Fundamental types in TypeScript :

1. **Primitive Types**:

Primitive types store **single** values directly in memory and are immutable (cannot be changed).
Examples:

- number
- string
- boolean
- null
- undefined
- symbol
- bigint

```
let age: number = 25;
let name: string = "Alice";
let isActive: boolean = true;
```

2. **Reference Types**

Reference types store **complex** objects in memory and are mutable (can be modified). Instead of storing the actual value, they store a reference (memory address).
Examples:

- object
- array
- function
- class

Array - Used to store multiple values of the same type.

```
let numbers: number[] = [1, 2, 3, 4];
let names: string[] = ["Alice", "Bob"];
```

Example in TS:

```
// Type definition for User objects
type User = {
  name: string;
```

```
    isActive: boolean;
};

// Arrays with specific types
const superHeros: string[] = [];
const heroPower: number[] = [];
const allUsers: User[] = [];

// Adding elements to arrays
superHeros.push("spiderman");
heroPower.push(2);
allUsers.push({ name: "", isActive: true });

// Multi-dimensional array example for ML models
const MLModels: number[][] = [
    [255, 255, 255],
    [128, 64, 32] // Added another row for better clarity
];
```

Converted JS:

```
// Arrays with specific types
var superHeros = [];
var heroPower = [];
var allUsers = [];
// Adding elements to arrays
superHeros.push("spiderman");
heroPower.push(2);
allUsers.push({ name: "", isActive: true });
// Multi-dimensional array example for ML models
var MLModels = [
    [255, 255, 255],
```

```
  [128, 64, 32] // Added another row for better clarity
];
```

Tuples - Fixed-length array with specific types at each index.

```
let person: [string, number] = ["Alice", 25];
```

Enums - Defines a set of named constants.

```
enum Color { Red, Green, Blue }
let color: Color = Color.Green;
```

Other special types:

Any - Can hold any type of value.

```
let value: any = "Nuha"; //string
console.log(value);  // Output: Nuha

value = 390;  //number
console.log(value);  // Output: 390

value = true; //boolean
console.log(value);  // Output: true
```

Here, a variable "value" can be reassigned to a string, number, or boolean without any type errors. However, excessive use of any can lead to runtime errors since TypeScript won't enforce type safety.

Unknown - Similar to any but requires type checking.

```
let data: unknown = "test";
```

Void - Represents functions that do not return anything.

```
function logMessage(): void { console.log("Hello!"); }
```

Null & Undefined - Represents absence of value.

```
let empty: null = null;
let notDefined: undefined = undefined;
```

Never - Represents values that never occur, used in functions that throw errors.

```
function error(message: string): never { throw new Error(message); }
```

Example Ts:

```
// Primitive Types (number, string, boolean)
let age: number = 25;
let username: string = "nuha";  // Changed from "harsh"
let isLoggedIn: boolean = true;

// Arrays
let numbers: number[] = [1, 2, 3, 4];
let names: string[] = ["Alice", "Bob", "Charlie"];

// Tuples (Fixed-length array with specific types)
let arr: [string, number] = ["nuha", 25];  // Changed from "harsh"

// Enums (Define named constants)
enum UserRoles {
    ADMIN = "admin",
    GUEST = "guest",
    SUPER_ADMIN = "super_admin"
}

// Using Enums
let userRole: UserRoles = UserRoles.ADMIN;
console.log(userRole);  // Output: "admin"

// Any, Unknown, Void, Null, Undefined, Never
let randomValue: any = 10;  // Can be any type
randomValue = "Now it's a string";  // No error
```

```
let something: unknown = "I am unknown"; // Safe than any, requires type checking
if (typeof something === "string") {
    console.log(something.toUpperCase());
}

function logMessage(): void {
    console.log("This function returns nothing");
}

let emptyValue: null = null;
let notDefined: undefined = undefined;

function throwError(message: string): never {
    throw new Error(message);
}
```

And converted js :

```
// Primitive Types (number, string, boolean)
var age = 25;
var username = "nuha"; // Changed from "harsh"
var isLoggedIn = true;
// Arrays
var numbers = [1, 2, 3, 4];
var names = ["Alice", "Bob", "Charlie"];
// Tuples (Fixed-length array with specific types)
var arr = ["nuha", 25]; // Changed from "harsh"
// Enums (Define named constants)
var UserRoles;
(function (UserRoles) {
    UserRoles["ADMIN"] = "admin";
    UserRoles["GUEST"] = "guest";
```

```
    UserRoles["SUPER_ADMIN"] = "super_admin";
})(UserRoles || (UserRoles = {}));
// Using Enums
var userRole = UserRoles.ADMIN;
console.log(userRole); // Output: "admin"
// Any, Unknown, Void, Null, Undefined, Never
var randomValue = 10; // Can be any type
randomValue = "Now it's a string"; // No error
var something = "I am unknown"; // Safe than any, requires type checking
if (typeof something === "string") {
    console.log(something.toUpperCase());
}
function logMessage() {
    console.log("This function returns nothing");
}
var emptyValue = null;
var notDefined = undefined;
function throwError(message) {
    throw new Error(message);
}
```

Alias:
A way to create a new name for a type, making it easier to reuse and manage complex types is called alias.
Example in TS:

```
type User = {
    name: string;
    email: string;
    isActive: boolean;
}
```

```
function createUser(user: User): User {
    return { name: "", email: "", isActive: true };
}

createUser({ name: "Alice", email: "alice@example.com", isActive: true });
```

And converted js :

```
function createUser(user) {
    return { name: "", email: "", isActive: true };
}
createUser({ name: "Alice", email: "alice@example.com", isActive: true });
```

Union in TS:
Union Type allows a variable to hold multiple types.

TS example:

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

  //declaring a user object
  let hitesh: User = { name: "Hitesh", id: 334 };
  //union
  function getDbId(id: number | string) {
    //making some API calls
    console.log(`DB id is: ${id}`);
  }

  // Function calls with different types
```

```
getDbId(101);
getDbId("abc");

let score: number | string = "55";
score = 55;
score = "Sixty";
```

Converted JS:

```
//declaring a user object
var hitesh = { name: "Hitesh", id: 334 };
//union
function getDbId(id) {
    //making some API calls
    console.log("DB id is: ".concat(id));
}
// Function calls with different types
getDbId(101);
getDbId("abc");
var score = "55";
score = 55;
score = "Sixty";
```

Intersection :

Example in TS:

```
let a: string | null; // Union

// user
type NewUser = {
    name: string;
    email: string;
};
```

```
type Admin = NewUser & { //intersection
    getDetails(userInfo: string): void;
};

function abcd(a: Admin) {
    a.getDetails("User details here");
}
// Example
const adminUser: Admin = {
    name: "Nuha",
    email: "n@h.com",
    getDetails: (userInfo: string) => {
        console.log("Admin Details:", userInfo);
    },
};

abcd(adminUser);
```

And converted JS:

```
var a; // Union
function abcd(a) {
    a.getDetails("User details here");
}
// Example
var adminUser = {
    name: "Nuha",
    email: "n@h.com",
    getDetails: function (userInfo) {
        console.log("Admin Details:", userInfo);
    },
};
```

```
abcd(adminUser);
```

Tuples:
These are used to define arrays with fixed types for each position.
Example in TS:

```
let tUser: [string, number, boolean] = ["hc", 131, true];

let rgb: [number, number, number] = [255, 123, 112];

// Creating a tuple type alias for user
type User = [number, string];

//User tuple
const newUser: User = [112, "example@google.com"];
```

Converted JS:

```
var tUser = ["hc", 131, true];
var rgb = [255, 123, 112];
//User tuple
var newUser = [112, "example@google.com"];
```

Named Tuples:

Named tuples allow us to give meaningful names to values at each index, improving code readability.

Destructuring Tuples

Since tuples are essentially arrays, we can use destructuring to extract values into separate variables.

Example in TS:

```
//named tuple

const graph: [x: number, y: number] = [55.2, 41.3];

console.log(graph[0]); // 55.2

console.log(graph[1]); // 41.3


//destructing tuple

const graph: [number, number] = [55.2, 41.3];

const [x, y] = graph;


console.log(x); // 55.2

console.log(y); // 41.3
```

And in converted JS :

```
//named tuple

var graph = [55.2, 41.3];

console.log(graph[0]); // 55.2

console.log(graph[1]); // 41.3

//destructing tuple

var graph = [55.2, 41.3];

var x = graph[0], y = graph[1];
```

```
console.log(x); // 55.2

console.log(y); // 41.3
```

Optional and ReadOnly:
A good practice is to make your tuple readonly.

Example Typescript:

```
type User = {
    readonly _id: string;
    name: string;
    email: string;
    isActive: boolean;
}

let myUser: User = {
    _id: "1245",
    name: "John",
    email: "john@example.com",
    isActive: true
};

// myUser._id = "5678";
// ❌ Error: Cannot assign to '_id' because it is a read-only property.
```

Here, readonly prevents modification of _id after the object is created.

And converted js :

```
var myUser = {
    _id: "1245",
    name: "John",
```

```
    email: "john@example.com",
    isActive: true
};
// myUser._id = "5678";
// ❌ Error: Cannot assign to '_id' because it is a read-only property.
```

Example of Both :

```
//ReadOnly
type User = {
    readonly _id: string;
    name: string;
    email: string;
    isActive: boolean;
}

let myUser: User = {
    _id: "1245",
    name: "John",
    email: "john@example.com",
    isActive: true
};

// myUser._id = "5678";
// ❌ Error: Cannot assign to '_id' because it is a read-only property.
//ReadOnly with Arrays
type ReadonlyArrayExample = {
    readonly values: number[];
}

let obj: ReadonlyArrayExample = {
    values: [1, 2, 3]
};
```

```
// obj.values.push(4);  ❌ Error: Cannot push to 'values' because it is a read-only array.


//optional
type User = {
  _id: string;
  name: string;
  email?: string; // Optional property
}

let user1: User = { _id: "123", name: "Alice" };  //Works without email
let user2: User = { _id: "456", name: "Bob", email: "bob@example.com" }; //Works with email
```

And Converted JS :

```
var myUser = {
  _id: "1245",
  name: "John",
  email: "john@example.com",
  isActive: true
};
var obj = {
  values: [1, 2, 3]
};
var user1 = { _id: "123", name: "Alice" }; // ✅ Works without email
var user2 = { _id: "456", name: "Bob", email: "bob@example.com" }; // ✅ Works with email
```

Enums:

Enums exist because there are certain times when you want to restrict somebody's choice or with the values that are offered here, making the code more readable and maintainable.

Example in TS

```
//ENUMS -> define funct , const, any string
// suppose in a plane ticket booking 3 aeroplane seat
//const aisle = 0
//const middle = 1
//const windoww = 2
//if(seat === aisle){}

//in enums first var is default as 0, then increement of 1,
// you can also set it but next will be as the same increement 1 of that number...
const enum SeatChoice{  //use const before enum to avoid long code
    aisle,
    middle,
    windoww,
    fourth
}

const hcSeat = SeatChoice.aisle
```

And converted JS :

```
//ENUMS -> define funct , const, any string
// suppose in a plane ticket booking 3 aeroplane seat
//const aisle = 0
//const middle = 1
//const windoww = 2
//if(seat === aisle){}
var hcSeat = 0 /* SeatChoice.aisle */;
```

Interfaces in TS: Interface create objects shape, while Type

```
interface User {
   readonly dbID: number;
   email: string;
   userID: number;
   googleID?: string;

   // Trial User
   startTrial(): string;

   // Discount
   getCoupon(couponname: string, value: number): number;
}

const nuha: User = {
   dbID: 23,
   email: "n@h.com",
   userID: 2211,

   // Trial User
   startTrial: () => {
      return "trial started";
   },

   // Discount
   getCoupon: (couponname, value) => {
      return value;
   }
};

// modifying allowed
```

```
nuha.email = "n@hdhdf.com";

console.log(nuha.startTrial());
console.log(nuha.getCoupon("nuha30", 30));
```

In converted JS:

```
var nuha = {
    dbID: 23,
    email: "n@h.com",
    userID: 2211,
    // Trial User
    startTrial: function () {
        return "trial started";
    },
    // Discount
    getCoupon: function (couponname, value) {
        return value;
    }
};
// modifying allowed
nuha.email = "n@hdhdf.com";
console.log(nuha.startTrial());
console.log(nuha.getCoupon("nuha30", 30));
```

Another example in TS:
Interfaces can be extended and reopened multiple times and Types cannot be changed once declared.

```
interface User {
    readonly dbID: number;
    email: string;
```

```
    userID: number;
    googleID?: string;

    // Trial User
    startTrial(): string;

    // Discount
    getCoupon(couponname: string, value: number): number;
}

//Interface vs Type -
// One can easily extend or reopen the interface & it has advantage in inheritance
interface User {
    githubToken: string;

 }

const nuha: User = {dbID: 23, email: "n@h.com", userID: 2211,
    githubToken: "NUHA24"
    startTrial: () => {
        return "trial started";  },
    // Discount
    getCoupon: (couponname, value) => {
        return value;
    }
};

// modifying allowed
nuha.email = "n@hdhdf.com";
```

And converted JS:

```
var nuha = { dbID: 23, email: "n@h.com", userID: 2211,
    githubToken: "NUHA24",
    startTrial: function () {
        return "trial started";
    },
    // Discount
    getCoupon: function (couponname, value) {
        return value;
    }
};
// modifying allowed
nuha.email = "n@hdhdf.com";
```

Some Basic Commands for a project :

Commands	Purposes
mkdir -Path src, dist	Creates src/ and dist/ folders
npm init -y	Initialize a package.json (If not present)
Move-Item index.ts src/	Moves index.ts into src/
tsc src/index.ts	tsc src/index.ts
tsc src/index.ts	Creates a valid package.json
npm install lite-server --save-dev	Install lite-server (For running project locally)
npm start	Runs lite-server
code .	Opens vs code in current directory

Class : In this case class members (both properties and methods) are typed using type annotations, just like regular variables.

Example of TS:

```
class User {
   email: string;
   name: string;

   constructor(email: string, name: string) {
       this.email = email;
       this.name = name;
   }
}

const nuha = new User("n@ee.com", "Nuha");

class APerson {
   name: string;
  }

  const person = new APerson();
  person.name = "Nuhaaaa";
  console.log(person.name);
```

And in converted JS:

```
var User = /** @class */ (function () {
   function User(email, name) {
       this.email = email;
       this.name = name;
   }
   return User;
}());
var nuha = new User("n@ee.com", "Nuha");
```

```
var APerson = /** @class */ (function () {
    function APerson() {
    }
    return APerson;
}());
var person = new APerson();
person.name = "Nuhaaaa";
console.log(person.name);
```

Access Modifiers : 3

public → Accessible everywhere.
private → Accessible only within the class.
protected → Accessible within the class and subclasses.

```
class User {
    public email: string; //everyone can access
    private password: string; //only user class can access
    protected age: number; //only this and its subclass can access
    //readonly -> something once set, can never change
    readonly city: string = "Dhaka";

    constructor(email: string, password: string, age: number)
    {
      this.email = email;
      this.password = password;
      this.age = age;
    }
    public getEmail(): string { //we dont need it as its already public
      return this.email;
    }
    private getPassword(): string {
      return this.password;
```

```
    //method to get password as its private accessed only in this class
  }
  protected getAge(): number {
    return this.age;
    //only this and its subclass can as its protected
  }
}

class Admin extends User { //suppose admin is extending user
  constructor(email: string, password: string, age: number) {
    super(email, password, age);
  }
  getAdminAge(): number {
    return this.age;
    // admin shall check their age and it works because age is protected
  } }
const user = new User("nuha@example.com", "superSecret", 25);
console.log(user.email);
console.log(user.city);
console.log(user.password); //none can access it and it's private
console.log(user.age);      // none can access it and it's protected

//suppose a new admin
const admin = new Admin("admin@example.com", "adminPass", 30);
console.log(admin.getAdminAge()); // works as its using a protected property
```

And in Converted JS:

```
var __extends = (this && this.__extends) || (function () {
    var extendStatics = function (d, b) {
        extendStatics = Object.setPrototypeOf ||
```

```
        ({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ =
b; }) ||
                        function (d, b) { for (var p in b) if
(Object.prototype.hasOwnProperty.call(b, p)) d[p] = b[p]; };
    return extendStatics(d, b);
  };
  return function (d, b) {
    if (typeof b !== "function" && b !== null)
        throw new TypeError("Class extends value " + String(b) + " is not a
constructor or null");
    extendStatics(d, b);
    function __() { this.constructor = d; }
     d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype,
new __());
  };
})();
var User = /** @class */ (function () {
  function User(email, password, age) {
    //readonly -> something once set, can never change
    this.city = "Dhaka";
    this.email = email;
    this.password = password;
    this.age = age;
  }
  User.prototype.getEmail = function () {
    return this.email;
  };
  User.prototype.getPassword = function () {
    return this.password;
    //method to get password as its private accessed only in this class
  };
  User.prototype.getAge = function () {
```

```
        return this.age;
        //only this and its subclass can as its protected
    };
    return User;
}());
var Admin = /** @class */ (function (_super) {
    __extends(Admin, _super);
    function Admin(email, password, age) {
        return _super.call(this, email, password, age) || this;
    }
    Admin.prototype.getAdminAge = function () {
        return this.age;
        // admin shall check their age and it works because age is protected
    };
    return Admin;
}(User));
var user = new User("nuha@example.com", "superSecret", 25);
console.log(user.email);
console.log(user.city);
console.log(user.password); //none can access it and it's private
console.log(user.age); // none can access it and it's protected
//suppose a new admin
var admin = new Admin("admin@example.com", "adminPass", 30);
console.log(admin.getAdminAge()); // works as its using a protected property
```

Getter & Setters:

```
class NUser {
    public email: string;
    private courseCount = 1;

    constructor(email: string, password: string, age: number)
    {
```

```
    this.email = email;
  }

  //annotate with get keyword ---> getter
  get getAppleEmail(): string{
      return `apple ${this.email}`
  }

    get coureCCount(): number{ //user for get any property --> mostly private property
      return this.courseCount
  }
  //setter cant be void/or any other data type!! no return type!
  set coureCCount(courseNum){
    if(courseNum <= 1)
    {
      throw new Error("Course Count Shall be more than 1")
    }
    this.courseCount=courseNum
  }
 }
```

In converted JS:

```
var NUser = /** @class */ (function () {
    function NUser(email, password, age) {
        this.courseCount = 1;
        this.email = email;
    }
    Object.defineProperty(NUser.prototype, "getAppleEmail", {
        //annotate with get keyword ---> getter
        get: function () {
```

```
        return "apple ".concat(this.email);
    },
    enumerable: false,
    configurable: true
});
Object.defineProperty(NUser.prototype, "coureCCount", {
    get: function () {
        return this.courseCount;
    },
    //setter cant be void/or any other data type!! no return type!
    set: function (courseNum) {
        if (courseNum <= 1) {
            throw new Error("Course Count Shall be more than 1");
        }
        this.courseCount = courseNum;
    },
    enumerable: false,
    configurable: true
});
return NUser;
}());
```

Abstract Class:

An abstract class in TypeScript is like a blueprint for other classes. An abstract class :

1. Cannot be instantiated
2. Defines common properties and methods
3. Abstract methods must be implemented in subclasses
4. Can have concrete methods with implementation
5. One cannot create an object from it directly. Instead, other classes must extend it and implement its properties and methods. Inherited using extends.
6. Requires super() in subclasses
7. Enforces structure and code reuse

Example TS:

```
// class TakePhoto {
 //constructor(
  // public cameraMode: string,
  // public filter: string
 //) {}
//}
abstract class TakePhoto {
   constructor(
     public cameraMode: string,
     public filter: string,
     public burst: number
   ) {}
 //method definition
 abstract getSepia(): void
 getReelTime(): number{
 //some complex calculation
 return 8
 }
 //if we write abstract before any method / feature and can have overriding
 }

//const hc = new TakePhoto("test", "Test");
// abstract classes cannot create object of their own
//but they help you define the classes who are inhereting them
//in the correct way we have to extend the abstract class
// --> extend is like having a inheritance here
class Instagram extends TakePhoto {
 constructor(
   public cameraMode: string,
```

```
    public filter: string,
    public burst: number) {
    super(cameraMode, filter, burst); //must use super
  }
  getSepia() : void{
    console.log("Sepia");
  }
}
const nh = new Instagram("testing", "testing", 7)
```

And in JS:

```
var __extends = (this && this.__extends) || (function () {
    var extendStatics = function (d, b) {
        extendStatics = Object.setPrototypeOf ||
            ({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ = b; }) ||
            function (d, b) { for (var p in b) if (Object.prototype.hasOwnProperty.call(b, p)) d[p] = b[p]; };
        return extendStatics(d, b);
    };
    return function (d, b) {
        if (typeof b !== "function" && b !== null)
            throw new TypeError("Class extends value " + String(b) + " is not a constructor or null");
        extendStatics(d, b);
        function __() { this.constructor = d; }
        d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
    };
})();
// class TakePhoto {
//constructor(
```

```
// public cameraMode: string,
// public filter: string
//) {}
//}
var TakePhoto = /** @class */ (function () {
    function TakePhoto(cameraMode, filter, burst) {
        this.cameraMode = cameraMode;
        this.filter = filter;
        this.burst = burst;
    }
    TakePhoto.prototype.getReelTime = function () {
        //some complex calculation
        return 8;
    };
    return TakePhoto;
}());
//const hc = new TakePhoto("test", "Test");
// abstract classes cannot create object of their own
//but they help you define the classes who are inhereting them
//in the correct way we have to extend the abstract class
// --> extend is like having a inheritance here
var Instagram = /** @class */ (function (_super) {
    __extends(Instagram, _super);
    function Instagram(cameraMode, filter, burst) {
        var _this = _super.call(this, cameraMode, filter, burst) || this; //must use
super
        _this.cameraMode = cameraMode;
        _this.filter = filter;
        _this.burst = burst;
        return _this;
    }
    Instagram.prototype.getSepia = function () {
```

```
    console.log("Sepia");
  };
  return Instagram;
}(TakePhoto));
var nh = new Instagram("testing", "testing", 7);
```

Generics: reusable components while maintaining type safety

1. for functions/arrays
2. Avoids specific type declarations
3. Uses <T> or any H, J, I, A for flexibility
4. Ensures type safety
5. Supports custom types
6. Prevents redundancy

Example TS:

```
//<>
const Score: Array<number> = []
const Name: Array<string> = []

function identityOne(val: boolean | number): boolean | string | number{
  return val
}

function identityTwo(val: any): any{
  return val
}
function identityThree<Type>(val: Type): Type{ //return type same
  return val
}

//function identityThree<"3">(val: "3"): "3"
identityThree("3")
```

```
function identityFour<T>(val: T): T{ //exact as Type
    return val
}

interface bottle{
    brand: string,
    type: number,
}
//identityFour<bottle>({})

function getSearchProducts<T>(products: T[]): T{
   //database operations
   const myIndex=5
    return products[myIndex]
}
//how to define a arrow function & definition <T>(): => {}
// , -> to ensure its not an ordinary syntax rather a syntax for generics
const getMoreSearchProducts = <T,>(products: T[]): T => {
    const myIndex=4
    return products[myIndex]
}
```

And in JS:

```
//<>
var Score = [];
var Name = [];
function identityOne(val) {
    return val;
}
function identityTwo(val) {
    return val;
```

```
}
function identityThree(val) {
    return val;
}
//function identityThree<"3">(val: "3"): "3"
identityThree("3");
function identityFour(val) {
    return val;
}
//identityFour<bottle>({})
function getSearchProducts(products) {
    //database operations
    var myIndex = 5;
    return products[myIndex];
}
//how to define a arrow function & definition <T>(): => {}
// , -> to ensure its not an ordinary syntax rather a syntax for generics
var getMoreSearchProducts = function (products) {
    var myIndex = 4;
    return products[myIndex];
};
```

Type parameters in & class type in generics Generic:

1. Scalability → This approach allows handling different types (Course, Product, etc.) without writing separate classes for each.
2. Organization → The code remains structured and clean, even as the project grows.
3. Flexibility → The Sellable<T> class can work with any type, adapting to project needs without major modifications.

In the example of TS:

```
//() input
```

```
function anotherFunction<T, U extends string>(valOne: T, valTwo: U): object{
 return{
   valOne,
   valTwo
 }
}
interface Database{
   connection: string,
   username: string,
   password: string

}
//function anotherFunction<T, U extends number>
anotherFunction(3, "4")
//if we use U extends number then it will show error when we use string value

function Function2<T, U extends Database>(valOne: T, valTwo: U): object{
   return{
     valOne,
     valTwo
   }
  }

  //Function2(3, {})

  //class type in generics
interface Quiz{
   name: string,
   type: string,
}

interface Course{
```

```
    name: string,
    author: string,
    subject: string
}

class sellable<T>{
    public cart: T[] = []

    addToCart(products: T){
        this.cart.push(products)
    }
}
```

And in JS:

```
//() input
function anotherFunction(valOne, valTwo) {
    return {
        valOne: valOne,
        valTwo: valTwo
    };
}
//function anotherFunction<T, U extends number>
anotherFunction(3, "4");
//if we use U extends number then it will show error when we use string value
function Function2(valOne, valTwo) {
    return {
        valOne: valOne,
        valTwo: valTwo
    };
}
var sellable = /** @class */ (function () {
```

```
    function sellable() {
        this.cart = [];
    }
    sellable.prototype.addToCart = function (products) {
        this.cart.push(products);
    };
    return sellable;
}());
```

Type narrowing :
It means to refine a variable's type within a specific block of code based on conditions.

Example In js:

```
function detectType(val: number | string) {
    if (typeof val === "string") {
        return val.toLowerCase();
    }
    return val + 3;
}

function provideId(id: string | null) {
    if (!id) {
        console.log("Please provide ID");
        return;
    }
    id.toLowerCase();
}
```

And in TS:

```
function detectType(val) {
    if (typeof val === "string") {
        return val.toLowerCase();
```

```
    }
    return val + 3;
}
function provideId(id) {
    if (!id) {
        console.log("Please provide ID");
        return;
    }
    id.toLowerCase();
}
```

“in” operator narrowing: “in” operator checks if a specific property exists in an object.

Example in TS:

```
interface UUser{
    name: string,
    email: string
}

interface Adminn{
    name: string,
    email: string,
    isAdmin: boolean
}

function isAdminAccount(account: UUser | Adminn)
{
    if("isAdmin" in account){
        return account.isAdmin
    }
}
```

And JS:

```
function isAdminAccount(account) {
   if ("isAdmin" in account) {
      return account.isAdmin;
   }
}
```

“instanceof” Narrowing: it helps narrow down types dynamically, checks if it was an instance of some class or maybe something like that

“Type Predicates” : type predicates are functions that return a boolean value and are used to narrow down the type of a variable.

```
function logValue(x: Date | string)
{
   if (x instanceof Date)  //almost like TypeOf
   {
     console.log(x.toUTCString()); //here ts knows x is a Date here
   }
   else
   {
     console.log(x.toUpperCase()); //and ts knows x is a string here
   }
}


//type Predicates
type Fish={swim: ()=> void};

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

function isFish(pet: Fish | Bird)
{
```

```
  return (pet as Fish).swim !== undefined
  //pet is a Fish if it has a method of .swim not undefined then true
}

//function getFood(pet: Fish | Bird){
   function getFood(pet: Fish | Bird): pet is Fish{
      //has to be boolean so pet is Fish
   if (isFish(pet)){
      pet
      //return "it is fish food"
      return true
   }
   else{
   pet
  // return "It is a birds food"
  return false
   }
}
```

And in JS:

```
function logValue(x) {
   if (x instanceof Date) //almost like TypeOf
   // so here we checked wether the x is an instance od date or not
   {
      console.log(x.toUTCString()); //here ts knows x is a Date here
   }
   else {
      console.log(x.toUpperCase()); //and ts knows x is a string here
   }
}
function isFish(pet) {
   return pet.swim !== undefined;
```

```
    //pet is a Fish if it has a method of .swim not undefined then true
}
//function getFood(pet: Fish | Bird){
function getFood(pet) {
    //has to be boolean so pet is Fish
    if (isFish(pet)) {
        pet;
        //return "it is fish food"
        return true;
    }
    else {
        pet;
        // return "It is a birds food"
        return false;
    }
}
```

DiscriminatedUnion:
In TS:

```
interface Circle {
    kind: "circle";
    radius: number;
}

interface Square {
    kind: "square";
    side: number;
}

interface Rectangle {
    kind: "rectangle";
    length: number;
```

```
    width: number;
}

type Shape = Circle | Square | Rectangle;

function getArea(shape: Shape): number {
    switch (shape.kind) {
        case "circle":
            return Math.PI * shape.radius ** 2;
        case "square":
            return shape.side * shape.side;
        case "rectangle":
            return shape.length * shape.width;
        default:
            const exhaustiveCheck: never = shape;
            throw new Error(`Unhandled case: ${exhaustiveCheck}`);
    }
}
```

And in JS:

```
function getArea(shape) {
    switch (shape.kind) {
        case "circle":
            return Math.PI * Math.pow(shape.radius, 2);
        case "square":
            return shape.side * shape.side;
        case "rectangle":
            return shape.length * shape.width;
        default:
            var exhaustiveCheck = shape;
```

```
        throw new Error("Unhandled case: ".concat(exhaustiveCheck));
    }
}
```

Overloading: means a function can have multiple ways to be called with different inputs and return types.

In TS:

```
// Overloads ->

// ts function signature
function abcd(a: string): void;
function abcd(a: string, b: number): number;

function abcd(a: any, b?: any) {
    if (typeof a === "string" && b === undefined) {
        console.log("hey");
    }
    if (typeof a === "string" && typeof b === "number") {
        return 123;
    } else throw new Error("something is wrong");
}
```

And in JS:

```
// Overloads ->
function abcd(a, b) {
    if (typeof a === "string" && b === undefined) {
        console.log("hey");
    }
    if (typeof a === "string" && typeof b === "number") {
        return 123;
    }
```

```
    else
        throw new Error("something is wrong");
}
```

Rest & Spread parameters: used with 3 dots …

Rest is used when collecting multiple values into an array.

Spread is used when expanding or copying an array or object.

In TS:

```
//function abcd(...args: number[]){
    // ...rest/spread -- only 3 dots
   //}
function func(...arr: number[]) {
    console.log(arr); // [1,2,3,4,5,6,7,8,9,10]
}

func(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

//spread
const num = [1, 2, 3, 4, 5];
console.log(...num); // 1 2 3 4 5

const moreNumbers = [...num, 6, 7, 8];
console.log(moreNumbers);
// [1, 2, 3, 4, 5, 6, 7, 8]

const person = { name: "Nuha", age: 434243 };
const copy = { ...person };

console.log(copy);
// {name: "Nuha", age: 434243 }
```

And in JS:

```
var __assign = (this && this.__assign) || function () {
    __assign = Object.assign || function(t) {
        for (var s, i = 1, n = arguments.length; i < n; i++) {
            s = arguments[i];
            for (var p in s) if (Object.prototype.hasOwnProperty.call(s, p))
                t[p] = s[p];
        }
        return t;
    };
    return __assign.apply(this, arguments);
};
var __spreadArray = (this && this.__spreadArray) || function (to, from, pack) {
    if (pack || arguments.length === 2) for (var i = 0, l = from.length, ar; i < l; i++) {
        if (ar || !(i in from)) {
            if (!ar) ar = Array.prototype.slice.call(from, 0, i);
            ar[i] = from[i];
        }
    }
    return to.concat(ar || Array.prototype.slice.call(from));
};
//function abcd(...args: number[]){
// ...rest/spread -- only 3 dots
//}
function func() {
    var arr = [];
    for (var _i = 0; _i < arguments.length; _i++) {
        arr[_i] = arguments[_i];
    }
```

```
    console.log(arr); // [1,2,3,4,5,6,7,8,9,10]
}
func(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
//spread
var num = [1, 2, 3, 4, 5];
console.log.apply(console, num); // 1 2 3 4 5
var moreNumbers = __spreadArray(__spreadArray([], num, true), [6, 7, 8], false);
console.log(moreNumbers);
// [1, 2, 3, 4, 5, 6, 7, 8]
var person = { name: "Nuha", age: 434243  };
var copy = __assign({}, person);
console.log(copy);
// {name: "Nuha", age: 434243 }
```