2-1: Type assertion/type narrowing

Type Assertion/ Narrowing কি?

আমি ডেভলপার হিসাবে Typescript এর চেয়ে যদি Batter Type Declared করতে পারি বা বুঝতে পারি এ টাই হচ্ছে Type Assertion / Narrowing.

একটি টাইপ Declared করলাম `let anything : any;` । এখন যেহেতু anything variable এর টাইপ আমি any দিয়েছি তার মানে এখন চাইলে আমি anything এর মান যে কোনো কিছু সেট করতে পারি যেমনঃ`( number , string , Boolean , etc.)` typescript কিছু মনে করবে না । এখন যদি আমি `anything = "learning next level web development"` নামে একটি string Declared করি তাহলে আমার `anything.` দিলে string এর সকল method পাওয়ার কথা কিন্তু আমি পাবো না কড়ন সে এখন পর্যন্ত জানে যে, anything এর টাইপ any দেওয়া যদি আমি চাই string এর মেথড গুলো নিয়ে কাজ করতে তাহলে আমাকে বলে দেওয়া লাগবে as দিয়ে (anything as number/string/Boolean) যদি আমি এইভাবে বলে দেই যে, anything এর টাইপ এখন `(string or number or Boolean etc.)` তখন সে ওই অনুযায়ি রেজাল্ট দিবে।

তার মানে আমি typescript কে বলে দিচ্ছি যে, anything এর মান কি হবে । মানে আমি typescript এর থেকে বেশি বুঝি আর এই টাই হলো Type Assertion

Try & Catch Block:

Typescript এ try catch block এ যদি -

```
catch (error) {
        console.log(error.message); // 'error' is of type 'unkn
        // console.log((error as CustomError).message)
    };
```

যদি error message টাকে console করতে চাই তখন আসলে একটি error দিবে। typescript error type কে নির্নয় করতে পারছে না। তাই এই সমস্যা টা কে সমাধান করার জন্য একটি `custom error type Declared` করতে হয় আর আমরা তো জানিই যে, error এর ভিতরে message নামক

একটি value belong করে তাই as দিয়ে error টাকে custom error type বলে দিচ্ছি যে, error এর ভিতরে একটি message নামক value আছে।

```ts
{
  ////////////////////////// Start /////////////////

  // Type Assertion

  let anything: any;

  anything = "learning Next Level Web Development";

  // (anything as string).length;

  anything = 265;

  (anything as number).toFixed();

  const kgToGm = (value: string | number): string | number | undefined => {
    if (typeof value === "string") {
      const convertedValue = parseFloat(value) * 1000;
      return `The converted Value is ${convertedValue}`;
    } else {
      return value * 1000;
    }
  };

  const result1 = kgToGm(1000) as number;
  const result2 = kgToGm("2000") as string;

  console.log(result2);

  // try catch

  type CustomError = {
    message: string;
  };

  try {
  } catch (error) {
    // console.log(error.message); // 'error' is of type 'unknown'.
    console.log((error as CustomError).message);
  }

  ////////////////////////// End //////////////////////////
}
```

2-2: Interface, type vs interface

Type Alias এর জাত ভাই এর নাম হচ্ছে Type Interface। Interface দিয়ে ও Typescript এ Type Declared করা যায়। Type alias এর ক্ষেত্রে Type লিখতে হতো আর Interface এর ক্ষেত্রে

লিখতে হবে Interface। চাইলে কোনো একটি variable এর object type টা Type alias এবং Type Interface উভয়ের মাধ্যমেই করা যাবে। কোন একটি variable এর object Type টা চাইলে Type Alias এবং Type Interface দুইটাই ব্যবহার করা যায় ।

Type alias এবং Type Interface উভয়ের মধ্যেই কিছু পার্থক্য রয়েছে -

Type alias	Type Interface
1. Type Alias Primitive Data Type এর ক্ষেত্রে ব্যবহার করা যায় `type RollNumber = number ;`	1. Interface Alias Primitive Data Type এর ক্ষেত্রে ব্যবহার করা যায় না। `interface User {};`
2. Type Alias এর মাধ্যমে কোন একটা property যদি আমরা বাড়াতে চাই তাহলে আমরা Intersection (&) ব্যবহার করতে পারি।	2. Type Interface এর মাধ্যমে কোন একটা property যদি আমরা বাড়াতে চাই তাহলে আমরা extends keyword ব্যবহার করতে পারি।
3. Type Alias এর মাধ্যমে Array, Function Declared করা যায়।	3. Type Interface এর মাধ্যমে Array, Function Declared করা যায়।
4. Type Alias সবচেয়ে গ্রহণযোগ্য । Object বাদে প্রায় সব গুলো তে এই টাইপ গুলো ব্যবহার করা Recommended।	4. Type Interface Object এর বেলায় ব্যবহার করা Recommended।

```
{
  ////////////////////////// Start ////////////////

  // Type Vs Interface

  // Type alias:-
  type User1 = {
    name: string;
    age: number;
  };

  // const user1 : User1 = {
  //     name : 'sifat',
  //     age : 23,
```

```
// };

const user1: User2 = {
  // type alias এর পরিবর্তে type interface ব্যবহার করলে ও কোন সমস
  name: "sifat",
  age: 23,
};

type RollNumber1 = number; // type alias

type UserWithRole1 = User1 & { role: string }; // added extra

const user11: UserWithRole1 = {
  name: "sifat",
  age: 23,
  role: "Developer",
};

// Array
type Roll2 = number[];

const roll2: Roll2 = [1, 2, 3, 4];

// Function
type Add1 = (num1: number, num2: number) => number;

const add1: Add1 = (num1, num2) => num1 + num2;

// Type Interface
interface User2 {
  name: string;
  age: number;
}

// const user2 : User2 = {
//     name : "sifat",
```

```
  //     age : 23
  // };

  type RollNumber2 = {}; // type interface

  interface UserWithRole2 extends User2 {
    // extends Keyword
    role: string; // added extra properties
  }

  const user2: UserWithRole2 = {
    name: "sifat",
    age: 23,
    role: "Developer",
  };

  // Array
  interface Roll3 {
    [index: number]: number;
  }

  const roll3: Roll3 = [1, 2, 3, 4];

  // Function
  interface Add2 {
    (num1: number, num2: number): number;
  }
  const add2: Add2 = (num1, num2) => num1 + num2;

  ////////////////////////// End //////////////////////////
}
```

2-3: Introduction to generics

Generic Type কি?

Generic শব্দের অর্থ কি জেনারালাইস। আমরা যদি কোন একটি টাইপকে জেনারালাইস করতে পারি অর্থাৎ আমরা ডায়নামিক ভাবে যে, কোন সময় আমাদের প্রয়োজন অনুসারে টাইপ তৈরি করতে পারি এবং সেটাকে re-use করতে পারি সেটাই হলো Generic Type ।

একটি টাইপ Declared করবো Generic Type । উদাহরন হিসাবে Number Array type কে দেখানো হচ্ছে কিন্তু এইটা সব টাইপ (string,number, boolen) এর জন্যই প্রসেস সেম।

`type GenericArray = Array<Number>;`

`const rollNumbers: Array<number> = [1, 2, 3, 4, 5, 6, 7, 8];` এই ভাবে array type বলে দেওয়া যা `const rollNumbers: GenericArray<number> = [1, 2, 3, 4, 5, 6, 7, 8];` এই ভাবে বলে দেওয়া তাই।

GenericArray বর্তমানে static অবস্থায় রয়েছে যদি এই টাইপকে dynamic করে হয় তাহলে তার ভিতরে parameter pass করতে হবে যেমন টি একটি function এর ক্ষেত্রে হয়।

`type GenericArray<param> = Array<param>;` এই ভাবে GenericArray কে ডাইনামিক রূপদেওয়া যায় কিন্তু এইটা ভালো উপায় না। Industry standard অনুযায়ি `type GenericArray<T> = Array<T>;` এই ভাবে লিখতে হয়। `T = Typescript, T = Type` ।

```
{
  ////////////////////////// Start ///////////////

  // Generic Type

  // type GenericArray = Array<Number>;

  // type GenericArray<param> = Array<param>;

  type GenericArray<T> = Array<T>;

  // const rollNumbers: number[] = [1, 2, 3, 4, 5, 6, 7, 8];

  // const rollNumbers: Array<number> = [1, 2, 3, 4, 5, 6, 7, 8]

  const rollNumbers: GenericArray<number> = [1, 2, 3, 4, 5, 6,

  // const mentors: string[] = ['Mr.x' , 'Mr.y' , 'Mr.z' , ];
```

```
// const mentors: Array<string> = ['Mr.x' , 'Mr.y' , 'Mr.z' ,

const mentors: GenericArray<string> = ["Mr.x", "Mr.y", "Mr.z"]

// const boolArray: boolean[] = [true, false, false];

// const boolArray: Array<boolean> = [true, false, false];

const boolArray: GenericArray<boolean> = [true, false, false];

// function
const add = (x: number, y: number): number => x + y;

add(10, 20);

// array of objects:-
// const users : GenericArray<object> = [ // GenericArray<obje
//         {
//             name: 'sifat',
//             age: 25
//         },
//         {
//             name: 'shoyon',
//             age: "23"
//         }

// ];

const users: GenericArray<{ name: string; age: number }> = [
  // Declared specifically value is most preferable
  {
    name: "sifat",
    age: 25,
  },
  {
    name: "shoyon",
```

```
      age: 23,
    },
  ];

  // Generic Tuple

  type GenericTuple1<x, y> = [x, y];
  type GenericTuple2<x, y, z> = [x, y, z];

  const manus: GenericTuple1<string, string> = ["Mr.x", "Mr.y"];

  const userWithId: GenericTuple2<
    number,
    { name: string; email: string },
    boolean
  > = [2356, { name: "sifat", email: "a@gmail.com" }, true];

  ////////////////////////// End //////////////////////////
}
```

2-4: Generic with Interface

একটি variable বা object এ generic Declared করার পরে অনেক সময় দেখা যায় যে, অনেক সময় অনেক গুলো টাইপ হওয়ার কাড়নে এক লাইনে লিখার ফলে variable বা object টি অনেক বড় হয়ে যায় এবং দেখতে ও ভালো দেখায় না তাই কোড গুলোকে clean রাখার জন্য টাইপ গুলো আলাদা alias বা interface এর মাধ্যমে ওপরে Declared করার ফলে কোডগুলো ভালো দেখায় এবং বুঝতে সহজ হয়।

Function এর parameter এ যেমন default value set করা যায় generic ও ঠিক তেমন ভাবে default value set করা যায় এবং dynamic ভাবে ও type pass করা যায়।

```
{
  ////////////////////////// Start ///////////////

  // Interface - Generic
```

```
interface Developer<T, X = null> {
  name: string;
  computer: {
    brand: string;
    model: string;
    releaseYear: number;
  };
  smartWatch: T;
  bike?: X;
}

type EmilabWatch = {
  brand: string;
  model: string;
  display: string;
};

const poorDeveloper: Developer<EmilabWatch> = {
  name: "sifat",
  computer: {
    brand: "msi",
    model: "ku996t",
    releaseYear: 2013,
  },
  smartWatch: {
    brand: "Emilab",
    model: "kw66",
    display: "OLED",
  },
};

interface AppleWatch {
  brand: string;
  model: string;
  display: string;
```

```
    heartTrack: boolean;
    sleepTrack?: boolean;
  }

  interface YamahaBike {
    model: string;
    engineCapacity: string;
  }

  const richDeveloper: Developer<AppleWatch, YamahaBike> = {
    name: "shoyon",
    computer: {
      brand: "asus",
      model: "ku996p",
      releaseYear: 2016,
    },
    smartWatch: {
      brand: "apple",
      model: "k6",
      display: "OLED",
      heartTrack: true,
      sleepTrack: true,
    },
    bike: {
      model: "Yamaha",
      engineCapacity: "150cc",
    },
  };

  ////////////////////////// End //////////////////////////
}
```

2-5 Function with generics

Generics এর সাথে function এর কাজ ও করা হয়। একটি function এ multiply type এর value আসতে পারে নানান রকমের টাইপের তাই চাইলে generics এর মাধ্যমে dynamic ভাবে type বলে দেওয়া যায়। চাইলে Tuple ব্যবহার করা যায় with alias & interface types.

```
{
  ////////////////////////// Start ///////////////

  // Function with generics:

  const createArray = (params: string): string[] => {
    return [params];
  };

  const createArrayWithGenerics = <T>(params: T): T[] => {
    return [params];
  };

  const res1 = createArray("Bangladesh");
  const res2 = createArrayWithGenerics<string>("programming hero
  // const res3 = createArrayWithGenerics<object>({id : 1 , name

  // const res3 = createArrayWithGenerics<{id : number; name: st

  type User = { id: number; name: string }; // Type Aliases & Ty

  const res3 = createArrayWithGenerics<User>({ id: 1, name: "sit

  // console.log(res3);

  // create Array With Tuple:-
  const createArrayWithTuple = <T, Q>(params1: T, params2: Q):
    return [params1, params2];
  };

  const res11 = createArrayWithTuple<string, number>("Bangladesh
  const res22 = createArrayWithGenerics<string>("programming her
```

```
// const res3 = createArrayWithGenerics<object>({id : 1 , name

// const res3 = createArrayWithGenerics<{id : number; name: st

type User1 = { id: number; name: string }; // Type Aliases & T

const res33 = createArrayWithTuple<string, { id: number; name
  "Hello Generics",
  { id: 1, name: "sifat" }
); // This Way is perfect

// console.log(res33);

const addCourseToStudent = <T>(student: T) => {
  const courser = "Next Level Courser";
  return {
    ...student,
    courser,
  };
};

// Type Declared
interface IstudentInfo {
  name: string;
  email: string;
  devType?: string;
}

const student1 = addCourseToStudent<IstudentInfo>({
  name: "student1",
  email: "studnet1@gmail.com",
  devType: "level 1 developer",
});
const student2 = addCourseToStudent<IstudentInfo>({
  name: "student2",
  email: "studnet2@gmail.com",
```

```
    devType: "next level developer developer",
  });
  const student3 = addCourseToStudent<IstudentInfo>({
    name: "student3",
    email: "studnet3@gmail.com",
  });

  ////////////////////////// End //////////////////////////
}
```

2-6: Constraints in Typescript

Constraints বলতে কি বোঝায়?

⇒ Constraints শব্দের অর্থ হচ্ছে force করা অর্থাৎ কোন একটি role বা কোন কিছু enforce করার মানেই হচ্ছে Constraints।

Generic এর ভিতরে যদি Constraints keyword থাকে এর পরে যদি কোনো (obj , string , number etc.) থাকে তাহলে অবশ্যই পরবর্তীতে Generic যেই সব স্থানে ব্যবহার করা হবে তার ভিতরে (obj , string , number etc.) এই জিনিস গুলো থাকতে হবে নয় তো কাজ করবে না error দিবে।

```
{
  ////////////////////////// Start ///////////////

  // Constraints

  const addCourseToStudent = <T extends IstudentInfo>(student: T
    const courser = "Next Level Courser";
    return {
      ...student,
      courser,
    };
  };

  // Type Declared
```

```
  interface IstudentInfo {
    id: number;
    name: string;
    email: string;
    devType?: string;
  }

  const student1 = addCourseToStudent({
    id: 1,
    name: "student1",
    email: "studnet1@gmail.com",
    devType: "level 1 developer",
  });
  const student2 = addCourseToStudent({
    id: 2,
    name: "student2",
    email: "studnet2@gmail.com",
    devType: "next level developer developer",
  });
  const student3 = addCourseToStudent({
    id: 3,
    name: "student3",
    email: "studnet3@gmail.com",
  });

  const student4 = addCourseToStudent({
    id: 4,
    name: "student4",
    email: "studnet4@gmail.com",
    devType: "python developer",
  });

  ////////////////////////// End //////////////////////////
}
```

2-7 Constraint using a key of

যদি এমন কখনো এমন হয় যে, একটি টাইপ তৈরি করতে হবে union type যে খানের মধ্যে

স্ট্রিং লিটালের হিসাবে ব্যবহার হবে একটি property এর কিছু value।

type Owner = 'bikes' | 'car' | 'ship'; এইটাকে বলে manually ভাবে করা। same জিনিস টা একটি অপারেটর (keyof) এর মাধ্যমে। দুইটা জিনিসই একই অর্থ বহন করে।

একটি function এর ভিতরে যদি একটি অবজেক্ট কে পাঠানো হয় এবং তার value এবং key গুলো কে যদি access করতে হয় তাহলে Consstraint keyword এর ভিতরে keyof operator ব্যবহার সহজে পাওয়া যায় আর যদি জিনিস টাকে dynamic করতে হয় তাহলে Generic<> ব্যবহার করা যায়।

```
{
  ////////////////////////// Start ///////////////

  // Generic Gonstraint with keyof operator

  type Vehicle = {
    bike: string;
    car: string;
    ship: string;
  };

  type Owner = "bikes" | "car" | "ship"; // Manually

  type Owner2 = keyof Vehicle; // using keyof operator

  const person1: Owner = "bikes";

  const person2: Owner2 = "ship";
  // console.log(person2);

  // const getPropertyValue = (obj : object , key : string) =>
  //     return obj[key];
  // }; // use  basic
```

```
  // const getPropertyValue = <X, Y extends 'name' | 'age' | 'a
  //     return obj[key];
  // }; use union operator

  const getPropertyValue = <X, Y extends keyof X>(obj: X, key: Y
    return obj[key];
  };

  // object exm:
  const user = {
    name: "sifat",
    age: 64,
    address: "dhk",
  };

  const car = { model: "Toyota 100", year: 200 };

  const result1 = getPropertyValue(user, "age");
  const result2 = getPropertyValue(car, "year");

  // obj[key]

  ////////////////////////// End //////////////////////////
}
```

2-8: Asynchronous typescript

Asynchronous Typescript এ promise contractor declared করার সময় অবশ্যই tsconfig.ts file এর "terget" : "ES6" করে নিতে হবে না হইলে error দিবে।

what's the meaning of simulate?

⇒ TypeScript-এ "simulate" বলতে সাধারণত কোনো কিছুর মক, অনুকরণ, বা মডেল তৈরি করা বোঝায়, যা পুরো প্রক্রিয়া সম্পন্ন না করেই এর আচরণ পরীক্ষা বা প্রদর্শন করতে সহায়ক।

** Nodejs এর আপডেট ভার্ষনে by default fetch api method support করে যা নোড এর আগের ভার্ষনে করতো না।

`new Promise(resolve,reject) ⇒ {};` Promise Contractor এই রূপভাবে defined করতে হয়। promise 2টি parameter নেয় resolve & reject । Promise function টি যদি সঠিক ভাবে run করলে resolve এর ভিতরে যা থাকবে তা return করবে এবং যদি সঠিক ভাবে run না করে তাহলে reject এর ভিতরে যা থাকবে তা return করবে । চাইলে Promise এর ভিতরে type allies ও interface দিয়ে প্রমিস যা রিটার্ন করবে তার টাইপ ও বলে দেওয়া যায়।

```
{
  ////////////////////////// Start ///////////////

  // asynchronousTypescript

  // promise

  type Ttodo = {
    userId: number;
    id: number;
    title: string;
    completed: boolean;
  };

  const getTodo = async (): Promise<Ttodo> => {
    const respone = await fetch("https://jsonplaceholder.typicod
    const data = await respone.json();
    //    console.log(data);
    return data;
  };

  getTodo();

  type Tsomething = { something: string };

  // simulate
  const createPromise = (): Promise<Tsomething> => {
```

```
    return new Promise<Tsomething>((resolve, reject) => {
      const data: Tsomething = { something: "something" };
      if (data) {
        resolve(data);
      } else {
        reject("Failed to load data");
      }
    });
  };

  // calling create promise function

  const showData = async (): Promise<Tsomething> => {
    const data: Tsomething = await createPromise();
    // console.log(data);
    return data;
  };

  showData();

  ////////////////////////// End //////////////////////////
}
```

2-9: Conditional types

Conditional types কি?

Conditional type মানে হলো একটি টাইপ অন্য একটি টাইপ এর উপর কোন একটি কনডিশিন এর উপর ভিত্তি করে type নির্ধারন করে।

extends কি?

⇒ extends মানে হলো কোন কিছু আছে কিন। type a1 এর ভিতরে যদি null থাকে তাহলে return true or return false

```
  // conditional
```

```
type a1 = number;
type b1 = undefined;

  type x = a1 extends null ? true : false;

  type y = a1 extends null ? true : b1 extends undefined ? und
```

keyof কি?

⇒ 'bike' | 'car' | 'ship' এই ভাবে লেখা যা কথা keyof মানে ও তাই।

```
type Tsheikh = {
  bike: string;
  car: string;
  ship: string;
};

// car ache kina | bike ache kina | ship ache kina
// type checkVehicle<T> = T extends 'bike' | 'car' | 'ship' ?

// keyof Tsheikh 'bike' | 'car' | 'ship'
type checkVehicle<T> = T extends keyof Tsheikh ? true : false

type hasCar = checkVehicle<"car">; // true

type hasTractor = checkVehicle<"tractor">; // false
```

2-10: Mapped types

Mapped Types কি?

Mapped Types এর সাহায্যে একটি variable এর টাইপকে looping এর সাহায্যে পরিবর্তন বা dynamic করা যায় । Mapped Types লিখার জন্য index signature use করতে হবে। index variable এর নাম চাইলে index বা key দেওয়া জায়। index signature JavaScript map() এর মতো কাজ করে।

look up type কি?

⇒ একটি অবজেক্ট টাইপ property থেকে যখন bracket notation obj[key] এর মাধ্যমে অন্য একটি property এর key যখন look up মাধ্যমে নিয়ে আসা হয় তখন তাকে বলা হয় look up type.

```
{
  ////////////////////////// Start ///////////////

  // Mapped Types:-
  const arrOfNumbers: number[] = [1, 2, 3, 4];

  // const arrOfNumbers: string[] = ["1", "2", "3", "4"];

  const arrOfStrings: string[] = arrOfNumbers.map((str) => str.t

  // console.log(arrOfStrings);

  type TareaNumber = {
    height: number;
    widht: number;
  };

  type Theight = TareaNumber["height"]; // look up type

  // type TareaString = {
  //     height: string;
  //     width: string;
  // };

  // type TareaString = {
  //     [key in 'height' | 'width' ] : string;
  // };

  type TareaString = {
    [key in keyof TareaNumber]: string;
  };
```

```
  // mapped with generic

  // T => {height : string ; width: number};
  // key => ["height"];
  // key => ["width"];
  type TareaString1<T> = {
    // [key in keyof TareaNumber] : string;
    [key in keyof T]: T[key];
  };

  const area1: TareaString1<{ height: string; width: number }> =
    height: "12",
    width: 65,
  };

  ////////////////////////// End //////////////////////////
}
```

2-11 Utility types

Pick কি?

⇒ pick হচ্ছে এমন একটি utility type যা existing type থেকে কোন একটি টাইপ কে এনে নিজের কাছে store করে রাখে। যদি একের অধিক property হয় ওই সেম টাইপের তাহলে union (|) operator দিয়ে access করতে হয়।

```
type Tperson = {
        name: string;
        age: number;
        email?: string;
        contactNo: string;
    };

// Pick
type TnameAge = Pick<Tperson , "name" | "age">;
```

omit কি?

⇒ omit হচ্ছে এমন একটি utility type যা existing type থেকে যেই property গুলো আমার দরকার ওই গুলো বাদে অন্য Property গুলো নিজের কাছে store করে রাখে। যদি একের অধিক property হয় ওই সেম টাইপের তাহলে union (|) operator দিয়ে access করতে হয়। যেমনঃ ধরা যায় একটি type থেকে আমি contact no & email property গুলোকে access করবো তাহলে এই দুই property বাদে অন্য গুলো সে নিজের কাছে store করে রাখবে।

```
type Tperson = {
        name: string;
        age: number;
        email?: string;
        contactNo: string;
    };

   // Omit
  type TcontactInfo = Omit<Tperson, "name" | "age">;
```

required কি?

⇒ একটি type এর সবগুলো property তো আর required থাকে না কিছু কিছু তো অপসনাল থাকে আমি যদি চাই যে, ঐ টাইপ এর সব গুলো property নিয়ে নতুন একটি type গঠন হবে তখন তাকে required utility types বলা হয় ।

```
type Tperson = {
        name: string;
        age: number;
        email?: string;   // email is optional
        contactNo: string;
    };

// Required
type TpersonRequired = Required<Tperson>; // Threre is no optio
```

partial কি?

⇒ Partial type হলো required টাইপ এর সম্পূর্ণ বিপরীত । partial হচ্ছে এমন একটি utility type যা existing type এর সব গুলো property কে optional করে নতুন একটি type গঠন

করে।

```
type Tperson = {
       name: string;
       age: number;
       email?: string;
       contactNo: string;
   };

// Partial
type TpersonPartial = Partial<Tperson>;
```

readonly কি?

⇒ যদি এমন টি চাই যে, একটি টাইপ এর property গুলো কে re-assign বা পুনরায় পরিবর্তন না করে নতুন একটি type declared করতে তখন এই readonly utility type ব্যবহার করতে হয়। readonly ব্যবহার করলে property এর value কখনো পরিবর্তন করা যায় না।

```
type Tperson = {
       name: string;
       age: number;
       email?: string;
       contactNo: string;
   };

// Readonly
type TpersonalReadonly = Readonly<Tperson>;
```

record কি?

⇒ Record type এর মাধ্যমে আমার চাইলে পরবর্তীতে কোন property, existing type এর ভিতরে নতুন করে declared করতে পারি। চাইলে property এর টাইপ dynamic ভাবে ও বলা যায়। obj এর ক্ষেত্রে প্রতিটি key 'string' হিসাবে value (string , number , Boolean, array , etc.) জিনিস আসতে পারে তাই Record Type এ key হিসাবে 'string' এবং value হিসাবে 'unknown' নেওয়া হয়েছে।

```
// Record
    type TmyObj = {
        a : string;
        b: string;
    };

    const obj1 : TmyObj = {
        a: "aa",
        b: "bb",
        // c: "cc", // Object literal may only specify known pro
    };

    type TmyObj2 = Record<string , string>;

    const emptyObj : Record<string , unknown> = {};

    const obj2 : TmyObj2 = {
        a : "aa",
        b : "bb",
        c : "cc"
    };

    type TemptyObj = Record<string , unknown>;

    const emptyObj2 : TemptyObj ={
        a: "aa",
        b: "bb",
        c: true,
        d: 'dd',
        e: 688
    };
```

Tips & Trick:

Industry-standard অনুযায়ী type alias দিয়ে কোন type declared করলে T দিয়ে শুরু করা টা ভালো কাড়ন তাহলে সহজেই বুঝা যায় যে, এইটা type alias. Example: type Tuser = {} ।

Industry-standard অনুযায়ী type interface দিয়ে কোন type declared করলে I দিয়ে শুরু করা টা ভালো কাড়ন তাহলে সহজেই বুঝা যায় যে, এইটা type interface. Example: interface Iuser {} ।