SECTION-1

-> typescript is superset of javascript.It is a language build upon javascript. It adds new features and advantages to javascript.but ts cannot be executed by js environments like browser.

-> install typescript *npm install -g typescript*

-> typescript with javascript is executed by following command
tsc filename.ts this then compiles and creates javascript files.

-> If we change anything in ts file then we need to again compile and refresh browser page.to avoid that we use a tool.To install that tool command *npm init*.we will get package.json file.Then execute the command *npm install* to install the dependencies.lite-server is simple development server,which always servers index.html file.so we have to set “start”:”lite-server” in package.json file.so that we can run npm start .

-> key difference between typescript and javascript is that js is dynamic type(resolved at run time).
Ts is static type(resolved during development).

SECTION-2

->**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!
-> by default number is float in ts and js.there is no difference between ts types and js types.

->Type assignment and type inference:
const a=5;
The variable ‘a’ is inferred as number here.

-> Object types:
Object types looks similar to object .
const person={
name:”Max”,
Age:30 }
Type inference would be :

```
const person:{
  name:string;
  age:numbr;
}
```

Object is a key value pair but object type is key type pair.

```
const person:{
  name:string;
  age:numbr;
}=
{
name:”Max”,
    Age:30
 }
```

-> **Nested Objects & Types**

Of course object types can also be created for **nested objects**.

Let's say you have this JavaScript **object**:

```
const product = {
 id: 'abc1',
 price: 12.99,
 tags: ['great-offer', 'hot-and-new'],
 details: {
   title: 'Red Carpet',
   description: 'A great carpet - almost brand-new!'
 }
}
```

This would be the **type** of such an object:

```
{
 id: string;
 price: number;
 tags: string[];
 details: {
```

```
    title: string;
    description: string;
  }
 }
```

So you have an object type in an object type so to say.

-> Array types

```
favourites:string[]
```

-> tuples:

```
Person:{
 Role:[string,number]
}
```

Above type only accepts 2 values and also first value must be string and second value must be number.

There is also type inference, in that case
Role=[1,"MAX"]

This accepts (number | string) . we can also push additional values.there is no specific order.

-> Enum:

Enum is a custom type so,enum variable should start with Capital letter.
Enum automatically increment values.

```
enum Role{ admin,read_only,author}
console.log(Role.admin) // 0
console.log(Role.read_only) // 1
console.log(Role.author) // 2
```

-> any type:

Can store any value.

->union types:

Value: number | string.

-> Literal types:

Literal types are the types which are based on core types (string ,number ..) but then we have a specific version of the type.

Ex: ResultConversion : ' as-number' | 'as-text'

Here we gave specific strings.ResultConversion will not accept any string other than ‘as-number’ or ‘as-text’.

-> type aliases / custom types :

```
type value = number | string;
input1:value;
input2:value;
```

-> **Type Aliases & Object Types**

Type aliases can be used to "create" your own types. You're not limited to storing union types though - you can also provide an alias to a (possibly complex) object type.

For example:

```
type User = { name: string; age: number };
const u1: User = { name: 'Max', age: 30 }; // this works!
```

This allows you to avoid unnecessary repetition and manage types centrally.

For example, you can simplify this code:

```
function greet(user: { name: string; age: number }) {
 console.log('Hi, I am ' + user.name);
}

function isOlder(user: { name: string; age: number }, checkAge: number) {
 return checkAge > user.age;
}
```

To:

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

function greet(user: User) {
 console.log('Hi, I am ' + user.name);
}

function isOlder(user: User, checkAge: number) {
 return checkAge > user.age;}
```

-> function return types and void:

Ex:

```
function add(input1:number,input2:number):number{
```

```
    return input1+input2;
}
```

If we don't return anything then return type would be void.if we don't return anything but there is return statement then return type would be 'undefined'.

```
function add():void{
  console.log("hi")
}

function add():undefined{
   return;
}
```

-> function as types: function types are types that define a function regarding return values,parameters.

Ex: const combineValues:(a:number,b:number)=>number;
combineValues=add; // add is function here.
console.log(combineValues(1,2));

-> function types and callbacks:
Ex:

```
function add(a:number,b:number,cb:(num:number)=>viod)
{
 const result=a+b;
  cb(result);
}
add(1,2,(result:number)=>{
console.log(result);
});
```

-> unknown type:
If we are not sure about values that we are going to store in a variable,then we can use unknown type.we can store anyvalue.it is better than 'any' type.
Ex: const userInput=unknown;
-> never type:
Never is another type functions can return.If a function is throwing an error then that function returns never.if we try to print value returned by that function,It does't print anything on screen.

```
function errorGenerator(message:string,code:number):never{
 throw { message:message,code:code};
}
```

SECTION-3

-> Watch mode:
   In order to execute file.ts automatically when something changes in file then it should executed in watch mode.
* npm tsc filename.ts -watch * (or)
* npm tsc filename.ts -w *
It automatically recompiles.
If there are more ts files then it would not work for that
tsc –init this will create tsconfig.json(crucial to manage this project) file.and then
tsc this will execute all ts files and create js files.

-> if we want to execute in watch mode then
tsc –watch or *tsc -w*

-> in tsconfig.json file, at the end of the page after compiler options if we give
“exclude”:[“filename.ts”]
the compiler would not execute that particular file and it would not effect the compiler default behaviour.
“exclude”:[“*.dev.ts”]
It will check for all files with this extension.

-> there will be “include” as well.
-> there will be “files”.It is bit like “include”.It points individual files.
->In compiler(in tsconfig.json file),
   -> “sourceMap” is for debugging and development
   -> “outDir”,It stores created files there (like js files).
   -> “rootDir”, It avoids same filestructure on output files(like js files).
   -> “removecoments” is to remove comments in compiled javascript files.
   -> “noEmit” is to avoid creating js files.when we want to check whether ts files are correct or not.
   -> “noEmitOnError” (default is false)set to true to avoid creating js files when there is error in ts files.

SECTION-4

-> difference between let and var is that var only know function scope and global scope.but let considers every block like if,function etc(variable defined with let is only available within block like func,if).If it is defined with var then it is available within function and global.

-> Arrow functions:
const printOutput:(a:number | string)=>void = output=>console.log(a);

->Rest parameters:
Const add=(...numbers)=>{
console.log(numbers) //numbers is array here.

}

add(1,2,3,4.5,6,7,8);

-> Array and object destructuring:
Array:
const [hobby1,hobby2,...remaining]= hobbies;
Object :
const {firstName:username,age}=person;
Here firstName and age are property names of person object.username is nothing but alias name of firstName.

SECTION-5

->constructor functions and this keyword:
'This' keyword refers to instance of class.with '.' notation we can access all properties and methods of that instance.

```
class Department {
 name: string;
 constructor(n: string) {
   this.name = n;
 }
 describe(){
```

```
    console.log("Depatment:",this.name); // here if we use ‘name’ then it
                                         //Will look for global variable
                                         // Or variable that is Local to
                                         //  describe function.so we have to
                                         //  Give ‘this.name’
  }
}
const Acounting=new Department('acounting')
Acounting.describe();
```

```
class Department {
  name: string;
  constructor(n: string) {
    this.name = n;
  }
  describe(this:Department){
    console.log("Depatment:",this.name);
  }
}
const Acounting=new Department('acounting')
Acounting.describe();

const accountingCopy={name:'DUMMY',describe:Acounting.describe}
accountingCopy.describe();
```

-> public and private modifiers.public is default access modifier.public properties are accessible from outside of class.whereas private properties are not.

->

```
constructor(public name: string) {
    this.name = name;
  }
```

This avoids creating variables at the top of class.public name: string this creates variable.

->readonly:
Variable which is declared readonly, cannot be modified.

private readonly name:string;

-> Inheritance:
-> super() is to invoke super class constructor.
-> It is important to call super in constructor before using ‘this’ to perform any operations.

-> overriding properties and “protected” modified.
-> variable declared as protected is like private but protected is accessible in child class whereas private is not accessible outside class.
-> getters and setters(provides more security):
get method should return something.we can access oit as property.
->static properties and methods:
Static meethods can be accessible without intantiating the class.we can access static method without new keyword.here static methods and variables are not accessible with ‘this’ keyword inside class.bcz static methods and variables are detached from instance of class.If we want to use then we need use with classname.

->abstract:
Abstract methods are the kind of methods that are only defined in base class and is implemented in inherited classes.implementation is not provided in base class.
Abstract classes cannot be intantiated, but inherited classes from abstract class can be instantiated.
Abstract methods can only appear within an abstract class.

-> Private constructors:
In oop there is a pattern which is called singleton pattern.singleton pattern means ensuring that a particular class always exactly have one instance.
Inorder to avoid creating instance multiple times use private before constructor of that class.It avoids using new keyword.

The singleton pattern is about ensuring
that you always only have exactly one instance
of a certain class.
This can be useful in scenarios where you somehow

can't use static methods or properties
or you don't want to,
but at the same time you want to make sure
that you can't create multiple objects based on a class
but that you always
have exactly one object based on a class.
Let's say for our AccountingDepartment
we wanna make sure that we can only create exactly
one object based on this class,
because we have exactly one accounting department
in our entire company.
We might have more than one IT department
but we have exactly one accounting department.
Now to enforce this and to avoid
that we manually call new AccountingDepartment
multiple times, we can turn the constructor
of the AccountingDepartment class
into a private constructor by adding the private keyword
in front of it.
Now what this does is,
it ensures that we can't call new on this.
Here you see I'm getting an error
because the constructor is private
so it's only accessible from inside the class,
which sounds strange because how do we get inside
of the class if we can't create objects based on it anymore.
The answer is, well, static methods.
A static method can be called on the class itself
so you don't have instantiate it for that.
So here we can add a static method which
we could call getInstance,
the name is totally up to you though.
Now getInstance will check if we already have an instance
of this class and if not, return a new one.
For that we can add a new static property instance,
a static private property
so you can put private in front of static
called instance which will be of type AccountingDepartment.
So in there we'll store an AccountingDepartment instance.
So that's what I'm saying here,

I have a static property
which is accessible on the class itself,
but only from inside the class
and the value we store in there will be
of type AccountingDepartment, so of the class itself.
Now we can use this instance property here in getInstance
and check if this.instance is set here inside
of static, if we use this, it will refer to the class itself
and then we can access all other static properties on that.
The alternative to that would be to use the class name.
And now if this is set I want to return this.instance,
or again classname.instance but this inside
of a static method works, it gives us access
to the class itself then,
unlike this in a non static method which gives us access
to the instance with which we're trying to work,
not what we're doing here.
If however we don't make it in here then
we have no instance yet,
then I set this.instance, so this static instance property
equal to new AccountingDepartment,
we can use this from inside here
because now we're inside of this class method,
so here we can access the private constructor
and pass in our ID,
and our reports array
and then return this.instance here.
So now we're either returning the one instance
we might already have,
or if we don't have it yet we create a new one.
But this code, the marked code here,
can only run once because once we have an instance
we make it into we make it into this if block
and we return the existing instance.
So now if you wanna work with the AccountingDepartment,
instead of creating it like this we could call,
const accounting AccountingDepartment.getInstance,
and this returns us a new instance
of the AccountingDepartment.
But if I do this again I will get the same instance as

you will see if I console log accounting,
and accounting2 here.
You will see that the two should be exactly equal,
if we save that and reload,
you see down there are my two AccountingDepartment objects,
they have the same ID, the exact same setup,
they are the same object, the same instance
because we only have one instance
with this singleton pattern which is created
with the help of the private keyword in front
of the constructor.
Now this is, arguably, an approach which
you won't use all the time.
The singleton pattern can sometimes be useful,
you don't need it all the time,
but it's definitely worth to know about it
because it is something interesting which
you can easily implement with TypeScript thanks
to private constructors.

-> Interfaces:
->Interface describes structure of an object.Its like custom type.
-> We can define structure but cannot assign values.
-> A class can implement more than one interface separated with coma.
-> working with interfaces is bit like abstract classes the difference being that an interface has no implementation details at all,whereas abstract classes can be a mixture of you have to overwrite these parts and I have a concrete implementation parts.That's an important difference between interfaces and abstract classes.

Example:

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

```
   greet(phase:string):void
}

let user:Person;
user={
  name:'Max',
  age:29,
  greet(phrase:string){
    console.log(phrase);
  }
}

user.greet("hello")
```

->we can only add readonly access modifier to properties in interface.readonly can also be used to type.we can implement more than one interface.but we cannot inherit more than one class.
->Interfaces can also be used to define structure of a function.
->

```
 interface AddFun{
    (a:number,b:number):number
}

let add:AddFun;
add=(a:number,b:number)=>{

  return a+b;
}
console.log(add(2,3))
```

->optional properties :

```
   Interface person={
       Age?:number;
      optional?=mymethod?(){....};
      }
```

SECTION-6

-> Intersection types:

Intersection types allows us to combine other types.

For intersection types we can use both types and interface.But it is preferred to use types here.

```
// interface Employee1={....}
type Employee1={
  name:string;
  privilages:string[];

}

// interface Employee2={....}
type Employee2={
  name:string;
  startDate:Date;

}
// interface Intersection extends Employee1,Employee2 {}
type Intersection= Employee1 & Employee2;

let Employee:Intersection;
Employee={
  name:'max',
  privilages:['access'],
  startDate:new Date()
}
console.log(Employee);
```

-> type Guards :

Type guards helps us with union types

```
let a: string | number;
let b: string | number;
```

```
function Add(a: string | number, b: string | number) {
  if (typeof a === "string" || typeof b === "string") {
    return a.toString() + b.toString();
  }
  return a + b;
}

type UnknownEmployee=Employee1 | Employee2;

function Display(emp:UnknownEmployee){
  if('privilages' in emp){
    console.log(emp.privilages)

  }
  else{
    console.log(emp.name);
  }
}
```

-> instanceof keyword can be used to check.

-> Discriminated unions:

It is a pattern,which you can use when working with union types.this makes implementing type guards easier.

All interfaces have common property.so that when comparing we can proceed with common property.

```
interface Bird{
  type:'bird';
  flyingSpeed:Number;
}

interface Animal{
  type:'Animal';
  RunningSpeed:number;
}

function moveAnimal(animal:Bird|Animal){
  switch(animal.type){
```

```
    case 'bird':console.log(animal.flyingSpeed);
    break;
    case 'Animal': console.log(animal.RunningSpeed);
    break;

  }
}
```

-> type casting:

We can use ‘as’ or <> to define type casting.

Ex:

```
Const userInput = document.getElementById('userInput') as HtmlInputElement
 Or
Const userInput = <HtmlInputElement>document.getElementById('userInput')
```

-> Index properties :

a feature that allows us to create objects which are more flexible regarding the properties they might hold.

If we are not sure about how many properties and what properties we are going to use then we can use index properties.

```
interface ErrorContainer{
   [prop:string]:string;
}

const errorBag:ErrorContainer={
    email:'email is not valid',
    username:'must start with capital letter'
}
```

-> function overloads:

Type casting syntax: prev_type as next_type

```

type combine=string|number;

function Adding(a:string,b:string):string;
function Adding(a:number,b:number):number;
function Adding(a:combine,b:combine){
  if(typeof a==='string' || typeof b==='string')
```

```
  {
    return a.toString()+b.toString();
  }
  return a+b;
}

console.log(Adding('suji','max'))
```

->optional chaining:
if(job.title && job.title.get)

&& and ? is used to avoid runtime error when there is no data.

-> Nullish Coalescing: (??)

```
Const userInput="";
const  stored =userInput || 'DEFAULT';
```

Here empty string is not null or undefined.but in second line compiler treats as undefined or null and stores DEFAULT.to avoid that nullist coalescing is used(??).

```
const  stored =userInput ?? 'DEFAULT';
```