Type Aliases and Interfaces

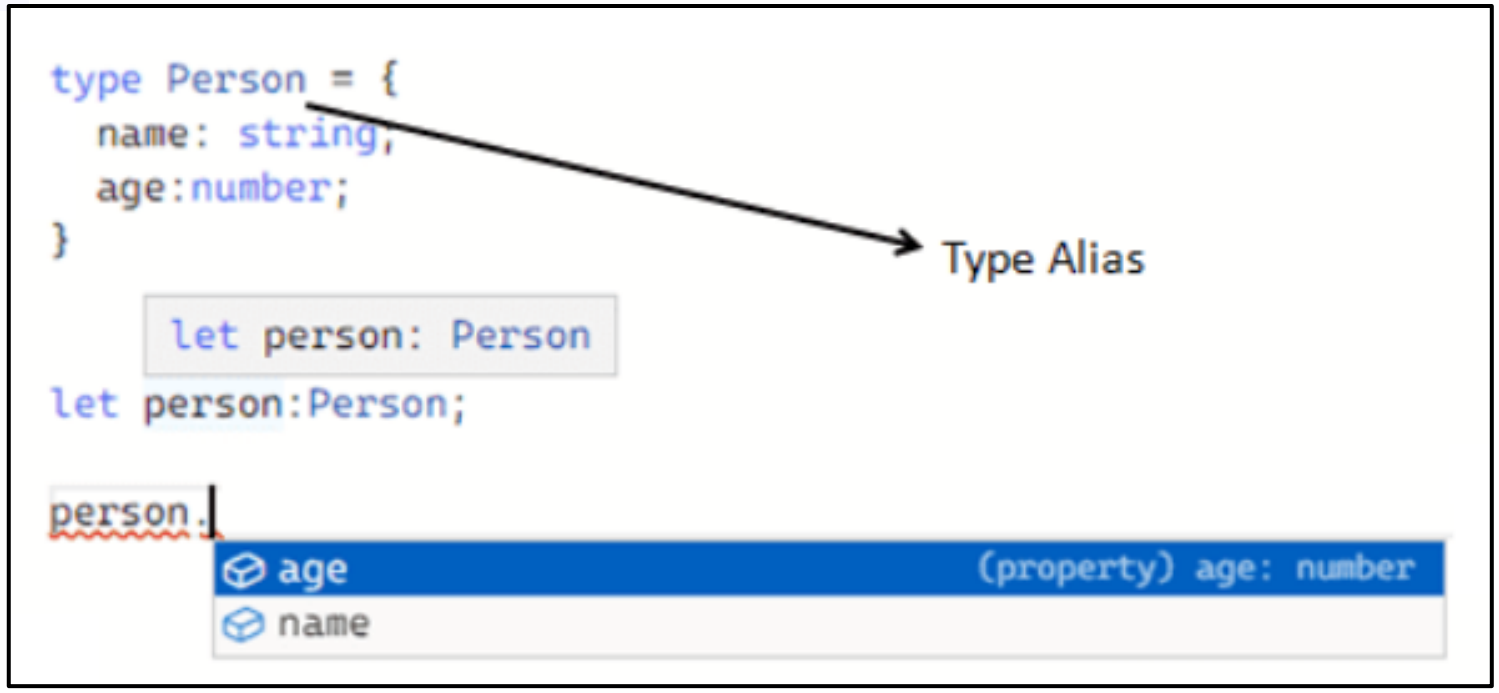
Type Aliases in Typescript

- It allows us to give a **custom name to an existing type**. This is useful when you wish to reuse an existing type as it provides a **reusable definition**.

```
type aliasName = anyType;
```

```
type Person = {
  name: string;
  age:number;
}
```

```
let person:Person;
```



Type Aliases in Typescript (Example)

```
type Person = {
  name: string;
  age:number;
}

let person:Person = {
  name:"Sachin",
  age:50
}
```

Type Aliases for Primitive Types

```
type numType= number;

let a:numType=10
let b:numType=20
let c=30

console.log(a+b+c)   //50
```

Union Types and Type Aliases

```
let a:number|string

a = 1        //ok
a = "hello"   //ok

a=true       //Type 'boolean' is not assignable to type 'string | number'
```

```
type stringOrNumber =  number | string;
type yesNoType = "yes" | "no";
type statusType = "Pending" | "Started" | "Finished"
```

```
type stringOrNumber =  number | string;      //Type Alias for a Union Type

let a:stringOrNumber

a = 1        //ok
a = "hello"    //ok
a=true       //Compile error Type 'true' is not assignable to type 'string | number'
```

Objects

```
type Marks = {
  name: string;
  marks:number;
}

let marks1: Marks = { name:'Tom',marks:90 }
let marks2: Marks = { name:'Poldark',marks:75 }
let marks3: Marks = { name:'Harry',marks:80 }
```

```
type Marks = {
  name: string;
  marks:number;
  pass?:boolean       //optional object members
}

let marks1: Marks = { name:'Tom',marks:90 }
let marks2: Marks = { name:'Poldark',marks:60, pass:false }
let marks3: Marks = { name:'Harry',marks:80 }
```

Nesting Types

```
type Person = {
  name: string;
};

type Company = {
  name: string;
  manager: Person;
};


let microsoft:Company = {
    name:'Microsoft',
    manager: {
        name:'Bill Gates'
    }
}
```

Arrays & Tuple

```
type stringArray= string[];


let arr:stringArray=[]
arr[0]="Hello"
```

```
type Status = [string, boolean];
let arr: Status = ["active", true];

arr[0]="Done";    //ok
arr[0]=false;     //Type 'boolean' is not assignable to type 'string'.

arr[1]=false;
arr[1]="false";   //Type 'string' is not assignable to type 'boolean'.

arr[2]=""         //Tuple type 'Status' of length '2' has no element at index '2'.
```

Type Alias for Functions

- We can also create a Type Alias for a **function expression**.

```
type FuncPrintString = (strToPrint:string) => void

// function expression
let printMe:FuncPrintString = function(foo) {
  console.log(foo)
}

printMe(1)  //Argument of type 'number' is not assignable to parameter of type 'string'.
printMe("Hello")  //ok
```

Type Alias for Functions

- But there is no option to apply a type alias to a **function declaration**.

```
type FuncPrintString = (strToPrint:string) => void

// function declaration. Cannot apply type alias here
function printMe(foo) {
  console.log(foo)
}
```

Interface in TypeScript

- The Interface in TypeScript defines the Shape of an **object**. i.e. it specifies what properties & methods a given object can have and their corresponding value types.
- TypeScript interfaces are **abstract types**. They do not contain any code.
- An object, function, or class that implements the interface must implement all the properties specified in the interface.
- They are similar to **type alias**.
- Both allow us to **name a type** and **use it elsewhere in the code**.

Creating an interface

- We create an interface using the keyword **interface** followed by the interface name. It is then followed by curly brackets **{ }.**
- We define the shape of the object inside the curly brackets.

```
interface IProduct {
   id:number,
   name:string,
   price:number,
   calculate:(qty:number)=> number
}
```

Creating an interface

- Once you have defined an interface, you can use it as a type.

```
let product: IProduct = {
    id : 1,
    name: "Samsung Galaxy",
    price:1000,
    calculate(qty:number): number {
        return this.price*qty;
    }
}

console.log(product.calculate(10))     //1000
```

Creating an interface

- If the Product variable does not contain any property or method defined in the Interface, then the **Typescript compiler immediately throws an error**.

```
let product: IProduct = {
    id : 1,
    name: "Samsung Galaxy",
    calculate(qty:number): number {
        return this.price*qty;
    }
}

//Property 'price' is missing in type '
//{ id: number; name: string; calculate(qty: number): number; }'
//but required in type 'IProduct'.
```

Hence the product object must adhere to an Interface. When it does that, we say that the object **implements the interface**.

JavaScript does not have Interfaces. Hence Interface definitions are not removed when the code is compiled to JavaScript.

Optional Properties

- In the previous example, when we removed the property price from the product object, the compiler flagged it as an error. But in real life, there could be situations where we may require to create objects without assigning values to all properties.
- In such cases, we can mark the **property as optional** with a question mark after the name.

```
interface IEmployee {
    firstName: string;
    lastName: string;
    address?:string
}

let employee:IEmployee= {
   firstName: "Allie",
   lastName: "Grater",
}
```

Read-Only Properties

- We can also mark the property as **read-only** by prefixing the property as **readonly**.
- When the property is marked as read-only, we can assign a value to the property only when initializing the object or in the class constructor.
- **Any subsequent assignments will result in a compiler error.**

```
interface IEmployee {
    readonly firstName: string;
    readonly lastName: string;
    address:string
}


let employee:IEmployee= {
   firstName: "Allie",
   lastName: "Grater",
   address:"Mumbai, India"
}

employee.firstName="Bill"      //Compiler Error
employee.address="Mumbai, Maharastra India"    //ok
```

Interface for an array of objects

- The simplest option is to create the interface for the object and use the [ ] to declare the variable.

```
interface Employee {
  id: number;
  name: string;
}

const arrEmployee: Employee[] = [
  { id: 1, name: 'Rebecca' },
  { id: 2, name: 'Akins' },
];
```

We can also create another interface to extend the Array interface

```
interface Employee {
  id: number;
  name: string;
}

interface EmployeeArray extends Array<Employee> {}


let arrEmployee: EmployeeArray = [
  { id: 1, name: 'Rebecca' },
  { id: 2, name: 'Akins' },
];
```

Interfaces for functions

- Interfaces are also capable of **describing the functions**.
- To create an interface for function type use the keyword **interface** followed by the interface name. Then follow It up with curly brackets { }
- Inside the curly braces { } add the list of parameters (comma-separated) inside parentheses. Follow it up with a semicolon : and return type.

```
//interface
interface IAdd {
   (arg1:number,arg2:number):number
}

//implementation
let add:IAdd = function(num1:number,num2:number) {
 return num1+num2;
}


//invoking
add(10,20) //30
```

Interfaces for functions

- Typescript compiler throws an error if we try to use more arguments or arguments of incompatible data types to the implementation function.

```
interface IAdd {
   (arg1:number,arg2:number):number
}

let add:IAdd

add=function(num1:number,num2:number,num3:number) {
  return num1+num2;
}

//Type '(num1: number, num2: number, num3: number) => number' is not assignable to

//invoking
add(10,20) //30
```

Interfaces for functions

- But allows less number of arguments.

```
interface IAdd {
   (arg1:number,arg2:number):number
}

let add:IAdd

add=function(num1:number) {     //No error
  return num1;
}


//invoking
add(10,20) //30
```

Interfaces for functions

```
interface IEmployee {
    firstName: string;
    lastName:string
    getName(fName:string,lName:string):string
}


let emp:IEmployee= {

  firstName:"",
  lastName:"",
  getName:function(fName:string,lName:string) {
    return fName+lName
  }
}
```

Interfaces for functions

```
interface IgetName {(fName:string,lName:string):string}     //function interface

interface IEmployee {
    firstName: string;
    lastName:string;
    getName:IgetName;
}


let emp:IEmployee= {

  firstName:"",
  lastName:"",
  getName:function(fName:string,lName:string) {
    return fName+lName
  }
}
```

Function Overloading

```
interface IProduct {
  id: number;
  name:string
  someFunc(arg1:string):string
  someFunc(arg1:number):number;
}

let product:IProduct = {
    id:1,
    name:"Mobile",
    someFunc(arg1:any) {
        return arg1;
    },
}
```

v4.7.4 ▾ Run Export ▾ Share

```
1   interface IProduct {
2     id: number;
3     name:string
4     someFunc(arg1:string):string
5     someFunc(arg1:number):number;
6   }
7
8   let product:IProduct = {
9       id:1,
10      name:"Mobile",
11      someFunc(arg1:any) {
12          return arg1;
13      },
14  }
15
16  product.someFunc()
```

Class Implementing Interface

- The Interfaces can be used with the Classes using the keyword **Implements**

```
interface IEmployee {
    firstName: string;
    lastName: string;
    address:string;
    getName():string
}

class Employee implements IEmployee {

    firstName: string;
    lastName: string;
    address:string;

    constructor(firstName: string, lastName: string, address:string) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.address=address;
    }

    getName():string {
        return this.firstName+' '+this.lastName
    }

}
```

Extending Interfaces

- We can extend the Interfaces to include other interfaces.
- This helps us to create a new interface consisting of definitions from the other interfaces.

```
interface Address {
    address: string;
    city:string
    state:string
}

interface Employee extends Address {
    firstName: string;
    lastName: string;
    fullName(): string;
}


let employee: Employee = {
    firstName : "Emil",
    lastName: "Andersson",
    fullName(): string {
        return this.firstName + " " + this.lastName;
    },
    address:"India",
    city:"Mumbai",
    state:"Maharastra",
}
```

Extending Interfaces

- We can extend an interface from several other interfaces.
- **Note:** if the properties with the same name and data type are merged.
- In the case of functions, the function signatures must match.
- i.e. number of arguments, their data type, and function return type must be compatible.

```
interface Address {
    address: string;
    city:string
    state:string
}

interface Employment {
    designation: string;
}

interface Employee extends Address, Employment {
    firstName: string;
    lastName: string;
    fullName(): string;
}


let employee: Employee = {
    firstName : "Emil",
    lastName: "Andersson",
    fullName(): string {
        return this.firstName + " " + this.lastName;
    },
    address:"India",
    city:"Mumbai",
    state:"Maharastra",
    designation:"CEO"
}
```

Interface Merging

- Declaration merging is one of the unique concepts of TypeScript. “declaration merging” means that the compiler merges **two or more separate declarations** declared with the **same name** into a single definition.

```
interface IEmployee {
    firstName: string;
    lastName: string;
    fullName(): string;
}

interface IEmployee {
    address: string;
    city:string
    state:string
}
```

```
//merged to
interface IEmployee {
    firstName: string;
    lastName: string;
    fullName(): string;
    address: string;
    city:string
    state:string
}
```

The compiler does not complain, but instead, it merges the interface into one.

Interface Merging

```
interface IEmployee {
    firstName: string;
    lastName: string;
    fullName(): string;
}

interface IEmployee {
    address: string;
    city:string
    state:string
}

//Compiler error
let employee: IEmployee = {
    firstName : "Emil",
    lastName: "Andersson",
    fullName(): string {
        return this.firstName + " " + this.lastName;
    }
}


//Type '{ firstName: string; lastName: string; fullName(): string; }'
//is missing the following properties
//from type 'Employee': address, city, state
```

- In the case of interface merging, the common properties with the same name and data type are merged. But in the case of functions, if the function signatures do not match, then overload functions are created. This is different from interface extending where it compiler throws an error.

Generic interfaces

- The generics in typescript allow us to work with a variety of types instead of a single one.

```
interface IaddString {(arg1:string,arg2:string):string}
interface IaddNum {(arg1:number,arg2:number):number}

let addString:IaddString= function(arg1:string,arg2:string) {
  return arg1+arg2;
}

let addNum:IaddNum= function(arg1:number,arg2:number) {
  return arg1+arg2;
}
```

Generic interfaces

- Using generics, we can reduce the number of interfaces into a single one. Here **T** represents a type. It could be anything string, number or array, etc. Here a single interface handles multiple related functions.

```
interface Iadd<T> {(arg1:T,arg2:T):T}


let addString1:Iadd<string> = function(arg1:string,arg2:string) {
  return arg1+arg2;
}
let addNum1:Iadd<number> = function(arg1:number,arg2:number) {
  return arg1+arg2;
}
let addArray:Iadd<String[]> = function(arg1:String[],arg2:String[]) {
  return [ ...arg1, ...arg2];
}
```

Creating Interface for Function Types

- To create an interface for function type use the keyword **interface** followed by the interface name. Then follow It up with curly brackets { }. Inside the curly braces { } add the list of parameters (comma-separated) inside parentheses.

```
//interface
interface Calculator {
   (arg1:number,arg2:number):number
}

//implementation
let calculator:Calculator = function(arg1:number,arg2:number) {
  return arg1+arg2;
}


//invoking
calculator(10,20) //30
```

Creating Interface for Function Types

- The parameter names need not match.

```
//interface
interface Calculator {
    (arg1:number,arg2:number):number
}

//implementation
let calculator:Calculator = function(num1:number,num2:number) {
  return num1+num2;
}


//invoking
calculator(10,20) //30
```

Creating Interface for Function Types

- But data types need not be the same but they must be **compatible**.

```
//interface
interface Calculator {
    (arg1:number,arg2:number):number
}

//implementation
let calculator:Calculator = function(num1:unknown,num2:unknown) {
  return (num1 as number)+(num2 as number);
}


//invoking
calculator(10,20) //30
```

Creating Interface for Function Types

- Typescript compiler throws an error if we try to use more arguments or arguments of incompatible data types to the implementation function.

```
//interface
interface Calculator {
    (arg1:number,arg2:number):number
}

//implementation
let calculator:Calculator

calculator = function(num1:number,num2:number, num3:number) {    //Error
  return num1+num2;
}

//Type '(num1: number, num2: number, num3: number) => number' is not assignable to type 'ICalc

//invoking
calculator(10,20) //30
```

Interface Vs Type Alias in TypeScript

- The syntax for both are similar, except in type alias we use the assignment operator
- Type alias does not create a new type. it just gives a new name to an existing type. The Interface creates a new type.
- We can assign a name to any type using a type alias. For example, we can assign a new name to primitive types. We cannot use an interface to create a primitive type.
- Type alias can give a name to a union type. We cannot do that using the interface.
- Type alias can give a name to an Intersection Type. But we cannot do that with an interface.

Interface Vs Type Alias in TypeScript

- **Declaration merging** means that the compiler merges **two or more separate declarations** declared with the **same name** into a single definition.
- Interfaces with the same name are merged. The two types with the same name are not allowed. The following code results in an error.
- A Type alias cannot extend an Interface, class, or Type alias.
- Both Type Alias and an interface can be used to describe a function.
- Tuples are easily described by using Type Alias. But, we can also use an interface to define a Tuple.

Intersection Type

```
interface Person {
    name: string;
    age: number;
}

interface Student {
    studentCode: string;
    division: string
}


let student: Student & Person= {
    studentCode:"1",
    division:"10",
    name:"Rahul",
    age:20
}
```