**Typescript - Decorators**  
**Project: course-starting-project**

1. **Decorator** - A [*Decorator*](https://www.typescriptlang.org/docs/handbook/decorators.html)is a special kind of declaration that can be attached to a [class declaration](https://www.typescriptlang.org/docs/handbook/decorators.html#class-decorators), [method](https://www.typescriptlang.org/docs/handbook/decorators.html#method-decorators), [accessor](https://www.typescriptlang.org/docs/handbook/decorators.html#accessor-decorators), [property](https://www.typescriptlang.org/docs/handbook/decorators.html#property-decorators), or [parameter](https://www.typescriptlang.org/docs/handbook/decorators.html#parameter-decorators). Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration. Decorators execute when the class is defined, not when it is instantiated. Decorators allow us to create utility functions that can be used by other developers to perform some function, when a target is declared.
   1. **Advantage**: Decorators do not have a direct impact on the end-user of a page but is an instrument for writing easy to use code for other developers.
   2. **Two Methods to Create Decorators:**
      1. **Decorator Function** - Has no return statement
      2. **Example**:

function Logger(constructor: Function) {

console.log('Logging...');

console.log("Constructor:", constructor);

}

// Example1: Add logger a decorator to the Person class

@Logger

class Person {}

* + 1. **Decorator Factory** - Has a return statement and allows us to pass in params to the decorator function.
    2. **Example**:

// Decorator Factory: Pass in string into decorator function from decorator factory

function Logger(logString: string) {

return function(constructor: Function) {

console.log(logString);

console.log(constructor);

};

}

// Example1: Decorator Factory: Returns a decorator function

// Allows configuration when we assign it as a decorator to a class

@Logger('LOGGING - PERSON')

class Person {}

* 1. **What can I do with decorators:**
     1. Create a decorator factory that renders an html template to the DOM, whenever a class is instantiated.
  2. **Applying Multiple Decorators to the same Class**: You can add more than one decorator to a given class.
     1. When more than one decorator function is applied to the same class, the creation of the decorator function happens in order, however, the execution (what is rendered in the decorator’s return statement), occurs from the bottom up (in the reverse order, in which they are applied).
     2. **Example:**

function Logger(logString: string) {

console.log('LOGGER FACTORY');

return function(constructor: Function) {

console.log(logString);

console.log(constructor);

};

}

function WithTemplate(template: string, hookId: string) {

console.log('TEMPLATE FACTORY');

return function(constructor: any) {

console.log('Rendering template');

const hookEl = document.getElementById(hookId);

const p = new constructor();

if (hookEl) {

hookEl.innerHTML = template;

hookEl.querySelector('h1')!.textContent = p.name;

}

}

}

// Example 1: Multiple Decorators:

@Logger('LOGGING')

@WithTemplate('<h1>My Person Object</h1>', 'app')

class Person {}

* 1. **Property decorators:** You can add a decorator to a class instance variable. It executes when the property is defined on the class.
     1. **When is the property decorator executed?**
        1. When the class definition is executed by JavaScript.
     2. **Decorator Function Syntax:** 
        1. DecoratorName(classTarget, instanceVariableName)
     3. **Example:**

// Example1: Decorator Function to be added to a property.

function Log(target: any, propertyName: string | Symbol) {

console.log('Property decorator!');

console.log("Target", target, `PropertyName: ${propertyName}`);

}

class Product {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Example 1: Calling a decorator on a property.

\* Currenty not receiving arguments

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

@Log

title: string; // instance variable, decorator name

constructor(){}

}

* 1. **Accessor (Getter or Setter) Decorators**:
     1. You can add a decorator to a getter and a setter.
     2. **Example:**

function Log2(target: any, name: string, descriptor: PropertyDescriptor) {

console.log('Accessor decorator!');

console.log("Target2:", target);

console.log("Name2:", name);

console.log("Descriptor2:", descriptor);

}

class Product {

// @Log

title: string;

private \_price: number;

@Log2

set price(val: number) {

if (val > 0) {

this.\_price = val;

} else {

throw new Error('Invalid price - should be positive!');

}

}

}

* 1. **Method Decorators:** You can add a decorator to a method.
     1. **Example:**

/\*\*

\* Method Decorator

\* @param target - instance method prototype of object (constructor)

\* @param name - name of the method

\* @param descriptor - method descriptor

\*/

function Log3(

target: any,

name: string | Symbol,

descriptor: PropertyDescriptor

) {

console.log('Method decorator!');

console.log("Target3:", target);

console.log("Name3:", name);

console.log("Descriptor3:", descriptor);

}

class Product {

constructor(){}

@Log3

getPriceWithTax(tax: number) {

return this.\_price \* (1 + tax);

}

}

* 1. **Parameter Decorators:** You can add a decorator to a parameter.
     1. **Example:**

/\*\*

\* Parameter Decorator

\* target - instance method prototype of object (constructor)

\* name - name of the method in which we use the paramenter

\* position - The argument position

\*/

function Log4(target: any, name: string | Symbol, position: number) {

console.log('Parameter decorator!');

console.log("Target4:", target);

console.log("Name4:", name);

console.log("Position4:", position);

}

class Product {

constructor(){}

getPriceWithTax(@Log4 tax: number) {

return this.\_price \* (1 + tax);

}

}

1. Note: Decorators run during class declaration, not initialization, or runtime. They allow you to do additional (background) work when setting up a class.
2. **Returning (and changing) a Class in a Class Decorator**:
   1. Decorators can return a value inside the decorator return function, that will override the class the decorator was added to, when that class is instantiated. Decorator can return a new constructor function, which will replace the class that you added the decorator to. So, the decorator code will run when the class is instantiated only (see WithTemplate Decorator).
   2. **Other Decorator Return Types**:
3. **When Do Decorators Execute?**
   1. **Decorators run when the class is defined.** Class instantiation has now effect on decorators.
   2. **Decorators are not run at run time or when you call a method.** They allow additional behind the scenes setup work when a class is defined. They are not event listeners. It’s a function when your class is defined to setup some code that should run whenever the function is called.
4. **Class Decorators that run at class declaration vs class instantiation:**
   1. **Video108:** Example of running decorator that runs at class declaration only.
   2. **Video112:** Example of overriding class constructor in a decorator and running decorators during class instantiation.
5. **Other Decorator Return Types (113):**
   1. [**PropertyDescriptor**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) is a JavaScript object. They allow you to define a property in more detail, and determine if it is a:
      1. **Method (functions) Decorators** (see getPriceWithTax, @Log2, and @Log3) – Can return data. (see video113)
         1. **Method Decorators** can return a brand new **propertyDescriptors.**
      2. **Accessor Decorators** (Getter/Setter) – Can return data. (see price video113)
      3. **Property and Params Decorator** return values are not respected by TypeScript.
   2. **PropertyDescriptors have the following properties:**
      1. **Writable** - Can be changed after the object is created.
      2. **Enumerable** – It is iterable. It is true if you can loop through the object, otherwise it is false.
      3. **Configurable** – You can change this property or delete it after it has been created.
      4. **Set** – Could return a new descriptor which assigns a new set
   3. **Useless Setter Decorator Example**:

function Log2(target: any, name: string, descriptor: PropertyDescriptor): PropertyDescriptor {

console.log('Accessor decorator!');

console.log(target);

console.log(name);

console.log(descriptor);

// return { set }

// return { get }

// return { configurable }

// return { enumerable}

}

1. **Example1: Creating an Auto Bind Method Decorator function**: PropertyDescriptors can be modified to auto bind the ‘this’ reference to a method called on an instantiated class through a decorator function.
   1. **Video114**: Create an AutoBind Decorator on a class method to retain the this reference on an button click eventListener.
2. **Example2: Validation Decorators** - see course validate example:
   1. **Video116:** Create a course instance from input data. Register the instance variables (title, price) in a registeredValidators object and validate the data with a validator function.
3. **Other Uses for Decorators:**
   1. TS [Class-Validator](https://www.npmjs.com/package/class-validator) – A more elaborate version of our **Validator in Video116**. Decorator and non-decorator-based validation. This package can be imported and used to validate any class instance variables.
      1. Add all your instance variables to an object then call validate method on it.
   2. [Nest js](https://docs.nestjs.com/) - server side js framework for nodejs that embraces typescript. Uses decorators to define controllers and routes, and methods users use on http routes to trigger certain types of functions. Here decorators is meta programming, to add extra configuration and logic to which is taken into account by the framework to execute your code.