Advanced TypeScript Generics and Decorators

CS569 – Web Application Development II

Maharishi University of Management

Department of Computer Science

Assistant Professor Asaad Saad

Maharishi University of Management - Fairfield, Iowa

Class Properties Example

```
interface Book {
        bookName: string;
        isbn: number;
}
class Course {
        name: string;
        code: number;

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

        useBook(book: Book) {
                console.log(`Course ${this.name} is using the textbook:
                                ${book.bookName} who's ISBN = ${book.isbn}`);
        }
}
```

Class Example - Shortcut

```
interface Book {
        bookName: string;
        isbn: number;
}
class Course {

        constructor(public name: string, public code: number) {}

        useBook(book: Book) {
                console.log(`Course ${this.name} is using the textbook:
                                ${book.bookName} who's ISBN = ${book.isbn}`);
        }
}
```

Adding **access modifiers** to the constructor arguments lets the class know that they're properties of a class. If the arguments don't have access modifiers, they'll be treated as an argument for the constructor function and not properties of the class.

Class Properties Getters and Setters

```
class Login {
        private _password: string = "Default";
        get password() {
                return this._password;
        }
        set password(value: string) {
                if (value.length > 10) {
                        this._password = value;
                } else {
                        this._password = "Default";
                }
        }
}
let login = new Login();
console.log(login.password); // getter
login.password = "myPassword"; // setter
```

Losing the Type to Any!

```
const last = (arr: Array<number>) => {
    return arr[arr.length - 1]
}
const last_element = last([1, 2, 3])  // number
```

The function only usable for this primitive type

```
const last = (arr: Array<any>) => {
    return arr[arr.length - 1]
}
const last_element = last(['a', 'b', 'c'])  // any, we lost the type
```

We are losing the ability to define which type should be returned

Generics

Generics offer a way to create **reusable components**. Generics provide a way to make components work with any data type and not restrict to one data type.

Generic types can also be used with other non-generic types.

The implementation of generics give us the ability to pass in a range of types to a component, adding an extra layer of abstraction and re-usability to your code.

Generics can be applied to **functions**, **interfaces** and **classes** in Typescript.

Using Generics

```
const last_generics = <T>(arr: Array<T>): T => {
    return arr[arr.length - 1]
}

function last_generics<T>(arr: Array<T>): T {
    return arr[arr.length - 1]
}
```

T stands for Type, and is commonly used as the first type variable name when defining generics. T can be replaced with any valid name.

While TypeScript will infer the correct type, we can explicitly set the type when calling the generics function

```
const last = last_generics<number>([1, 2, 3])
const last = last_generics<string>(['a', 'b', 'c'])
```

Multiple Generic Types

```
function display<T, U>(id:T, name:U): void {
  console.log(typeof(id) + ", " + typeof(name));
}

display<number, string>(1, "Asaad");
```

We are not limited to only one type variable, we can bring in any amount we wish to define.

Generic Constraints

```
// this will cause an error
function display<T>(arg: T): T {
   console.log(arg.length);
   return arg;
}
```

We can also extend from multiple types by separating our constraints with a comma. <T extends Length, Type2, Type3>.

Not all Types have a .length property. What we need to do is extend our type variable to an interface that has our required property:

```
interface ILength {
    length: number;
}

function display<T extends ILength>(arg: T): T {
   // length property can now be called
   console.log(arg.length);
   return arg;
}
```

We are telling the compiler that we can support any type that implements the properties within ILength.
Now the compiler will let us know when we call the function with a type that does not support .length.

Generic Constraints

To restrict the generic type to certain types.

```
class Person {
    constructor(public firstName:string,  public lastName:string) {}
}

function display<T extends Person>(person: T): void {
    console.log(`${ person.firstName} ${person.lastName}` );
}

const asaad = new Person("Asaad", "Saad");

display(asaad); // Asaad Saad
display("Asaad Saad"); // Compiler Error
```

Constraints Real-World Use Case

```
const parseResponse= (obj: { status: number, payload: string }) => {
    // obj.date error
}
parseResponse({ status: 200, payload: 'Asaad' })
```


Interface?

```
const parseResponse = <T extends {status : number, payload : string }>(obj: T) => {
    // obj.status okay
}
parseResponse({status : 200, payload : 'Asaad', date: 'May 2020' })
```

Generic Interface

```
interface KeyPair<T, U> {
    key: T;
    value: U;
}

let kv1: KeyPair<number, string> = { key:1, value:"Asaad" }; // OK
let kv2: KeyPair<number, number> = { key:1, value: 1 }; // OK
```

Generic Class

```
class KeyValuePair<T,U> {
    private key: T;
    private val: U;
    setKeyValue(key: T, val: U): void {
        this.key = key;
        this.val = val;
    }
    display():void {
        console.log(`Key = ${this.key}, val = ${this.val}`);
    }
}
let kvp1 = new KeyValuePair<number, string>();
kvp1.setKeyValue(1, "Asaad");
kvp1.display(); // Key = 1, Val = Asaad

let kvp2 = new KayValuePair<string, string>();
kvp2.SetKeyValue("Asaad", "Mike");
kvp2.display(); // Key = Asaad, Val = Mike
```

We use angled brackets with the specific type when instantiating a new instance

Class Types are Compulsory

For instantiating classes, there is not much the compiler can do to guess which type we want assigned to our instance, **it is compulsory to pass the type here**.

```
class Programmer<T> {
   private languageName: string;
   private languageInfo: T;
   constructor(lang: string) {
      this.languageName = lang;
   }
   ...
}
let myObj = new Programmer<Type>("args");
```

Function Types can be Inferred

```
function identities<T, U> (arg1: T, arg2: U): void {}
```

With functions, the compiler can guess which type we want our generics to be. Calling the function will assign the `string` and `number` types to **`T`** and **`U`** respectively:

```
display<string, number>("argument 1", 100);
```

It is more commonly practiced that the compiler will pick up on types automatically, making cleaner code to read:

```
display("argument 1", 100);
```

The compiler is smart enough to pick up on the types of our arguments, and assign them to **`T`** and **`U`** without the developer needing to explicitly define them.

When to Use Generics

- When your function, interface or class **will work with a variety of data types**
- When your function, interface or class **uses that data type in several places**

You will not have a component that needs using generics early on in a project. But as the project grows, a component's capabilities often expand. This added extensibility may eventually meet with above two scenarios, in which case introducing generics would be a cleaner alternative than to duplicate components just to satisfy a range of data types.

Real-World Use Case

API services are a strong use case for generics, allowing you to wrap your API handlers in one class, and assigning the correct type when fetching results from various endpoints.

```
class APIService {
   public getRecord<T, U> (endpoint: string, params: T[]): U {}
   public getRecords<T, U> (endpoint: string, params: T[]): U[] {}
  ...
}
```

Decorators

Decorators are functions called **when the class is declared** (compile time)—not when an object is instantiated (runtime).

Decorators will change or add functionality to its destination.

Multiple decorators can be applied on the same **Class/Property/Method/Parameter**.

Decorators are not allowed on constructors.

In order for TS to understand decorators we should add this to the `tsconfig.json` *(proposed for ES7)*

```
"compilerOptions":{"experimentalDecorators": true}
```

Decorator Pattern

```
const course = { name: 'CS569' };

function addLevel(obj){
   return {
        level: 500,
        name: obj.name
   }
}

const decoratedObj = addLevel(course);

console.log(decoratedObj); // object {level: 500, name: 'CS569'}
```

Factory Decorator Pattern

```
const course = { name: 'CS569' };

function addLevel(level){
   return function(obj){
        return {
           level: level,
           name: obj.name
        }
   }
}

const decoratedObj = addLevel(500)(course);

console.log(decoratedObj); // object {level: 500, name: 'CS569'}
```

Simple Decorator in TS

```
@addLevel
class Course { name = "CS569" }

function addLevel(targetClass){
   return class {
        level = 500;
        name = new targetClass().name;
   }
}

console.log(new Course()); // object {level: 500, name: 'CS569'}
```

Simple Factory Decorator in TS

```
@addLevel(500)
class Course { name = "CS569" }

function addLevel(val){
   return function(targetClass){
        return class {
            level = val;
            name = new targetClass().name;
        }
   }
}

console.log(new Course()); // object {level: 500, name: 'CS569'}
```

Method Decorator

```
class MyClass {
        @log
        myMethod(arg: string) { return "Arg: " + arg; }
}
function log(target: Object, propertyKey: string, descriptor: PropertyDescriptor<any>) {
        let originalMethod = descriptor.value; // save a reference to the original method
        descriptor.value = function(...args: any[]) {
                console.log("The method args are: " + JSON.stringify(args));
                console.log("The return value is: " + originalMethod.apply(this, args[0]));
        };
        return descriptor;
}

new MyClass().myMethod(["CS569", "CS571"]);
// The method args are: ["CS569", "CS571"]
// The return value is: Arg: – CS569
```

Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time.
It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

Higher-Order Method Decorator

```
function editable(value: boolean) {
     return function (target: any, propName: string, descriptor: PropertyDescriptor){
               descriptor.writable = value;
     }
}
class Project {
       projectName: string;
       constructor(name: string) {
               this.projectName = name;
       }
       @editable(false)
       getGrade() {
               console.log("A");
       }
}
```

Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time.
It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

```
const project = new Project("CS569 Final Project");
project.getGrade(); // A
project.getGrade = function () { console.log("B"); };
project.getGrade(); // B
```

Property Decorator

```
class Project {
        @logProperty
        projectName: string;
        constructor(name: string) {
                this.projectName = name;
        }
}

function logProperty(target: any, propName: string): any {
        var myGetter = function () {
                console.log(`Get: ${propName} => ${this.propName}`);
                return this.propName;
        };
        var mySetter = function (newVal) {
                console.log(`Set: ${propName} => ${newVal}`);
                this.propName = newVal;
        };
        if (delete this.propName) {
                // Create new property with getter and setter
                Object.defineProperty(target, propName, {
                        get: myGetter,
                        set: mySetter,
                        configurable: true });
        }
}
```

Parameter Decorator

```
class Course {
        name: string;
        constructor(name: string) {
                this.name = name;
        }
        printStudentNumbers(mode: string, @printInfo isPrint: boolean) {
                if (isPrint) {
                        console.log(12345);
                } else {
                        console.log(null);
                }
        }
}

function printInfo(target: any, paramName: string, paramIndex: number) {
        console.log("Target: ", target);
        console.log("paramName: ", paramName);
        console.log("paramIndex: ", paramIndex);
}
```

Decorator Composition

```
function printable(targetClass) {
        targetClass.prototype.print = function () {
                                        console.log(this);
                                    }
}

@logging(false)
@printable
class Plant {
        name = "Green Plant";
}

const plant = new Plant();
plant.print();
```

We can use multiple decorators
When we have two decorators @A and @B they are similar to A(B(class))

- A evaluated
- B evaluated
- B called
- A called

TypeScript Modules

Depending on the module target specified during compilation, the TypeScript compiler generates appropriate code and supports many kinds of **module-loading systems**:

- Node.js (CommonJS)
- require.js (AMD)
- isomorphic (UMD)
- SystemJS
- ECMAScript 2015 native modules (ES6)

TypeScript supports ES6 Modules

```
//------ helper.ts ------
export function add(val) {}
export function delete(val) {}
export default function main(){}

//------ app.ts ------
import { add, delete }, main from './helper';
```

Test in the Browser

```
export class Best{
        constructor(){ console.log(`Best course ever!`); }
}
```

./src/b.ts

```
import {Best} from "./b";
class App{
        constructor(){ console.log(`CS569`); }
}
new App();
new Best();
```

./src/main.ts

```
npm install -g http-server
tsc –w
http-server -c-1
```

(never cache)

What's the output of this code?

The need for a Bundler!

Unfortunately none of the major JavaScript runtimes support ES2015 modules in their current stable branches. This means no support in Firefox, Chrome.

Many transpilers do support modules and a polyfill is also available.

Solutions to use modules in browsers today are: Parcel, Gulp, Browserify, **Webpack**

Webpack – Module Bundler

It bundles our code into one JS file. This file will still need compilation in order to make our app to work (to be done later in the browser JiT).