions can be created both as a named function or as an anonymous function.
- ***Named function:***

```
function addTwoNumbers(num1: number, num2: number): number {
  return num1 + num2;
}
```

- ***Anonymous function:***

```
(num1: number, num2: number): number => {
  return num1 + num2;
};
```

Writing the Function Type

□ We can declare a function type variable as:

```
let myFun: (num1: number, num2: number, num3: number) => number;
```

□ Now myFun variable can contain only function type value

```
myFun = (val1: number, val2: number, val3: number): number => {
  return val1 + val2 + val3;
};
```

Optional and Default Parameters

- In TypeScript functions, every parameter is required.
- When the function is called, the compiler will check that the user has provided a value for each parameter.
- The number of arguments given to a function has to match the number of parameters the function expects.

```
let fullName = (firstName: string, lastName: string): string => {
  if (lastName) {
    return firstName + " " + lastName;
  } else {
    return firstName;
  }
};
fullName("Abhinav");
```

Error: ***`Expected 2 arguments, but got 1`***

- To resolve this problem we can use optional or default parameters.

Optional Parameters

□ In JavaScript, every parameter is optional.

□ We can get this functionality in TypeScript by adding a ? to the end of parameters we want to be optional.

□ Any optional parameters must follow required parameters.

```
let fullName = (firstName: string, lastName?: string): string =>
{
  if (lastName) {
    return firstName + " " + lastName;
  } else {
    return firstName;
  }
};
console.log(fullName("Abhinav"));
```

Now it will work fine because lastName is optional

Default Parameters

- We can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined in its place.
- These are called default-initialized parameters.
- Default-initialized parameters that come after all required parameters are treated as optional.

```
let fullName = (firstName: string, lastName: string = ""): string
=> {
  if (lastName) {
    return firstName + " " + lastName;
  } else {
    return firstName;
  }
};
console.log(fullName("Abhinav"));
```

It will work fine because lastName default value is set.
If we pass any value then it will override the default value

Function Return Type

- □ We can define the return type of a function after function parameter list.

```
let sub = (num1: number, num2: number, num3: number): string => {
  let result = num1 + num2 + num3;
  return result.toString();
};
```

This is the return type of the function

- □ If you do not want to return any value from a function then you can use void

```
let message = (): void => {
  console.log("Thsi is a function without any return value");
};
```

Function Overloading

- TypeScript provides the concept of function overloading
- In TypeScript you can have multiple functions :
  - With the same name
  - But different parameter types and return type
  - However, the number of parameters should be the same.
- Provide the prototype/declaration of the function

```
function addTwoValues(param1: string, param2: number): string;
function addTwoValues(param1: number, param2: number): number;
```

- Provide the implementation of the function only one time

```
function addTwoValues(param1: any, param2: any): any {
  return param1 + param2;
}
```

- Now we can use this function as:

```
console.log(addTwoValues(10, 20));
console.log(addTwoValues("Test", 20));
```

Type Aliases in TypeScript

- Type aliases is just giving another name for a type.
- Aliasing doesn’t create a new type; instead, it gives that type a new name.
- Consider the following example:

```
let userDetail = (name: string, uid: string | number) => {
  console.log(`The name is ${name} and uid is ${uid}`);
};
let anotherUserDetail = (user: {
  name: string;
  age: number;
  uid: string | number;
}) => {
  console.log(
    `The user name is ${user.name}, age is ${user.age} and uid is
${user.uid}`
  );
};
```

Here the parameters are complex so we can use type aliases

Type Aliases in TypeScript

□ We can do type aliases in the previous code as:

```
type stringOrNumber = string | number;
type objUserAliases = { name: string; age: number; uid:
stringOrNumber };
```

□ We can rewrite the previous function as:

```
let userDetailFun = (name: string, uid: stringOrNumber) => {
  console.log(`The name is ${name} and uid is ${uid}`);
};
let anotherUserDetailFun = (user: objUserAliases) => {
  console.log(
    `The user name is ${user.name}, age is ${user.age} and uid is
${user.uid}`
  );
};
anotherUserDetailFun({ name: "Abhi", age: 15, uid: 2455636773 });
```

Classes in TypeScript

□ TypeScript offers full support for the class keyword introduced in ES2015.

□ We can define a class as:

```
class Test {}
let myData: Test;
```

□ A class can contain ***fields***, ***methods***, ***constructor***.

□ **Field:** A field declaration creates a public writeable property on a class.

□ **Constructor:** Class constructors are very similar to functions. You can add parameters with type annotations, default values.

□ **Method:** A function property on a class is called a method. Methods can use all the same type annotations as functions and constructors.

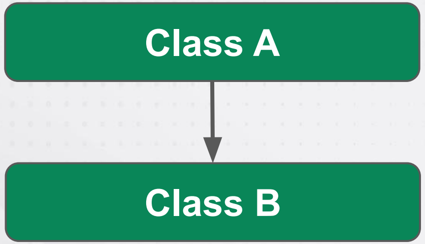
Class Example

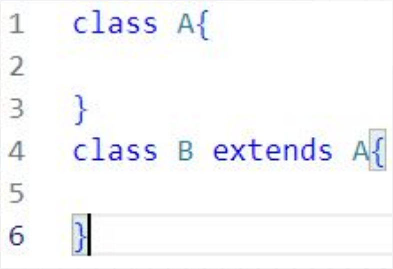
```
class Student {
 name: string;
 branch: string;
 age: number;
 marks: number;
 isActive: boolean;
 address!: String;
constructor(
  name: string,
  branch: string,
  age: number,
  marks: number,
  isActive: boolean
 ) {
  this.name = name;
  this.branch = branch;
  this.age = age;
  this.marks = marks;
  this.isActive = isActive;}
```

```
displayInfo() {
  console.log(`Name is ${this.name}, branch
is ${this.branch}, age is ${
   this.age
  }
  marks are ${this.marks} and status is ${
   this.isActive ? "Active" : "not Active"
  } address is ${this.address}`);
 }
}
let studentOne: Student;
let studentTwo: Student;
studentOne = new Student("Abhinav", "CSE", 25,
87, true);
studentTwo = new Student("Aman", "IT", 20, 91,
false);
studentOne.displayInfo();
studentTwo.displayInfo();
```

Inheritance in TypeScript

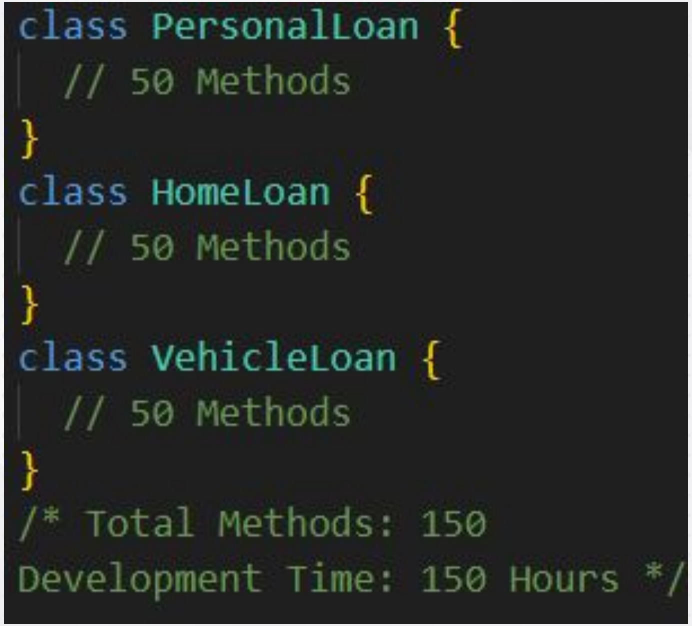
- Inheritance is acquiring all the variables and methods from one class to another class.
- It helps to reuse the code and establish a relationship between different classes.
- A class which inherit the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.
- TypeScript classes can be extended to create a new class with inheritance using the keyword extends.



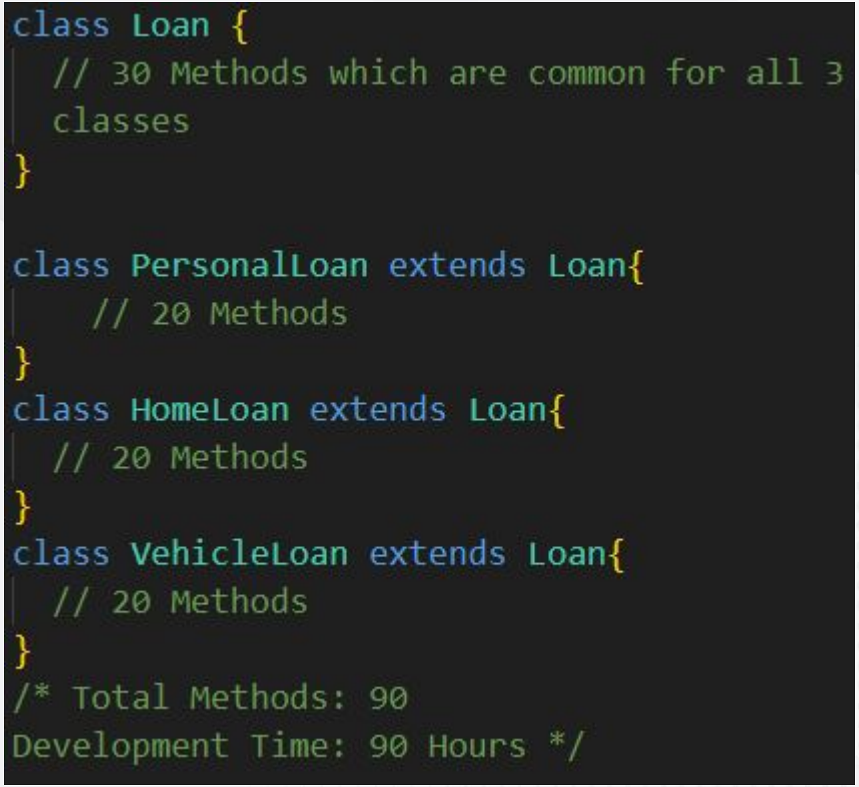


```
class A{

}
class B extends A{

}
```

Advantages of Inheritance



```
class PersonalLoan {
  // 50 Methods
}
class HomeLoan {
  // 50 Methods
}
class VehicleLoan {
  // 50 Methods
}
/* Total Methods: 150
Development Time: 150 Hours */
```



```
class Loan {
  // 30 Methods which are common for all 3
  classes
}

class PersonalLoan extends Loan{
    // 20 Methods
}
class HomeLoan extends Loan{
  // 20 Methods
}
class VehicleLoan extends Loan{
  // 20 Methods
}
/* Total Methods: 90
Development Time: 90 Hours */
```

Example of Inheritance

```
class Person {
  name: string;
  uid: number;
  constructor(name: string, uid: number){
    this.name = name;
    this.uid = uid;   }
  displayBasicInfo() {
    console.log(`The name is ${this.name}
and uid is ${this.uid}`);
  } }
class Employee extends Person {
  empId: string;
  empSalary: number;
  constructor(name: string, uid: number,
empId: string, empSalary: number) {
    super(name, uid);
    this.empId = empId;
    this.empSalary = empSalary;  }
```

```
  displayBasicInfo() {
    console.log(
      `The name is ${this.name}
and uid is ${this.uid},
employee id is ${this.empId}
and salary is
${this.empSalary}`
    );
  }
}
let emp = new Employee("Mukesh
Singh", 647578785,
"LNXIND00001", 57000);
let person = new Person("Aman
Verma", 8657657775);
emp.displayBasicInfo();
person.displayBasicInfo();
```

Access Modifiers in TypeScript

- Access modifiers are used to define the accessibility of properties and methods.
- In TypeScript we have three access modifiers:
  - Private
  - Public
  - Protected
- In TypeScript, we can also define a property as read only by using ***readonly*** keyword.
- ***Private*** properties and methods are accessible only inside the class in which they are declared.
- ***Public*** properties and methods are accessible from anywhere.
- ***Protected*** properties and methods are accessible only inside the class in which they are declared and inside the derived class.

Private Access Modifiers in TypeScript

```
class Student {
  private name: string;
  private age: number;
  constructor(name: string, age:
number) {
    this.name = name;
    this.age = age;
  }
  getName() {return this.name;}
  getAge() {return this.age;}
  setName(name: string) {
    this.name = name;
  }
  setAge(age: number) {
    this.age = age;
  }
}
```

```
let studentOne = new
Student("Abhi", 21);
let studentTwo = new
Student("Manoj", 25);
studentTwo.setName("Abhay");
console.log(studentOne.getName()
);
console.log(studentTwo.getName()
);
```

□ Private properties are accessible inside the class only.

□ We can access the private properties outside the class by using getter and setter.

Private Access Modifiers with Method

```
class Student {
  private name: string;
  private age: number;
  constructor(name: string, age: 
number) {
    this.name = name;
    this.age = age;
  }
  displayInfo() {
    let nameLength = 
this.getNameLength();
    console.log(
      `Student name is ${this.name} 
and age is ${this.age}and the name 
length is ${nameLength}`
    );
  }
  private getNameLength() {
    return this.name.length;
  }
}
let studentOne = new 
Student("Abhi", 21);
let studentTwo = new 
Student("Manoj", 25);
studentOne.displayInfo();
studentTwo.displayInfo();
```

□ Here ***getNameLength()*** method is private so we can not access this method from outside the class.

Public Access Modifiers in TypeScript

```
class Student {
  public name: string;
  age: number;
  constructor(name: string, age:
number) {
    this.name = name;
    this.age = age;
  }
  displayInfo() {
    let nameLength =
this.getNameLength();
    console.log(
      `Student name is
${this.name} and age is
${this.age}and the name length is
${nameLength}`
    );
  }
public getNameLength() {
    return this.name.length;
  }
}
let studentOne = new Student("Abhi", 21);
let studentTwo = new Student("Manoj", 25);
studentOne.displayInfo();
console.log(studentTwo.getNameLength());
```

□ Public properties are accessible from anywhere.

□ If we do not specify any access modifier with property then it is always public by default..

Protected Access Modifiers in TypeScript

```
class Person {
  protected name: string;
  constructor(name: string) {
    this.name = name;
  }
}
class Employee extends Person {
  empId: string;
  constructor(name: string,
empId: string) {
    super(name);
    this.empId = empId;
  }
```

```
 displayBasicInfo() {
    console.log(`The name is
${this.name} and employee id is
${this.empId}`);
  }
}
let emp = new Employee("Mukesh
Singh", "LNX308840");
emp.displayBasicInfo();
```

□ Protected properties are accessible within the same class and inside the child class.

Read Only Properties in TypeScript

```
class Person {
  readonly name: string;
  constructor(name: string) {
    this.name = name;
  }
}
class Employee extends Person {
  empId: string;
  constructor(name: string,
empId: string) {
    super(name);
    this.empId = empId;
  }
```

```
  displayBasicInfo() {
    console.log(`The name is
${this.name} and employee id is
${this.empId}`);
  }
}
let emp = new Employee("Mukesh
Singh", "LNX308840");
emp.name = "test";
emp.displayBasicInfo();
```

```
Cannot assign to 'name' because
it is a read-only property.
```

- If a property is read only then you can not modify the values of that property.

Static Members in TypeScript

□ The static members of a class are accessed using the class name and dot notation, without creating an object.

□ The static members can be defined by using the keyword static.

□ Static variables exist within the class context, and are not carried forward to the object of the class.

```
class Person {
  static objCount: number = 0;
  constructor(public name:
string, public age: number) {
    Person.objCount += 1;
  }
  static displayCount() {
    console.log(`The object
count is ${Person.objCount}`);
  }
}
let p = new Person("Mohit", 35);
let p1 = new Person("Abhay",
30);
Person.displayCount();
```

Abstract Class in TypeScript

- □ Define an abstract class in Typescript using the abstract keyword.
- □ Abstract classes are mainly for inheritance.
- □ We cannot create an instance of an abstract class.
- □ An abstract class typically includes one or more abstract methods or property declarations.
- □ The class which extends the abstract class must define all the abstract methods.

Abstract Class in TypeScript

```
abstract class Person {
  constructor(readonly name: string,
readonly age: number) {}
  displayPersonInfo() {
    console.log(`Person name is
${this.name} and age is
${this.age}`);
  }
  abstract displayDetails(): void;
}
class Employee extends Person {
  constructor(
    readonly name: string,
    readonly age: number,
    private empId: string
  ) {
    super(name, age);
  }
displayDetails(): void {
    console.log(
      `Employee name is
${this.name}, age is ${this.age}
and employee id is ${this.empId}`
    );
  }
}
let emp = new Employee("Mohit",
20, "te002993");
emp.displayDetails();
```

Interface in TypeScript

□ Interfaces in TypeScript allows us to apply certain structures on a class or object.

□ We use ***interface*** keyword to define an interface.

```
interface IsPerson {
  name: string;
  age: number;
  speak(text: string): void;
  spend(amount: number): number;
}
const me: IsPerson = {
  name: "Mohit",
  age: 37,
  speak(text: string): void {
    console.log(text);
  },
  spend(amount: number): number {
    console.log(`Spent amount
${amount}`);
    return amount;
  },
};
console.log(me);
```

Interface with Classes in TypeScript

- Use ***implements*** keyword to implement an interface.

```
interface HasFormater {
  name: string;
  age: number;
  uid: string;
  format(): string;
}
class Person implements
HasFormater {
  constructor(
    readonly name: string,
    readonly age: number,
    readonly uid: string
  ) {}
```

```
format(): string {
    return `Name:
${this.name.toUpperCase()}
      Age: ${this.age}
      Uid:
${this.uid.toUpperCase()}`;
  }
}
let p: HasFormater = new
Person("Mohit", 37,
"canuid65789");
console.log(p.format());
```

Organizing Code

□ Create two folders: ***public*** and ***src*** inside your project.

□ Put the **HTML**, **CSS** and **JS** file inside the public folder.

□ Put TS file inside ***src*** folder.

□ Now create a ***tsconfig.json*** file using command ***tsc - -init***

□ Now inside ***tsconfig.json*** file change `"outDir": "./public"` and `"rootDir": "./src"`

□ Now run the command ***tsc -w*** to compile the code in watch mode.

□ If you do not want to compile the files automatically that are outside the ***src*** folder then add one more property to ***tsconfig.json*** file.

```
"include": ["src"]
```

Modules in TypeScript

- In TypeScript, any file containing a top-level import or export is considered a module.
- Modules are executed within their own scope, not in the global scope.
- Variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported.
- To consume a variable, function, class, interface, etc. exported from a different module, it has to be imported.

Modules in TypeScript

- Open ***tsconfig.json*** file and find the property `module` and change to `"module": "ES2015"`.
- Goto html file and set the type property as module

```
<script type="module" src="app.js"></script>
```

- Suppose I want to put the Person class in a module.
- Create a new folder named ***classes*** inside the ***src*** folder and put a file named ***person.ts*** inside this class folder.

```
export class Person {
  constructor(public name: string, public uid: string) {}
  displayInfo() {
    console.log(`Person name is ${this.name} and uid is
${this.uid}`);
  }
}
```

Modules in TypeScript

- Now I want to use this Person class inside the ***app.ts*** file.
- First import this file inside the ***app.ts*** file.

```
import { Person } from "./classes/person.js";


let p = new Person("Test Name", "uid-001");
p.displayInfo();
```

- While importing the file use ***<file-name>.js*** not ***<file-name>.ts.***