**Typescript-Generics – Functions, Classes, Type Constraints, Special Types**

**Projects: course-starting-project**

1. [**What are Generics**](https://www.typescriptlang.org/docs/handbook/2/generics.html)**:**
   1. A component that can work over a variety of types rather than one.
2. **Built-in Generics** (Array<T>) – An array is a data structure that holds specific types of data. To identify which types of data the array will hold, you must specify it between angle brackets (<>).
   1. **Example 1:Arrays**
      1. **Syntax:**
      2. const names: Array<string> = ["Tony", "Max"]; // string[]
   2. **Example 2: Promises –** A promise is an object that represents a value that may not be available yet but will be resolved in the future. Promises are used to handle asynchronous operations, such as making network requests or accessing databases, where the result is not immediately available.
      1. **Intro to Promises:** [**video**](https://www.youtube.com/watch?v=QWiLfQHksdQ)
      2. [**Handling Multiple Promises**](https://dev.to/codeofrelevancy/all-about-promises-in-javascript-39lj)
      3. **Syntax:**

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

setTimeout(() => {

resolve('This is done!');

}, 2000);

});

* 1. **Advantage:** We get better type safety with generic types, because typescript knows what type of data will be held in a data structure or returned.

1. **Generic Functions** - Used when we want the parameters and return type of a function to be flexible. TypeScript should be able to infer the types when values are given.
   1. **Example 1: Problem: TypeScript cannot infer the properties of object types**

function merge(objA: object, objB: object) {

// merges a series of objects into an object array and returns the results

return Object.assign(objA, objB);

}

const results = merge({name: "Tony"},{age: 89});

console.log(results);

* 1. Example 2: Solution: Specify types as generics:
     1. The letters between the angle brackets, after the function name <T, U> indicate the function parameters. The “T,” indicates a flexible Type. They allow TypeScript to infer the functions’ return type.
     2. NOTE: In Generic functions, the types are set dynamically when the function runs, because the actual type depends on the parameters passed to the function.
     3. **Syntax:**

function merge<T extends object, U extends object>(objA: T, objB: U) {

return Object.assign(objA, objB);

}

console.log(merge({name: "Tony"}, {age: 153}));

const mergedObj1 = merge({name: "Tony"}, {age: 153});

1. **Type Constraint** – Use the extend keyword, to enable generic parameter types to inherit the properties of custom, built-in types.
   1. **Example:**

function merge<T extends object, U extends object>(objA: T, objB: U) {

return Object.assign(objA, objB);

}

const mergedObj = merge({ name: 'Max', hobbies: ['Sports'] }, {age: 30});

console.log(mergedObj);

1. **keyof Constraint** - Used to inform TypeScript that a variable must be a key within an object, so we do not attempt to access a property that doesn’t exist.
   1. **Example:**

function extractAndConvert<T extends object, U extends **keyof** T>(

obj: T,

key: U

) {

return 'Value: ' + obj[key];

}

extractAndConvert({ name: 'Max' }, 'name');

1. **Generic Classes** – A class that is flexible about the type of data it contains. Although the data must be uniform, which means the type of data must remain consistent throughout class execution.
   1. **Advantage:** They give us flexibility and type safety, during creation, that can be inferred during instantiation.
   2. **Example:**

class DataStorage<T> {

private data: T[] = [];

addItem(item: T) {

this.data.push(item);

}

removeItem(item: T) {

if (this.data.indexOf(item) === -1) {

return;

}

this.data.splice(this.data.indexOf(item), 1); // -1

}

getItems() {

return [...this.data];

}

}

// Example1: Create an instance of the DataStorage class that contains an array of strings

const textStorage = new DataStorage<string>();

textStorage.addItem('Tony');

textStorage.addItem('Max');

textStorage.removeItem('Tony');

console.log(textStorage.getItems());

1. **Generic Utility types:**
   1. **Partials**: A build in TypeScript type, that allows you to wrap a custom interface, making the properties on that interface optional, so you can assign values to them individually. Note, you will have to cast the return type back to the original type when done.
      1. **Example**:

interface CourseGoal {

title: string;

description: string;

completeUntil: Date;

}

function createCourseGoal(title: string, description: string, date: Date): CourseGoal{

// Tell TypeScript the variable with eventually be a CourseGoal

// Note: The partial is a wrapper that makes all properties optional

let courseGoal: Partial<CourseGoal> = {};

courseGoal.title = title;

courseGoal.description = description;

courseGoal.completeUntil = date;

// requires you to convert from Partial to CourseGoal

return courseGoal as CourseGoal;

}

* 1. **Readonly**: Does not allow anything to be added or removed from the data structure. Useful when you want to lock the data down.
     1. **Example:**

const names: Readonly <string[]> = ["Tony", "Anna"];

1. **Generics and Union types are not the same**. Unions tell us we are free to use any types. Generics, say, you must decide what type of data the data structure contains, and then stick with it. When we want to lockdown the type for every call.
2. **More on Generics**: <https://www.typescriptlang.org/docs/handbook/generics.html>