To run any typescript with decorator do following :
First, create an empty folder:
Next, initialize npm project by running **npm init** `-y` in the root dir of your empty project: (`-y` flag means that you are answering **yes** to all npm questions)
Next, we will need to add **typescript** npm package to our dev dependencies:
npm add typescript -DNext, we will need to add **tsconfig.json** file by running **npx tsc** –`init` in the root of our project:

```
 {
    "compilerOptions": {
      "target": "es6",
      "module": "commonjs",
      "sourceMap": true,
      "outDir": "out",
      "strict": true,
      "esModuleInterop": true,
      "experimentalDecorators": true,
      "emitDecoratorMetadata": true,
      "skipLibCheck": true,
      "forceConsistentCasingInFileNames": true
    }
```

Void display(String);//prototype for display function “c”

Void display(String sname)

{

//

}

Example #1: Basic TypeScript Decorator

```
// Requires enabling of `experimentalDecorators`
//
http://www.typescriptlang.org/docs/handbook/decorators.
html
function Self(constructorFunction: Function) {
console.log('----invoking decorator function here----
');
constructorFunction.prototype.selfEmp = true;
}
@Self
class Employee {
private _empName: string;
constructor(empName: string) {
```

```
console.log('----Inovking constructor here----');
this._empName = empName;
}
}
console.log('----Instance creation here----');
let employee: Employee = new Employee("Karthik");
console.log(employee);
console.log(`self: ${employee['selfEmp']}`);
console.log('----Creation of one more instance----');
employee = new Employee("Chaitanya");
console.log(employee);
console.log(`self: ${employee['selfEmp']}`);
```

Output:

Decorator Factory: User can customize how decorator is being applied to a declaration, for which Decorator factory comes into picture. It is a function that returns the expression that is called by decorator at runtime.

Decorator function can be written as below,

```
function decorSample(value: string) {
// decorator factory
return function (target) {
// decorator
// 'target' and 'value' are used for coding...
};
}
Multiple decorators can be applied to a single
declaration,
On single line; @f, @g, x
On multiple lines;
@f
```

```
@g
```

```
x
```

When there are multiple decorators applied to the same declaration, composing function f and g are composed to (f *g)(x) equal to f(g(x))

Steps followed to apply multiple decorators,

1. Expressions of each decorator are evaluated in top to bottom manner
2. Results are called functions in bottom to top manner.

Example #2: Using Multiple Decorators

```
// Enabling `experimentalDecorators`
//
http://www.typescriptlang.org/docs/handbook/decorators.html
function function1() {
console.log("function1(): function 1 has been evaluated");
return function (target,propertyKey: string,descriptor:
```

```
PropertyDescriptor) {

console.log("function1(): function 1 has been called");

};

}

function function2() {

console.log("function2(): function 2 has been

evaluated");

return function (target,propertyKey: string,descriptor:

PropertyDescriptor) {

console.log("function2(): function 2 has been called");

};

}

class sampleFunct {

@function1()

@function2()

method() {}

}
```

Output:

Property Decorator: Has priority 3

It is defined just before declaring a property, most similar to method decorators. There is a difference between both property and method decorators, which is, they do not accept the property descriptor as argument and return nothing

The expression for property decorator requires 2 arguments,

- Constructor function of the class for static members or prototype for the class of instance members.
- And the other one is a Member name.

Method Decorator: Has priority 2

It is defined just before method declaration, which is applicable to property descriptors. It is used for modifying, observing, or replacing a method definition. No method decorator can be used in declaration file.

Method decorator accepts 3 arguments as below,

- Constructor function of a static member of the class or instance member of a class prototype.
- Member name
- Property descriptor of Member

Parameter Decorator: Has priority 1

It is defined just before Parameter declaration and applied to the functions of the class constructor or the method declarations. Cannot be used in a declared class or a declaration file.

Parameter decorator required 3 parameters as below,

- Constructor function of a static member of the class or instance member of class prototype
- Member name

- Index of the parameters in a list of arguments of a function.

Accessor Decorator: Has priority 3

It is defined just before declaring an accessor, which is applied to accessor's property descriptor. It can be used to replace, modify or observe accessor's definition

Accessor decorator required 3 parameters as below,

- Constructor function of a static member of the class or instance member of class prototype
- Member name
- Property descriptor for member

Example #3: Accessor Decorator

```
// Enabling `experimentalDecorators`
//
http://www.typescriptlang.org/docs/handbook/decorators.
html
function Enumerable(target: any, propertyKey: string,
descriptor: PropertyDescriptor) {
```

```
//making the method enumerable
descriptor.enumerable = true;
}
class Emp {
_empname: string;
constructor(empname: string) {
this._empname = empname;
}
@Enumerable
get empname() {
return this._empname;
}
}
console.log("Accessor instance creation");
let emp = new Emp("Sandeep");
console.log("Looping string name here");
for (let key in emp) {
console.log(key + " = " + emp[key]);
}
```

Output:

Example #4: Method Decorators

```
// Enabling `experimentalDecorators`

//

http://www.typescriptlang.org/docs/handbook/decorators.

html

const log = (target: Object, key: string | symbol,

descriptor: TypedPropertyDescriptor<Function>) => {

return {

value: function( ... args: any[]) {

console.log("Arguments: ", args.join("; "));

const value = descriptor.value.apply(target, args);

console.log("NumberResult: ", value);

return value;

}

}
```

```
}

class sampleCalC {

@log

multiply(x: number, y: number) {

return x * y;

}

}

new sampleCalC().multiply(4, 6);
```

Output:

Rules and Regulations for TypeScript Decorators

- Decorator implementation allows users to decorate the class, parameters, or functions inside the class.
- For the class methods, user can enhance the behavior of the decorators by adding preconditions and postconditions to the logic.
- User can even decorate a single method if required.

- Need to keep in mind if there are both getter and setter for a single attribute. One of them can be decorated which applies for both.
- Can decorate a single attribute inside the class.

Conclusion

With this, we conclude our topic ‘TypeScript Decorators’. We have seen the syntax and its use in TypeScript. This was a very clear explanation of Decorators. Topic here is a very interesting feature of TypeScript which allows user to add behavior on top of existing logics. Hopefully, the examples listed above would help you to understand the behavior of each specific decorator. Thanks! Happy Learning!!

To run any typescript with decorator do following :

First, create an empty folder:

Next, initialize npm project by running **npm init** `-y` in the root dir of your empty project: (`-y` flag means that you are answering **yes** to all npm questions)

Next, we will need to add **typescript** npm package to our dev dependencies:

```
npm add typescript -D
```

Next, we will need to add **tsconfig.json** file by running **npx tsc** `–init` in the root of our project:

```
{
    "compilerOptions": {
      "target": "es6",
      "module": "commonjs",
      "sourceMap": true,
      "outDir": "out",
      "strict": true,
      "esModuleInterop": true,
      "experimentalDecorators": true,
      "emitDecoratorMetadata": true,
      "skipLibCheck": true,
      "forceConsistentCasingInFileNames": true
    }
```

npx ts-node file1.ts