# Life Cycle Hooks

[Hooks for the Component](https://codecraft.tv/courses/angular/components/lifecycle-hooks/#_hooks_for_the_component)

**constructor**

This is invoked when Angular creates a component or directive by calling new on the class.

**ngOnChanges**

Invoked **every** time there is a change in one of th input properties of the component.

This is the very first lifecycle hook, it is called right after your class gets initialized and the component is created the ngOnChanges() is called. You might want to wonder why ngOnInit hook is not called first, but that is because Angular counts that very first class initialization as a data property change. So the hook that gets called once a data property change (like resetting values) occurs is ngOnChanges(). This hook is basically called after the constructor is called and any other time there is a property change inside your component.

**ngOnInit**

Invoked when given component has been initialized.  
This hook is only called **once** after the first ngOnChanges

**When to use –**

If you have a lot of processing to do when the component gets created, it’s good practice to do it in the **ngOnInit** hook rather than in the constructor. Example: Setup initial data

This is the second lifecycle hook called by Angular, it is called right after the very first ngOnChanges hook is called. It is only called once, it initializes the component, sets and displays component input properties. It is the most important lifecycle hook in Angular as it signals the activation of the created component. For the fact that this hook is called only once, it is therefore great for fetching data from external sources like servers and APIs.

**ngDoCheck**

Invoked when the change detector of the given component is invoked. It allows us to implement our own change detection algorithm for the given component.

This is the third Angular lifecycle hook that gets called on a component. It is called during every change detection run, Angular has an internal system that goes around the component processes every so often looking for changes that the compiler cannot detect on its own. This hook is called at every change detection run, usually after the ngOnInit hook is called.

**ngOnDestroy**

This method will be invoked just before Angular destroys the component.  
Use this hook to unsubscribe observables and detach event handlers to avoid memory leaks.

This is the last Angular lifecycle hook, it is called just before the component is removed from the DOM. Inside it clean up of the component is done, from detaching event handlers to unsubscribing from observables.

Exercise: To test this hook, follow the pattern in the previous hooks above and achieve create the alert for this. You will also notice that ngOnDestroy hook is not called, that is because the DOM has not been removed, if you have an unsubscribe statement inside it for instance, it will get cal

**Hooks on children**

These hooks are only called for components and not directives.

Note

We will cover the difference between Components and Directives in the next section.

**ngAfterContentInit**

Invoked after Angular performs any content projection into the component’s view

This is the fourth lifecycle hook Angular calls after a component has been initialized. This hook is called only once immediately after the first ngDoCheck hook is called, it is a kind of ngDoCheck but for content projected into the component view with ng-content

**ngAfterContentChecked**

Invoked each time the content of the given component has been checked by the change detection mechanism of Angular.

This is the fifth lifecycle hook Angular calls after a component has been initialized. It is called after the content projected into a component view is initialized, after the ngAfterContentInit hook and every subsequent ngDoCheck hook is called.

**ngAfterViewInit**

Invoked when the component’s view has been fully initialized.

This is the sixth lifecycle hook Angular calls after a component has been initialized. It is called only once after the very first ngAfterContentChecked hook is called. It is called after Angular initializes component views and the subsequent child views under each component, this will have to include the views displayed through content projection too and that is why it is called after the ngAfterContentChecked hook.

**ngAfterViewChecked**

Invoked each time the view of the given component has been checked by the change detection mechanism of Angular.

This is the seventh lifecycle hook Angular calls after a component has been initialized. It is called after Angular checks the component views and the subsequent child views under each component for changes, this includes the views displayed through content projection too. It is called after the ngAfterViewInit hook and every subsequent ngAfterContentChecked hook.

The ideal behavior is for the ngDoCheck, ngAfterContentChecked and ngAfterViewChecked hooks to be called multiple times as changes occur so do not be surprised they get called more than once.

<https://itnext.io/understanding-angular-life-cycle-hooks-91616f8946e3>

# Reactive Programming using RxJs

Streams are just a *sequence of values* over time.

In [computing](https://en.wikipedia.org/wiki/Computing), **reactive programming** is a [declarative](https://en.wikipedia.org/wiki/Declarative_programming) [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) concerned with [data streams](https://en.wikipedia.org/wiki/Stream_(computing)) and the propagation of change. With this paradigm it is possible to express static (e.g., arrays) or dynamic (e.g., event emitters) *data streams* with ease, and also communicate that an inferred dependency within the associated *execution model* exists, which facilitates the automatic propagation of the changed data flow

Reactive programming is the idea we can define an application as a series of different streams with operations that connect the different streams together and which are automatically called when new values are pushed onto those streams.

Angular uses RxJS for some parts of its internal functioning. If you want you can also choose to use RxJS but you don’t need to at all.

Streams are a sequence of values over time, that’s it. We could even have a stream to represent a user filling in a form on a website.

* Mouse click or Mouse hover events with x & y positions
* Keyboard events like keyup, keydown, keypress, etc
* Form events like value changes etc
* Data which 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.
* can also emit errors.
* must emit the complete signal, when completes (finite streams).
* can be infinite, that they never complete

Machine generated alternative text:
Hot and Cold Event Streams 
• PULL-based (Cold Event Streams) — Cold streams are 
streams that run their sequence when and if they are 
subscribed to. They present the sequence from the start 
to each subscriber. 
• PUSH-based (Hot Event Streams) — Hot streams emit 
values independent of individual subscriptions. They 
have their own timeline and events occur whether 
someone is listening or not. An example of this is 
mouse events. A mouse is generating events regardless 
of whether there is a subscription. When subscription is 
made observer receives current events as they happen. 

The RxJs has two main players

1. Observable
2. Observers ( Subscribers)

**Observable**

Observable converts the ordinary stream of data into an observable stream of data. It observes the stream of data and emits the value, complete or error signals to the consumers of the stream

You can think of Observable it as a wrapper around the stream of data.

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

We link streams together using operators,

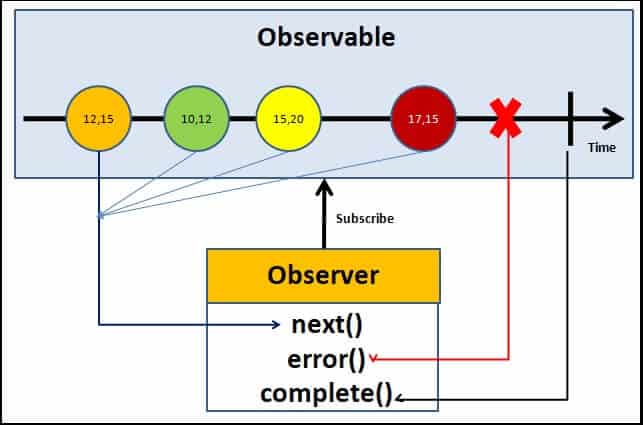
*Observables* is a new primitive type which acts as a *blueprint* for how we want to create streams, subscribe to them, react to new values, and combine streams together to build new ones.

**Observers (subscribers)**

The Observable on its own is useless unless someone consumes the value delivered by the observable. We call them observers or subscribers.

The observable communicates with the observers using callbacks

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

Angular Observable Tutorial how observable and observers communicates with callbacks

The observable starts emitting the value as soon as 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
* 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**

RxJS stands for \*R\*eactive E\*x\*tensions for \*J\*ava\*S\*cript, and it’s a library that gives us an implementation of Observables for JavaScript.

## RxJS Operators

Understanding Operator or Marble diagram

Machine generated alternative text:
This is time flowing from 
left to right to represent 
the execution of the 
input Observable. 
6 
60 
This Observable is the 
output of the operator 
call. 
These are values emitted 
by the Observable. 
8 
multiplyByTen 
This vertical ine represents 
the "complete" notification 
and indicates that the 
Observable has completed 
successfully. 
This box indicates the 
tweralor which 
the input Observable 
(above) to produce the 
output Observable 
(below). Tha text inside 
the box shows the 
nature of the 
transformation. 
This X represents an error entted by 
the output Observable, indicating 
abnormal termination. Neither values 
nor the vertical will be delivered 
thereafter. 

[**interval**](https://codecraft.tv/courses/angular/reactive-programming-with-rxjs/observables-and-rxjs/#_interval)

An observable **isn’t** a stream. An observable is a blueprint which describes a set of streams and how they are connected together with operations.

Lets assume for our use case, we want our observable to create a single stream and push onto that stream a number every second, incremented by 1.

For this purpose, we can use the interval operator which creates an Observable that emits numbers in sequence based on the provided time interval

[**subscribe**](https://codecraft.tv/courses/angular/reactive-programming-with-rxjs/observables-and-rxjs/#_subscribe)

In RxJS land no one can hear you *stream*, unless you subscribe.

**Of operator –**

Parameters can be – Array, string, sequence of numbers, value

**From operator –**

Parameter – array, string, collection, iterable, promise

**fromEvent** –

can be created from button, scroll, keyup

**pipe** –

The pipe method of the [**Angular Observable**](https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/) is used to chain multiple operators together. We can use the pipe as a standalone method, which helps us to reuse it at multiple places or as an instance method

**Map**

operator applies a given project function to each value emitted by the source Observable and emits the resulting values as an Observable.

**Filter**

operator filter items from the source observable based on some condition and returns the filtered value as a new observable

**tap**

Perform a side effect for every emission on the source Observable, but return an Observable that is identical to the source. Used for debugging,

# Forms

Common form foundation classes

Both reactive and template-driven forms are built on the following base classes.

* [FormControl](https://angular.io/api/forms/FormControl) tracks the value and validation status of an individual form control.
* [FormGroup](https://angular.io/api/forms/FormGroup) tracks the same values and status for a collection of form controls.
* [FormArray](https://angular.io/api/forms/FormArray) tracks the same values and status for an array of form controls.
* [ControlValueAccessor](https://angular.io/api/forms/ControlValueAccessor) creates a bridge between Angular [FormControl](https://angular.io/api/forms/FormControl) instances and native DOM elements.

### **Data flow in reactive forms**

In reactive forms each form element in the view is directly linked to the form model (a [FormControl](https://angular.io/api/forms/FormControl) instance). Updates from the view to the model and from the model to the view are synchronous and do not depend on how the UI is rendered.

The view-to-model diagram shows how data flows when an input field's value is changed from the view through the following steps.

1. The user types a value into the input element, in this case the favorite color Blue.
2. The form input element emits an "input" event with the latest value.
3. The control value accessor listening for events on the form input element immediately relays the new value to the [FormControl](https://angular.io/api/forms/FormControl) instance.
4. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the new value through the valueChanges observable.
5. Any subscribers to the valueChanges observable receive the new value.

The model-to-view diagram shows how a programmatic change to the model is propagated to the view through the following steps.

1. The user calls the favoriteColorControl.setValue() method, which updates the [FormControl](https://angular.io/api/forms/FormControl) value.
2. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the new value through the valueChanges observable.
3. Any subscribers to the valueChanges observable receive the new value.
4. The control value accessor on the form input element updates the element with the new value.

### **Data flow in template-driven forms**

In template-driven forms, each form element is linked to a directive that manages the form model internally.

The view-to-model diagram shows how data flows when an input field's value is changed from the view through the following steps.

1. The user types Blue into the input element.
2. The input element emits an "input" event with the value Blue.
3. The control value accessor attached to the input triggers the setValue() method on the [FormControl](https://angular.io/api/forms/FormControl) instance.
4. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the new value through the valueChanges observable.
5. Any subscribers to the valueChanges observable receive the new value.
6. The control value accessor also calls the [NgModel.viewToModelUpdate()](https://angular.io/api/forms/NgModel" \l "viewToModelUpdate) method which emits an ngModelChange event.
7. Because the component template uses two-way data binding for the favoriteColor property, the favoriteColor property in the component is updated to the value emitted by the ngModelChange event (Blue).

The model-to-view diagram shows how data flows from model to view when the favoriteColor changes from Blue to Red, through the following steps

1. The favoriteColor value is updated in the component.
2. Change detection begins.
3. During change detection, the ngOnChanges lifecycle hook is called on the [NgModel](https://angular.io/api/forms/NgModel) directive instance because the value of one of its inputs has changed.
4. The ngOnChanges() method queues an async task to set the value for the internal [FormControl](https://angular.io/api/forms/FormControl) instance.
5. Change detection completes.
6. On the next tick, the task to set the [FormControl](https://angular.io/api/forms/FormControl) instance value is executed.
7. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the latest value through the valueChanges observable.
8. Any subscribers to the valueChanges observable receive the new value.
9. The control value accessor updates the form input element in the view with the latest favoriteColor value.

### **Mutability of the data model**

The change-tracking method plays a role in the efficiency of your application.

* **Reactive forms** keep the data model pure by providing it as an immutable data structure. Each time a change is triggered on the data model, the [FormControl](https://angular.io/api/forms/FormControl) instance returns a new data model rather than updating the existing data model. This gives you the ability to track unique changes to the data model through the control's observable. Change detection is more efficient because it only needs to update on unique changes. Because data updates follow reactive patterns, you can integrate with observable operators to transform data.
* **Template-driven** forms rely on mutability with two-way data binding to update the data model in the component as changes are made in the template. Because there are no unique changes to track on the data model when using two-way data binding, change detection is less efficient at determining when updates are required.

The difference is demonstrated in the previous examples that use the favorite-color input element.

* With reactive forms, the [FormControl](https://angular.io/api/forms/FormControl)**instance** always returns a new value when the control's value is updated.
* With template-driven forms, the **favorite color property** is always modified to its new value.

# Dependency Injection

Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

In Angular, the DI framework provides declared dependencies to a class when that class is instantiated. This guide explains how DI works in Angular, and how you use it to make your apps flexible, efficient, and robust, as well as testable and maintainable.