**Angular Decorators**

Decorators are design patterns used to isolate the modification or decoration of a class without modifying the source code.

In AngularJS, decorators are functions that allow a **service, directive,** or **filter** to be modified before it is used.

1. **Class Decorators: @Component and @NgModule**
2. **Property Decorators: @Input and @Output (These two decorators are used inside a class)**
3. **Method Decorators: @HostListener (This decorator is used for methods inside a class like a click, mouse hover, etc.)**
4. **Parameter Decorators: @Inject (This decorator is used inside class constructor).**

**Class Decorators**

This is the first type of decorator which tells us about a particular class’s intent and helps answer questions like whether the class is a component or a module. There are various Class decorators in Angular, and among them @Component and @NgModule are widely used.

The below code snippet uses @Component, which is a type of class decorator provided by Angular.

----------- we use @component in app.component.ts

@Component({

  selector: 'app-plotly-example-component',

  templateUrl: './plotly-example-component.component.html',

  styleUrls: ['./plotly-example-component.component.css']

})

export class PlotlyExampleComponentComponent implements OnInit {

}

**---------- we use ngmodule in app.module.ts**

**@NgModule({**

**imports: [],**

**declarations: [],**

**})**

**export class ExampleModule {  }**

**Property Decorators**

Property decorators are used to decorate specific properties inside a class. Using a property decorator, we can easily identify why we use any particular property of a class Like @Input (), @Output, @ReadOnly (), @Override ()

**import** { Component, Input } from '@angular/core';

@Component({

  selector: 'example-component',

  template: '<div>Woo a component!</div>'

})

export **class** ExampleComponent {

  @Input()

  exampleProperty: string;

}

**<example-component**

**[exampleProperty]="exampleData">**

**</example-component>**

### Method Decorator

Method decorators, as the name indicates, are used to decorate the method defined inside our class with functionality. A common example of method decorator is @HostListene



### Parameter Decorators

Parameter decorators allow us to decorate parameters in our class constructors.  @Inject () is a common and widely used parameter decorator. Using this decorator, we can inject services in Angular classes.



**Angular Directive**

The Angular directive helps us to manipulate the DOM. You can change the appearance, behavior, or layout of a DOM element using the Directives. They help you to extend HTML.

There are three kinds of directives in Angular:

1. Component Directive
2. Structural directives
3. Attribute directives

**Component Directives:** Component directives are used in main class. They contain the detail of how the component should be processed, instantiated and used at runtime.

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

There are three parameters discussed below:

* + **Selector:** Tells the template tag which specifies the beginning and end of the component.
  + **templateURL:** Consists of the template used for the component.
  + **styleUrls:** It is of array type which consists of all the style format files used for the template.

**Structural Directives:** Structural directives start with a \* sign. These directives are used to manipulate and change the structure of the DOM elements. For example, \*ngIf and \*ngFor.

The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection).

**Example of ngFor**

<tr \*ngFor="let customer of customers;">

    <td>{{customer.customerNo}}</td>

    <td>{{customer.name}}</td>

    <td>{{customer.address}}</td>

    <td>{{customer.city}}</td>

    <td>{{customer.state}}</td></tr>

**Attribute Directives:** Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass, ngStyle etc. t provides the facility to create our own directive.

## **Difference between Attribute Directive and Structural Directive**

|  |  |
| --- | --- |
| **Attribute Directives** | **Structural Directives** |
| Attribute directives look like a normal HTML Attribute and mainly used in databinding and event binding. | Structural Directives start with a \* symbol and look different. |
| Attribute Directives affect only the element they are added to. | Structural Directives affect the whole area in the DOM. |

Data Binding

 used to define the communication between a component and the DOM.

Data binding is a technique, where the data stays in sync between the component and the view. Whenever the user updates the data in the view, Angular updates the component. When the component gets new data, the Angular updates the view.

Data binding can be either one-way data binding or two-way data binding.

**One way binding**

In one way binding data flows from one direction. Either from view to component or from component to view

### From Component to View

To bind data from component to view, we make use of Interpolation & Property Binding.

[Interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/) allows us to include expressions as part of any string literal, which we use in our HTML. The angular evaluates the expressions into a string and replaces it in the original string and updates the view. You can use interpolation wherever you use a string literal in the view

The Angular uses the {{ }} (double curly braces) in the template to denote the interpolation.

Welcome,  {{firstName}} {{lastName}} --------------html

import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  firstName= 'Sachin';

  lastName=”Tendulkar”

}

Also, whenever the values of firstName & lastName change, Angular updates the view. But not the other way around.

Property Binding

In property binding, we bind a property of a DOM element to a field which is a defined property in our component TypeScript code.

import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = "Data binding using Property Binding";

  imgUrl="https://static.javatpoint.com/tutorial/angular7/images/angular-7-logo.png";

}

**<h2>**{{ title }}**</h2>** <!-- String Interpolation -->

**<img** [src]="imgUrl" **/>** <!-- Property Binding --> -----html

### Event Binding

### Event binding allows us to bind events such as keystrokes, clicks, hover, touch, etc to a method in component.

event binding is used to handle the events raised from the DOM like button click, mouse move etc. When the DOM event happens (eg. click, change, keyup), it calls the specified method in the component.

**<h2>** Event Binding Example**</h2>**

**<button** (click)="onSave($event)"**>**Save**</button>** <!--Event Binding--> ------html

import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  onSave($event){

    console.log("Save button is clicked!", $event);

  }

}

Two way Data Binding

In two-way databinding, automatic synchronization of data happens between the Model and the View. Here, change is reflected in both components. Whenever you make changes in the Model, it will be reflected in the View and when you make changes in View, it will be reflected in Model.

import { Component } from "@angular/core";

@Component({

  selector: "app-root",

  templateUrl: "./app.component.html",

  styleUrls: ["./app.component.css"]

})

export class AppComponent {

  fullName: string = "Hello JavaTpoint"; ---this we can give a value or can be empty

}

**<h2>**Two-way Binding Example**</h2>**

**<input** [(ngModel)]="fullName" **/>** **<br/><br/>**    ---in textbox whatever value is entered that will be shown below.

**<p>** {{fullName}} **</p>**

Life Cycle Hooks in Angular

The life cycle hooks are the methods that angular invokes on the [directives](https://www.tektutorialshub.com/angular/angular-directives/) and [components](https://www.tektutorialshub.com/angular/angular-component/) as it creates, changes, and destroys them. Using lifecycle hooks we can fine-tune the behavior of our components during its creation, updating, and destruction.

The Angular life cycle hooks are nothing but callback functions, which angular invokes when a specific event occurs during the component’s life cycle.

* [ngOnInit](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) when Angular initializes the component for the first time.
* When a component’s input property change, Angular invokes [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/)
* If the component is destroyed, Angular invokes [ngOnDestroy](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/)

## Constructor

The life cycle of a component begins when Angular creates the component class. The first method that gets invoked is class Constructor.

Constructor is neither a life cycle hook nor is it specific to Angular.  It is a Javascript feature. It is a method that is invoked when a class is created.

## ngOnChanges

The Angular invokes the [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) life cycle hook whenever any data-bound input property of the component or directive changes. Initializing the Input properties is the first task angular carries during the change detection cycle. And if it detects any change in property, then it raises the [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) hook.

## ngOnInit

The Angular raises the [ngOnInit](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) hook after it creates the component and updates its input properties. It raises it after the [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) hook.

This hook is fired **only once** and immediately after its creation (during the first change detection).

## ngDoCheck

The Angular invokes the [ngDoCheck](https://www.tektutorialshub.com/angular/angular-ngdocheck-life-cycle-hook/) hook event during every change detection cycle. This hook is invoked even if there is no change in any of the properties.

Angular invoke it after the [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) & [ngOnInit](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) hooks.

Use this hook to Implement a custom change detection whenever Angular fails to detect the changes made to Input properties. This hook is convenient when you opt for the Onpush change detection strategy.

The Angular [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) hook [does not detect all the changes made to the input properties](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/#onchanges-does-not-fire-always).

## ngAfterContentInit

ngAfterContentInit Life cycle hook is called after the Component’s [projected content](https://www.tektutorialshub.com/angular/ng-content-content-projection-in-angular/) has been fully initialized. Angular also updates the properties decorated with the [ContentChild and ContentChildren](https://www.tektutorialshub.com/angular/contentchild-and-contentchildren-in-angular/) before raising this hook. This hook is also raised, even if there is no content to project.

## ngAfterContentChecked

ngAfterContentChecked Life cycle hook is called during every change detection cycle after Angular finishes checking of component’s projected content. Angular also updates the properties decorated with the [ContentChild and ContentChildren](https://www.tektutorialshub.com/angular/contentchild-and-contentchildren-in-angular/) before raising this hook. Angular calls this hook even if there is no projected content in the component.

This hook is very similar to the ngAfterContentInit hook. Both are called after the external content is initialized, checked & updated. The only difference is that ngAfterContentChecked is raised after every change detection cycle. While ngAfterContentInit during the first change detection cycle.

## ngAfterViewInit

ngAfterViewInit hook is called after the Component’s View & all its child views are fully initialized. Angular also updates the properties decorated with the [ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) & [ViewChildren](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) properties before raising this hook.

## ngAfterViewChecked

The Angular fires this hook after it checks & updates the component’s views and child views. This event is fired after the ngAfterViewInit and after that, during every change detection cycle.

This hook is very similar to the ngAfterViewInit hook. Both are called after all the child components & directives are initialized and updated. The only difference is that ngAfterViewChecked is raised during every change detection cycle. While ngAfterViewInit during the first change detection cycle.

## ngOnDestroy

This hook is called just before the Component/Directive instance is [destroyed by Angular](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/)

You can Perform any cleanup logic for the Component here. This is where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

## How to Use Lifecycle Hooks

1. Import Hook interfaces
2. Declare that Component/directive Implements lifecycle hook interface
3. Create the hook method

Step-1:import { Component,OnInit } from '@angular/core'

Step-2:export **class** AppComponent implements OnInit {

Step-3: ngOnInit() {

    console.log("AppComponent:OnInit");

  }

## The Order of Execution of Life Cycle Hooks

The Angular executes the hooks in the following order

On Component Creation

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
6. AfterViewInit
7. AfterViewChecked

When the Component with Child Component is created

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
   1. Child Component -> OnChanges
   2. Child Component -> OnInit
   3. Child Component -> DoCheck
   4. Child Component -> AfterContentInit
   5. Child Component -> AfterContentChecked
   6. Child Component -> AfterViewInit
   7. Child Component -> AfterViewChecked
6. AfterViewInit
7. AfterViewChecked

After The Component is Created

1. OnChanges
2. DoCheck
3. AfterContentChecked
4. AfterViewChecked

## Component Communication

There are few ways in which components can communicate or share data between them. And methods depend on whether the components have a Parent-child relationship between them are not.

Here are the three Possible scenarios

1. Parent to Child Communication
2. Child to Parent Communication
3. Interaction when there is no parent-child relation

## Parent to Child Communication

If the Components have a parent-child relationship then, then the parent component can pass the data to the child using the [@input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/#input) Property.

**Listen for Input Changes**

The Child Component can get the values from the someProperty. But it also important for the child component to get notification when the values changes.

There are two ways in which we can achieve that.

1. Using OnChanges life Cycle hook or
2. Using a Property Setter on Input Property

## Child to Parent Communication

The Child to Parent communication can happen in three ways.

1. Listens to Events from Child
2. Uses [Local Variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/) to access the child in the Template
3. Uses a [@ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) to get a reference to the child component

### Listens to Child Event

This is done by the child component by exposing an [EventEmitter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) Property. We also decorate this Property with [@Output](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator. When Child Component needs to communicate with the parent it raises the emit event of the [EventEmitter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) Property. The Parent Component listens to that event and reacts to it.

### Uses [Local Variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/) to access the child

Using [Local Variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/) is to refer to the child component is another technique

<child-component #child></child-component>

<p> current count **is** {{child.count}} </p>

### Uses a [@ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) to get the reference to the child component

<child-component></child-component>

@ViewChild(ChildComponent) child: ChildComponent;

 increment() {

**this**.child.increment();

  }

## Communication when there is no relation

If the Components do not share the Parent-child relationship, then the only way they can share data is by using the services and observable.

The advantageous of using service is that

1. You can share data between multiple components.
2. Using observable, you can notify each component, when the data changes.

## What is Angular Routing

Routing allows you to move from one part of the application to another part or one View to another View.

In Angular, Routing is handled by the Angular Router Module.

## Angular Router

The Router is a separate module in Angular. It is in its own library package, [**@angular/router**](https://angular.io/api/router). The Angular Router provides the necessary service providers and directives for navigating through application views.

## How to configure Angular Router

To Configure the Router in Angular, you need to follow these steps

* Set the <base href>
* Define routes for the view
* Register the Router Service with Routes
* Map HTML Element actions to Route
* Choose where you want to display the view

### Set the <base href>

The HTML <base> element specifies the base URL to use for all relative URLs contained within a document.

<base href="/">

### Define the routes

Next, create an array of route objects. Each route maps the path (URL Segment) to the component

const appRoutes={ path: 'product', component: ProductComponent }

Where

**path:** The URL path segment of the route. We will use this value to refer to this route elsewhere in the app

**component:** The component to be loaded.

This route tells angular to render ProductComponent when the user navigates to the URL “/product”

### Register the Routes

Import the Angular Router from **@angular/router** library in the root module of the application

import { RouterModule } from '@angular/router';

**Map Action to Routes**

Next, we need to bind the click event of the link, image, or button to a route. This is done using the routerlink directive

<li><a [routerLink]="['product']">Product</a></li>

**Choose where you want to display**

Finally, we need to tell the angular where to display the view. This is done using the RouterOutlet directive as shown. We will add the following directive to the root component.

<router-outlet></router-outlet>

# **Angular Pipes**

Angular Pipes takes data as input and formats or transform the data to display in the template. We use them to change the appearance of the data before presenting it to the user

The syntax of the pipe is as follows

Expression | pipeOperator[:pipeArguments]

Where

Expression: is the expression, which you want to transform  
| : is the Pipe Character  
pipeOperator : name of the Pipe  
pipeArguments: arguments to the Pipe

Some commonly used predefined Angular pipes are:

1. DatePipe: Formats a date value.
2. UpperCasePipe: Transforms text to uppercase.
3. LowerCasePipe: Transforms text to lowercase.
4. CurrencyPipe: Transforms a number to the currency string.
5. PercentPipe: Transforms a number to the percentage string.
6. DecimalPipe: Transforms a number into a decimal point string.

## **Pure and Impure Pipes**

Pipes in Angular are classified into Pure and Impure types. Let’s have a closer look at them.

### Pure Pipes

These pipes use pure functions. As a result of this, the pipe doesn’t use any internal state and the output remains the same as long as the parameters passed remain the same.

### Impure Pipes

An impure pipe in Angular is called for every change detection cycle regardless of the change in the input fields. Multiple pipe instances are created for these pipes and the inputs passed to these pipes are mutable.

## **Why Do We Need HttpClient?**

The front-end of applications communicate with back-end services to get or send the data over HTTP protocol using either XMLHttpRequest interface or fetch API. This communication is done in Angular with the help of HttpClient.

## **What Is HttpClient?**

HttpClient is a built-in service class available in the @angular/common/http package. It has multiple signature and return types for each request. It uses the RxJS observable-based APIs, which means it returns the observable and what we need to subscribe it. This API was developed based on XMLHttpRequest interface exposed by browsers.

## **Features of HttpClient**

* Provides typed request and response objects
* Contains testability features
* Intercepts request and response
* Supports RxJS observable-based APIs
* Supports streamlined error handling
* Performs the GET, POST, PUT, DELETE operations

## What is RxJS

The [RxJS](https://rxjs.dev/guide/overview) (Reactive Extensions Library for JavaScript) is a Javascript library that allows us to work with asynchronous data streams.

A **data stream** is the data that arrives over some time. The stream of data can be anything. Like variables, user inputs, properties, caches, data structures, and even failures, etc

the stream may emit the following three things.

**Value:** i.e., the next value in the stream  
**Complete**: The stream has ended  
**Error**: The error has stopped the stream

As said earlier the stream of data can be anything. For Example

* Mouse click or Mouse hover events with x & y positions
* Keyboard events like keyup, keydown, keypress, etc
* Form events like value changes etc
* Data that arrives after an HTTP request
* User Notifications
* Measurements from any sensor

Important Points regarding streams can

* Emit zero, one or more values of any time.
* It can also emit errors.
* Must emit the complete signal when completed (finite streams).
* Can be infinite, and they never complete

The RxJs has two main players

1. Observable
2. Observers ( Subscribers)

**What is an Observable in Angular**

**An observable stream** or simple Observable emits the **value from the stream** asynchronously. It emits the **complete** signals when the stream completes or an **error** signal if the stream errors out.

Observables are declarative. You define an observable function just like any other variable. The observable starts to emit values only when **someone subscribes to it**.

## Who are the observers (subscribers)

The Observable is only useful if someone consumes the value emitted by the observable. We call them observers or subscribers.

The observers communicate with the Observable using callbacks

The observer must subscribe to the observable to receive the value from the observable. While subscribing, it optionally passes the three callbacks. next(), error() & complete()

The observable emits the value as soon as the observer or consumer subscribes to it.

The observable invokes the next() callback whenever the value arrives in the stream. It passes the value as the argument to the next callback. If the error occurs, then the error() callback is invoked. It invokes the complete() callback when the stream completes.

* Observers/subscribers subscribe to Observables.
* The observer registers three callbacks with the observable at the time of subscribing. i .e next(), error() & complete()
* All three callbacks are optional
* The observer receives the data from the observer via the next() callback
* They also receive the errors and completion events from the Observable via the error() & complete() callbacks

# **RxJS Subjects**

An RxJS Subject is like an Observable. It is a special type of Observable that allows values to be multicasted to many Observers. In simple words, we can say that an RxJS subject is an Observable can multicast or talk to many observers.

According to its official definition, "A Subject is like an Observable, but can multicast to many Observers. Subjects are like EventEmitters: they maintain a registry of many listeners."

An [RxJS](https://www.javatpoint.com/rxjs) subject can be subscribed to, just like we usually do with Observables. It also has methods such as next(), error() and complete(), which we have already seen and used in our Observable creation function.

### Difference between RxJS Observable and RxJS Subject

Every Subject is an Observable. We can subscribe to a given Subject just like an observable, and it will start receiving values usually. From the perspective of the Observer, it cannot be decided whether the Observable execution is coming from a plain unicast Observable or a Subject.

The main difference between an Observable and a Subject is that a plain Observable by default is unicast. It means that each subscribed Observer owns an independent execution of the Observable. On the other hand, Subjects are multicast. A Subject is like an Observable, but it can multicast to many Observers. The main reason behind using Subjects is to multicast.

## Promise

A [Promise](https://www.geeksforgeeks.org/javascript-promise/) represents a single value in the future, that may not be available at present but is expected to be resolved or rejected in the future. It is more readable and maintainable in asynchronous. A Promise object has two possible states, i.e. the **resolve** and **reject**. It offers a structured way to handle resolved or rejected states. It has “[**then ()**](https://www.geeksforgeeks.org/javascript-promise-then-method/)” to handle resolved states and “[**catch ()**](https://www.geeksforgeeks.org/javascript-promise-catch-method/)” to handle rejected ones. These help in making promises a suitable choice for single asynchronous operations. Suitable for activities such as reading data from server, files. There are 4 phases in it, namely, fulfilled, rejected, pending, and settled.

* **Pending**: In this, action is still pending and not yet fulfilled or rejected.
* **Fulfilled**: This state represents that the asynchronous operation is successfully completed.
* **Rejected**: In this action is rejected or failed.
* **Settled**: In this result is determined successfully, either fulfilled or rejected.

const myPromise = new Promise((resolve, reject) => {

// Asynchronous operation

if (success) {

resolve(response);

} else {

reject(error);

}

});

myPromise.then(result => {

// Handle success here

}).catch(error => {

// Handle error here

});

**Disadvantages of Promises**

* These promises are not cancellable in between the process. Once implemented, we must wait till a result is obtained. i.e., settled.
* These execute only once and don’t repeat them again.
* Multiple values are not retrieved over time.
* When working on large applications, it’s complicated.

### Different ways to add css in an Angular component.

1. Adding style url into the component.  
2. Adding css inside the component.  
3. Adding css along with template.  
4. Inline style.

**1)**

**import** { Component } from '@angular/core';

@Component({

selector: 'app-general',

templateUrl: './general.component.html',

styleUrls: ['./general.component.scss']

})

**export** **class** GeneralComponent {

}

**2)**

**import** { Component } from '@angular/core';

@Component({

selector: 'app-general',

templateUrl: './general.component.html',

styles: [

`

.content{

color:**red**;

}

`

]

})

**export** **class** GeneralComponent {

}

**3)**

**import** { Component } from '@angular/core';

@Component({

selector: 'app-general',

template: `

<style>

.content {color :**red**;}

.active {color:**blue**;}

</style>

<div **class**="content">Test</div>

`,

})

**export** **class** GeneralComponent {

}

**4)**

**import** { Component } from '@angular/core';

@Component({

selector: 'app-general',

template: '<div style="color:red">Test</div>',

})

**export** **class** GeneralComponent {

}

## What is Angular Dependency Injection

**Dependency Injection (DI) is a technique in which a class receives its dependencies from external sources rather than creating them itself**.

What if ProductService depends on another Service. And then we decide to change the service to some other service. Again we need to search and replace the code manually

It is hard to test this [Component](https://www.tektutorialshub.com/angular/angular-component/) as it is difficult to provide the Mock for the ProductService. For Instance, what if we wanted to substitute out the implementation of ProductService with MockProductService during testing.

Our Component Class has now tied one particular implementation of ProductService. It will make it difficult to reuse our components.

## Benefits of Dependency Injection

### loosely coupled

Our Component is now loosely coupled to the ProductService.

AppComponent does not know how to create the ProductService.  Actually, it does not know anything about the ProductService. It just works with the ProductService passed onto it. You can pass ProductService, BetterProductService or MockProductService. The AppComponent does not care.

### Easier to Test

AppComponent is now easier to Test. Our AppComponent is not dependent on a particular implementation of ProductService anymore. It will work with any implementation of ProductService that is passed on to it. You can just create a mockProductService Class and pass it while testing.

### Reusing the Component

Reusing of the component is becomes easier. Our Component will now work with any ProductService as long as the interface is honored.

Dependency injection pattern made our AppComponent testable, maintainable, etc

## Angular Dependency Injection Framework

Angular Dependency Injection framework implements the Dependency Injection in Angular. It creates & maintains the Dependencies and injects them into the Components, Directives, or Services.

There are five main players in the Angular Dependency injection Framework.

### Consumer

The Consumer is the class (Component, Directive, or Service) that needs the Dependency. In the above example, the AppComponent is the Consumer.

### Dependency