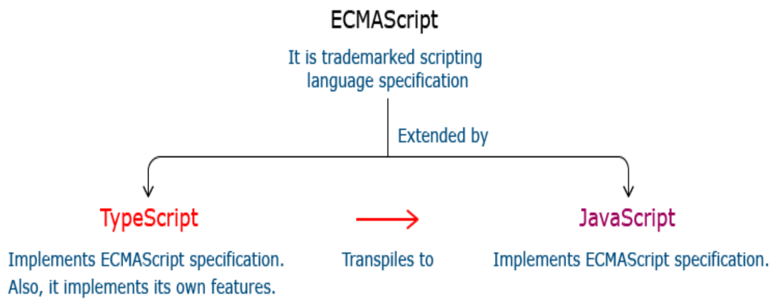
Contents

1. Introduction

1.1 What is TypeScript?

TypeScript can be considered as a typed superset of JavaScript, that transpiles to JavaScript.

- Transplier converts the source code of one programming language to the source code of another programming language.
- TypeScript makes the development of JavaScript more of a traditional object-oriented experience.
- TypeScript is based on ECMAScript 6 and 7 proposals.
- Any valid JavaScript is TypeScript.


ECMAScript
It is trademarked scripting language specification
Extended by
TypeScript
Implements ECMAScript specification. Also, it implements its own features.
Transpiles to
JavaScript
Implements ECMAScript specification.

1.2 Features of TypeScript

- Static Typing: It adds static typing to JavaScript, due to which the readability of the code improves and helps in finding more early compilation errors than run time errors.
- Modules support: TypeScript provides an option to create modules to modularize the code for easy maintenance. Modules help in making the application scalable.
- Object-Oriented Programming: TypeScript supports Object-Oriented Programming features such as class, encapsulation, interface, inheritance and so on which helps in creating highly structured and reusable code.
- Open Source: TypeScript is open source. The source code of TypeScript can be downloaded from Github.
- Cross-Platform: It works across the platform.
- Tooling Support: TypeScript works extremely well with Sublime Text, Eclipse, and almost all major IDEs compared to JavaScript.
- Decorator Support: Decorator is used for providing metadata and they are used to specify extra behavior of a class, method, or property of a class.

1.3 Why TypeScript?

Pitfalls of JavaScript:

- Dynamic Typing: It decides the data type of the variable dynamically at run time.
- Interpreted Language: It is a language in which the code instructions are executed directly without prior compilation to machine-language instructions.
- Minimal Object Oriented support: Object-Oriented Programming (OOP) is a programming methodology based on the concept of objects. Object-Oriented concepts like classes, encapsulation, inheritance help in the readability and reusability of the code.
- Minimal IDE support: Integrated Development Environment (IDE) is a software application that provides all necessary options like code refactoring, intelliSense support, debugging support to software programmers for software development.

As the complexity of the JavaScript code increases, it gradually becomes difficult in coding and maintaining. This is because of the limitations of the JavaScript language. There is no option to change the language for the client-side scripting as the browser understands only JavaScript.
The solution is to choose a language that is rich in features and the code can be converted to JavaScript. This process of converting the code written in one language into another language is called Transpilation. TypeScript is one such language where its code can get transpiled to JavaScript.

1.4 Installing TypeScript

npm is a command-line tool that comes along with Node.js installation with which you can download node modules. TypeScript is also such a node module that can be installed using npm.

Open a Node.js command prompt and use the command to download the TypeScript module from the npm repository.

```
npm i –g typescript
```

```
tsc -v
//or
tsc --version
```

To complie a ts file

```
tsc filename.ts
```

After compilation of the TypeScript file, the corresponding JavaScript file gets generated

1.5 Project Configuration

Project configuration in TypeScript is used to set the compiler options and helps in specifying the files to be included or excluded while performing the compilation.

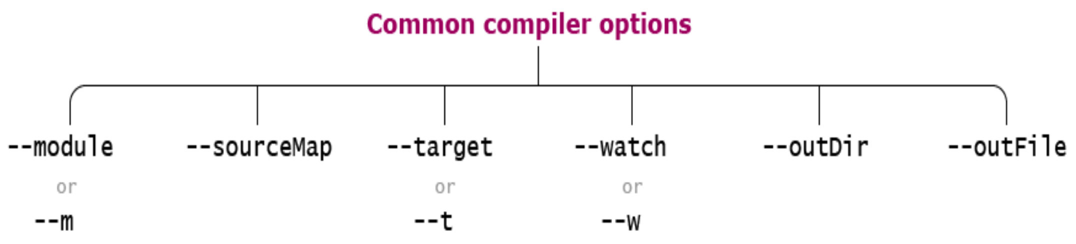
TypeScript project configuration can be done using one of the below approaches:

- tsc: With command line along with the tsc command
- IDE: Setting options using the IDE with which you have coded TypeScript
- Build Tool: Using any task builder like Grunt or Gulp
- tsconfig.json: Inside a file with the name tsconfig.json

Compiler option is used to specify configurations like target ES version to be used to compile, module loader to be used, and so on.

There are many compiler options available which you can refer to from the TypeScript documentation.

Some of the Common compiler options used are:



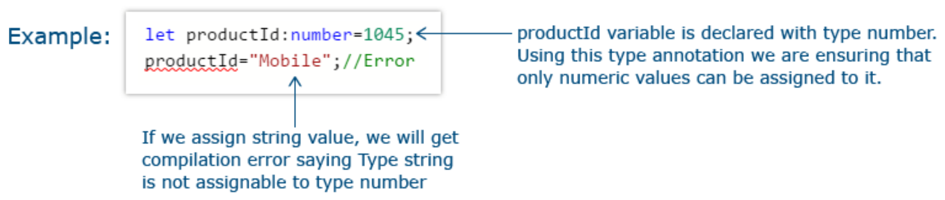
Option	**Description**	**Example**
--target	A developer can specify ECMAScript target version:'es3'(default),'es5', or 'es6'	tsc --target ES2015 filename.ts
--module	A developer can specify module code generation: 'none','commonjs', 'amd', 'system', 'umd', 'es6', 'es2015', or ‘es2022’	tsc --module commonjs filename.ts
--outDir	A developer can redirect output structure to the directory	tsc --outDir foldername filename.ts
--outFile	Helps in concatenation, and emits the output to a single file. Order of concatenation is determined by the list of files passed to the compiler on the command line along with triple-slash references and imports	tsc --outFile foldername filename.ts
--sourceMap	Generates corresponding '.map'file which is used to perform debugging.	tsc --sourceMap filename.ts
--watch	Run the compiler in watch mode. Watch input files and trigger recompilation on changes.	tsc --watch filename.ts

It is used to provide compiler options to a TypeScript project
It helps in specifying the files to be included or excluded from the project
You can also find the root of a TypeScript application using the tsconfig.json file as this file resides in the root folder.

2. TypeScript Basics

2.1 Type Annotations

Type Annotation is a way to enforce type restriction to a specific variable or a function.
If a variable is declared with a specific data type and another type of value is assigned to the variable, a compilation error will be thrown.

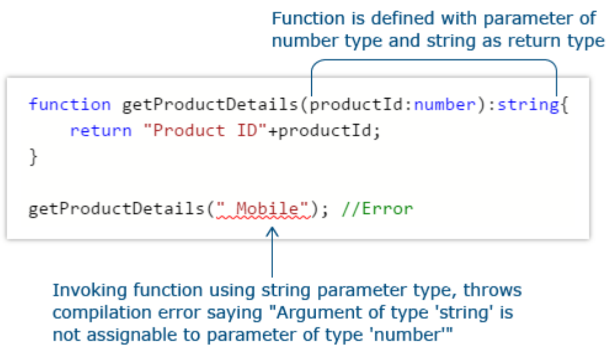


Example:

```
let productId:number=1045;
productId="Mobile";//Error
```

productId variable is declared with type number. Using this type annotation we are ensuring that only numeric values can be assigned to it.

If we assign string value, we will get compilation error saying Type string is not assignable to type number



Function is defined with parameter of number type and string as return type

```
function getProductDetails(productId:number):string{
    return "Product ID"+productId;
}

getProductDetails(" Mobile"); //Error
```

Invoking function using string parameter type, throws compilation error saying "Argument of type 'string' is not assignable to parameter of type 'number'"

2.2 Basic DataTypes

2.2.1 boolean

boolean is a data type that accepts only true or false as a value for the variable it is been declared.
In a shopping cart application, you can use a boolean type variable to check for the product availability, to show/hide the product image, etc.

```
let showImage:boolean = true;
```

2.2.2 number

number type represents numeric values. All TypeScript numbers are floating-point values.
In a shopping cart application, you can use number type to declare the Id of a product, price of a product, etc.

```
let productId:number = 1045;
```

2.2.3 string

A string is used to assign textual values or template strings. String values are written in quotes – single or double.
In a mCart application, you can use string type to declare variables like productName, productType, etc.

```
        let productName:string="Samsung Galaxy J7";
```

Template strings are types of string value that can have multiple lines and embedded expressions.
They are surrounded by the backquote\backtick (`) character and embedded expressions of the form ${expr}.

```
let productName:string = "Television";
let message:string = `The product name is ${productName}`;
let catalog = `The products in the catalog are
                                TV
                                Refigerator
                                Airconditioner
                                Geyser`;
console.log(catalog);
```

2.2.4 any

any type is used to declare a variable whose type can be determined dynamically at runtime with the value supplied to it. If no type annotation is mentioned while declaring the variable, any type will be assigned by default.

In a mCart application, you can use any type to declare a variable like screenSize of a Tablet.

```
let screenSize:any;
screenSize=13.97;
screenSize="5.5-inch";
```

2.2.5 void

void type represents undefined as the value. the undefined value represents "no value".
A variable can be declared with void type as below:

```
        let product:void=undefined;
```

In the above example , variable declared with void datatype. It is not preferred as we can assign only undefined or null as value.

void type is commonly used to declare function return type. The function which does not return a value can be declared with a void return type. If you don’t mention any return type for the function, the void return type will be assigned by default.

```
function displayProductDetails():void{
console.log("Product category is Gadget");
}
```

2.2.6 never

Never is a new data type introduced in TypeScript in the year 2016. Never data type indicates that the situation will never occur. Never can be used in functions that run in infinite loop or that in which the end of function is not reached at all.

Consider a function which executes recursively as shown below:

```
function sayHello() {
  while (true) {
    console.log(Hello World')
  }
}
```

Consider the below function Demo(), which accepts string parameter named eMsg as shown in Line no 1.

This function accepts string input and throws it as an error message but never reaches the end of the function. Hence, we can add the never return type to it.

```
function Demo(msg: string): never {
throw new Error(msg);
}
```

The difference between void and never is that never does not take null and undefined as values, but void does take null and undefined as values.

```
let test: never = null; //error
let demo: never = undefined; //error
let test: void = null;
let demo: void = undefined;
```

2.2.7 ReadOnly

"Readonly" keyword can be used with Class declaration, type or interface in any TypeScript application development.

- These members are normally assigned within the constructor of the Class.
- They cannot be modified outside the class, but you access them outside the Class.

```
class Student {
readonly Id: number;
Name: string;
constructor(code: number, name: string)     {
this.Id = code;
this.Name = name;
}
}
let student  = new Student(1, "Mike");
student.Id = 4; //Compiler Error
student.Name = 'Bill';
```

```
interface IStudent {
readonly Id: number;
Name: string;
     }
let studentObj:IStudent = {
Id:1,
Name:"Mark"
}
studentObj.Id = 9; // Compiler Error: Cannot change readonly 'Id'
```

2.3 Enums

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript. Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

By default, enum's first item will be assigned with zero as the value and the subsequent values will be incremented by one.

In the mCart application, you can use a MobilePrice enum to store different prices of the mobile depending on the mobile color.

```
        enum MobilePrice {Black, Gold, White};
```

In the above example, value of first item will be 0 (default value) and subsequent items will have sequential increment from first value.

```
To get the value from an enum use one of the following:
//Syntax EnumName.item
MobilePrice.Black
//Syntax EnumName["item"]
MobilePrice["Black"]
```

Enum items can also be initialized with different values than the default value. If you set a different value for one of the variables, the subsequent values will be incremented by 1.

```
//Initial value is set as 25000, so subsequent values will be 25001 and 25002 respectively
enum MobilePrice {Black=25000, Gold, White};
```

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}
```

2.4 Array

An Array is used to store multiple values in a single variable. You can easily access the values stored in an array using the index position of the data stored in the array.
TypeScript array is an object to store multiple values in a variable with a type annotation. They are like JavaScript arrays.
Arrays can be created using one of the below:

- Using datatype[] declaration:

```
//String array is created using string[] declaration
let manufacturers:string[] = ["Samsung","Apple","Sony"];
```

- Using Array<type> declaration

```
//String array is created using Array<type> declaration
let manufacturers : Array<string> = ["Samsung","Apple","Sony"];
```

- Using any[] declaration

```
//It accepts any type of data
let products : any[] = ["Mobile",12500,true];
```

A TypeScript array defined with a specific type does not accept data of different types. TypeScript arrays can be accessed and used much like JavaScript arrays.

2.5 Tuples

A tuple type is another sort of Array type that knows exactly how many elements it contains, and exactly which types it contains at specific positions.

```
//customerCreditInfo tuple with 3 different types of data
var customerCreditInfo:[string, Customer, number];
customerCreditInfo =["I342", new Customer("I342"),3000];
```

In TypeScript, now, tuple types can have labels as mentioned below:

```
        type Progress = [start: number, end: number];
```

You can have Labeled Tuple along with rest parameters and optional elements as below:

```
        type Demo = [one: number, two?: string, ...rest: any[]];
```

Rules for using labeled tuples as below :

All elements in the tuple must be labeled otherwise the code would throw error as shown below.

```
        type Test = [first: string, number];//error
```

Labels used while destructuring doesn’t require variables to be named differently as shown below.

```
function Demo(x: [first: string, second: number]) {
   // we need not name these parameters as ‘first’ and 'second'
   const [a, b] = x;
```

2.6 Union Types

TypeScript’s type system allows you to build new types out of existing ones using a large variety of operators.
Now that we know how to write a few types, it’s time to start combining them in interesting ways.

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}
// OK
printId(101);
// OK
printId("202");
// Error
printId({ myID: 22342 });
```

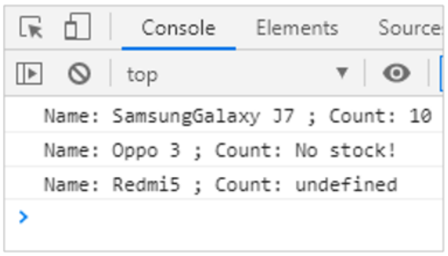
2.7 Type Aliases

Type Aliases is a custom or hand-written name provided to an existing data type of Typescript.
The alias names can be used for several types including simple, union, tuple, or other existing primitive types. This feature makes the declaration clear and easy.

```
type MeaningfulName = < an existing Typescript type>
```

Let us try to create a type to define the pCount parameter of getProductDetail function.

```
type ProdCountType = number | string | undefined;
function getProductDetail(pName: string, pCount: ProdCountType){
   console.log(`Name: ${pName} ; Count: ${pCount}`)
}
getProductDetail("SamsungGalaxy10", 2);
getProductDetail("LenovoNote3", "No stock!");
getProductDetail("Redmi5", undefined);
```



```
Name: SamsungGalaxy J7 ; Count: 10
Name: Oppo 3 ; Count: No stock!
Name: Redmi5 ; Count: undefined
```

- Simple Types

The Developer can add a custom name for the simple data type.
In the below code the ProductIdNum variable can accept only a number data type only. The type has been used to add a custom name.

```
type ProductIdNum = number;
function getProduct(id: ProductIdNum) {
   console.log(`Product found for id ${id}`)
}
getProduct(1002);
getProduct(null); // Error: argument type is not a number
```

- Union Types

If the developer needs to hold more than one data type, then union types can be used for this requirement.
In the below code, the ProdCountType can hold a number, string, or undefined values.

```
type ProdCountType = number | string | undefined;
function getProductDetail(pName: string, pCount: ProdCountType){
   console.log(`Name: ${pName} ; Count: ${pCount}`)
}
getProductDetail("SamsungGalaxy10", 2);
getProductDetail("LenovoNote3", "No stock!");
getProductDetail("Redmi5", undefined);
```

- Function Types

If the developer needs to define a name for a function definition, then Function types can be used.
In the below code a function (PromoCodeGenerator) is defined with a set of parameters and return type to specify a type.

```
type PromoCodeGenerator = (pName: string, pId: number) => string;
let generator: PromoCodeGenerator = function (pName: string, pId: number): string {
   return pName.substr(0, 4).toLocaleUpperCase() + pId;
}
let offer = generator("Lenovo 3", 1002);
console.log(`Please use the Promocode: ${offer}`)
```

- Object Types

Object types help the developer to enforce Type Checking on the object without interfaces. This feature benefits developers in writing interface independent object type.

```
type ProductStatusObject = { pId: string, availablity: boolean }
function checkProductAvailablity(product: ProductStatusObject) {
   if (product.availablity == true) {
      console.log(`${product.pId} is available`);
   }
   else {
       console.log(`${product.pId} is not available`);
   }
}
let product1 : ProductStatusObject = {pId: "LENOVO9089", availablity: true}
checkProductAvailablity(product1);
let product2 : ProductStatusObject = { availablity: false, pId: "SAMSUN8811"}
checkProductAvailablity(product2);
```

- Array Types

It helps developers to create an array of dissimilar records.
In the below code an array (AvailabiltyArray) is created for checking the availability of the product.

```
type ProductStatusObject = { pId: string, availablity: boolean }

type AvailabiltyArray = [ ProductStatusObject ]

let product1: ProductStatusObject = { pId: "LENOV9912", availablity: true  };
let product2: ProductStatusObject = { pId: "REDMI8878", availablity: false };
let product3: ProductStatusObject = { pId: "SAMSU5633", availablity: false }
let product4: ProductStatusObject = { pId: "OPPO1128",availablity: true}

let products: AvailabiltyArray = [product1];
products.push(product2,product3, product4);

let availableProducts = products.filter( (product) => product.availablity)
console.log(availableProducts)
```

- Generic Types

The developer can also work with generic types that have type alias name.
In the below example ProductCatelog is created to work with different types of product category code.

```
type ProductCatelog<T> = { pCategoryCode: T, availableProducts: Array<T> }
function availableProducts<T>(pCatelog : ProductCatelog<T>) {
console.log(`Products on category code ${pCatelog.pCategoryCode} :
        ${(pCatelog.availableProducts)}`)
}
//ProductCatelog by code number
let productByCodeNumber: ProductCatelog<number>;
productByCodeNumber = {pCategoryCode : 12321, availableProducts :[4566,7788,9090]}
availableProducts(productByCodeNumber)
//ProductCatelog by code string
let productByCodeString: ProductCatelog<string>;
productByCodeString = {pCategoryCode : "PCC2020",
 availableProducts :["Pr4566","Pr7788","Pr9090"]}
availableProducts(productByCodeString)
```

- Type Literals

Literals help a developer to create possible static values as the data to a variable.
In the below example ratings variable is defined with possible union types.

```
type ratings = "Good" | "Average" | "Excellent" | 0;
let customerFeedback1: ratings = "Good";
let customerFeedback2: ratings = 0;
let customerFeedback3: ratings = "Not bad";
// Error: Could not find literal value match.
```

2.8 Type Assertions

Sometimes you will have information about the type of a value that TypeScript can't know about.
For example, if you're using document.getElementById, TypeScript only knows that this will return some kind of HTMLElement, but you might know that your page will always have an HTMLCanvasElement with a given ID.
In this situation, you can use a type assertion to specify a more specific type:

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

Like a type annotation, type assertions are removed by the compiler and won't affect the runtime behavior of your code.

You can also use the angle-bracket syntax (except if the code is in a .tsx file), which is equivalent:

```
const myCanvas = <HTMLCanvasElement>document.getElementById("main_canvas");
```

TypeScript only allows type assertions which convert to a more specific or less specific version of a type. This rule prevents "impossible" coercions like:

```
const x = "hello" as number;
Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently
overlaps with the other. If this was intentional, convert the expression to 'unknown' first.
```

2.9 Type Narrowing

TypeScript follows possible paths of execution that our programs can take to analyze the most specific possible type of a value at a given position. It looks at these special checks (called type guards) and assignments, and the process of refining types to more specific types than declared is called narrowing

2.9.1 typeof

As we've seen, JavaScript supports a typeof operator which can give very basic information about the type of values we have at runtime. TypeScript expects this to return a certain set of strings:

```
function printAll(strs: string | string[] | null) {
  if (typeof strs === "object") {
    for (const s of strs) {
'strs' is possibly 'null'.'strs' is possibly 'null'.
      console.log(s);
    }
  } else if (typeof strs === "string") {
    console.log(strs);
  } else {
    // do nothing
  }
}
```

2.9.2 Truthiness Narrowing

Truthiness might not be a word you'll find in the dictionary, but it's very much something you'll hear about in JavaScript.

In JavaScript, we can use any expression in conditionals, &&s, ||s, if statements, Boolean negations (!), and more. As an example, if statements don't expect their condition to always have the type boolean.

```
function getUsersOnlineMessage(numUsersOnline: number) {
  if (numUsersOnline) {
    return `There are ${numUsersOnline} online now!`;
  }
  return "Nobody's here. :(";
}
```

2.9.3 Equality Narrowing

TypeScript also uses switch statements and equality checks like ===, !==, ==, and != to narrow types. For example:

```
function example(x: string | number, y: string | boolean) {
  if (x === y) {
    // We can now call any 'string' method on 'x' or 'y'.
    x.toUpperCase();

(method) String.toUpperCase(): string
    y.toLowerCase();

(method) String.toLowerCase(): string
  } else {
    console.log(x);

(parameter) x: string | number
    console.log(y);

(parameter) y: string | boolean
  }
}
```

2.9.4 IN Operator Narrowing

JavaScript has an operator for determining if an object or its prototype chain has a property with a name: the in operator. TypeScript takes this into account as a way to narrow down potential types.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    return animal.swim();
  }

  return animal.fly();
}
```

2.10 Type Manipulation

TypeScript's Type Aliasing feature helps you to define the several types like variables, functions, objects, and return parameters.
To implement this concept, TypeScript provides you with various type operators.

2.10.1 keyof operator

This operator accepts an object type as a parameter and generates numeric or string literal union keys. It helps you to get all the keys of the object in a type and returns the union type of that object.

For example, in the below code, type Mobile would hold "string" | "number."

```
type Product = { pName: string; pCount: number};
type Mobile = keyof Product;
```

This operator can be used with primitive data types like any type, enumerated and Generics constraint types as shown below:

Example 1 : any type

```
type no1 = keyof any;
let x1:no1;
x1="hello"; // hello
x1=123; // 123
x1= true;//error
```

In this code you can observe that x1 accepts only string and number values and while assigning boolean value the code throws an error.

Example 2 : enumerated type

```
enum ProductColors {
    white = '#ffffff',
    black = '#000000',
}
type Color = keyof typeof ProductColors; // 'white' | 'black'
let pColor: Color
pColor = "white" // white
pColor = "black" // black
pColor = "gold"  // error
```

In this code snippet,

Line no 4 helps you in creating enumerated type variable named Color.
Line no 9 throws error as you have non-enumerated value.

Example 3 : Generics constraint types

```
function getProductDetails<T, K extends keyof T>
 (object: T, key: K) {
 let propertyValue = object[key];
 console.log(`${String(key)} = ${propertyValue}`);
}
let Mobile = {
 pName:"Samsung Galaxy", pCount:2
}
getProductDetails(Mobile, "pName");//"pName = Samsung Galaxy"
getProductDetails(Mobile, "pColor");//error
```

In the above code, Line no 1 a Generic constraint function named getProductDetails has been implemented to populate the values.

2.10.2 typeof operator

This operator allows you to add a reference to the type of property or variable used in the application development.
For example, when you declare a pName as mentioned below.

```
let pName = "Samsung Galaxy";
let mobileName: typeof pName;
```

On mousehover of second statement of the above given code-snippet in any TypeScript application supporting editor, a message “let mobileName: string” would be automatically populated on your screen.

Using typeof operator individually would not be helpful for you, but when it is combined with other type operators, you can easily use different patterns.

For example, if you require creating a predefined String data type named Sample using function type then you can use the below-given sample.

```
type CheckValue = (x: unknown) => String;
type Sample = ReturnType<CheckValue>;
let check: Sample ="hello";
console.log(check);
```

Please note that in the above code ReturnType<T> used is a predefined type that is available in TypeScript. If the same ReturnType<T> is used with function name, the code would throw an error.
You can observe the error while implementing the below-given code.

```
function productDetails() {
   return { pName:"Samsung", pCount: 2};
  }
  type P = ReturnType<productDetails>; //error
```

To overcome this error, you should use typeof operator as mentioned below.

```
function productDetails() {
   return { pName:"Samsung", pCount: 2};
  }
type P = ReturnType<typeof productDetails>; // type P ={ pName:"Samsung", pCount: 2}
```

Note : typeOf operator should be used only with variable declaration or properties, that helps you in debugging the code easily with help of any TypeScript supporting editor Intellisense feature.

If you have the requirement to refer to a specific property existence of another type, then indexed access type can be used.

For example, if you want to validate whether pCount property exists as part of the ProductDetails type then indexed access type can be implemented in your code as shown below:

```
type ProductDetails ={pName: "Samsung", pCount: 2};
type Pcount= ProductDetails["pCount"];// type Pcount = 2
```

If you try to validate a non-existing property, the indexed access type will automatically throw error as shown below.

```
        type Pprice = ProductDetails["pPrice"]; //Property pPrice does not exist on type ProductDetails
```

This operator can be used with other operators like keyof, unions, other types, and arbitrary types.

Example 1: indexed access type with keyof

```
type ProductDetails = {pName: "Samsung", pCount: 2};
type Mobile1 = ProductDetails[keyof ProductDetails];
// type Mobile1 = Samsung | 2
```

Example 2: indexed access type with union types

```
type ProductDetails = {pName: "Samsung", pCount: 2};
type Mobile2 = ProductDetails["pName" | "pCount" ];
// type Mobile2 = Samsung | 2
```

Example 3: indexed access type with other types

```
type ProductDetails = {pName: "Samsung", pCount: 2};
type GadgetCategory = "pName" ;
type Gadget = ProductDetails[GadgetCategory]
//type Gadget = “Samsung”
```

Arbitrary type using Indexed accessed type helps you in type of array elements using number data type. This concept can be achieved using typeof operator.

Consider the below code snippet:

```
//Arbitrary type
const MyProduct = [
   {pName:"Samsung", pCount:2},
   {pName: "Apple iphone", pCount:3}
];
type P1 = typeof MyProduct[number]; // type P1 = { pName: string, pCount: number}
type ProductCount = typeof MyProduct[number]["pCount"]; // type Productcount = number
```

In this code snippet, you can observe that

- P1 is created using number data type with typeof operator
- Productcount is created specifically using pCount property of MyProduct object with the usage of number and typeof operator.

Note: Indexing concept cannot be used with const keyword.

2.10.3 Conditional Types

Conditional types help you in predicting the relation between the input and output parameters.
To implement this concept, use syntax like Conditional (Ternary) expression but with types parameters.

Conditional types help you in predicting the relation between the input and output parameters.

To implement this concept, use syntax like Conditional (Ternary) expression but with types parameters.

For example,

```
type Product = { productName: string };
type Category = { categoryName: string };
type ProductDetails = { productName: string, productId: number };
type CategoryDetails = { categoryName: string, categoryType: string };
let product1: ProductDetails ={ productName: 'Samsung', productId: 123};
let productCategory:CategoryDetails = { categoryName: 'Electronics', categoryType: 'Mobiles' };
//Conditional Types implementation
type ExtractDetails<T> = T extends { productName: string } ? Product: Category;
type IsProduct = ExtractDetails<Product> // returns Product type
let item1: IsProduct = product1;
console.log(item1); // { productName: 'Samsung', productId: 123};
type NotProduct = ExtractDetails<CategoryDetails> //returns Category type
let item2: NotProduct = productCategory;
console.log(item2); // { categoryName: 'Electronics', categoryType : 'Mobile'}
```

In this code, you observe that at Line no: 8 a type named ExtractDetails is defined using Generics concept using T parameter, based on type of T this line would return either type Product if extends condition is True or Category type if this condition is False.

To create a reusable conditional type, you can use union type. This concept is called the Distributive conditional type.
By using Distributive conditional type, the same conditional can be iterated over the union members.

For example,

```
        type TypeofProduct = Category |  CategoryDetails| ProductDetails | Product
```

This Line would help you to apply the above-discussed conditional type on members of Category or CategoryDetails or ProductDetails or Product type and it would return the details of the respective type accordingly.

2.10.4 Mapping Types

Mapped types build on the syntax for index signatures, which are used to declare the types of properties which have not been declared ahead of time:

```
type Product = { productName: string};
let p: Product ={ productName: 'Samsung'}
type ProductDetails = {
[key: string]: boolean | Product;
};
const product1: ProductDetails = {
status: true,
productName: p,
productColor:'color' //compile-time error
};
```

You can use keyof keyword while creating Generic Mapped type value.
This concept would help you in iterating over all the property of the Generic type parameter passed value and assign the default value based on your requirement.

```
// Generic Mapped Type which helps in assigning all the properties to boolean value
type Product<Type> = {
[Property in keyof Type]: boolean;
}
type ProductDetails =
{
availableFeatures : string;
productCost : number;
};
type MobileDetails = Product<ProductDetails>;
let mobile1: MobileDetails ={ availableFeatures : true, productCost: true}
console.log(mobile1); // { availableFeatures : true, productCost: true}
```

readonly modifier implementation:

```
// Mapped Modifier with readonly
type Product<T> = {
readonly [P in keyof T]: T[P];
};
type ProductDetails = {
availableFeatures: string;
productColor?: "white" | "black" | "gold";
}
const product1: Product<ProductDetails>  = {
availableFeatures: "Camera",
productColor: "gold"
};
// below line throws compile-error as you are accessing readonly mapped types
product1.availableFeatures="iPhone";
product1.productColor = "black";
```

In this code, Line 14 and 15 would throw compile-error as you are accessing readonly mapped types
To overcome this issue prefix - symbol as shown below at the Line no 3 as shown below

```
type Product<T> = {
 - readonly [P in keyof T]: T[P];
};
```

optional(?) modifier implementation:

```
// Removes Optional attributes because of -? usage
type Product<T> = {
[P in keyof T] -? T[P];
};
type ProductDetails = {
availableFeatures: string;
productColor?: "white" | "black" | "gold";
}
// below line throws compile-error as 'productColor' property declaration is missing
const product1: Product<ProductDetails>  = {
availableFeatures: "Camera",
};
```

In this code, you can observe the Line no 10 would throw compile-error due to usage of -? declaration at Line no 3 enforces you to declare the optional parameters.

3. Functions

A function is a block of statements to perform a particular task.
A sequence of statements written within function forms function body.
Functions are executed when it is invoked by a function call. Values can be passed to a function as function parameters and the function returns a value.
Functions in TypeScript are like functions in JavaScript with some additional features.

	TypeScript	**JavaScript**
Types:	Supports	Do not support
Required and optional parameters:	Supports	All parameters are optional
Function overloading:	Supports	Do not support
Arrow functions:	Supports	Supported with ES2015
Default parameters:	Supports	Supported with ES2015
Rest parameters:	Supports	Supported with ES2015

3.1 Paramteres and return types

The parameter type is the data type of the parameter and the return type is the data type of the returned value from the function.
With the TypeScript function, you can add types to the parameter passed to the function and the function return types.
While defining the function, return the data with the same type as the function return type. While invoking the function you need to pass the data with the same data type as the function parameter type, or else it will throw a compilation error.

```
//Function is defined with parameter of number type and string as return type
function getProductDetails(productId:number):string{
return "Product ID"+productId;
}
//Invoking function using string parameter type,throws compilation error saying "Argument of type
'string' is not assignable to parameter of type 'number'"
getProductDetails("Mobile"); //Error
```

3.2 Optional and default parameters

TypeScript treats every function parameter as mandatory. So when a function is compiled, the compiler will check whether there is a value supplied to all the parameters of the function, or else it will throw a compilation error.

Consider the code below:

```
function getProductDetails(productName:string,productId:number):string{
        return "Product: "+productName + " "+productId;
    }
 let productName:string=getProductDetails("Mobile");//Error
```

In the above example , at line no. 4 we are invoking getProductDetails function with single parameter.This throws compilation error "Supplied parameters don not match any signature of call target" .

In the above example, you have tried to invoke a function with only a single parameter, whereas the definition of the function accepts two parameters. Hence, it will throw a compilation error. Also, optional parameter can be used to tackle this issue.

The Optional parameter is used to make a parameter, optional in a function while invoking the function.

If you rewrite the previous code using an optional parameter, it looks like the below:

```
//Adding ? after parameter name makes parameter , optional
function getProductDetails(productName:string,productId?:number):string{
        return "Product: "+productName + " "+productId;
    }
 let productName:string=getProductDetails("Mobile");
```

An Optional parameter must appear after all the mandatory parameters of a function.

Default parameter is used to assign the default value to a function parameter.
If the user does not provide a value for a function parameter or provide the value as undefined for it while invoking the function, the default value assigned will be considered.
If the default parameter comes before a required parameter, you need to explicitly pass undefined as the value to get the default value.

```
//Clothing is assigned as deafult value to productName parameter
function getProductDetails(productName:string="Clothing",productId:number):string{
        return "Product: "+productName + " "+productId;
    }
    //Access function without default parameter, throws compilation error : "Supplied parameters do not match any signature of call target
 let productName:string=getProductDetails(101);//Error
 //Pass undefined as value for default parameter.Since we have already set default value for the same , function takes that value to process the same.
  let productName:string=getProductDetails(undefined,101);
```

3.3 Rest parameter

The rest Parameter is used to pass multiple values to a single parameter of a function. It accepts zero or more values for a single parameter.
The rest Parameter should be declared as an array.
Precede the parameter to be made as the rest parameter with triple dots.
The rest parameter should be the last parameter in the function parameter list.

```
 //Preceding parameter with triple dots makes it a rest parameter. Rest parameter by default will have
array type declaration
 function getProductDetails(productId:number,...productName:string[]):string{
        return "Product: "+productName + " "+productId;
    }
    //We can pass number of values for the Rest parameter or even leave it //empty
 let productName:string=getProductDetails(101,"Mobile","Furniture");
```

3.4 Arrow function

Arrow function is a concise way of writing a function. Whenever you need a function to be written within a loop, the arrow function will be the opt choice.
Do not use the function keyword to define an arrow function.
In a mCart application, you can use the arrow function to perform filtering, sorting, searching operations, and so on.

Syntax : (parameter)=>function body

Example : Arrow function with number parameter, string return type and function body

```
          var getProductDetails = (productId:number):string=>{return "Product ID"+productId;}
```

3.5 Function Overloads

Typescript Function overloads allow you to define multiple function signatures for a single function, each with different parameter types or return types. Function overloads help provide more accurate type checking and enable TypeScript to infer and enforce the correct types when you call the function in different ways.

```
// Function signature for the first overload
function functionName(parameter1: type1, parameter2: type2): returnType1;
// Function signature for the second overload
function functionName(parameter1: type3, parameter2: type4): returnType2;
// Implementation of the function that matches one of the overloads
function functionName(parameter1: any, parameter2: any): any {
   // Function implementation here
```

```
function greet(name: string): string;
function greet(name: string, age: number): string;

function greet(name: string, age?: number): string {
   if (age === undefined) {
      return `Hello, ${name}!`;
   } else {
      return `Hello, ${name}! You are ${age} years old.`;
   }
}

console.log(greet("Geeks"));
console.log(greet("xyz", 30));
```

4. Classes

Class is a template from which objects can be created.
It provides behavior and state storage.
It is meant for implementing reusable functionality.
Use a class keyword to create a class.

```
class Product{
productId : number;
getProductDetails(productId:number):string{
return "Product ID is"+productId;
}
}
```

4.1 Constructor

A constructor is a function that gets executed automatically whenever an instance of a class is created using a new keyword.
To create a constructor, a function with the name as a "constructor" is used.
A class can hold a maximum of one constructor method per class.
You can use optional parameters with a constructor function as well.

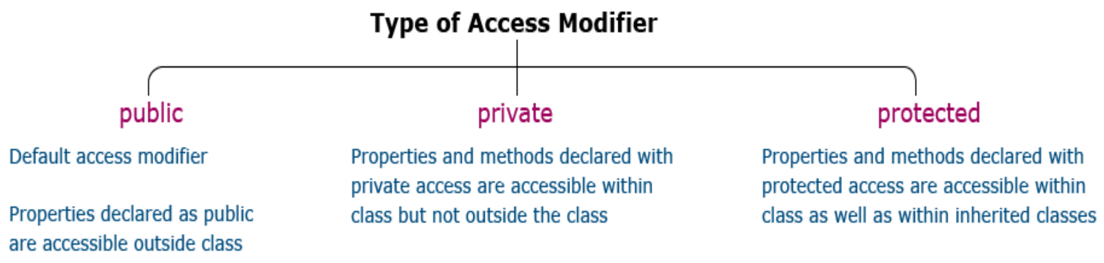
```
class Product{
productId : number;
//Parameterized constructor is used to initialize productId property
constructor(productId:number){
          this.productId = productId;
          }
          getProductId() : string{
          }
}
//Creating instance of class with parameterized constructor
var product:Product = new Product(1234);
```

There are just a few differences between class constructor signatures and function signatures:

- Constructors can't have type parameters - these belong on the outer class declaration.
- Constructors can't have return type annotations - the class instance type is always what's returned

4.2 Access Modifiers

Access modifiers are used to provide certain restriction of accessing the properties and methods outside the class.


Type of Access Modifier
public
Default access modifier
Properties declared as public are accessible outside class
private
Properties and methods declared with private access are accessible within class but not outside the class
protected
Properties and methods declared with protected access are accessible within class as well as within inherited classes

```
class Product{
        //Declaring productId property using public keyword
        public productId : number;
        constructor(productId : number){
            this.productId = productId;
            }
     }
var product : Product = new Product(1234);
//Accessing productId outside class since it is declared using public keyword
console.log(product.productId);
```

```
class Product{
        //Declaring productId property using private keyword
        private productId : number;
        constructor(productId : number){
            this.productId = productId;
            }
    }
var product : Product = new Product(1234);
//Accessing productId outside class.Since it is declared using private keyword it is not accessible
outside class.This line throws compilation error
console.log(product.productId);
```

```
class Product{
 //Declaring productId property using protected keyword
       protected productId : number;
       constructor(productId : number){
           this.productId = productId;
           }
     }
class Gadget extends Product{
       getProduct() : void{
//Accessing productId inside inherited class since it is declared using protected keyword
       console.log("ProductID"+this.productId);
       }
     }
var g:Gadget=new Gadget(1234);
g.getProduct();
```

4.3 Static Members

TypeScript provides an option to add a static keyword. This keyword can be used to declare a class variable or method.

- A static variable belongs to the class and not to the instance of the class.
- A variable or function declared with a static keyword can be accessed using the class name instead of the instance of the class.
- Static variables are initialized only once, at the start of the execution.
- A single copy of the static variables would be shared by all instances of the class.
- A static variable can be accessible inside the static function as well as the non-static function.

A static function can access only static variables and other static functions.

```
class Product{
  static productName:string="Mobile";
  static getProductDetails():string{
  return "Product Name is"+Product.productName;
  }
  getProduct():string{
  return "Product Name is"+Product.productName;
  }
  }
 Product.productName="Tablet";
 console.log(Product.productName);
 console.log(Product.getProductDetails());
```

4.4 Properties and Methods

4.4.1 Parameter Properties

Instead of declaring instance variables and then passing parameters to the constructor and initializing it, you can reduce the code by adding access specifiers to the parameters passed to the constructor.

Consider the below code:

```
class Product{
private productId : number;
constructor(productId : number){
this.productId = productId;
}
}
```

In the above example, in Line no 4 you are passing parameter to constructor and setting private property with value passed to constructor.
Instead of this declare the parameter itself with any of the access modifiers and reduce the lines of code used for the initialization.

Let us rewrite the above code:

```
class Product{
constructor(private productId:number){
}
}
```

4.4.2 Index Signatures

A type in TypeScript usually describes an exact set of fields to match on an object. An index signature is a way to define the Shape of fields which are not known ahead of time.

```
class MyClass {
  [s: string]: boolean | ((s: string) => boolean);

  check(s: string) {
    return this[s] as boolean;
  }
}
```

4.4.3 Getters/Setters

TypeScript provides an option to add getters/setters to control accessing the members outside the class or modifying the same.
There is no direct option available to access private members outside the class but use accessors in TypeScript for the same.
If you need to access private properties outside the class, you can use the getter and if you need to update the value of the private property, you can use the setter.

```
class Product{
private _productName : string;
get productName():string{
return this._productName;
}
set productName(newName:string){
this._productName=newName;
}
}
let product = new Product();
product.productName = "Fridge";
 if(product.productName){
    console.log(product.productName);
}
```

You can have different types for the setter block and getter block within the class which should be compatible.

In the below code snippet, the getter block accepts a string whereas the setter block accepts string, boolean or number.

```
let passcode = "secret passcode";
class Product {
   private _productName: any;
   get productName(): string {
      return this._productName;
   }
   set productName(newName: boolean | string | number) {
      if (passcode && passcode == "secret passcode") {
         this._productName= newName;
      }
      else {
         console.log("Error: Unauthorized update of employee!");
      }
   }
}
let product:Product = new Product();
product.productName = "Fridge";
if (product.productName) {
   console.log(product.productName);
}
product.productName = true;
```

```
if (product.productName) {
   console.log(product.productName);
}
product.productName = 123;
if (product.productName) {
   console.log(product.productName);
}
```

TypeScript has some special inference rules for accessors:

- If get exists but no set, the property is automatically readonly
- If the type of the setter parameter is not specified, it is inferred from the return type of the getter
- Getters and setters must have the same Member Visibility

4.4.4 Singletone Instance

Singleton is a creational design pattern that lets you ensure that a class has only one instance while providing a global access point to this instance.

```
class Singleton {
  private static instance: Singleton;

  private constructor() {
    // Private constructor to prevent direct instantiation
  }

  public static getInstance(): Singleton {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }

  public someMethod(): void {
    console.log("Singleton method called");
  }
}

// Usage:
const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();

console.log(instance1 === instance2); // Output: true

instance1.someMethod(); // Output: "Singleton method called"
```

4.4.5 Inheritance

Inheritance is the process of extending a class from an already existing class and reusing the functions and properties of the inherited class in the subclass.
TypeScript supports inheritance with class, wherein the superclass functionalities can be reused within the subclass.
The subclass constructor function definition should invoke the superclass constructor using the super function.

```
class Product{
  protected productId:number;
  constructor(productId:number){
  this.productId=productId;
 }
}
class Gadget extends Product{
    constructor(public productName:string,productId:number){
    super(productId);
  }
  getProduct():void{
  }
}
```

Use the super keyword to access the methods of the super class inside the subclass methods.
Override the superclass methods inside the subclass by specifying the same function signature.

```
class Product{
     protected productId:number;
     constructor(productId:number){
     this.productId=productId;
    }
   getProduct():void{
   }
}
class Gadget extends Product{
    getProduct():void{
      super.getProduct();
    }
}
```

4.4.6 Abstract Class

Abstract classes are base classes that may not be instantiated.
An abstract class can be created using abstract keyword.
Abstract methods within an abstract class are methods declared with abstract keyword and does not contain implementation.
Abstract methods cannot be private.
They must be implemented inside the derived classes.
Abstract classes can contain both abstract methods and its implementations.

```
abstract class Product{
  getFeatures():void{
  }
    abstract getProductName():string;
 }
 class Gadget extends Product{
 getProductName():string{
     }
}
 var g=new Gadget();
 g.getProductName();
```

5. Interfaces

An interface in TypeScript is used to define contracts within the code.

- Interfaces are used to enforce type checking by defining contracts.
- It is a collection of properties and method declarations.
- Interfaces are not supported in JavaScript and will get removed from the generated JavaScript.
- Interfaces are mainly used to identify issues upfront as we proceed with coding.
- Interfaces cannot have protected and private members.

```
interface Interfacename
 { properties;
method declarations;
}
```

```
//Declaring Product interface with productId and productName as properties. Function or Object
which is going to use this interface should contain properties declared here.
 interface Product{
                productId : number;
                      productName : string;
                      }
 //In this function Product object is passed as parameter, if object passed does not have any of the
properties declared in the Interface code throws error.
 function getProductDetails(productobj:Product):string{
                  }
 //Correct way of using the interface type
 let productobj={productId:1001,productName:'Mobile'};
 //InCorrect way of using the interface type, as this declaration does not have productId property of
Interface, code throws compilation error
 let productobj={productCategory:'Gadget',productName:'Mobile'};
 getProductDetails(productobj);
```

5.1 Duck Typing

Duck-Typing is a rule for checking the type compatibility for more complex variable types.

TypeScript compiler uses the duck-typing method to compare one object with the other by comparing that both the objects have the same properties with the same data types.

TypeScript interface uses the same duck typing method to enforce type restriction. If an object that has been assigned as an interface contains more properties than the interface-mentioned properties, it will be accepted as an interface type and additional properties will be ignored for type checking.

Let us rewrite the previous example to a pass additional parameter.

```
interface Product{
   productId:number;
   productName:string;
   }
  function getProductDetails(productobj: Product): string {
   return 'The product name is : ' + productobj.productName;
}
   //Incorrect way of using the interface duck type
   let prodObject={productName:'Mobile',productCategory:'Gadget'};
   //Correct way of using the interface duck type by adding an additional property productCategory
to demonstrate DuckTyping
  let prodObject:Product={productId:1001,productName:'Mobile',productCategory:'Gadget'};
  console.log(getProductDetails(prodObject));
```

In the Line no.11 of the above code, it will throw error because of the duck typing.

5.2 Function Types

Interfaces can be used to define the structure of functions like defining structure of objects.

Once the interface for a function type is declared, you can declare variables of that type and assign functions to the variable if the function matches the signature defined in the interface.

Function type interface is used to enforce the same number and type of parameters to any function which is been declared with the function type interface.

```
function CreateCustomerID(name:string,id:number):string{
}
interface StringGenerator{
```

```
(chars: string, nums : number):string;
}
let IdGenerator : StringGenerator;
IdGenerator=CreateCustomerID;
IdGenerator("Infy",101);
```

5.3 Extending Interfaces

An interface can be extended from an already existing one using the extends keyword.
In the code below, extend the productList interface from both the Category interface and Product interface.

```
interface Category{
categoryName:string;
}
interface Product{
productName:string;
productId:number;
}
interface productList extends Category, Product{
list:Array<string>;
}
let productDetails:productList={
        categoryName :'Gadget',
        productName:'Mobile',
        productId:1234,
        list:['Samsung','Motorola','LG']
}
```

5.4 Class Types

Make use of the interface to define class types to explicitly enforce that a class meets a particular contract.
Use implements keyword to implement interface inside a class.
To enforce interface type on a class, while instantiating an object of a class declare it using the interface type.
The only interface declared functions and properties will be available with the instantiated object.

```
interface Product{
   getProductDetails(productId:number):string;
   displayProductName:(productName:string)=>string;
   }
   class Gadget implements Product{
     getProductDetails(productId:number):string{
       return  "The product id is"+" "+productId;
   }
   displayProductName(productName:string):string{
   return "The product name is"+ " "+productName;
   }
   }
 var myproduct:Product=new Gadget();
```

```
console.log(myproduct.getProductDetails(9861));
console.log(myproduct.displayProductName("Samsung galaxy s23"))
```

5.5 Merging Interfaces

Two or more separate declarations with the same definition can be merged. This is called Declaration Merging.

```
interface Phone {
  color: string;
  vendor: string;
}
interface Phone {
  price: number;
}
let phone: Phone = { color: "Black", vendor: "Samsung", price: 10000 };
```

5.6 Interfaces v/s Abstract Classes

Aspect	**Interfaces**	**Abstract Classes**
Purpose	Define contractual structure.	Provide common functionality and structure.
Implementation	Contains only method signatures.	Can contain implemented methods and abstract methods.
Multiple Inheritance	Supports multiple interface implementation.	Supports single class inheritance.
Implementation Flexibility	No implementation code in interfaces.	Mixes implemented and abstract methods.
Extensibility	Easily extendable by adding new properties/methods.	Can provide shared methods for derived classes.
Constructors	No constructors in interfaces.	Can have constructors for initialization.
Type Checking	Ensures objects adhere to the structure.	Provides a common type and functionality.
Instantiation	Interfaces can’t be instantiated.	Abstract classes can’t be instantiated directly.
Usage	Designing contracts and structure.	Sharing functionality among related classes.

5.7 Interfaces v/s Type Aliases

Interfaces	Type Aliases
In Typescript, interfaces focus on the Type Checking feature. It defines and shapes a construct	Type Aliases is a custom or hand-written name provided to an existing data type of Typescript
It allows optional, readOnly properties	It allows optional, readOnly properties
It allows callable and static properties	It allows callable and static properties

Interfaces can be extended by other Interfaces and Type Aliases
Ex: interface Result extends score, Grade{
}
	Type Aliases cannot be extended by other Interfaces and Type Aliases.
	Intersections will help in Type Aliases to do extending
	Ex: type Result = Score & Grade

It can be implemented by a class	It can be implemented by a class and extended by an Interface.
Interface addresses declaration merging	Type Aliases do not address declaration merging

6. Generics

Generics is a concept using which you can make the same code work for multiple types.
It accepts type parameters for each invocation of a function or a class or an interface so that the same code can be used for multiple types.
Consider the below code where you implement a similar function for two different types of data:

```
function printString(stringData:string):string{
return stringData;
}
```

```
function printNumber(numberData:number):number{
return numberData;
}
```

Avoid writing the same code again for different types using generics. Let us rewrite the above code using generics:
In the below example, shows that <T> represents the type parameter. It is a generalizing type of parameter and function return type. The same works for number or string or any other type of parameter.

```
function printData<T>(data:T):T{
return data;
}
```

Generics helps us to avoid writing the same code again for different data types.

In TypeScript, generics are used along :

- with function to invoke the same code with different type arguments
- with Array to access the same array declaration with a different set of typed values
- with Interface to implement the same interface declaration by different classes with different types
- with a class to access the same class with different types while instantiating it

6.1 Generic Array

Array<T> provides us the option to use the same array declaration for different types of data when used along with a function.
<T> here represents the type parameter.
The below syntax shows creating a function to accept any type of array as a parameter and return type.

```
function functionname<T>(arg:Array<T>):Array<T>{}
```

```
function orderDetails<T>(Arg:Array<T>):Array<T>{
   return Arg;
}
let orderId:Array<number>=[101,102,103,104];
let orderName:Array<string>=['footwear','dress','cds','toys'];
console.log(orderDetails(orderId));
console.log(orderDetails(orderName));
```

6.2 Generic Interface

Generic interface is an interface that works with multiple types.
Below is the syntax for the generic interface

```
interface InterfaceName<T>{
        functionname(arg T):T;
        variablename:T;
        }
```

Example :

```
interface Inventory<T>{
        addItem:(newItem:T)=> void;
        getProductList:()=>Array<T>;
        }
```

This interface can be implemented by different classes with their own types.

```
class Gadget implements Inventory<string>{
addItem(newItem:string):void{
console.log("Item added");
}
productList:Array<string>=["Mobile","Tablet","Ipod"];
getProductListt():Array<string>{
return this.productList;
}
}
let productInventory:Inventory<string>=new Gadget();
let allProducts:Array<String>=productInventory.getProductList();
```

6.3 Generic Class

Generic class is a class that works with multiple types.
The below is the syntax for the Generic class.

```
class className<T>{
functionname(arg T):T;
variablename:T;
}
```

```
class Gadget<T>{
productList:Array<T>;
addItems(newItemList:Array<T>):void{
this.productList=newItemList;
console.log("Item added");
}
getPorductList():Array<T>{
return this.productList;
}
}
```

```
let product=new Gadget<string>();
let productList:Array<string>=["Mobile","Tablet","Ipod"];
product.addItems(productList);
let allProducts:Array<string>=product.getProductList();
console.log(allProducts);
let product2=new Gadget<number>();
let shippingList:Array<number>=[123,234,543];
product2.addItems(shippingList);
let allItems.Array<number>=product2.getProductList();
console.log(allItems);
```

The same class instance can be instantiated and invoked with different type parameters.

6.4 Generic Constraints

Generic constraints are used to add constraints to the generic type.
Generic constraints are added using the 'extends' keyword.
Below is the syntax for the same and it shows how to add generic constraints with extend keywords.

```
class classname<T extends constraint type>{
 functionname (arg T):T;
 variablename:T;
 }
```

Consider the below code:

Here you are trying to access the length property of the function type parameter.
Since the parameter type will get resolved only at run time, this line will throw compilation.
Line no 1 shows creating a generic function and accessing the length property of the parameter.
Line no 2 shows that since parameter type will get resolved only at run time, this line throws compilation error “Property length does not exist on type T”

```
function orderDetails<T>(arg:T):T{
console.log(arg.length);
return arg;
}
```

To resolve this, you can add a constraint with the type parameter.

If you need to access a property on the type parameter, add those properties in an interface or class and extend the type parameter from the declared interface or class.

Let us rewrite the previous code:

```
interface AddLength {
length:number;
}
//Adding constraint by extending from AddLength interface having length property
function orderDetails<T extends AddLength>(arg:T):T{
console.log(arg.length);
return arg;
}
```

6.5 Utility Types

TypeScript provides you with several built-in utility types, that helps you to implement common type transformation easily. These utility types are globally available for the developer.

Some of the commonly used utilities are:

6.5.1 Partial<Type>

This utility helps you in creating a type with all the properties set to optional type. Return type of this utility would be type consisting of all the subset combinations of invoked type.

```
type Product =
        {
        productName: string;
        productColor: string;
        productAvailability: boolean;
        productPrice: number;
}

// below-line generates PartialProduct type with all the Product type parameters as optional
type PartialProduct = Partial<Product>;
// type PartialProduct ={productName?: string; productColor?: string; productAvailability?: boolean;
productPrice?: number;}

const product1:PartialProduct ={};
product1.productAvailability=true;
console.log(product1.productAvailability); // true
```

6.5.2 Required<Type>

This utility type is completely opposite to Partial utility type. This type helps you to create a type from with all the declared properties as mandatory of passed Type parameter.

```
type Product =
{
productName: string;
productColor: string;
productAvailability: boolean;
productPrice: number;
}

// below-line generates CompleteProduct type with all the Product type parameters as mandatory
type CompleteProduct = Required<Product>;

// type CompleteProduct = {productName: string; productColor: string; productAvailability: boolean;
productPrice: number;}
// below-line throws the error as productName, productColor, productAvailability and productPrice
properties declaration are missing
const product1:CompleteProduct ={};
```

6.5.3 ReadOnly<Type>

This utility type helps in creating a type value with all the passed type parameters set to readonly value. You can reassign the newly created type value throughout the application.

```
type Product =
{
productName: string;
productColor: string;
productAvailability: boolean;
productPrice: number;
}

// below-line generates Readonly utility type
type ReadonlyProduct = Readonly<Product>;
const product1:ReadonlyProduct ={
productName: "Samsung",
productColor: "White",
productAvailability: true,
productPrice: 1000
};

// below-line throws the error as productColor cannot be reassigned
product1.productColor="Gold";
```

6.5.4 Record<Keys,Type>

This utility type helps in creating an Object type with key and value parameter where property key would be constructed using passed Keys (first) parameter, and property value with the help of passed Type (second) parameter value. Record utility type help you in mapping the one object values with another object values.

```
interface CatInfo {
  age: number;
  breed: string;
}

type CatName = "miffy" | "boris" | "mordred";

const cats: Record<CatName, CatInfo> = {
  miffy: { age: 10, breed: "Persian" },
  boris: { age: 5, breed: "Maine Coon" },
  mordred: { age: 16, breed: "British Shorthair" },
};
```

6.5.5 Pick<Keys,Type>

This utility type helps in creating a new type value by picking the set properties from the Keys (second) parameter of the passed Type (first) value. While implementing this utility type Keys parameter can hold string literal value or union of string literal value.

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Pick<Todo, "title" | "completed">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
};
```

6.5.6 Omit<Keys,Type>

This utility type exhibits opposite behavior to Pick utility type. It helps in creating type value by picking up all the value from Type (first) parameter and then deletes all the Keys (second) value. You can pass string literal or even union of string literal as Keys(second) parameter input while implementing any logic using this utility type in your code.

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
  createdAt: number;
}

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
  createdAt: 1615544252770,
};

todo;

const todo: TodoPreview

type TodoInfo = Omit<Todo, "completed" | "createdAt">;

const todoInfo: TodoInfo = {
  title: "Pick up kids",
  description: "Kindergarten closes at 5pm",
};
```

7. Decorators

Decorator is used for providing metadata and they are used to specify extra behavior of a class, method, or property of a class.
Decorators are used for declarative programming.
Decorators are used for implementing cross-cutting concerns.

Decorators use @expression form, where expression can represent a business logic which would be called at runtime with needed information respectively.

While developing custom decorators, you will be using the below two arguments:

- target - constructor function of the class for a static member or the prototype of the class for an instance member
- context - context object consists parameters like : kind, name, and addInitializer method.

7.1 Class Decorator

A Class Decorator is used just before a class declaration.
It can be used to modify, observe, or replace any class definition.
The class decorator can be applied to constructor of user defined class.
The class decorator at runtime overrides the original constructor logic with a new one, returns the only argument.
You can log, modify, or replace the original constructor within the class decorator function.

```
function decoratorName(constructor:function, context:any){
...
}
@decoratorName
class className{}
```

In the below example:

- Lines no 1 - 6 shows overriding original constructor with new one and returning new constructor using logClass decorator.
- Line no 8 shows applying decorator using @logClass.

```
function logClass (constructor:Function, context:any){
const newconstructor :any =function(...args){
this.productId=875;
this.productName="Tablet";
}
return newconstructor;
}

@logClass
class Product{
public productId:number;
public procudtName:string;
constructor(productId:number, productName:string){
```

```
this.productId=productId;
this.productName=productName;
}
}
```

7.2 Method Decorator

A Method Decorator are declared before needed method declaration.
They are used to modify, observe, or replace a method definition.
The decorator logic is applied with the help of the context object.

The below example shows the syntax for the method decorators, and

- Line no 1 indicates the method decorator function with the target, and context object has been assigned a type named ClassMethodDecoratorContext that helps to models the passed argument value .
- Line no 8 indicates applying the method decorator on top of the method.

```
function methodDecoratorName(target: any, context: ClassMethodDecoratorContext) {
    function displayMethod(this: any, ...args: any[]) {
      // write your custom logic
     }
     return displayMethod;
 }
class Product {
    @methodDecoratorName
    functionName(param1...)
    {
    //write your custom logic
    }
}
```

```
function logMethod(target: any, context: ClassMethodDecoratorContext) {
    function displayMethod(this: any, ...args: any[]) {
        console.log('Arguments: ', args.join(', '));
        const result = target.call(this, ...args);
        console.log('Total Payable Amount is: ', result);
        return result;
     }
     return displayMethod;
 }
class Product {
    @logMethod
    calculateAmountPayable(price: number, quantity: number): number {
        return price * quantity;
    }
}
const p1: Product = new Product();
let res=p1.calculateAmountPayable(220, 3);
```