**Typescript Introducton**

**1.So what is TypeScript? What is it all about, why would we use it?**

TypeScript is a JavaScript Superset. Now, what does this mean? It means that TypeScript is in the end a language, a programming language, building up on JavaScript. It's not a brand new language, instead it takes the JavaScript language and adds new features and advantages to it. It makes writing JavaScript code easier and more powerful,

But, we have one huge disadvantage. TypeScript can't be executed by JavaScript environments like the browser. Browsers can't execute TypeScript, and for example, Node.js also can't execute TypeScript, so the environments where we can execute JavaScript don't support TypeScript. What's the idea behind TypeScript then? It's a better version of JavaScript and we can't use it?

TypeScript is a programming language, but it's also a tool. It's a powerful compiler which you run over your code to compile your typescript code to JavaScript. So, what you get as a result when writing code in TypeScript is JavaScript. But you didn't write that JavaScript code, you wrote TypeScript code with all the new features and all the advantages, and you get normal JavaScript code.

how can TypeScript add new features if what you get in the end is regular avaScript?

And the answer is, the TypeScript compiler compiles these new features to JavaScript workarounds, so in the end it might give you a nicer syntax, an easier way of doing something, and it will then compile that nicer, easier way to a more complex JavaScript snippet, which you would have to write otherwise.

**Typescript=Javascript + Type system**

All the knowledge you have around JavaScript like arrays, objects, functions and even is 2015 syntax like de structuring and arrow functions and classes. All that knowledge still applies to the world of TypeScript as well. The only thing that we're really doing with TypeScript is adding in a little bit additional syntax to our code to handle something called the type system.

So let's get a quick overview on the type system and understand what's going on with it. The goal of the type system is to help us catch errors during development. In other words, when we're actually writing our code in our code editor, think about how we catch errors right now with JavaScript code.

Let's imagine that you're writing out some amount of JavaScript inside of your editor and maybe there's a bug inside of it. How would you find that bug? Well, really, with JavaScript, the only way to do that is to actually execute your code and see that error up here. And that's not super efficient. So as an improvement to the development workflow, we use the type system to help us catch errors during development.

**// Defining a function that takes two numbers and returns their sum**

**function add(x: number, y: number): number {**

**return x + y;**

**}**

**// Calling the function with numbers**

**let result = add(3, 5);**

**console.log(result); // Output: 8**

**// Calling the function with a string (will produce a type error)**

**// TypeScript will flag this as an error because 'Hello' is not a number**

**// and violates the type annotation for the function parameters**

**let invalidResult = add(3, 'Hello'); // Error: Argument of type 'string' is not assignable to parameter of type 'number'**

**console.log(invalidResult);**

**https://www.typescriptlang.org/play**

While you and I are writing our code, TypeScript is going to be constantly analyzing it and looking for bugs. If it finds any possible bug, it's then going to pop open an error message inside of your code editor and tell you, Hey, something might be wrong here.And that's going to be essentially be a signal to you as the developer that you might need to fix up your code. to do this error checking the TypeScript compiler is going to use something called type annotations to analyze our code base.

The type system is only active during development. So in other words, once we go to deploy our application or even run it inside of our browser in a development environment, the entire type system falls away.

The TypeScript compiler does not do any performance optimization whatsoever. Now, this is very different than many other languages. In many other languages, the type system can be used to optimize some code that you write using the compiler. But that is not the case here with TypeScript.

**What is Type?**

The properties or functions value type .we can assign strings, numbers, booleans, null, undefined objects, functions, classes, arrays and so on. So all of those different things have types. An array has a type, a object has a type. A function has a type A class as a type and so on. So let's now break down the first part of the statement.

“red” -> is string and it has properties and methods

indexOf()

lastIndexOf()

charAt()

concat()

endsWith()

match()

So type is a shortcut to describe what properties and function this things have. We summarize this as a string

**2.How to install Typescript?**

The Node Package Manager, a tool which we then can use to install TypeScript globally on our machine. So simply visit nodejs.org and there, install the latest version you find here. Simply click on this button. It will then download and install or you can walk through that installer. It is supported for all operating systems. And once you have Node JS installed, you will be able to run this command.

1.Create a folder in your local machine

2.Open node command prompt

3.Goto that folder

4.npm install -g typescript

Here What we have installed we installed typescript compiler everything we need to understand typescript and conver into Javascript.

we have the compiler installed and we can run the TSC command now which invokes this TypeScript compiler to compile a TypeScript file to JavaScript. So to see this in this project we worked on, let's simply add a new file using ts.

Create a file with .ts extention

var *x*=23;

x="12"; // *Error: Type '12' is not assignable to type 'number'.*

In the code immediately we will get the error.To see these errors need to add extention error lens.

<!DOCTYPE *html*>

<html>

  <head>

    <title>Sample HTML</title>

    <script *src*="using-ts.js" *defer*></script>

  </head>

  <body>

    <input *type*="number" *id*="n1" />

    <input *type*="number" *id*="n2" />

    <button *onclick*="*myFunction*()">Click Me</button>

  </body>

</html>

function *myFunction*() {

  var *textboxValue* = *document.getElementById*("myTextbox");

*alert*("Textbox value: " + textboxValue*.*value);      // *Error: Property 'value' does not exist on type 'HTMLElement'.*

}

function *myFunction*() {

  var *textboxValue* = *document.getElementById*("myTextbox")! *as HTMLInputElement*;

*alert*("Textbox value: " + textboxValue*.*value);

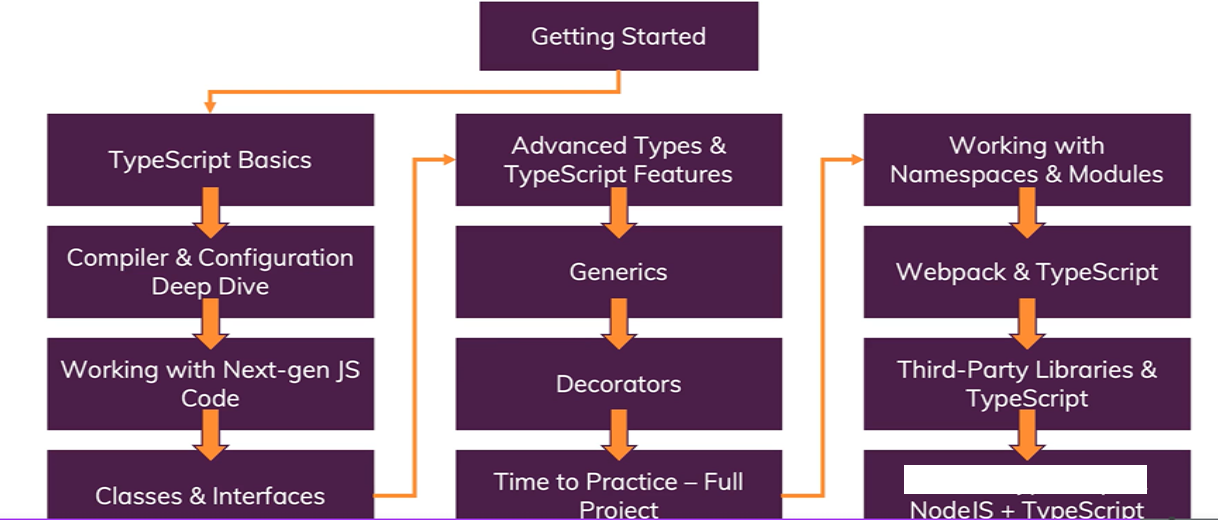
}

Let ,var ,const difference

Function why we need

Arraow function

ES6



**3.New enabled features**

ESLint

Material Icom

Path intelisence

Prettier

**4.For live server**

**To get package.json**

npm init

npm install –save -dev lite-server

“start”:”lite-server”

**5.Core types which Javascript already knows**

We can discuss the difference between Javascript knowing the type and Typescript using the type

number - 1,1.5.-10

string - “a”,’a’,`a`

boolean – true or false

undefined

null

Object

Date

**var** a = 10;

if (a) { where a is truthy

*//Whaat is this a  is it truthy or falsy*

  console.log("a is true");

}

Var a=0;

It is falsy

**undefined:**

in js

console.log(typeof a);

didn’t declare a

in ts

console.log(typeof a) -> will get error

so

let a:undefined

console.log(typeof a)

after compilation this will be convertesa as

var a;

console.log(typeof a);//js not support let keyword

**Note:**

When we learn optional parametetr it will be used.

**Object**

**Typescript**

**let** d: Date = new Date();

console.log(d);

console.log(typeof d);

let d:Date

console.log(d);

console.log(typeof d);

**undefined**

**Javascript**

**var** d = new Date();

console.log(d);

console.log(typeof d);

**var** d;

console.log(d);

console.log(typeof d);

**null**

var x=null

**typeof x ->Object**

where null is used to represent absence of value in object

and also it represent falsy type

var x;

typeof x is undefined

**let** a: null = null;

console.log(typeof a); object

console.log(a);null

**var** a = null;

console.log(typeof a);

console.log(a);

**let** a: null;

console.log(typeof a); undefined

console.log(a);undefined

**var** a;

console.log(typeof a);

console.log(a);

Fetching data from database with name and email for some data email can be nullin that case we can use email:null;

*// Nullable type example*

**let** myString: string | null;

myString = "Hello";

console.log(myString); *// Output: Hello*

myString = null;

console.log(myString); *// Output: null*

*// With strictNullChecks enabled*

**let** anotherString: string;

*// Error: Type 'null' is not assignable to type 'string'.*

*// anotherString = null;*

*// Using optional parameters and properties*

**function** greet(name: string | null) {

  if (name !== null) {

    console.log("Hello, " + name);

  } else {

    console.log("Hello, guest");

  }

}

greet("John"); *// Output: Hello, John*

greet(null); *// Output: Hello, guest*

*// Nullable properties in objects*

**type** User = {

  name: string;

  age?: number; *// Age is optional*

};

**let** user1: User = { name: "Alice" };

**let** user2: User = { name: "Bob", age: 30 };

console.log(user1); *// Output: { name: 'Alice' }*

console.log(user2); *// Output: { name: 'Bob', age: 30 }*

**Array**

// Declaring an array of numbers

let numbers: number[] = [1, 2, 3, 4, 5];

// Declaring an array of strings

let fruits: string[] = ['apple', 'banana', 'orange'];

// Accessing elements of an array

console.log(numbers[0]); // Output: 1

console.log(fruits[1]); // Output: banana

// Modifying elements of an array

numbers[2] = 10;

console.log(numbers); // Output: [1, 2, 10, 4, 5]

// Iterating over an array using a for loop

for (let i = 0; i < fruits.length; i++) {

console.log(fruits[i]);

}

// Iterating over an array using forEach

fruits.forEach(fruit => {

console.log(fruit);

});

// Adding elements to an array

fruits.push('grape');

console.log(fruits); // Output: ['apple', 'banana', 'orange', 'grape']

// Removing elements from an array

let removedFruit = fruits.pop();

console.log(removedFruit); // Output: grape

console.log(fruits); // Output: ['apple', 'banana', 'orange']

// Using array methods like map

let doubledNumbers = numbers.map(num => num \* 2);

console.log(doubledNumbers); // Output: [2, 4, 20, 8, 10]

**Object**

// Creating an object

let person = {

  firstName: "John",

  lastName: "Doe",

  age: 30,

  hobbies: ["reading", "hiking", "cooking"],

  address: {

    street: "123 Main St",

    city: "Anytown",

    country: "USA",

  },

  greet: function () {

    return "Hello, my name is " + this.firstName + " " + this.lastName + ".";

  },

};

// Accessing object properties

console.log(person.firstName); // Output: John

console.log(person["age"]); // Output: 30

console.log(person.address.city); // Output: Anytown

// Modifying object properties

person.age = 35;

person["address"]["country"] = "Canada";

console.log(person.age); // Output: 35

console.log(person.address.country); // Output: Canada

// Adding new properties to an object

person.email = "john@example.com";

person["occupation"] = "Engineer";

console.log(person.email); // Output: john@example.com

console.log(person["occupation"]); // Output: Engineer

// Accessing object methods

console.log(person.greet()); // Output: Hello, my name is John Doe.

Evenif we get the error we cn execute why because it execute only javascript files.

Typescrit includes

Any

Void

Unknown

Arrays

Tuples

Class

Interface

Enum

Union Types

Type Alias

All the above we need to discuss in detail

**any**

**In both Js and Ts**

let a;//in typescript it is any

console.log(typeof a) //it will be converted as js so js does'nt support any so it disply undefined

a=10;

console.log(typeof a);//number

a="abc"

console.log(typeof a);//string

But any is bad why becalues

let a:any=’welcome’;

console.log(a.fun()); //We will not get an error but at runtime we will get an error like **“Uncaught type error a.fun() is not like a function”.**

**Unknown**

To overcome the above error we can use unknown

**let** a:unknown;

console.log(typeof a);//undefined

a = 10;

console.log(typeof a);

a = "abc";

console.log(typeof a);

a.foo();

**Unknown Vs any**

**var** a:any=10

**var** b:unknown=20

**var** c:string=a;

**var** d:string=b as string; *// Error: Type 'unknown' is not assignable to type 'string'.*

**Void**

**function** doSomething(): void {

  console.log("Hello");

*//return "aBoolean";*

}

**let** x = () **=>** {

  console.log("Hello");

*//return "aBoolean";*

};

doSomething();

x();

**function** doSomething(): void {

  console.log("Hello");

*//return "aBoolean";*

}

**let** x = function() {

  console.log("Hello");

*//return "aBoolean";*

};

doSomething();

x();

**Union Type:**

**let** userInput:number |string

userInput=10;

userInput='10';

userInput=true; *//error*

**let** a;//here we will get on suggestion int playground it wont tell anything but in vscode it gives suggestion like infere type from usage

a = [1, "2", 3]; *// OK*

console.log(a);

let a:number | string

**Tuples**

let studentRecord=[1,"abc",23,true];

studentRecord=["abc",2,23,false];

let studentRecod1:[number,string,number,boolean];

studentRecod1=[2,"abc",34,true];

let studentRecod2:[number,string,number,boolean]=[3,"lkm",87,false];

TYpescript  
**let** studentRecord = [1, "abc", 23, true];

studentRecord = ["abc", 2, 23, false];

**let** studentRecod1: [number, string, number, boolean];

studentRecod1 = [2, "abc", 34, true];

**let** studentRecod2: [number, string, number, boolean] = [3, "lkm", 87, false];

for (**let** v in studentRecod2) {

  console.log(v);

}

Javascript

**var** studentRecord = [1, "abc", 23, true];

studentRecord = ["abc", 2, 23, false];

**var** studentRecod1;

studentRecod1 = [2, "abc", 34, true];

**var** studentRecod2 = [3, "lkm", 87, false];

for (**var** v in studentRecod2) {

    console.log(v);

}

**Enum**

Enums in TypeScript allow developers to define a set of named constants, making it easier to work with a group of related values. Here's an example demonstrating enums in

**// Enum example**

**enum Direction {**

**Up,**

**Down,**

**Left,**

**Right**

**}**

**// Using enum values**

**let playerDirection: Direction = Direction.Right;//it acts as both datatype and value**

**console.log(playerDirection); // Output: 3**

Javascript code

**var** Direction;

(**function** (Direction) {

    Direction[Direction["Up"] = 0] = "Up";

    Direction[Direction["Down"] = 1] = "Down";

    Direction[Direction["Left"] = 2] = "Left";

    Direction[Direction["Right"] = 3] = "Right";

})(Direction || (Direction = {}));

*// Using enum values*

**var** playerDirection = Direction.Right; *//it acts as both datatype and value*

console.log(playerDirection); *// Output: 3*

**Intersection type**

Intersection types in TypeScript allow you to combine multiple types into a single type that has all the characteristics of each type. You represent an intersection type by using the **&** operator between the types you want to intersect. Here's an example demonstrating intersection types in

// Intersection type example

interface Employee {

id: number;

name: string;

}

interface Manager {

department: string;

role: string;

}

type ManagerEmployee = Employee & Manager;

// Creating an object of intersection type

let managerEmployee: ManagerEmployee = {

id: 101,

name: "John Doe",

department: "Engineering",

role: "Team Lead"

};

console.log(managerEmployee);

// Output: { id: 101, name: 'John Doe', department: 'Engineering', role: 'Team Lead' }

// Accessing properties

console.log(managerEmployee.id); // Output: 101

console.log(managerEmployee.name); // Output: John Doe

console.log(managerEmployee.department); // Output: Engineering

console.log(managerEmployee.role); // Output: Team Lead

**Never Vs Void**

1. **void**:
   * Indicates that a function does not return any value.
   * It's commonly used as the return type of functions that perform an action but don't return a meaningful value.
   * Variables of type **void** can only be assigned **undefined** or **null**.
   * You might use **void** when you want to explicitly indicate that a function doesn't return anything.
2. **never**:
   * Represents the type of values that never occur.
   * It's typically used to indicate that a function never returns normally, for example, if it always throws an error or enters an infinite loop.
   * Variables of type **never** can't have any value assigned to them.
   * You might use **never** when you want to express that something will never happen or that a function will never complete normally.

// Function returning never

function throwError(message: string): never {

throw new Error(message);

}

// This function never returns normally

function infiniteLoop(): never {

while (true) {

// Infinite loop

}

}

// Variable with never type

let unreachable: never;

// Assigning a value to a never type results in an error

// Error: Type 'string' is not assignable to type 'never'.

// unreachable = "This line is unreachable";

**6.Adding Type annotation**

**function** add(n1, n2) {

  return n1 + n2;

}

**const** n1 = 12.5;

**const** n2 = "13";

console.log(add(n1, n2));

**We will get 12.513**

**To avoid this We can change the code as**

**function** add(n1:number, n2:number) { //This is type annotation

  return n1 + n2;

}

**const** n1 = 12.5;

**const** n2 = "13";

console.log(add(n1, n2));

**7.Javascript Vs Typescript**

where

typeof(n1) will be number

we can check

if(typeof(n1)!==’nummber’) in javascript

But we no needs to check this in typescript we can identify the error earlier.Javascript knows very few types where as typescript knows more types

Javascript uses “dynamic types “(resolved at runtime ) where as type script use “static types” (resolved at development time)

**8.Important: Type Casing**

In TypeScript, you work with types like string or number all the times.

**Important**: It is string and number (etc.), **NOT** String, Number etc.

**The core primitive types in TypeScript are all lowercase!**

**9.Working with Number,Boolean, String**

function getResult(t:number,e:number,name:srtring,showResult:Boolean,resultPhrase:string){

}

getResult();

**10.Typescript Basics & Basic Types**

**1.Type Annotation or Type inference or Type Assertion**

Type Annotation means When we specify the data type of variable, parameters return type explicitly.

**function** add(n1: number, n2: number) {*//type annotation*

  return n1 + n2;

}

Const n1:number

**const** n1 = 12.5;// *Type inference*

**const** n2 = 13;

console.log(add(n1, n2));

**Type Assignment**, on the other hand, involves assigning a specific type to a variable at the time of declaration or initialization. It's the act of associating a particular type with a variable or entity. Type assignment can be implicit or explicit, depending on the language and the context. In statically typed languages, type assignment usually occurs explicitly through type annotations or through the type inference mechanism of the language.

So, while type annotation involves explicitly stating the type of a variable in the source code, type assignment refers to the act of assigning a type to a variable, which can be done explicitly through type annotations or implicitly through type inference.

**Type assertion** is used when you know more about the type of a value than TypeScript can infer or check. It's like telling TypeScript, "Trust me, I know what I'm doing." Here's an example:

let someValue: any = "Hello, TypeScript!";

// We know that someValue is a string, but TypeScript doesn't

// We can use type assertion to tell TypeScript the type of someValue

let strLength: number = (someValue as string).length;

console.log(strLength); // Output: 17

let z: any = "Hello";

let strLength: number = (z as string).length; // Type assertion

console.log(strLength); // Output: 5

n this example, **z** is explicitly typed as **any**, so TypeScript doesn't know that it's a string. By using **(z as string)**, we're asserting to TypeScript that **z** should be treated as a string, allowing us to access its **length** property without TypeScript throwing an error.

// Assume data is received from an API and is of type any

let userData: any = {

id: 1,

username: "john\_doe",

email: "john@example.com"

};

// Type assertion to inform TypeScript about the structure of the data

let user = userData as { id: number, username: string, email: string };

// Now TypeScript knows the structure of user

console.log(user.id); // TypeScript knows user.id is a number

console.log(user.username); // TypeScript knows user.username is a string

console.log(user.email); // TypeScript knows user.email is a string

**11.Questions?**

**1.Why are "Types" useful and offer an advantage compare to vanilla JavaScript?**

Types allows you tod detect errors early and avoid some runtime errors.

2. Will the following code throw a compilation error?

1. let userName: string;
2. userName = 'Maximilian';
3. userName = false;

Yes

3.const a=5;

How the typescript will infer this variable type

const a:5

4. Does this code rely on type inference?

1. const age: number = 29;
2. Type is assigned explicitly as well
3. What's the difference between JavaScript types (e.g. typeof 'Max' => 'string') and TypeScript types (e.g. const name: string = '...')?

That's correct. JS has no compilation step but at runtime, you can check for certain types (e.g. in if conditions). TS on the other hand allows you to catch certain errors during development since it checks types during compilation as well.

**12.Object type and Type inference**

**const** person = {

  name: "John",

  age: 30,

  isEmployed: true,

};

*// TypeScript infers the type of 'person' as { name: string, age: number, isEmployed: boolean }*

**13.Function type And Type inference**

*// Function type declaration*

**type** MathOperation = (a: number, b: number) **=>** number;

*// Function with type inference*

**const** add: MathOperation = (a, b) **=>** a + b;

*// Function with explicit type annotation*

**const** subtract: MathOperation = (a: number, b: number) **=>** a - b;

*// Usage*

console.log(add(5, 3)); *// Output: 8*

console.log(subtract(10, 4)); *// Output: 6*

**14.Watchmode**

1.tsc app.ts -w

**2.for all**

tsc –init (create tsconfig.json in which we can configure details like hoe it could compile the files.)

tsc -w

**3.to exclude the file to watch the changes**

add

"exclude": ["\*\*/\*.ts"]

“exclude”:[“node\_modules”]

“include”:[“\*\*/a.ts”]

In ts.config

4.“target”:es5 what if es6

1. such a button will exist. "We will get a value here." So now we can add a click event here. And then find some anonymous function maybe where a console.log clicked. Now, the interesting thing here is not so much that I had to add the exclamation mark, as I said I will come back to that, but that if I run tsc this just compiles. Now, shouldn't TypeScript complain that document is unknown here?
2. How does it know that we have such a document, constant or variable vailable? How does it know that even if we have that available that it holds an object which has our querySelector method? How does it know that button is something which has addEventListener method? How does TypeScript know all of that? Now you might say, "Of course it knows." Because in Vanilla avaScript this would be valid code but keep in mind that when you write TypeScript code, you don't necessarily write it for the browser. You could be writing your Node.js application with TypeScript and there indeed this would all not work. So, the reason why this works is this lib option and as you see it's not even set here, but if it isn't set then some defaults are assumed. If it's not set the defaults depend on your JavaScript target here, and for es6 it, by default, includes all the features that are globally available in es6. For xample, the Map object which is available in es6. Therefore it wouldn't complain if you use Map. So it assumes all the es6 features which are made available globally in JavaScript, that they are available in TypeScript as well. And in addition it assumes that all DOM API's are available. You'll find the detailed descriptions about all these options, by the way, in the official box which are linked in the lecture at the end of this module. So, long story short, if the lib option is not set some defaults are assumed and these are typically the defaults you need
3. to have TypeScript run in the browser. So, all the DOM API's are gone. If we comment this in and I now compile everything I definitely get an error, because now they're commenting it in we don't have the defaults settings anymore. Instead, we now say, "Hey, please include "some default libraries," some default type definitions you could say, which I will give you in

“lib”:[

“Scripthost”

“dom”,

“es6”,

“dmiterable”,

“

]

How to debug

Run-> debug

Launch change localhost:3000

**15. Important es6**

Let and const

Desturctuing(array & Object)

Arrow function

Default function parameter

Rest and spread

**Classes and Interfaces**

1.ProductList->Product->ShoppingCart

2. What is class and Object

3.First class

Department.ts

**class** Department {

  name: string;

**constructor**(n: string) {

    this.name = n;

  }

    describe(this: Department) {

        console.log("Department: " + this.name);

    }

}

**const** accounting = new Department("Accounting");

console.log(accounting);

accounting.describe();

Department.js(es6)

"use strict";

**class** Department {

**constructor**(n) {

        this.name = n;

    }

    describe() {

        console.log("Department: " + this.name);

    }

}

**const** accounting = new Department("Accounting");

console.log(accounting);

accounting.describe();

*//# sourceMappingURL=app.js.map*

Es5

"use strict";

**var** Department = */\*\** **@class** *\*/* (**function** () {

**function** Department(n) {

        this.name = n;

    }

    Department.prototype.describe = **function** () {

        console.log("Department: " + this.name);

    };

    return Department;

}());

**var** accounting = new Department("Accounting");

console.log(accounting);

accounting.describe();

*//# sourceMappingURL=app.js.map*

Ts

**class** Department {

  name: string;

**constructor**(n: string) {

    this.name = n;

  }

  describe(this: Department) {

    console.log("Department: " + this.name);

  }

}

**const** accounting = new Department("Accounting");

console.log(accounting);

accounting.describe();

**const** accountingCopy = { describe: accounting.describe };

accountingCopy.describe();//undefined

**public and private**

**class** Department {

**public** name: string;

**private** employees: string[] = [];

**constructor**(n: string) {

    this.name = n;

  }

  describe(this: Department) {

    console.log("Department: " + this.name);

  }

  addEmployee(employee: string) {

    this.employees.push(employee);

  }

  printEmployeeInformation() {

    console.log(this.employees.length);

    console.log(this.employees);

  }

}

**const** accounting = new Department("Accounting");

accounting.addEmployee("Max");

accounting.addEmployee("Manu");

accounting.employees[2] = "Anna";//at compiletime we will get error

accounting.printEmployeeInformation();

*//console.log(accounting);*

accounting.describe();

*//const accountingCopy = { describe: accounting.describe };*

*//accountingCopy.describe();*

Readonly

**class** Department {

*//public name: string;*

**private** employees: string[] = [];

**constructor**(**private** **readonly** id: string, **public** name: string) {

*//this.name = n;*

  }

  describe(this: Department) {

    console.log("Department: " + this.name);

  }

  addEmployee(employee: string) {

    this.employees.push(employee);

  }

  printEmployeeInformation() {

    console.log(this.employees.length);

    console.log(this.employees);

  }

}

**const** accounting = new Department("Accounting");

accounting.addEmployee("Max");

accounting.addEmployee("Manu");

accounting.employees[2] = "Anna";

accounting.printEmployeeInformation();

*//console.log(accounting);*

accounting.describe();

*//const accountingCopy = { describe: accounting.describe };*

*//accountingCopy.describe();*

1. Classes are blueprint of Javascript Object
2. A variable in a class is called property
3. Private makes property not accisble from outside of the class

**Inheritence,get,static ,Abstract claa,Singleton**

**abstract** **class** Department {

*//public name: string;*

**static** fiscalYear = 2020;

**protected** employees: string[] = [];

**constructor**(**protected** **readonly** id: string, **public** name: string) {

*//this.name = n;*

  }

**static** createEmployee(name: string) {

    return { name: name };

  }

*// describe(this: Department) {*

*//   console.log("Department: " + this.name);*

*// }*

**abstract** describe(this: Department): void;

  addEmployee(employee: string) {

    this.employees.push(employee);

  }

  printEmployeeInformation() {

    console.log(this.employees.length);

    console.log(this.employees);

  }

}

**class** ITDepartment **extends** Department {

  admins: string[];

**constructor**(id: string, admins: string[]) {

    super(id, "IT");

    this.admins = admins;

  }

  describe() {

    console.log("IT Department - ID: " + this.id);

  }

}

*//const accounting = new Department("d1", "Accounting");*

**const** it = new ITDepartment("d1", ["Max"]);

*//console.log(it);*

**const** employee1 = Department.createEmployee("Max");

console.log(employee1, Department.fiscalYear);

it.addEmployee("Max");

it.addEmployee("Manu");

it.printEmployeeInformation();

console.log(it);

it.describe();

**class** AccountingDepartment **extends** Department {

**private** lastReport: string;

**private** **static** instance: AccountingDepartment;

**private** **constructor**(id: string, **private** reports: string[]) {

    super(id, "Accounting");

    this.lastReport = reports[0];

  }

**static** getInstance() {

    if (this.instance) {

      return this.instance;

    }

    this.instance = new AccountingDepartment("d2", []);

    return this.instance;

  }

**get** mostRecentReport() {

    if (this.lastReport) {

      return this.lastReport;

    }

    throw new Error("No report found.");

  }

**set** mostRecentReport(value: string) {

    if (!value) {

      throw new Error("Please pass in a valid value!");

    }

    this.addReport(value);

  }

  describe(): void {

    console.log("Accounting Department - ID: " + this.id);

  }

  addEmployee(name: string) {

    if (name === "Max") {

      return;

    }

    this.employees.push(name);

  }

  addReport(text: string) {

    this.reports.push(text);

    this.lastReport = text;

  }

  printReports() {

    console.log(this.reports);

  }

}

*//const accountingCopy = { describe: accounting.describe };*

*//accountingCopy.describe();*

**const** accounting2 = AccountingDepartment.getInstance();

**const** accounting3 = AccountingDepartment.getInstance();

console.log(accounting2, accounting3);

accounting2.mostRecentReport = "Year End Report";

accounting2.addReport("Something went wrong...");

console.log(accounting2.mostRecentReport);

accounting2.printReports();

accounting2.addEmployee("Max");

accounting2.addEmployee("Manu");

accounting.describe();

*//accounting2.printEmployeeInformation();*

*//console.log(accounting2);*

*//console.log(accounting3);*

**Interfaces**

[TypeScript: Documentation - Object Types (typescriptlang.org)](https://www.typescriptlang.org/docs/handbook/2/objects.html)

In TS

**function** greet(person: { name: string; age: number }) {

  return "Hello " + person.name;

}

greet({ id: 1, name: "John", age: 21 });

In Js

**function** greet(person) {

  return "Hello " + person.name;

}

greet({ id: 1, name: "John", age: 21 });

In ts

*// function greet(person: { id:number,name: string; age: number }) {*

*//   return "Hello " + person.name;*

*// }*

**function** greet1(person: { name: string; age: number }) {

  return "Hello " + person.name;

}

*//In another function I may misss the id property*

*//to avoid this we can use interface*

**interface** Person {

  id: number;

  name: string;

  age: number;

}

**function** greet(person: Person) {

  return "Hello " + person.name;

}

greet({ id: 1, name: "John", age: 21 });

*//or we can use type*

**type** Person = {

  id: number;

  name: string;

  age: number;

};

**function** greet2(person: Person) {

  return "Hello " + person.name;

}

*//or we can use class*

**class** PersonClass {

  id: number;

  name: string;

  age: number;

**constructor**(id: number, name: string, age: number) {

    this.id = id;

    this.name = name;

    this.age = age;

  }

}

*// function greet(person: { id:number,name: string; age: number }) {*

*//   return "Hello " + person.name;*

*// }*

**function** greet1(person: { name: string; age: number }) {

  return "Hello " + person.name;

}

*//In another function I may misss the id property*

*//to avoid this we can use interface*

*// interface Person {*

*//   id: number;*

*//   name: string;*

*//   age: number;*

*// }*

*//or we can use type*

**type** Person = {

  id: number;

  name: string;

  age: number;

};

**function** greet2(person: Person) {

  return "Hello " + person.name;

}

*//or we can use class*

**class** PersonClass {

  id: number;

  name: string;

  age: number;

**constructor**(id: number, name: string, age: number) {

    this.id = id;

    this.name = name;

    this.age = age;

  }

}

*//Difference between interface and class*

*//interface is a contract that an object can implement*

*//class is a blueprint for objects*

*//interface is used for defining types*

*//class is used for defining objects*

*//interface is used for defining the shape of an object*

*//class is used for defining the behavior of an object*

*//interface is used for defining the structure of an object*

*//class is used for defining the implementation of an object*

*//interface is used for defining the capabilities that an object must have*

*//class is used for defining the properties and methods that an object will have*

*//interface is used for defining the public API of an object*

*//class is used for defining the private implementation of an object*

*//interface is used for defining the public contract of an object*

*//class is used for defining the private details of an object*

*//interface is used for defining the public interface of an object*

*//class is used for defining the internal implementation of an object*

*//interface is used for defining the public properties and methods of an object*

*//class is used for defining the private properties and methods of an object*

*//interface is used for defining the public properties and methods of an object*

*//class is used for defining the private properties and methods of an object*

*//Example for interface with methods*

**interface** Greetable {

  name: string;

  greet(): string;

}

**class** PersonClass1 **implements** Greetable {

  name: string;

**constructor**(name: string) {

    this.name = name;

  }

  greet() {

    return "Hello " + this.name;

  }

}

*//Interface and Type comparision*

*//Type aliases can act as interfaces*

*//Type aliases can be used to define object types*

*//Example for type alias with object types*

**type** PersonType1 = { id: number; name: string; age: number };

*//Type aliases can be used to define union types*

*//Example for type alias with union types*

**type** GreetableType1 = PersonType | { name: string; greet(): string };

*//Type aliases can be used to define intersection types*

*//example for type alias with intersection types*

**type** GreetableType2 = PersonType & { greet(): string };

**type** PersonType = { id: number; name: string; age: number };

**type** GreetableType = PersonType | { name: string; greet(): string };

*//In TypeScript, GreetableType4 is a type that can either be a PersonType or an object type with a greet() method that returns a string. This means that a variable of type GreetableType4 can either hold a PersonType object or any other object that has a greet() method returning a string.*

**type** GreetableType4 = PersonType | { greet(): string };

*//In TypeScript, GreetableType5 is a new type created by combining two existing types: PersonType and an object type with a greet() method that returns a string. This means that any object of type GreetableType5 must have all the properties of PersonType as well as a greet() method that returns a string.*

**type** GreetableType5 = PersonType & { greet(): string };

*//Explaination for type alias with conditional types*

*//In TypeScript, both type and interface can be used to define object shapes, but they have some differences in behavior.*

*/\*Extension vs. Intersection:*

*With type, when you use &, you're doing an intersection, meaning you're combining two types to create a new one that has all the properties of both.*

*With interface, when you use extends, you're extending an existing interface to add more properties.*

*Open vs. Closed:*

*interface is open-ended, meaning you can add new properties to it later by extending it.*

*type is closed-ended; once defined, you can't add new properties to it. You'd need to create a new type.*

*Declaration Merging:*

*Only interface can be merged. If you define two interfaces with the same name, TypeScript will merge them into a single interface.*

*type declarations cannot be merged.*

*So, if you want to do an intersection of two types and ensure they have all properties, including methods, using type with & is the way to go. If you want to extend an existing shape or create an open-ended structure, interface would be more suitable.*

*\*/*

*//Type aliases can be used to define function types*

*//Example for type alias with function types*

**type** GreetFunction = (name: string) **=>** string;

*//Type aliases can be used to define constructor types*

*//Example for type alias with constructor types*

**type** PersonConstructor = new (name: string) **=>** PersonType;

*//Type aliases can be used to define tuple types*

*//Example for type alias with tuple types*

**type** PersonTuple = [number, string, number];

*//Type aliases can be used to define array types*

*//Example for type alias with array types*

**type** PersonArray = PersonType[];

*//Type aliases can be used to define generics*

*//Type aliases can be used to define conditional types*

*/\*Scenario:*

*You're developing an application where you need to handle different shapes, such as circles and squares. Each shape has a name property and either a radius property for circles or a sideLength property for squares. You want to create a type alias that, given a shape type, returns the corresponding type of the shape's area.*

*Question:*

*Using TypeScript conditional types, how would you define a type alias that takes a shape type and returns the corresponding type of the shape's area?\*/*

*//Solution:*

**interface** Circle {

  name: string;

  radius: number;

}

**interface** Square {

  name: string;

  sideLength: number;

}

**type** Area<T **extends** Shape> = T **extends** { radius: number }

  ? number

  : T **extends** { sideLength: number }

  ? number

  : never;

*// Example usage:*

**const** circle: Circle = { name: "Circle", radius: 5 };

**const** square: Square = { name: "Square", sideLength: 4 };

**type** CircleArea = Area<Circle>; *// CircleArea will be number*

**type** SquareArea = Area<Square>; *// SquareArea will be number*

*/\**

*Explanation:*

*In this example, we have two interfaces, Circle and Square, representing different shapes.*

*We define a type alias Area that takes a generic type T extending Shape and returns the corresponding type of the*

*shape's area. We use TypeScript conditional types to check if T has a radius property or a sideLength property,*

*and based on that, we return the area type as a number.*

*If T does not match either of the shape types, we return never.*

*Why never ?*

*The never type is used as the result when neither the condition for a circle nor the condition for a square is met. This is a way to ensure exhaustiveness in the conditional type definition. It indicates that if T does not have either a radius property or a sideLength property, then the resulting type is never, meaning there is no valid type for the area in this context.*

*Using never in this manner helps ensure that the conditional type covers all possible cases, providing type safety and preventing unintended scenarios where the type may be left unspecified or incorrect.*

*\*/*

*//Type aliases can be used to define mapped types*

*//Example for type alias with mapped types*

**type** PersonKeys = "id" | "name" | "age";

**type** PersonMapped = { [K in PersonKeys]: string };

*//Type aliases can be used to define recursive types*

*//Example for type alias with recursive types*

**type** TreeNode<T> = {

  value: T;

  left?: TreeNode<T>;

  right?: TreeNode<T>;

};

*//Type aliases can be used to define indexed types*

*//Example for type alias with indexed types*

**type** PersonIndex = {

  [key: string]: PersonType;

};

*//Type aliases can be used to define conditional types*

*//Example for type alias with conditionaltypes*

*//Type aliases can be used to define conditional types that depend on other types or values in the program  logic*

*//Type aliases can be used to define conditional types that depend on other types or values in the program  logic*

*//Type aliases can be used to define mapped types*

*//Type aliases can be used to define recursive types*

*//Type aliases can be used to define indexed types*

*//Type aliases can be used to define type assertions*

*//Type aliases can be used to define type guards*

*//Type aliases can be used to define type inference*

*//Type aliases can be used to define type compatibility*

*//Type aliases can be used to define type widening*

*//Type aliases can be used to define type narrowing*

*//Type aliases can be used to define type assertions*

*//Type aliases can be used to define type aliases*

**function** greet3(person: PersonClass) {

  return "Hello " + person.name;

}

**function** greet(person: Person) {

  return "Hello " + person.name;

}

greet({ id: 1, name: "John", age: 21 });

**GENERICS**

In this module, we'll delve into a concept called generics, which is a more advanced feature of TypeScript.

Generics serve as an extra layer of abstraction over regular types, allowing us to build stronger types by providing type parameters. Similar to how functions have parameters, types can also have parameters, and a type with parameters is called a generic type.

**Extra Layer of Abstraction**:

* Generics allow us to create components (functions, classes, interfaces) that can work with a variety of types.
* This means we can write code without committing to a specific type, making our code more flexible and reusable.
* Without generics, we would have to define separate versions of the same code for each type we want to work with, leading to code duplication.

**Example**: Suppose we want to create a function to log the length of an array. Without generics, we might create separate functions for each type of array:

function logArrayLengthString(arr: string[]): void {

console.log(`Length: ${arr.length}`);

}

function logArrayLengthNumber(arr: number[]): void {

console.log(`Length: ${arr.length}`);

}

// Usage

const stringArray: string[] = ["apple", "banana", "cherry"];

const numberArray: number[] = [1, 2, 3];

logArrayLengthString(stringArray); // Output: Length: 3

logArrayLengthNumber(numberArray); // Output: Length: 3

With generics, we can create a single function that works with any type of array:

function logArrayLength<T>(arr: T[]): void {

console.log(`Length: ${arr.length}`);

}

// Usage

const stringArray: string[] = ["apple", "banana", "cherry"];

const numberArray: number[] = [1, 2, 3];

logArrayLength(stringArray); // Output: Length: 3

logArrayLength(numberArray); // Output: Length: 3

**Building Stronger Types**:

* Generics allow us to create types that are more precise and adaptable to different scenarios.
* By using generics, we can enforce type safety while still maintaining flexibility in our code.

**Example**: Suppose we have a function to return the first element of an array. Without generics, the function might return **any**, which sacrifices type safety:

function firstElement(arr: any[]): any {

return arr[0];

}

// Usage

const stringArray: string[] = ["apple", "banana", "cherry"];

const firstString: any = firstElement(stringArray); // No type safety

console.log(firstString.toUpperCase()); // Potential runtime error

With generics, we can ensure that the function returns the same type as the elements of the array:

function firstElement<T>(arr: T[]): T | undefined {

return arr[0];

}

// Usage

const stringArray: string[] = ["apple", "banana", "cherry"];

const firstString: string | undefined = firstElement(stringArray); // Type safety

console.log(firstString.toUpperCase()); // No compilation error

The necessity of generics becomes apparent when we aim to reduce code duplication and create more robust types, especially when dealing with advanced TypeScript types.

Understanding generics is crucial if you want to grasp advanced TypeScript types, which we'll explore in subsequent lectures.

What can you expect from this section? First, we'll provide a simple definition and examples of generics. Then, we'll delve into popular generics, generics constraints, and using generics with multiple types, classes, interfaces, and types.

Now, let's begin with an introduction to TypeScript generics. Generics exist in TypeScript, not in vanilla JavaScript. They enable us to write more reusable and flexible code by parameterizing types.

While generics might seem unfamiliar at first, they are fundamental to understanding and harnessing the power of TypeScript's advanced type system. We'll explore what generics are, how they are utilized in TypeScript, and why

they are essential.

Throughout this section, we'll build our own generic functions and classes, explore generic constraints, and examine special TypeScript types that are typically generic.

By the end of this module, you'll have a comprehensive understanding of generics in TypeScript and be well-prepared to tackle more advanced TypeScript concepts. Let's dive in!

**Example 1:**

**// Define a function called 'toArray' that accepts an argument of any type and returns an array containing that argument**

**function toArray(arg: any): any[] {**

**return [arg]; // Wraps the argument into an array**

**}**

**// Use the 'toArray' function with a string argument**

**const stringArray = toArray("Hello");**

**console.log(stringArray); // Output: ["Hello"]**

**// Use the 'toArray' function with a number argument**

**const numberArray = toArray(123);**

**console.log(numberArray); // Output: [123]**

**// Define a generic function called 'toArrayGeneric' that accepts an argument of type 'T' and returns an array of type 'T[]'**

**function toArrayGeneric<T>(arg: T): T[] {**

**return [arg]; // Wraps the argument into an array**

**}**

**// Use the 'toArrayGeneric' function with a string argument**

**const stringArrayGeneric = toArrayGeneric("Hello");**

**console.log(stringArrayGeneric); // Output: ["Hello"]**

**// Use the 'toArrayGeneric' function with a number argument**

**const numberArrayGeneric = toArrayGeneric(123);**

**console.log(numberArrayGeneric); // Output: [123]**

**// Use type inference to automatically determine the type of the argument and return value**

**const inferredStringArray = toArrayGeneric("World");**

**console.log(inferredStringArray); // Output: ["World"]**

**// Use type inference to automatically determine the type of the argument and return value**

**const inferredNumberArray = toArrayGeneric(456);**

**console.log(inferredNumberArray); // Output: [456]**

**Example 2:**

// Define an array of names

const names = ["Max", "Manuel"];

// The type of 'names' constant is inferred as 'string[]'

// TypeScript knows it's an array of strings

// Define a promise that eventually resolves to a string

const promise: Promise<string> = new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Data loaded successfully");

}, 2000);

});

// The type of 'promise' constant is explicitly set as 'Promise<string>'

// TypeScript knows it's a promise that resolves to a string

// Generic types provide additional type information to TypeScript

// Array and Promise are examples of built-in generic types in TypeScript

// Generic types are indicated by angle brackets <>

// They allow for more precise type annotations and better type safety

// Using generics with arrays

// Array is a generic type, it works with different types of data

// The type parameter T specifies the type of data stored in the array

const stringArray: string[] = ["Hello", "World"];

// TypeScript infers the type 'string[]' for 'stringArray'

// Using generics with promises

// Promise is a generic type, it represents an asynchronous operation that eventually yields a value

// The type parameter T specifies the type of data returned by the promise

const stringPromise: Promise<string> = new Promise<string>((resolve, reject) => {

setTimeout(() => {

resolve("Data loaded successfully");

}, 2000);

});

// TypeScript infers the type 'Promise<string>' for 'stringPromise'

Explanation:

* The code demonstrates the use of generics with built-in generic types such as Array and Promise in TypeScript.
* Arrays in TypeScript are generic types, represented as **Array<T>** or **T[]**, where **T** specifies the type of data stored in the array.
* Similarly, Promises in TypeScript are generic types, represented as **Promise<T>**, where **T** specifies the type of data returned by the promise when resolved.
* By using generics, developers can provide additional type information to TypeScript, improving type safety and enabling better IDE support and code analysis.
* In the provided example, a string array (**string[]**) and a promise that resolves to a string (**Promise<string>**) are created using generics to specify the expected data types.
* TypeScript infers the types of variables based on the provided generic type annotations, ensuring type correctness and providing better code assistance during development.

[ChatGPT](https://chatgpt.com/c/8b02758e-dc1b-4f46-a31d-5a8634a69a65)(for promises)