# Angular

## Q) tsconfig.json

The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

## Q) polyfills.ts

Polyfills in angular are few lines of code which make your application compatible for different browsers. The code we write is mostly in ES6(New Features: Overview and Comparison) and is not compatible with IE or firefox and needs some environment setups before being able to be viewed or used in these browsers.

## Module

### Q) metadata

NgModule metadata does the following:

* Declares which components, directives, and pipes belong to the module.
* Makes some of those components, directives, and pipes public so that other module's component templates can use them.
* Imports other modules with the components, directives, and pipes that components in the current module need.
* Provides services that the other application components can use.

Every Angular app has at least one module, the root module. You bootstrap that module to launch the application.

### Q) Types of modules

There are five general categories of feature modules which tend to fall into the following groups:

* Domain feature modules.
* Routed feature modules.
* Routing modules.
* Service feature modules.
* Widget feature modules.

### Q) What is a *declarable*?

Declarables are the class types—components, directives, and pipes—that you can add to a module's [declarations](https://angular.io/api/core/NgModule#declarations) list. They're the only classes that you can add to [declarations](https://angular.io/api/core/NgModule#declarations).

Do not declare the following:

* A class that's already declared in another module, whether an app module, @NgModule, or third-party module.
* An array of directives imported from another module. For example, don't declare FORMS\_DIRECTIVES from @angular/forms because the [FormsModule](https://angular.io/api/forms/FormsModule) already declares it.
* Module classes.
* Service classes.
* Non-Angular classes and objects, such as strings, numbers, functions, entity models, configurations, business logic, and helper classes.

### Q) What should I import?

Import NgModules whose public (exported) [declarable classes](https://angular.io/guide/bootstrapping#the-declarations-array) you need to reference in this module's component templates.

This always means importing [CommonModule](https://angular.io/api/common/CommonModule) from @angular/common for access to the Angular directives such as [NgIf](https://angular.io/api/common/NgIf) and NgFor. You can import it directly or from another NgModule that [re-exports](https://angular.io/guide/ngmodule-faq#q-reexport) it.

Import [FormsModule](https://angular.io/api/forms/FormsModule) from @angular/forms if your components have [([ngModel](https://angular.io/api/forms/NgModel))] two-way binding expressions.

Import *shared* and *feature* modules when this module's components incorporate their components, directives, and pipes.

Import only [BrowserModule](https://angular.io/guide/ngmodule-faq" \l "q-browser-vs-common-module) in the root AppModule.

### Q) Should I import [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) or [CommonModule](https://angular.io/api/common/CommonModule)?

The root application module, AppModule, of almost every browser application should import [BrowserModule](https://angular.io/api/platform-browser/BrowserModule)from @angular/platform-browser.

[BrowserModule](https://angular.io/api/platform-browser/BrowserModule) provides services that are essential to launch and run a browser app.

[BrowserModule](https://angular.io/api/platform-browser/BrowserModule) also re-exports [CommonModule](https://angular.io/api/common/CommonModule) from @angular/common, which means that components in the AppModule module also have access to the Angular directives every app needs, such as [NgIf](https://angular.io/api/common/NgIf) and NgFor.

Do not import [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) in any other module. *Feature modules* and *lazy-loaded modules* should import [CommonModule](https://angular.io/api/common/CommonModule) instead. They need the common directives. They don't need to re-install the app-wide providers.

Importing [CommonModule](https://angular.io/api/common/CommonModule) also frees feature modules for use on *any* target platform, not just browsers.

### Q) What is the [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) method?

The [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) static method is a convention that makes it easy for developers to configure services and providers that are intended to be singletons. A good example of [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) is the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) method.

Apps pass a [Routes](https://angular.io/api/router/Routes) object to [RouterModule.forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) in order to configure the app-wide [Router](https://angular.io/api/router/Router) service with routes. [RouterModule.forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) returns a [ModuleWithProviders](https://angular.io/api/core/ModuleWithProviders). You add that result to the [imports](https://angular.io/api/core/NgModule" \l "imports)list of the root AppModule.

Only call and import a .forRoot() result in the root application module, AppModule. Importing it in any other module, particularly in a lazy-loaded module, is contrary to the intent and will likely produce a runtime error. For more information, see [Singleton Services](https://angular.io/guide/singleton-services).

For a service, instead of using [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot), specify [providedIn](https://angular.io/api/core/Injectable" \l "providedIn): 'root' on the service's @[Injectable](https://angular.io/api/core/Injectable)()decorator, which makes the service automatically available to the whole application and thus singleton by default.

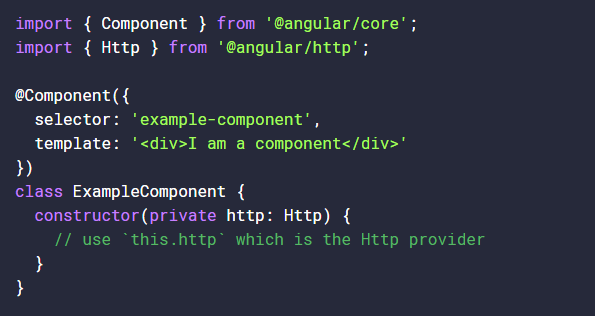
[RouterModule](https://angular.io/api/router/RouterModule) also offers a forChild static method for configuring the routes of lazy-loaded modules.

[forRoot()](https://angular.io/api/router/RouterModule#forRoot) and [forChild()](https://angular.io/api/router/RouterModule" \l "forChild) are conventional names for methods that configure services in root and feature modules respectively.

Angular doesn't recognize these names but Angular developers do. Follow this convention when you write similar modules with configurable service providers.

## DI

### Q) Decorator is essential to include dependency for $injector to work.



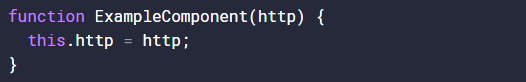
Inside our tsconfig.json files we’ll likely have emitDecoratorMetadata set to true. This emits metadata about the type of the parameter into a decorator in our compiled JavaScript output.



From here, we can see the compiled code knows about http being equal to the Http service provided by @angular/http - it’s added as a decorator for our class here:

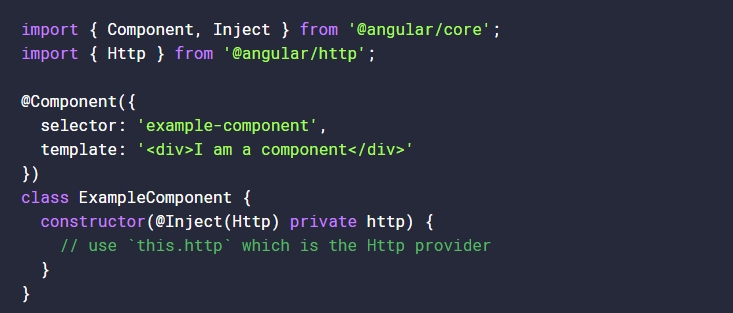


So essentially, the @Component decorator is transformed into plain ES5, and some additional metadata is supplied through the \_\_decorate assignment. Which in turn tells Angular to lookup the Http token and supply it as a first parameter to the Component’s constructor - assigning it to this.http



<https://toddmotto.com/angular-dependency-injection>

### Q) Inject()

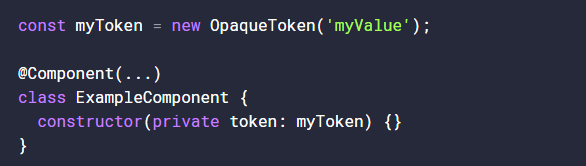


At this point, @Inject is a manual way of specifying this lookup token, followed by the lowercase http argument to tell Angular what to assign it against.

This could (and will) get very messy when a component or service requires a lot of dependencies. As Angular supports resolving dependencies from the emitted metadata, there’s no need to use @Inject most of the time.

The only time we’d need to use @Inject is alongside something like an [OpaqueToken](https://angular.io/docs/ts/latest/api/core/index/OpaqueToken-class.html) - which creates a unique blank token to be used as a dependency injection provider.

The reason we use @Inject is because we cannot use an OpaqueToken as the *type* of a parameter, for instance this will not work:



Here, myToken is not a Type, it’s a value - which means TypeScript cannot compile it. However, when we introduce @Inject alongside an OpaqueToken, things will work out nicely:

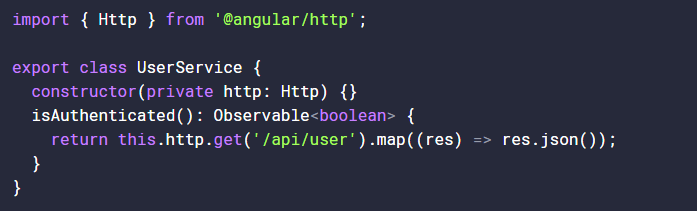


### Q) Injectable()

When using Angular decorators, the decorated class stores metadata about itself in a format that Angular can read - this includes the metadata about what dependencies it needs to fetch and inject.

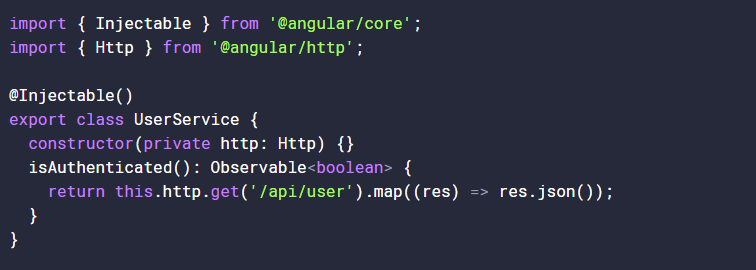
If no Angular decorator has been used on a class there is no way for Angular to read what dependencies it requires. This is why we need to use @Injectable().

If our service injects providers we must add @Injectable(), which providers no extra functionality, to tell Angular to store that metadata it needs.



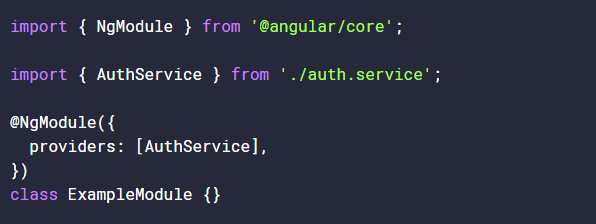
This would break as the Http provider metadata would not be stored for Angular to compose it correctly.

We can simply add @Injectable() to solve this:

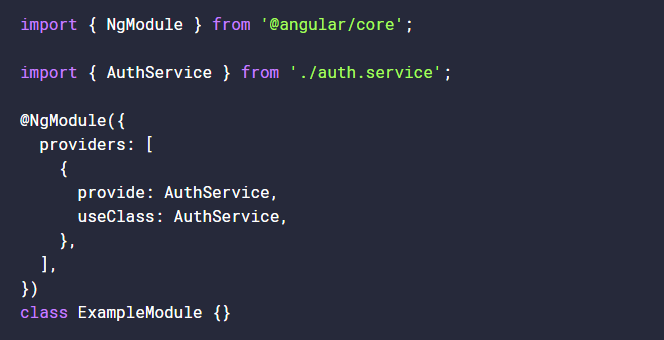


At this point, Angular is aware of the Http token and can supply it to http.

### Q) How Angular solves dependencies and instantiates them



The above is shorthand for this:



The provide property in the object is the token for the provider that we’re registering. This means that Angular can look up what is stored under the token for AuthService using the useClass value.

This provides many benefits. The first, we can now have two providers with the exact same class name and Angular will not have any issues in resolving the correct service. Secondly, we can also override an existing provider with a different provider whilst keeping the *token* the same.

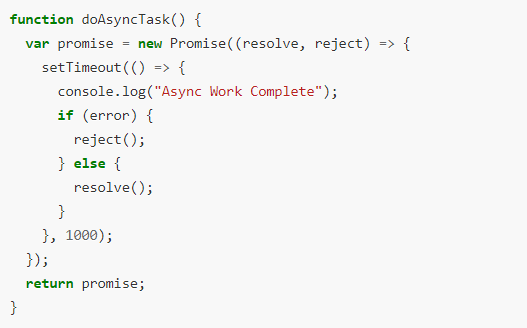
### Q) Understand Injectors

## Promise

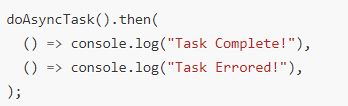
### Q) Promise syntax

A promise is a placeholder for a future value.

It serves the same function as callbacks but has a nicer syntax and makes it easier to handle errors.



We can get notified when a promise resolves by attaching a *success* handler to its then function, the second argument is an *error* handler that gets called if the promise is rejected.



### Q) Immediate Resolution or Rejection

We can create an immediately *resolved* Promise by using the Promise.resolve() method



And an immediately *rejected* Promise by using the Promise.reject() method, like so

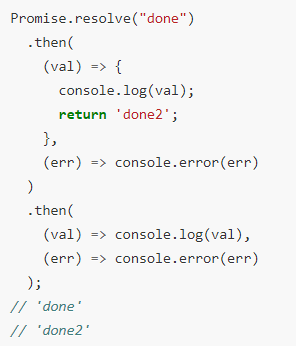


One of the nice things about Promises is that if we add a then handler **after** the promise resolves or rejects the handler **still** gets called.



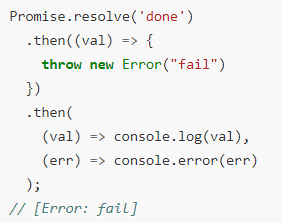
### Q) Chaining

We can also connect a series of then handlers together in a chain, like so:

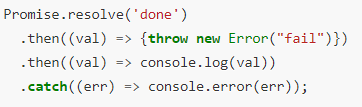


### Q) Error handling

If we throw an exception from our promise function or one of the success handlers, the promise gets rejected and the error handler is called, like so:



The catch function works exactly the same way as the then error handler, it’s just clearer and more explicitly describes our intent to handle errors.



## Router

### Q) Router Configuration



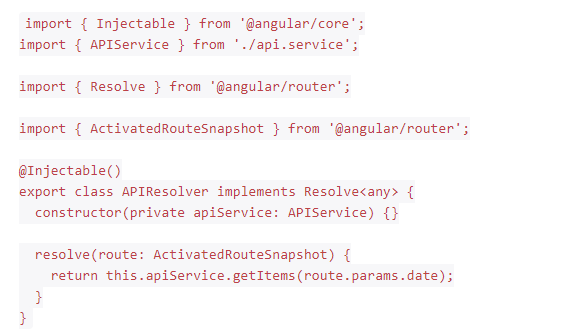
The order of the routes in the configuration matters and this is by design. The router uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes.

#### Q) Data

The data property in the third route is a place to store arbitrary data associated with this specific route. The data property is accessible within each activated route. Use it to store items such as page titles, breadcrumb text, and other read-only, *static* data

#### Q) Route Resolver

Route resolves are nothing more than a way to pre-fetch the data a component needs before it is initialized.





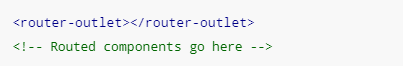
https://www.callibrity.com/blog/angular-2-route-resolves

#### Q) **PathMatch**

**pathMatch** is used to specify the matching strategy **full** or **prefix**. **full** means that the whole URL's path needs to match by the matching algorithm. **prefix** means the first route where path matches the start of the URL will be chosen. In the case of empty path if we don't set the **full** matching strategy then we won't get the desired behaviour as any path starts with an empty path.

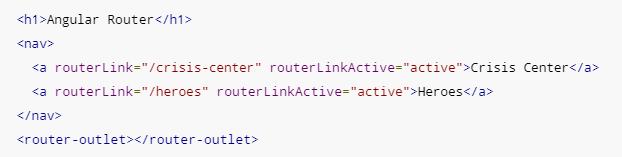
### Q) RouterOutlet

It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet.

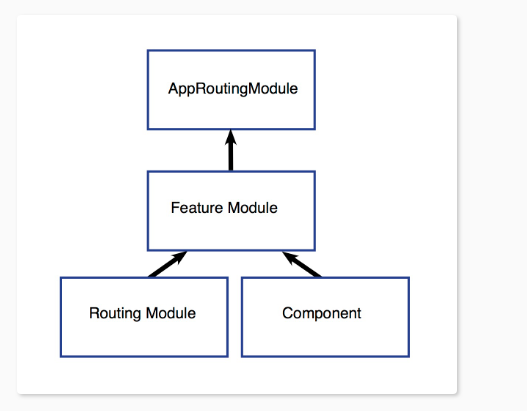


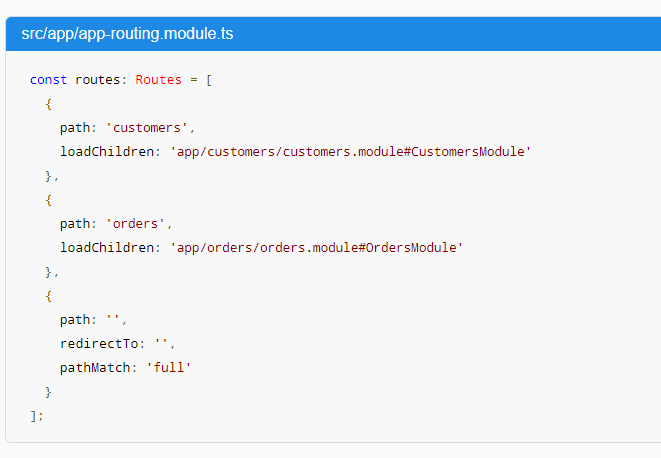
### Q) RouterLink

routerLink: this directive is used instead of *href* in the <a> tags, routerLinkActive: this directive is used to add a CSS class to an element when the link's route becomes active.



### Q) Lazy loading

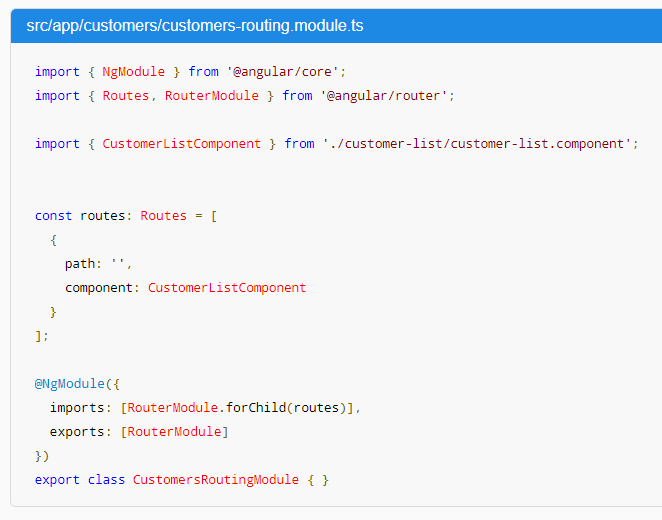




Routes at the top level

The first two paths are the routes to the CustomersModule and the OrdersModulerespectively. Notice that the lazy loading syntax uses [loadChildren](https://angular.io/api/router/Route" \l "loadChildren) followed by a string that is the path to the module, a hash mark or #, and the module’s class name.

Routes at the feature module level.



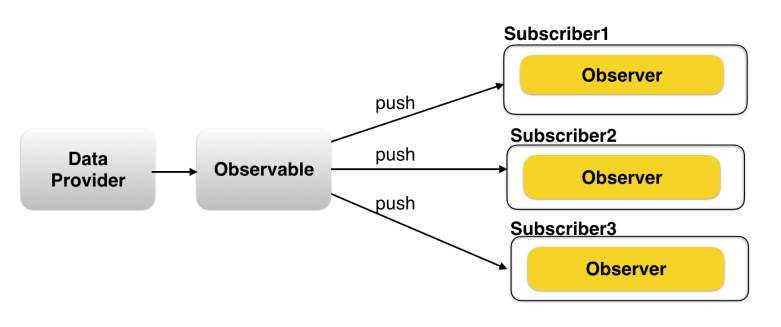
### Q) forRoot vs forChild

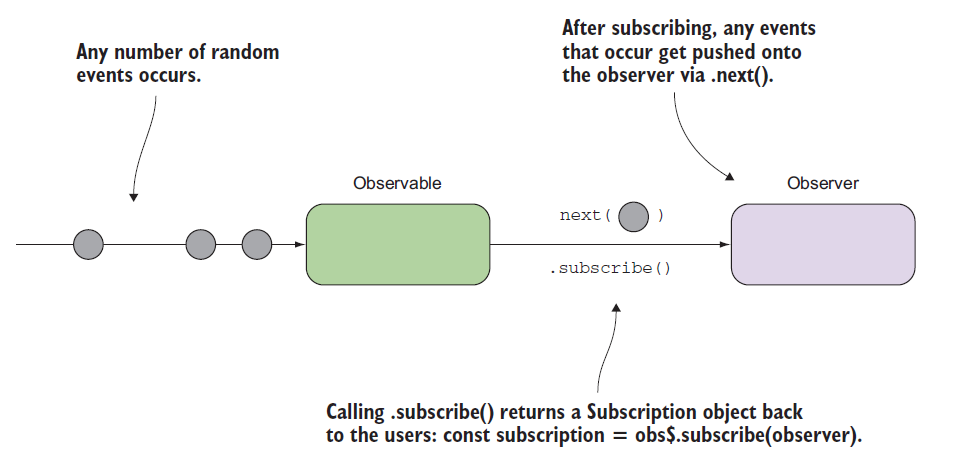
You might have noticed that the CLI adds RouterModule.forRoot(routes) to the app-routing.module.ts imports array. This lets Angular know that this module, AppRoutingModule, is a routing module and [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) specifies that this is the root routing module. It configures all the routes you pass to it, gives you access to the router directives, and registers the RouterService. Use [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) in the AppRoutingModule—that is, one time in the app at the root level.

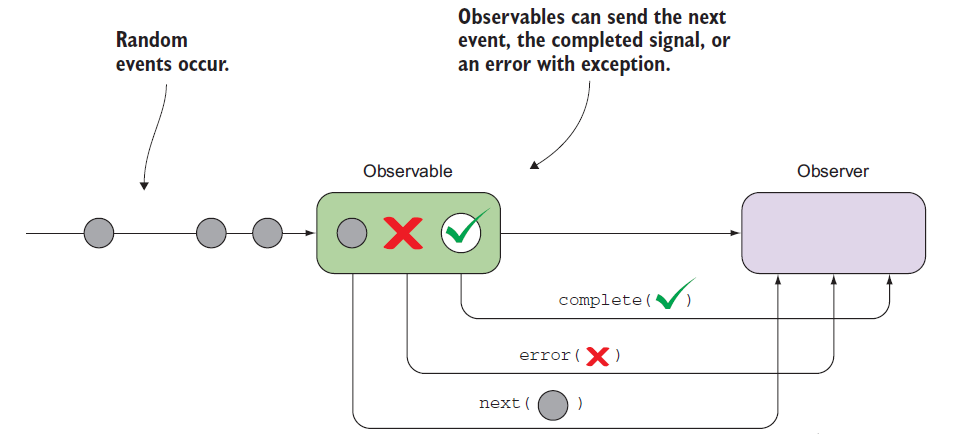
The CLI also adds RouterModule.forChild(routes) to feature routing modules. This way, Angular knows that the route list is only responsible for providing additional routes and is intended for feature modules. You can use [forChild()](https://angular.io/api/router/RouterModule" \l "forChild) in multiple modules.

[forRoot()](https://angular.io/api/router/RouterModule#forRoot) contains injector configuration which is global; such as configuring the Router. [forChild()](https://angular.io/api/router/RouterModule" \l "forChild) has no injector configuration, only directives such as [RouterOutlet](https://angular.io/api/router/RouterOutlet) and [RouterLink](https://angular.io/api/router/RouterLink).

# RxJS







## Q) Players

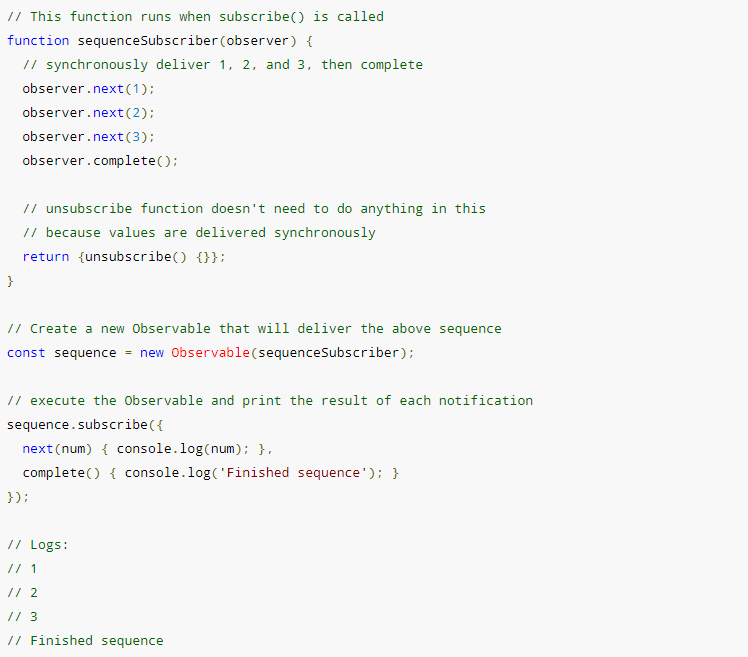
An observable gets data from some data source (a socket, an array, UI events) one element at a time. To be precise, an observable knows how to do three things:

1. Emit the next element to the observer
2. Throw an error on the observer
3. Inform the observer that the stream is over

Accordingly, an observer object provides up to three callbacks:

1. The function to handle the next element emitted by the observable
2. The function to handle errors thrown by the observable
3. The function to handle the end of stream

The subscriber connects an observable and observer by invoking the method subscribe() and disconnects them by invoking unsubscribe().



## Q) Observer

A handler for receiving observable notifications implements the Observer interface. It is an object that defines callback methods to handle the three types of notifications that an observable can send:

* next() - here's a new value from the stream
* error() - here's an error happened in the stream
* complete() - the stream's over

## Q) Observable

An object or a function that emits sequences of data over time.

Type

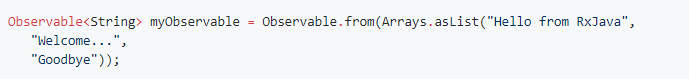
* **Hot observables** begin emitting items as soon as it is created, and so any observer who later subscribes to that Observable may start observing the sequence somewhere in the middle. (ex: watch livestream)
* **Cold observables** waits until an observer subscribes to it before it begins to emit items, and so such an observer is guaranteed to see the whole sequence from the beginning. (ex: watch video)

Cold observables usually refer to arrays or single values that have been converted to be used within RxJS.

Examples:

HTTP Requests.

UI Events



Use the Observable constructor to create an observable stream of any type.

A subscriber function receives an Observer object, and can publish values to the observer's next() method.

### Q) O**bservables Constructors**

RxJS offers multiple ways of creating an observable depending on the type of the data producer.

* Observable.of(1,2,3) – turns the sequence of numbers into an Observable
* Observable.create(myObserver) – returns an Observable that can invoke  methods on myObserver that you will create and supply as an argument
* Observable.from(myArray) – converts an array represented by the variable myArray into an Observable. You can also use any an iterable data collection or a generator function as an argument of from().
* Observable.fromEvent(myInput, ‘keyup’) – converts the keyup event from some HTML element represented by myInput into an Observable
* Observable.interval(1000) – emits a sequential integer (0,1,2,3…) every second

### Q) Async Observable

By default, the operator from() returns a synchronous observable, but if you want an asynchronous one, use a second argument specifying an async scheduler:



### Q) Observable merge

1. *Interleave events by merging streams*

This strategy is useful for forwarding events from multiple streams and is ideal for handling different types of user interaction events like mouse or touch.

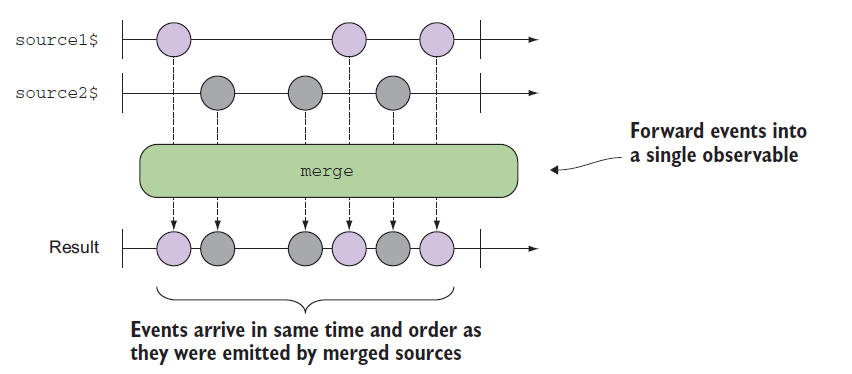
1. *Preserve order of events by concatenating streams*

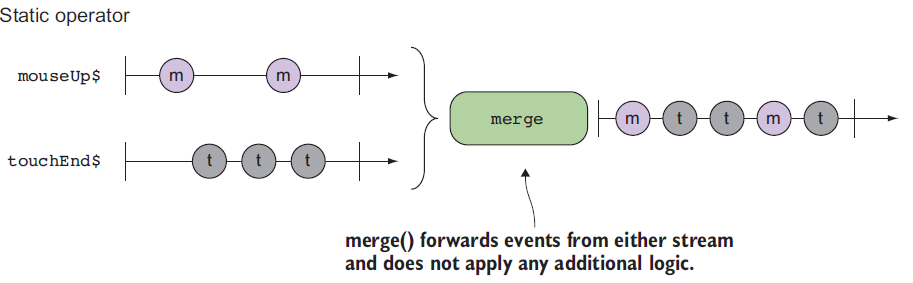
This one is used when the order of the events emitted by multiple streams needs to be preserved.

1. *Switch to the latest stream data*

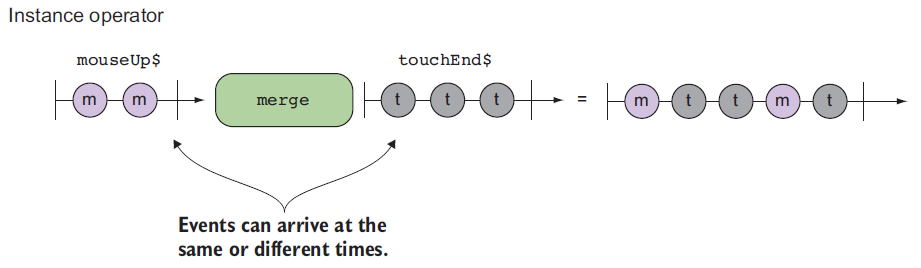
This is used when one type of event kicks off another, such as a button click initiating a remote HTTP call or beginning a timer.

##### Q) Interleave events by merging streams



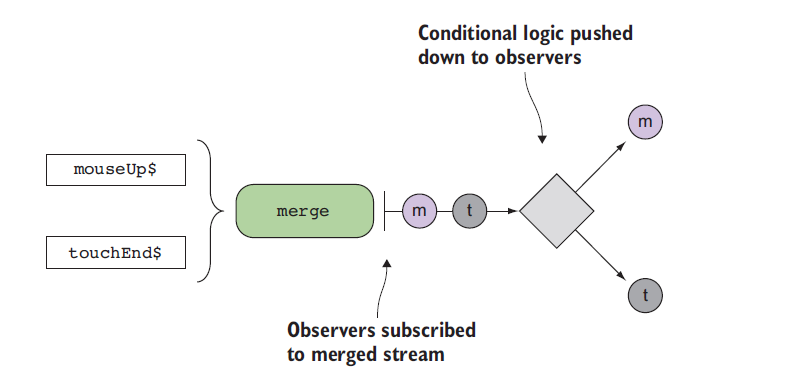


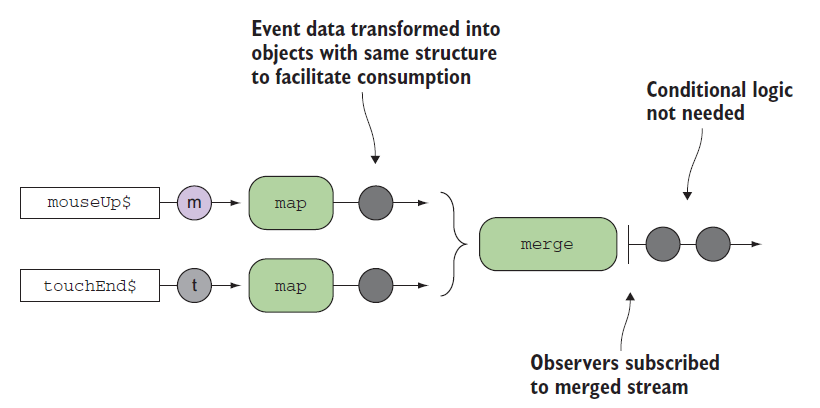






Two ways to handle stream





##### Q) Preserve order of events by concatenating streams

## Q) **Subscriber**

An Observable instance begins publishing values only when someone subscribes to it. You subscribe by calling the subscribe() method of the instance, passing an observer object to receive the notifications.

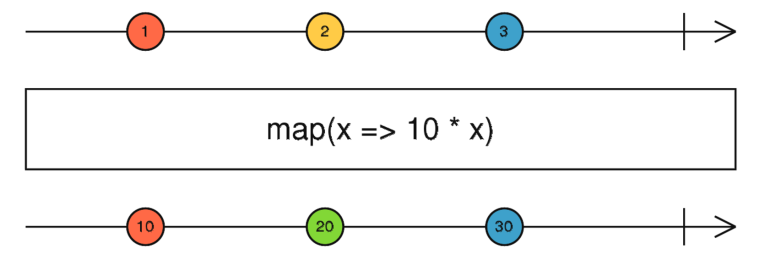
## Q) Operators

* Operators are functions that can transform the stream data between the moments when the Observable sent them and the function subscribe() received them.
* Each operator is a function that takes an Observable as an argument, transforms it and returns another Observable



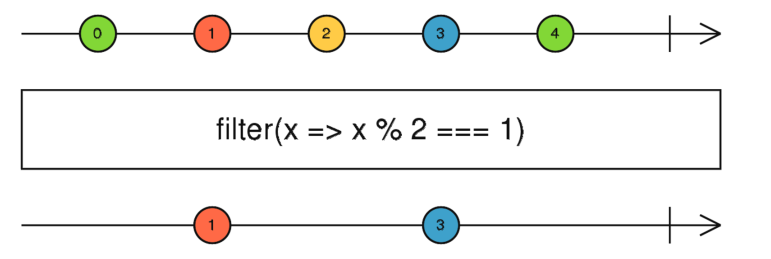
#### **Q) map()**

The map() operator transforms one value to another. It takes a given value from the observable stream and applies the provided transforming function to it.



#### **Q) filter()**

The filter() operator takes a function predicate as an argument, which returns true if the emitted value meets the criteria, or false otherwise. Only the values that meet the criteria will make it to the observer.

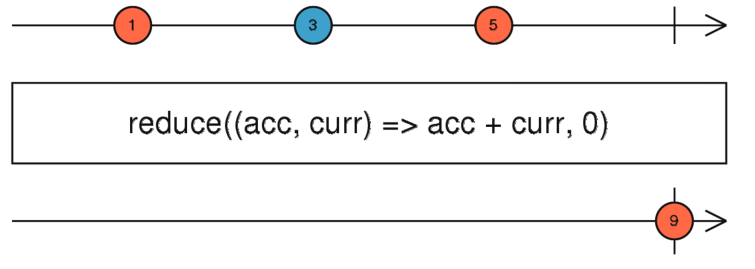


#### **Q) reduce()**

The operator reduce() that allows you aggregate values emitted by an observable.

As you see from the above diagram, the accumulator function also has two arguments:

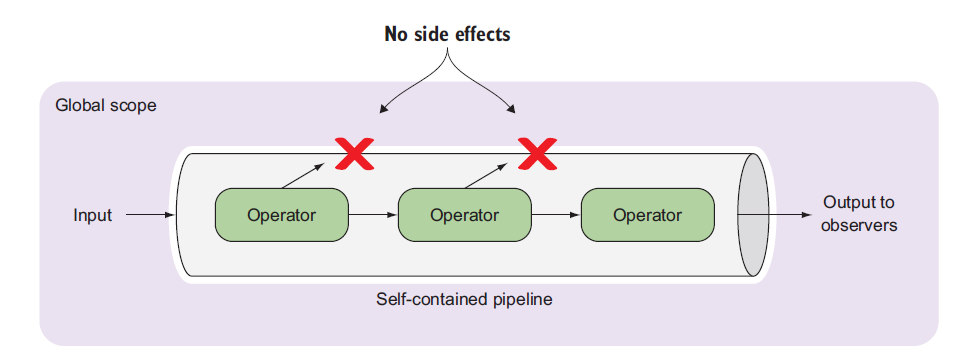
* acc stores the currently accumulated value, which is available for each emitted element
* curr stores the currently emitted value.



#### **Q) pipe()**

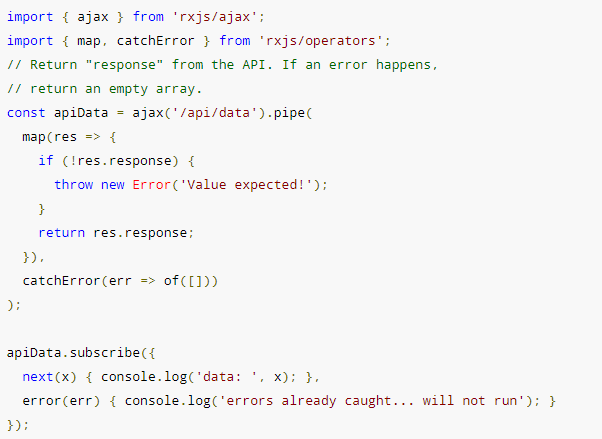
You can use pipes to link operators together. Pipes let you combine multiple functions into a single function. The pipe() function takes as its arguments the functions you want to combine, and returns a new function that, when executed, runs the composed functions in sequence.

A set of operators applied to an observable is a recipe—that is, a set of instructions for producing the values you’re interested in. By itself, the recipe doesn’t do anything. You need to call subscribe() to produce a result through the recipe.



#### Q) Error Handling

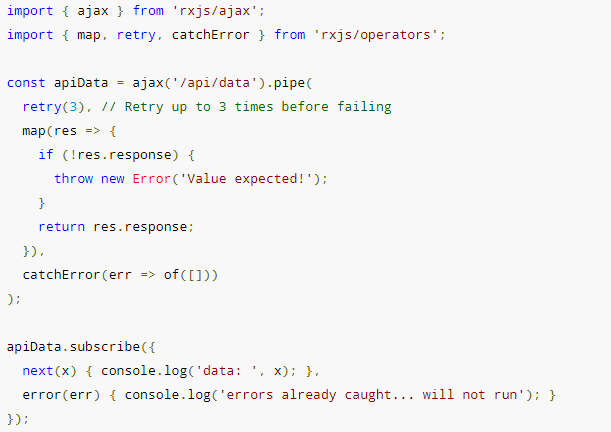
RxJS provides the catchError operator that lets you handle known errors in the observable recipe.



#### Q) Retry failed observable

Where the catchError operator provides a simple path of recovery, the retry operator lets you retry a failed request.

Use the retry operator before the catchError operator. It resubscribes to the original source observable, which can then re-run the full sequence of actions that resulted in the error. If this includes an HTTP request, it will retry that HTTP request.



## Q) Subject

* AsyncSubject: Subjects that will only emit the last item emitted by the source Observable when the source Observer completes the stream by calling onComplete()
* PublishSubject: The Subject only delivers to the Observers the events emitted after their subscription
* ReplaySubject: Emits all the events emitted by the source Observable, even those that were emitted before the subscription is made
* BehaviorSubject: Emits the last emitted item by the source Observable when the subscription is done, then continues to any other items emitted by the source observable

# Build

## Q) --prod flag

The --prod meta-flag engages the following optimization features.

* [Ahead-of-Time (AOT) Compilation](https://angular.io/guide/aot-compiler): pre-compiles Angular component templates.
* [Production mode](https://angular.io/guide/deployment#enable-prod-mode): deploys the production environment which enables production mode.
* Bundling: concatenates your many application and library files into a few bundles.
* Minification: removes excess whitespace, comments, and optional tokens.
* Uglification: rewrites code to use short, cryptic variable and function names.
* Dead code elimination: removes unreferenced modules and much unused code.