**Advanced Types –Intersection Types, Type Guards, Discriminating Unions, Type Casting, Function Overloads**

**Project: course-starting-project**

1. **Module Content:**
   1. **Intersection Types (&)**:
      1. When used on object types, the & combines multiple types into one. This allows us to combine types, to create a new type (can also use with interface).
      2. When used on union types, the & is an intersection type, which only allows values the types have in common.
   2. **Example1: Combining Two Object Types**

type Admin = {

name: string;

privileges: string[];

};

type Employee = {

name: string;

startDate: Date;

}

type ElevatedEmployee = Admin & Employee;

* 1. **Example2: Define the Common Type between Union types:**

type Combinable = string | number;

type Numeric = number | boolean;

type Universal = Combinable & Numeric; // Types in Combinable & Numeric

* 1. A **Union Type** describes a value that can be one of several types. We use the vertical bar (|) to separate each type, so number | string | boolean is the type of a value that can be a number, a string, or a boolean.
  2. **Type Guards**: Checks if a method or property exists before you attempt to use it.
     1. **Type Check** - Type check to make sure your variables have the correct type, so they execute the correct function. Attempting to execute a function on a variable of the wrong type, can lead to runtime errors during execution, when using union or intersection types.
     2. **Primitive Type Guards -** Use “typeof” to check the type of primitive vars, to make sure they are used correctly.
        1. Example:
           1. let a = “Tony”;
           2. if (typeof a === “string”){}
     3. **Property Guard - Uses “in” to determine if a property exists on an object.** 
        1. Since using “typeof” on an object always returns “object” and custom types only exist during runtime, you cannot use “typeof” to check if a property exists, you must use the JavaScript in operator:
        2. **Example:**

if ("privileges" in emp) {

// both employees do not have a privilege property

console.log("Privileges: " + emp.privileges);

}

* + 1. **Instance Type Guard – Use “instanceof” with a class, to determine if an instance is derived from a custom class definiton.** NOTE: This method will not work with interfaces since they are not instantiated, only classes.
       1. **Example:**

if (vehicle instanceof Truck) {

// only a truck has loadCargo

vehicle.loadCargo(1000);

}

* 1. **Discriminated Unions**: A pattern that makes implementing type guards easier, when working with objects, union types, and interfaces.
     1. Adding a **type property** to an interface, which describes the type of interface. This property can be used to distinguish what props and methods are available for a particular type.
     2. **Example:**

switch(animal.type) {

case 'bird':

speed = animal.flyingSpeed;

break;

case 'horse':

speed = animal.runningSpeed;

}

* 1. **Type Casting**: When TypeScript cannot detect the type, Typecasting allows you to tell typescript that a value is of a specific type. For Instance, the generic HTMLElement does not have properties specific to HTMLInputElement, like value, therefore we must cast HTMLElements into HTMLInputElement.
     1. **Typecasting Options:**
        1. **Angle Bracket Type casting:** 
           1. const userInputElement = **<HTMLInputElement>**document.getElementById(‘user’input”);
        2. **As Type Casting:**
           1. const userInputElement = document.getElementById(‘user-input’) **as HTMLInputElement;**
        3. **As Type Casting, with attribute access:**
           1. **if ( userInputElement) {**

**(userInputElement as HTMLInputElement).value = “Hi there!”;**

**}**

* 1. **Index Types:** A feature that allows us to create objects, that may contain one or more properties, whose property keys may be different, but whose key and value types are the same.
     1. **Benefits: The name of the key maybe unknown.**
     2. **Example:**

interface ErrorContainer {

// index type where the key is a string and value returned is

// string (string, number, object is allowed)

**[key: string]: string;**

}

* 1. **Function Overloads:** When using union types, typescript may not know the type of value returned by the function, so we use function overloads to define multiple function signatures for one function. Creates multiple ways to call a function with different parameter types.
     1. **Example:**

function add(a: number, b: number): number;

function add(a: string, b: number): string;

function add(a: number, b: string): string;

function add(a: Combinable, b: Combinable) {

// Type Guard for string or number

if (typeof a === "string" || typeof b === "string") {

return a.toString() + b.toString();

}

return a + b;

}

* 1. **Optional Chaining:** When you're getting data from a source, but you don’t know for certain, if a given property is defined within the object, you can use optional chaining, which checks if the property exists, before calling a nested property.
     1. **Example:**

console.log(fetchedUserData?.job?.title);

* 1. **Nullish Coalescin**g: If you don’t know if data is null, undefined, or valid data.
     1. **Example:**

const userInput = null;

// if userInput is null or undefined then use the Default

const storedData = userInput ?? 'DEFAULT';

1. **Useful Links:**
   1. More on Advanced Types: <https://www.typescriptlang.org/docs/handbook/advanced-types.html>