# TypeScript

## Q) three dots

Javascript's ECMA6 came out with some cool new features; ... is one of these new Javascript functionalities. It can be used in two different ways; as a spread operator OR as a rest parameter.

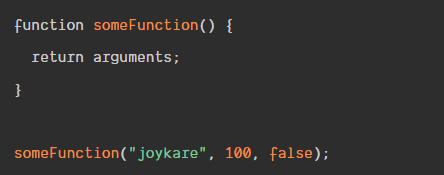
* Rest parameter: collects all remaining elements into an array.
* Spread operator: allows iterables( arrays / objects / strings ) to be expanded into single arguments/elements.

**Rest parameter**

Rest parameters have to be at the **last argument**. This is because it collects all remaining/ excess arguments into an array.



Arguments keywords



someFunction returns the arguments and their indexes,

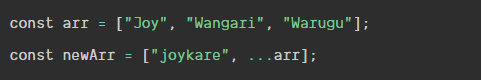
[Arguments] { '0': 'joykare', '1': 100, '2': false }.

The downside of using the arguments keyword is that, it returns an array-like **object**; this means you essentially cannot perform any array-methods like; Array.filer, Array.map.

Another pitfall, is that we cannot use arguments in arrow functions. This is because arrow-functions do not have their own this, and hence no arguments object either.

[**Spread operators**](https://scotch.io/bar-talk/javascripts-three-dots-spread-vs-rest-operators543#toc-spread-operators)

The spread operator allows us to expand elements. It let us unpack elements in an array to single/individual arguments.



The value of newArr will be [ 'joykare', 'Joy', 'Wangari', 'Warugu' ].

Note:Unlike rest parameters you can use the spread operator as the first argument.

**We can use the spread operator to copy an array.**

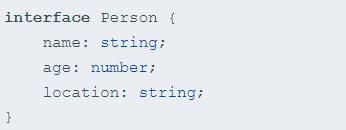


This copies arr into arr2. Now we can do things on arr2 and any changes done to arr2 will not have any effect arr.

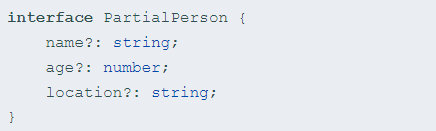
## Q) keyof

## Mapped types

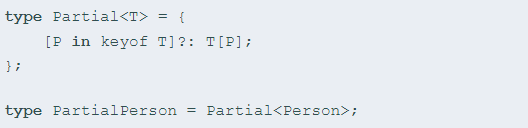
### Q) Partial



A partial version of it would be:



With Mapped types, PartialPerson can be written as a generalized transformation on the type Person as:



# 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.

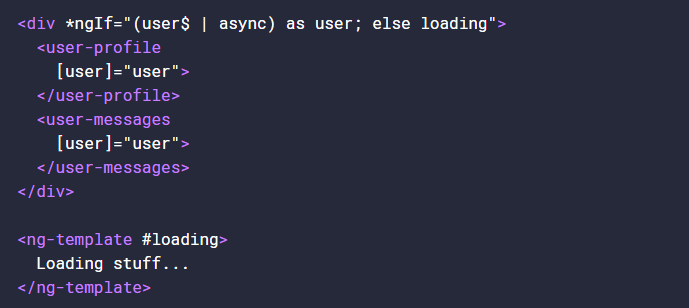
[RouterModule](https://angular.io/api/router/RouterModule) also offers a forChild static method for configuring the routes of lazy-loaded modules.

[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.

## Pipe

### Q) async

The [async](https://angular.io/api/core/testing/async) pipe subscribes to an Observable or Promise and returns the latest value it has emitted. When a new value is emitted, the [async](https://angular.io/api/core/testing/async) pipe marks the component to be checked for changes. When the component gets destroyed, the [async](https://angular.io/api/core/testing/async) pipe unsubscribes automatically to avoid potential memory leaks.

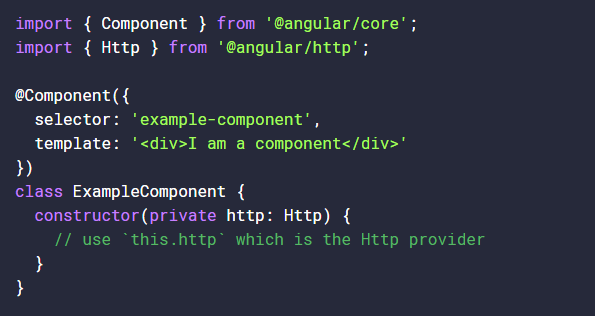
What “as user” will do is wait until user$ | async has evaluated, and bind the result to the value of user.

**Problems**

Everytime we use the async pipe, we create a subscription. If you’re going to subscribe directly to Observables that initiate data transfer, it’s likely you’ve come across unwanted issues such as duplicate Http requests.

## DI

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



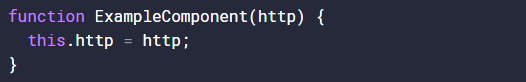
Inside our tsconfig.json files we’ll likely have emitDecoratorMetadata set to true. This emits metadata about the type of the parameter into a decorator in our compiled JavaScript output.



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:

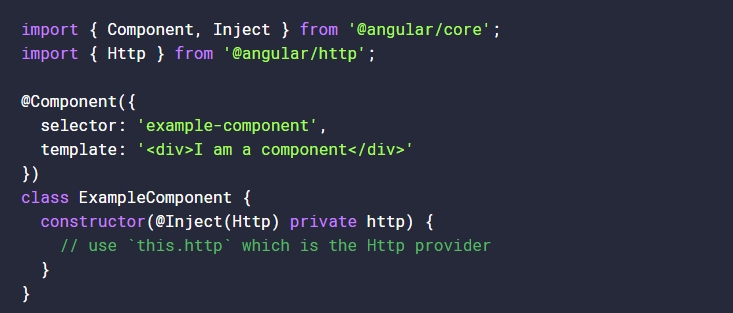


So essentially, the @Component decorator is transformed into plain ES5, and some additional metadata is supplied through the \_\_decorate assignment. Which in turn tells Angular to lookup the Http token and supply it as a first parameter to the Component’s constructor - assigning it to this.http



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

### Q) Inject()

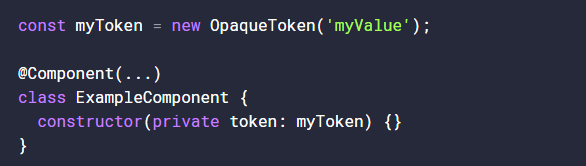


At this point, @Inject is a manual way of specifying this lookup token, followed by the lowercase http argument to tell Angular what to assign it against.

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:

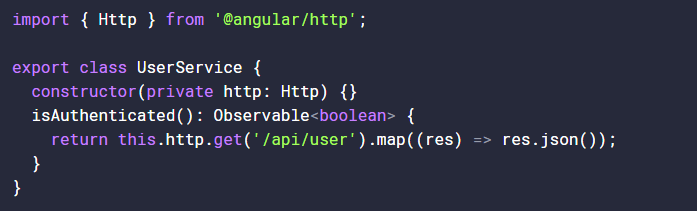


### Q) Injectable()

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.

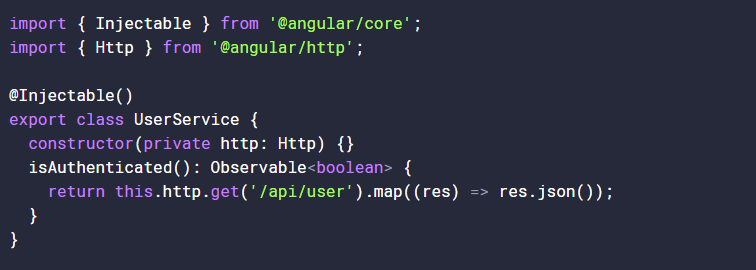
If no Angular decorator has been used on a class there is no way for Angular to read what dependencies it requires. This is why we need to use @Injectable().

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



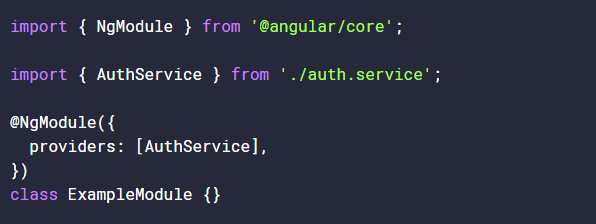
This would break as the Http provider metadata would not be stored for Angular to compose it correctly.

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

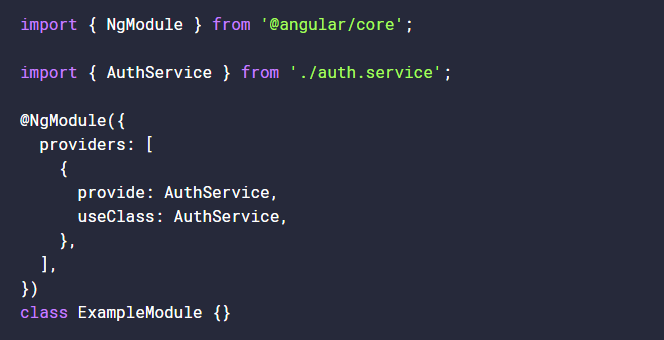


At this point, Angular is aware of the Http token and can supply it to http.

### 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.

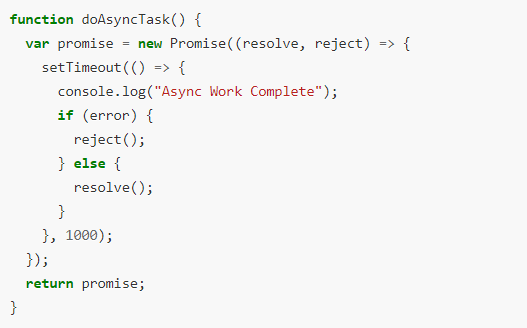
### Q) Understand Injectors

## Promise

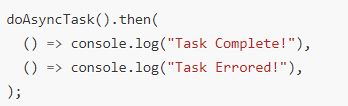
### Q) Promise syntax

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

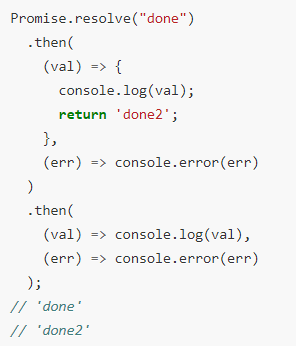


One of the nice things about Promises is that if we add a then handler **after** the promise resolves or rejects the handler **still** gets called.



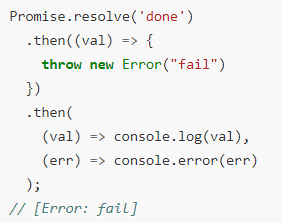
### Q) Chaining

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

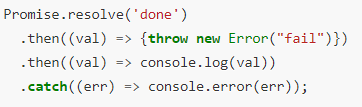


### Q) Error handling

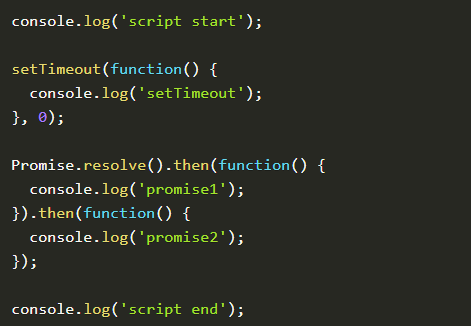
If we throw an exception from our promise function or one of the success handlers, the promise gets rejected and the error handler is called, like so:

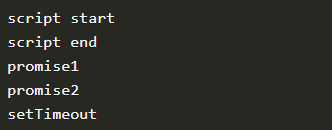


The catch function works exactly the same way as the then error handler, it’s just clearer and more explicitly describes our intent to handle errors.



### Q) Calling order





There are different queues. There is a microtask and a macrotask queue. A promise gets added to the microtask queue and a setTimeout to the macrotask queue. Whenever the call stack is cleared, the microtasks queue gets cleared first. Hence the ‘promise1’ is the next log statement we see.

When a microtask is finished, the rest of the microtasks queue gets executed until the microtask queue is empty. That’s the reason ‘promise2’ is logged next. So even if the second promise gets scheduled during the execution of the first, it still gets executed before the setTimeout.

Lastly, when the microtask queue is cleared, the next task is picked from the macrotask queue, and the ‘setTimeout’ is logged.

## Router

### Q) Router Configuration



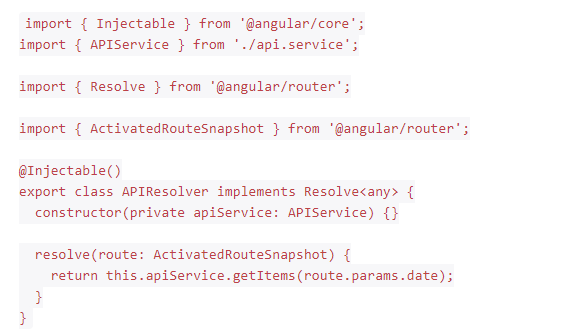
The order of the routes in the configuration matters and this is by design. The router uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes.

#### 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.





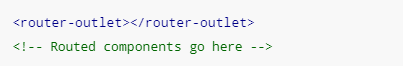
https://www.callibrity.com/blog/angular-2-route-resolves

#### 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.

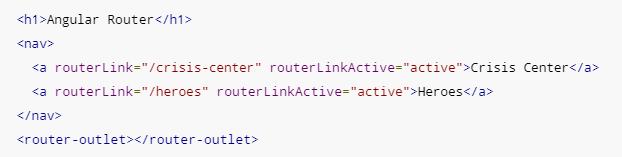
### Q) RouterOutlet

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

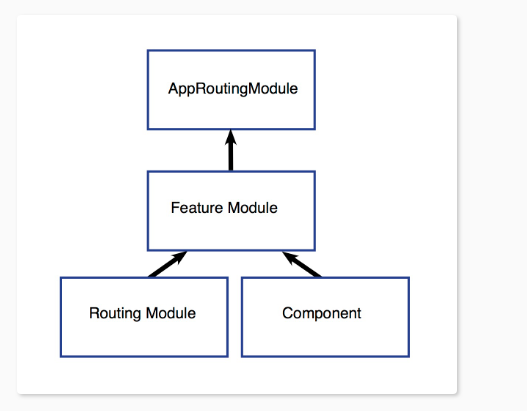


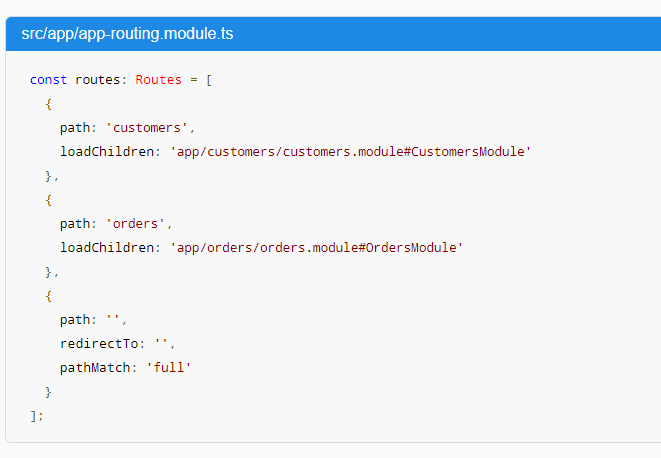
### Q) RouterLink

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

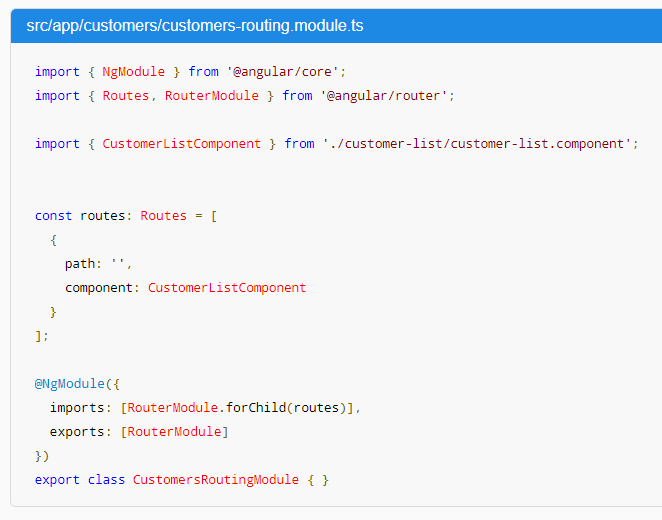




Routes at the top level

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

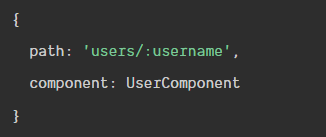
You might have noticed that the CLI adds RouterModule.forRoot(routes) to the app-routing.module.ts imports array. This lets Angular know that this module, AppRoutingModule, is a routing module and [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) specifies that this is the root routing 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).

### Q) Two ways to grab route parameters

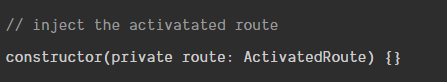
* The Snapshot Way: The router provides us with a snapshot of the current route
* The Observable/Stream Way: Since Angular employs Observables heavily, the router also returns an Observable that we can listen to.

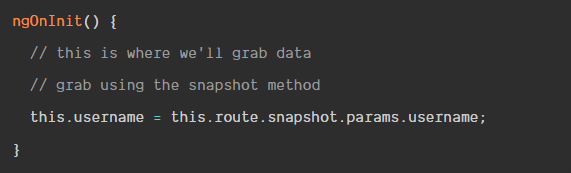




#### Q) the snapshot way

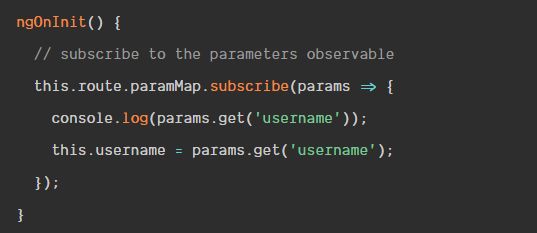
We'll inject the ActivatedRoute into this component via the constructor. Next we'll use the ActivatedRoute to grab the username route parameter in ngOnInit()





A scenario where the snapshot method wouldn't work is when we are on one profile and click a link to second profile. The snapshot method only runs one time when the component is initiated. The component wouldn't update if we travelled from (/users/chris) to another profile (/users/nick).

#### Q) the observe way



## 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.

# RxJS

## Q) Players

1. Producers
2. Consumers
3. Data pipeline
4. Time

### Q) Producer

Producers are the sources of your data. A stream must always have a producer of data, which will be the starting point for any logic that you’ll perform in RxJS. In practice, a producer is created from something that generates events independently (anything from a single value, an array, mouse clicks, to a stream of bytes read from a file). In RxJS, we call them *observables*, as in something that’s *able to be observed*.

Observables are in charge of pushing notifications, so we refer to this behavior as fire-and-forget, which means that we’ll never expect the producer to be involved in the *processing* of events, only the emission of them.

### Q) Consumer

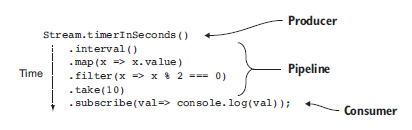
Consumer to accept events from the producer and process them in some specific way. When the consumer begins listening to the producer for events to consume, you now have a stream, and it’s at this point that the stream begins to push events; we’ll refer to a consumer as an *observer*.

### Q) Data Pipeline

One advantage of RxJS is that you can manipulate or edit the data as it passes from the producer to the consumer. This is where the list of methods (known as observable operators) comes into play. Manipulating data en route means that you can adapt the output of the producer to match the expectations of the consumer.

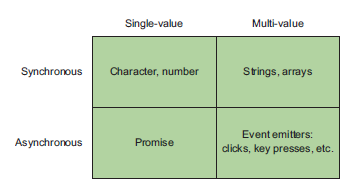
### Q) Time

The time need not always run at normal speed, and you can build streams that run slower or faster depending on your requirements.



## Q) Observable

### Q) Type of data source



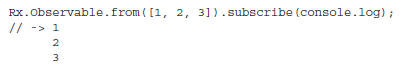
#### SINGLE-VALUE, SYNCHRONOUS

The simplest case is that you have only a single piece of data. In programming, you know there are operations that return a single value for each invocation. This is the category of any function that returns a single object. You can use the Rx.Observable.of() function to wrap a single, synchronous value. As soon as the subscriber is attached, the value is emitted.



#### MULTI-VALUE, SYNCHRONOUS

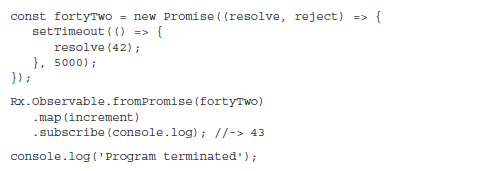
You can also group single items together to form collections of data, mainly for arrays. In order to apply the same operation that you used on the single item on all of the items, you would traditionally iterate over the collection and repeatedly apply the same operation to each item in the collection.



Both of these groups operate synchronously, which means each subsequent block of code must wait for the previous block to complete before executing. In the multi value example, each item will be processed serially (one by one) until the collection is exhausted.

#### SINGLE-VALUE, ASYNCHRONOUS

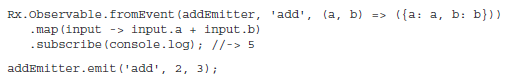
Like with the first dimension, you also have a single-value case, where the result of a task will result in a single return value. This kind of operation is usually used to load some remote resource via an AJAX call or wait on the result of some non-local calculation wrapped in a Promise, without blocking the application. In either case, after the operation is initiated, it will expect a single return value or an error.



**Promises are single-value and immutable, they’re never run again.**

#### MULTI-VALUE, ASYNCHRONOUS

An event emitter doesn’t stop after a single event; instead, it can continue to invoke the registered callbacks for each event that arrives, creating a practically infinite stream of events. The emitter will fulfill both of your criteria for handling multi-value, asynchronous events.

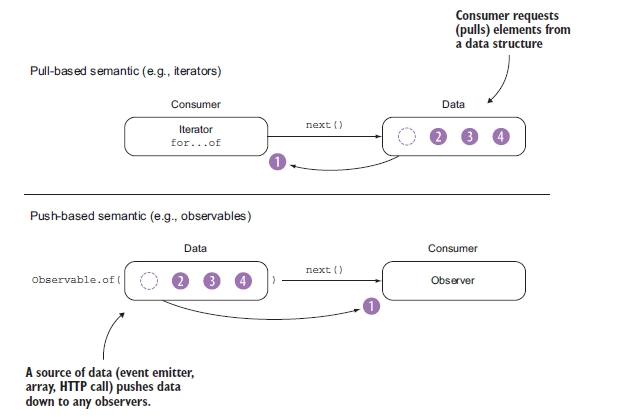


### Q) O**bservables Constructors**

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

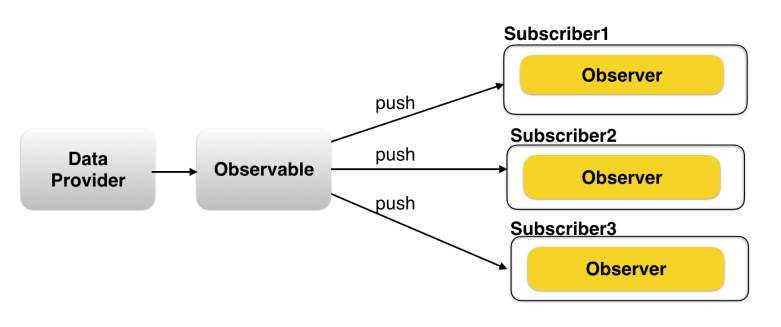
* Observable.of(1,2,3) – turns the sequence of numbers into an Observable
* Observable.create(myObserver) – returns an Observable that can invoke  methods on myObserver that you will create and supply as an argument.
* Observable.from(myArray) – converts an array represented by the variable myArray into 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) Data push pattern



Iterators use a pull*-*based semantic. This means that the consumer of the iterator is responsible for requesting the next item from the iterator.

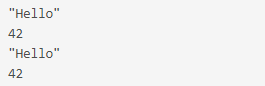
RxJS observables use push-based notifications, which means they don’t request data; rather, data is pushed onto them so that they can react to it. Push notifications bring the reactive paradigm to life.

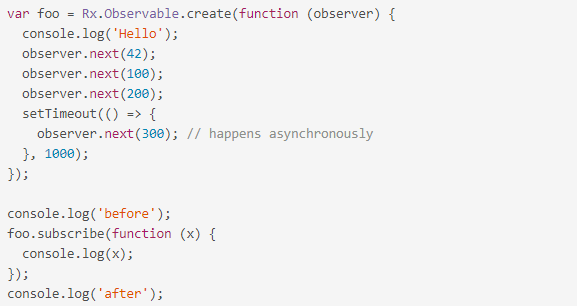


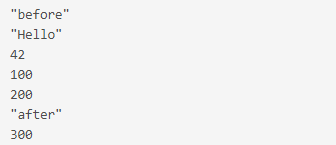
### Q) Observable vs Function

Observables are like functions with zero arguments, but generalize those to allow multiple values. Subscribing to an Observable is analogous to calling a Function. Observables are able to deliver values either synchronously or asynchronously.





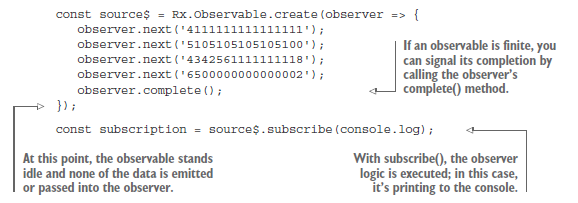




Conclusion:

* func.call() means "give me one value synchronously"
* observable.subscribe() means "give me any amount of values, either synchronously or asynchronously"

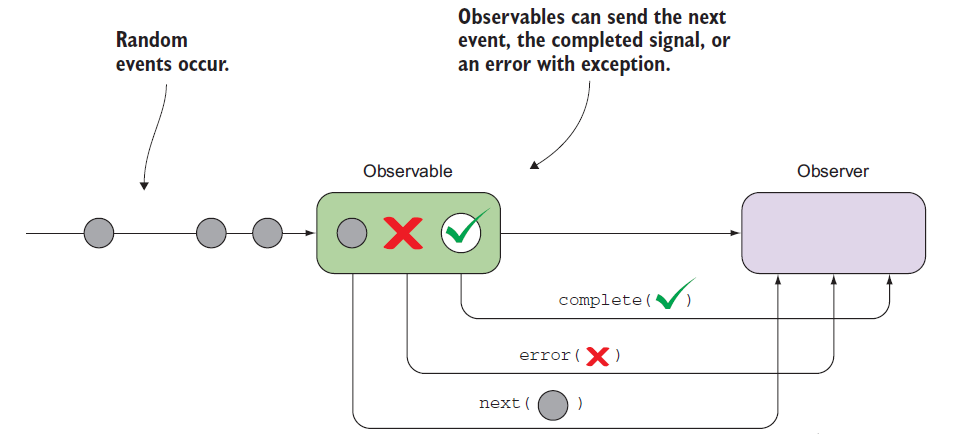
### Q) Rx.Observable.create()



The code inside Observable.create(function subscribe(observer) {...}) represents an "Observable execution", a lazy computation that only happens for each Observer that subscribes. The execution produces multiple values over time, either synchronously or asynchronously.

There are three types of values an Observable Execution can deliver:

* "Next" notification: sends a value such as a Number, a String, an Object, etc.
* "Error" notification: sends a JavaScript Error or exception.
* "Complete" notification: does not send a value.

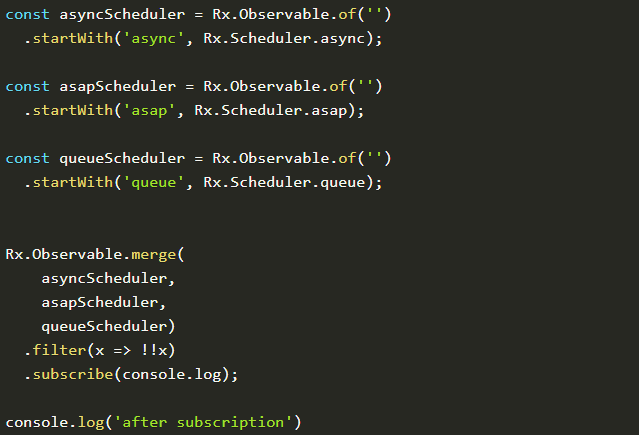


Next notifications are the most important and most common type: they represent actual data being delivered to an Observer. Error and Complete notifications may happen only once during the Observable Execution, and there can only be either one of them. In an Observable Execution, zero to infinite Next notifications may be delivered. If either an Error or Complete notification is delivered, then nothing else can be delivered afterwards.

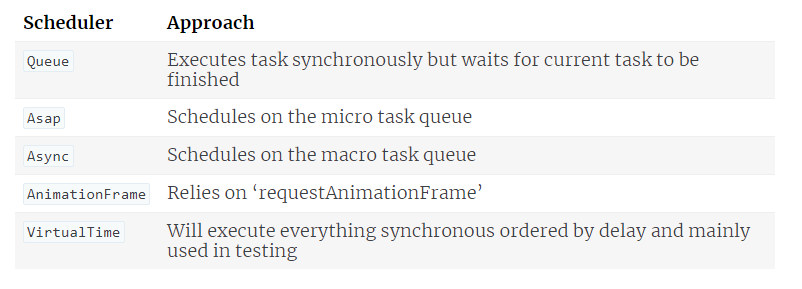


### Q) Schedulers

Schedulers influence the timing on which tasks get executed. You can change the default schedulers of some operators by passing in an extra scheduler argument.







### 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.

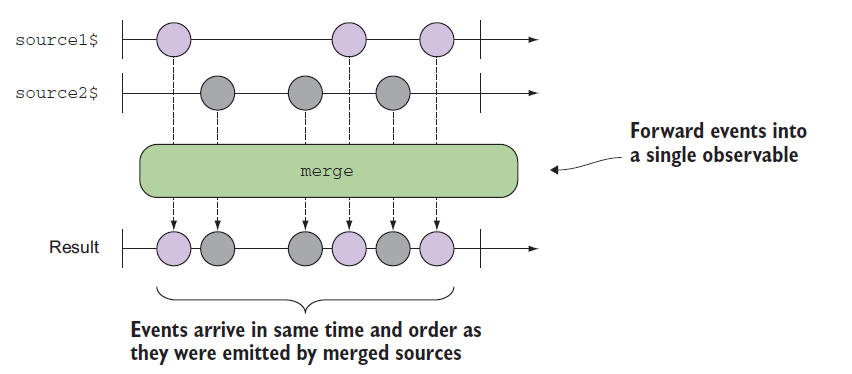
1. *Preserve order of events by concatenating streams*

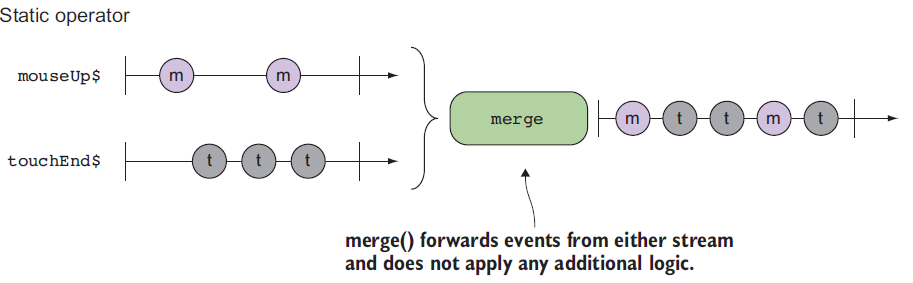
This one is used when the order of the events emitted by multiple streams needs to be preserved.

1. *Switch to the latest stream data*

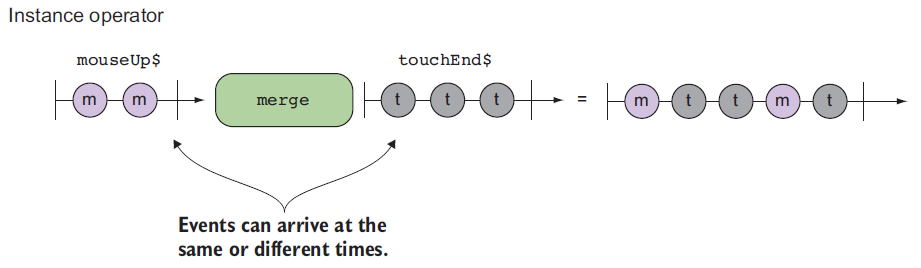
This is used when one type of event kicks off another, such as a button click initiating a remote HTTP call or beginning a timer.

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



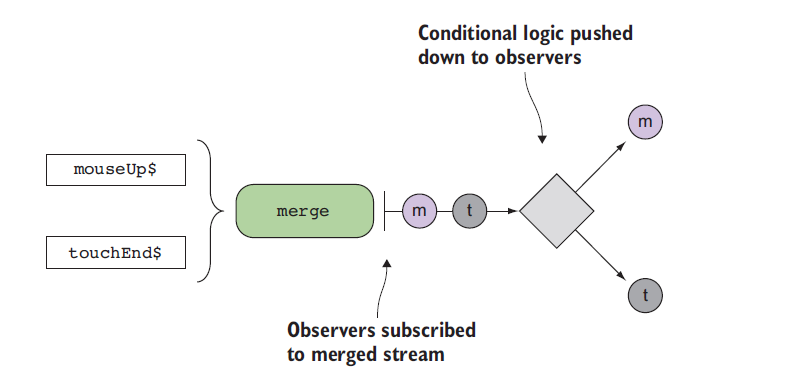


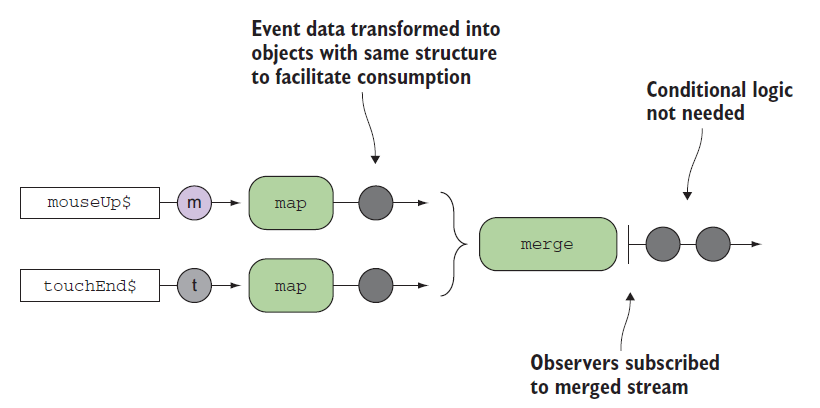






Two ways to handle stream





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

### Q) Observable types

#### Q) Cold Observable

**A cold observable is one that doesn’t begin emitting all of its values until an observer subscribes to it.** Cold observables are typically used to wrap bounded data types such as numbers, ranges of numbers, strings, arrays, and HTTP requests, as well as unbounded types like generator functions. These resources are known as passive in the sense that their declaration is independent of their execution. This also means that these observables are truly lazy in their creation and execution.

With cold observables, all subscribers, no matter at what point the subscription occurred, will observe the same events.

A cold observable is one that, when subscribed to, emits the entire sequence of events to any active subscribers.

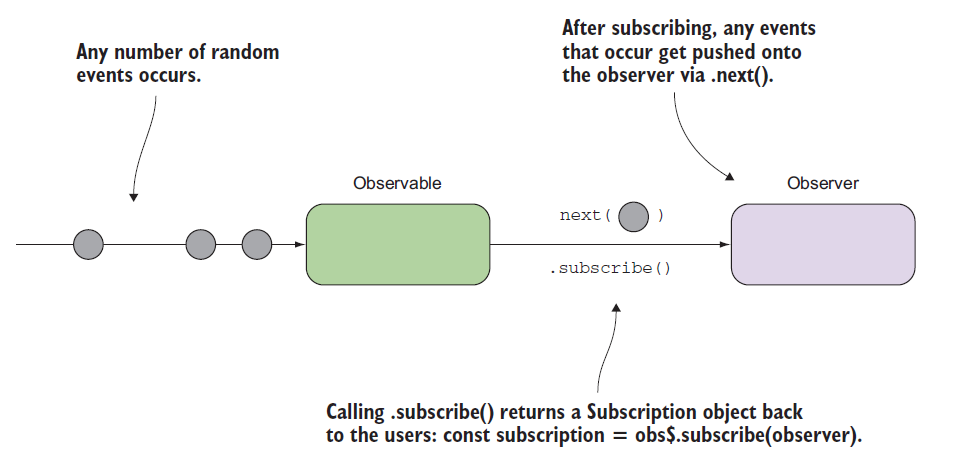
#### Q) Hot Observable

Hot observables are those that produce events regardless of the presence of subscribers— they are active. In the real world, hot observables are used to model events like clicks, mouse movement, touch, or any other events exposed via event emitters.

Unlike cold observables that create independent copies of the data source to emit to every subscriber, a hot observable shares the same subscription to all observers that listen to it. Therefore, you can conclude that a hot observable is one that, when subscribed to, emits the ongoing sequence of events from the point of subscription and not from the beginning.

## Q) Observer

Observers are created within the context of a subscription, which means that the result of calling subscribe() on an observable source is a Subscription object. Because observables operate synchronously or asynchronously, the consumer of an observable must in some way support the inversion of control that also happens with callbacks. This is consistent with its push-based mechanism.



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

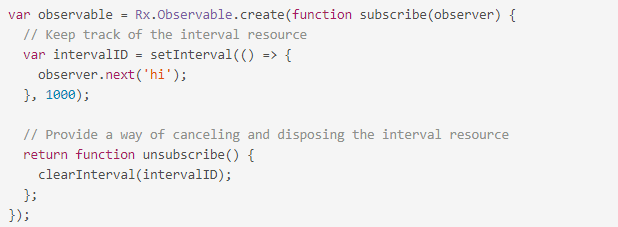
## Q) Subscription

Because Observable Executions may be infinite, and it's common for an Observer to want to abort execution in finite time, we need an API for cancelling an execution. Since each execution is exclusive to one Observer only, once the Observer is done receiving values, it has to have a way to stop the execution, in order to avoid wasting computation power or memory resources.

When you subscribe, you get back a Subscription, which represents the ongoing execution. Just call unsubscribe() to cancel the execution.



Each Observable must define how to dispose resources of that execution when we create the Observable using create(). You can do that by returning a custom unsubscribe function from within function subscribe().



Just like observable.subscribe resembles Observable.create(function subscribe() {...}), the unsubscribe we return from subscribe is conceptually equal to subscription.unsubscribe. In fact, if we remove the ReactiveX types surrounding these concepts, we're left with rather straightforward JavaScript.

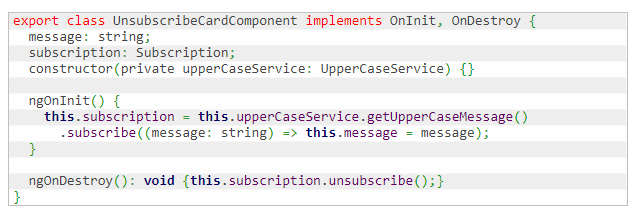


## Q) Unsubscribe

Manually unsubscribe from all custom Observables when a component/directive gets destroyed. The best place to unsubscribe is inside functions that handle the *OnDestroy* lifecycle hook. Some subscriptions like router and http don’t need manual unsubscribe, for the rest of them there are various solutions:

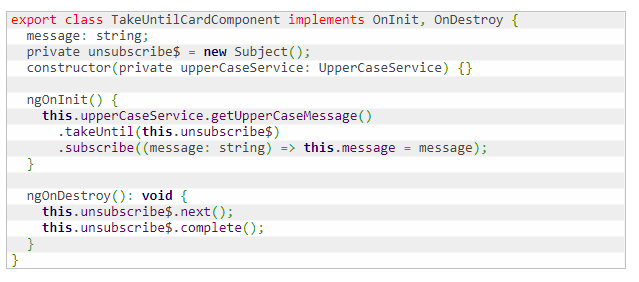
* execute unsubscribe over the subscription object
* using takeUntil operator
* using async pipe

**Unsubscribe**



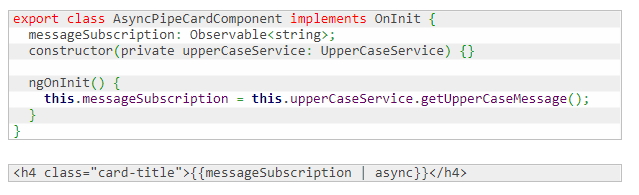
Subscription represents a disposable resource, it has an unsubscribe method that can be used to dispose the resource held by the subscription.

**TakeUntil**



TakeUntil takes a second Observable as an argument, it monitors the second Observable and discard subscription after it emits a value or terminates.

**AsyncPipe**



The async pipe subscribes to an Observable and returns the latest value it has emitted. When a component is destroyed, the async pipe unsubscribes automatically.

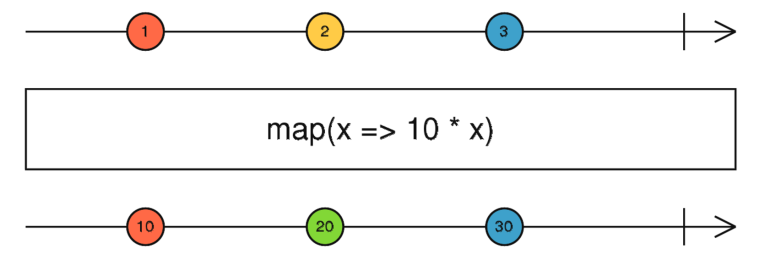
## Q) Operators

* 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



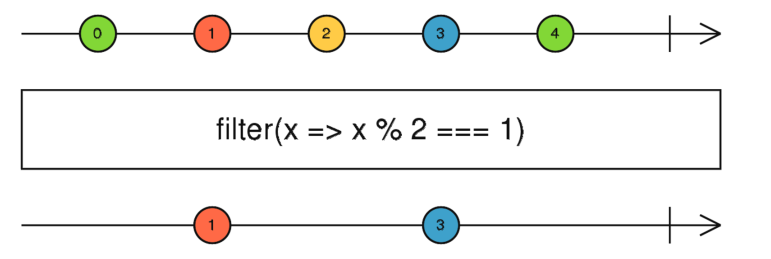
### **Q) map()**

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.

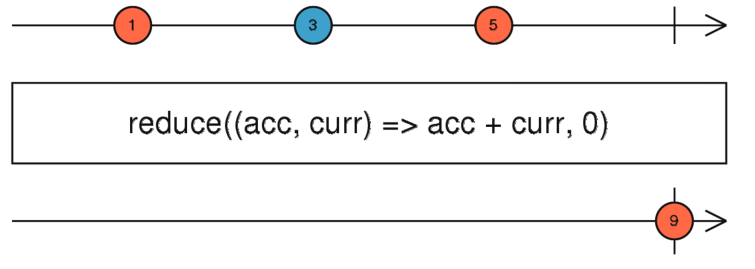


### **Q) reduce()**

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

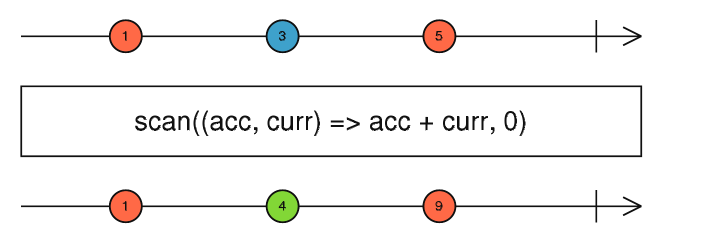
As you see from the above diagram, the accumulator function also has two arguments:

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



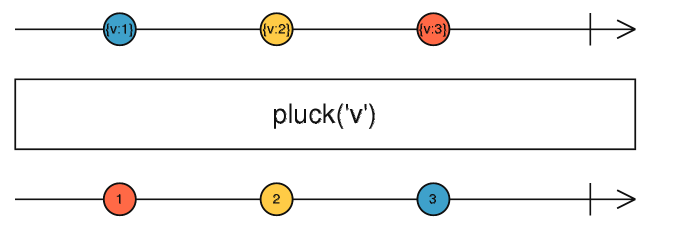
### Q) scan()

RxJS uses scan() to apply an accumulator function over an observable sequence (just like reduce()) but returns each intermediate result as the accumulation process is happening and not all at once.



### Q) pluck()

Maps each source value (an object) to its specified nested property.

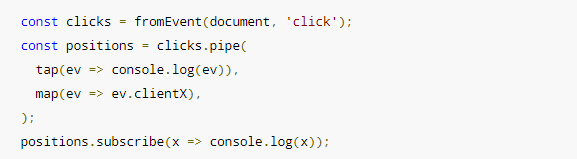


### Q) tap()

Returns a mirrored Observable of the source Observable, but modified so that the provided Observer is called to perform a side effect for every value, error, and completion emitted by the source. Any errors that are thrown in the aforementioned Observer or handlers are safely sent down the error path of the output Observable.

This operator is useful for debugging your Observables for the correct values or performing other side effects.

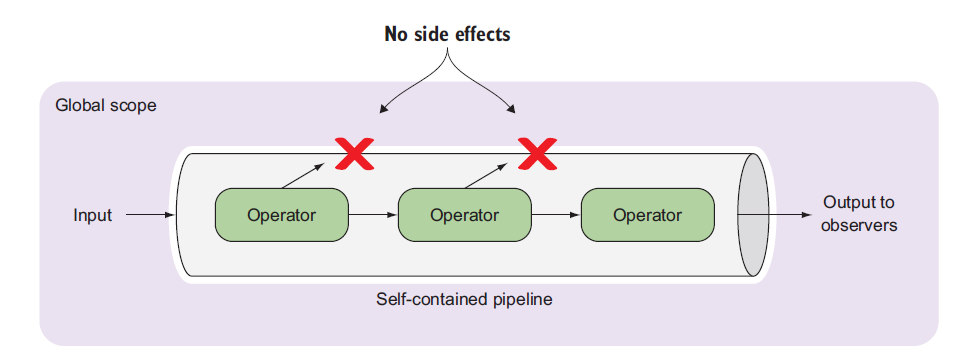
Note: this is different to a subscribe on the Observable. If the Observable returned by tap is not subscribed, the side effects specified by the Observer will never happen. tap therefore simply spies on existing execution, it does not trigger an execution to happen like subscribe does.



### **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.

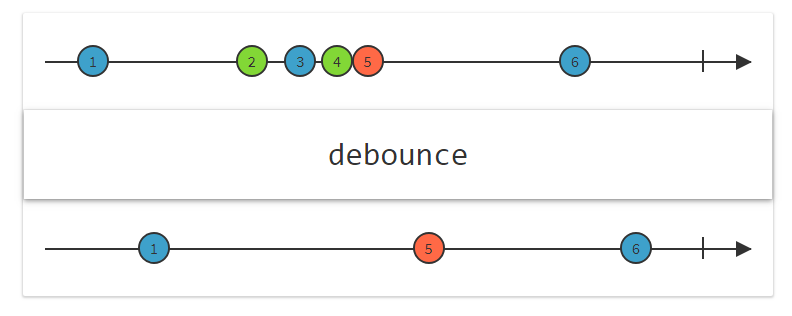


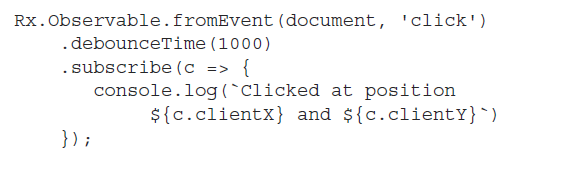
Problems with the patched operators for dot-chaining are:

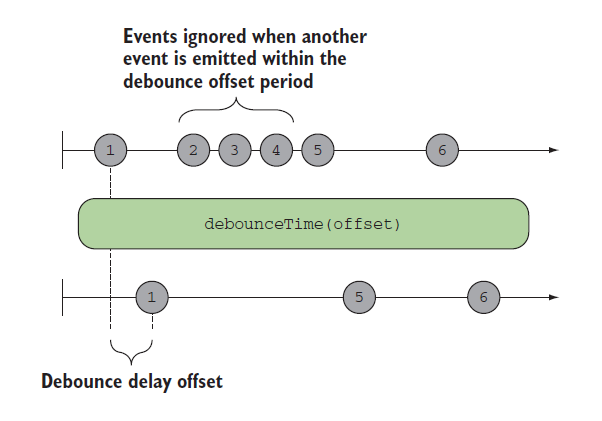
1. Any library that imports a patch operator will augment the Observable.prototype for all consumers of that library, creating blind dependencies. If the library removes their usage, they unknowingly break everyone else. With pipeables, you have to import the operators you need into each file you use them in.
2. Operators patched directly onto the prototype are not "tree-shakeable" by tools like rollup or webpack. Pipeable operators will be as they are just functions pulled in from modules directly.
3. Unused operators that are being imported in apps cannot be detected reliably by any sort of build tooling or lint rule. That means that you might import scan, but stop using it, and it's still being added to your output bundle. With pipeable operators, if you're not using it, a lint rule can pick it up for you.
4. Functional composition is awesome. Building your own custom operators becomes much, much easier, and now they work and look just like all other operators from rxjs. You don't need to extend Observable or override lift anymore.

### Q) debouncetime()

Only emit an item from an Observable if a particular timespan has passed without it emitting another item



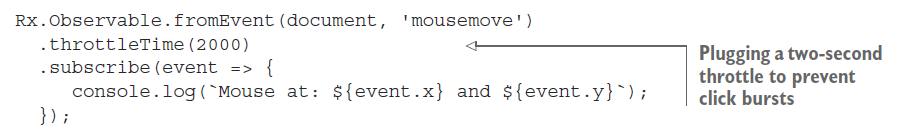


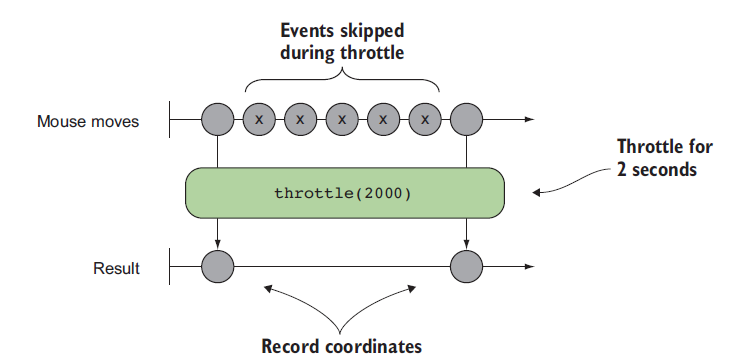


With this code, the user can generate a burst of click events, but only the last one will be emitted after a second of inactivity.

### Q) throttleTime()

Throttling ignores values from an observable sequence that are followed by another value before a certain time. In simple terms, this means “execute a function at most once every period.”





### Q) debounceTime vs throttleTime

debounceTime

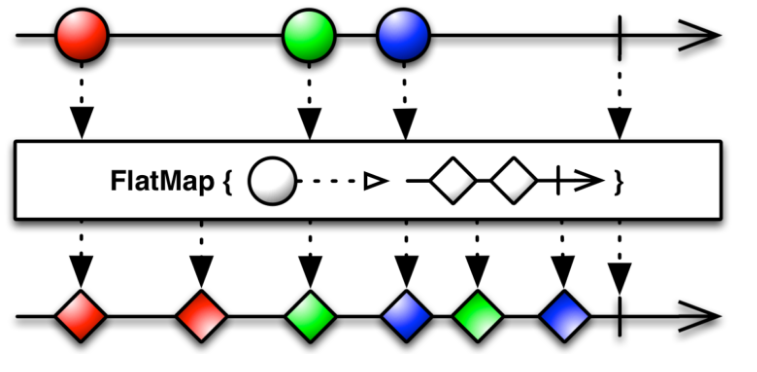
* accepts 1 input stream, time period number
* Whenever an event is emitted, the time period of silence measured restarts from zero
* It waits for a time period of silence and then emits the latest value of the input stream
* returns a new stream of debounced values

throttleTime

* accepts 1 input stream, time period number
* It starts by emitting the first values of the input stream
* Then, it limits the rate of values to at most one per time period
* returns a new stream of throttled values

### Q) flatmap()

flatMap() is used to “transform the items emitted by an observable into observables, then flatten the emissions from those into a single observable”.

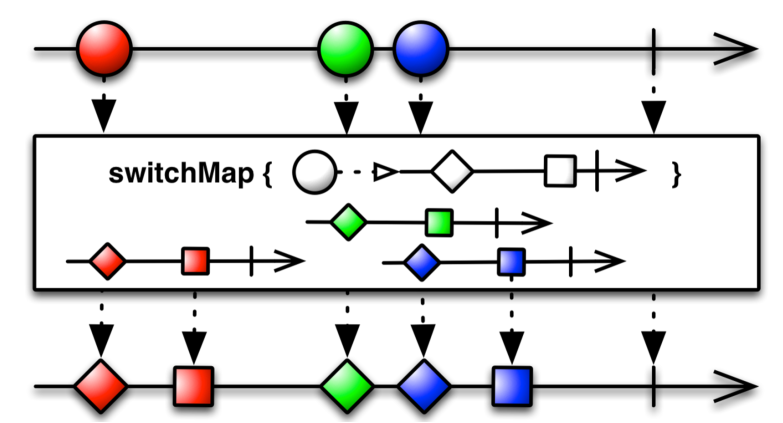


The flatMap() operator takes an emitted item from the outer observable (the circle) and unwraps its content (the inner observable of diamonds) into the flattened output observable stream. The flatMap() operator merges the emissions of the inner observables so their items may interleave.



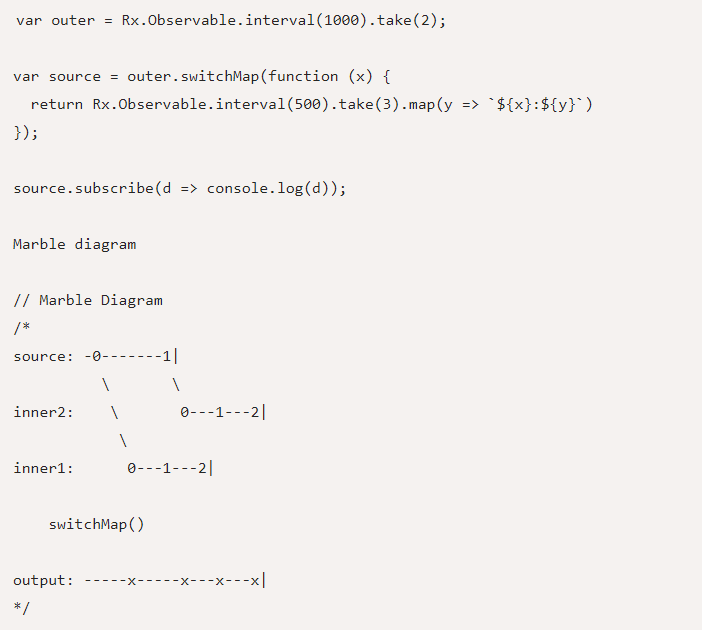
### Q) switchmap()

The main difference between switchMap and other flattening operators is the cancelling effect. On each emission the previous inner observable (the result of the function you supplied) is cancelled and the new observable is subscribed. You can remember this by the phrase switch to a new observable.



The outer observable emits the red circle, and switchMap() emits the item from the inner observable (red diamond and square) into the output stream. The red circle was processed without any interruptions because the green circle was emitted after the inner observable finished processing.

The situation is different with the green circle. The switchMap() managed to unwrap and emit the green diamond, but the blue circle arrived *before* the green square was processed. So the subscription to the green inner observable was cancelled, and the green square was never emitted into the output stream. In other words, the switchMap() operator *switched* from processing of the green inner observable to the blue one.

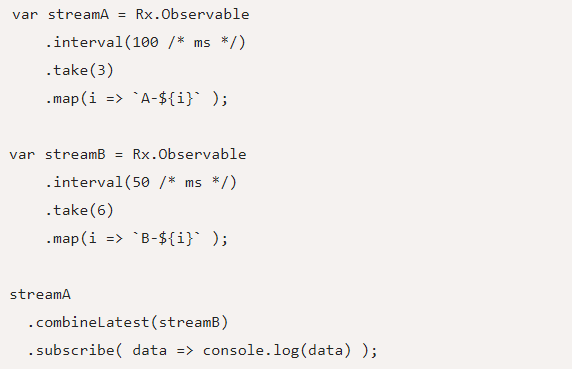


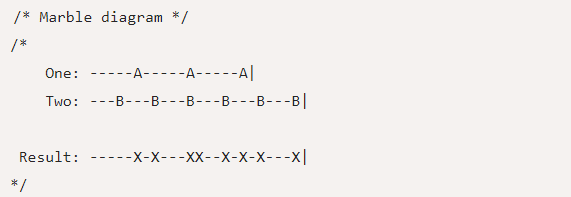
The moment the inner2 stream became available, the output no longer listens to data from inner1. switchMap effectively switches over to inner2 and unsubscribes from the previous one.

This also implies that even though the inner streams may be producing data concurrently in switchMap, the output is only determined by the latest observable that is still producing data.

### Q) combineLatest()

This operator is best used when you have multiple, long-lived observables that rely on eachother for some calculation or determination. Be aware that combineLatest will not emit an initial value until each observable emits at least one value.



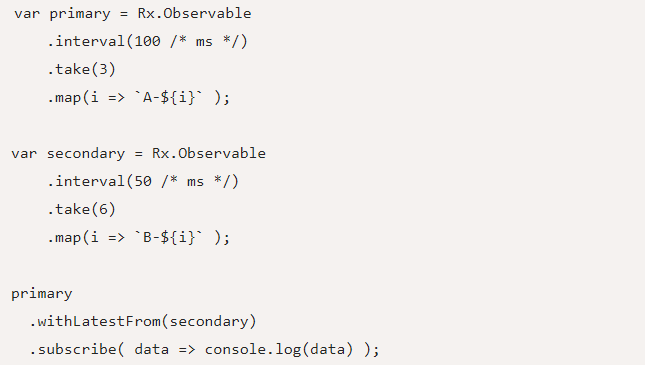


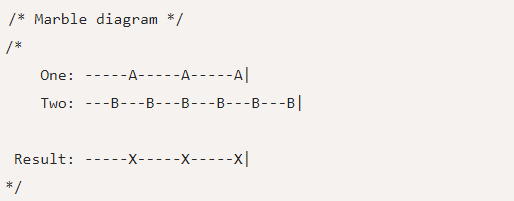
In the above example, since stream B emits faster than stream A, there is no output until stream A also starts emitting. Also notice that when both the streams emit data at the same time (as in the code example above), the result stream sees two immediate data points.

When using combineLatest, the data of both the streams have an equal weightage in terms of producing an output given that data is present on both the streams.

### Q) withLatestFrom ()

You use a primary stream to control when the data is emitted on the result.

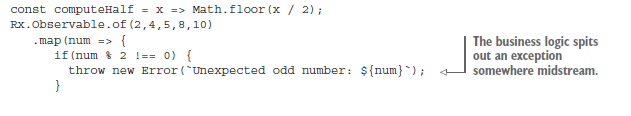


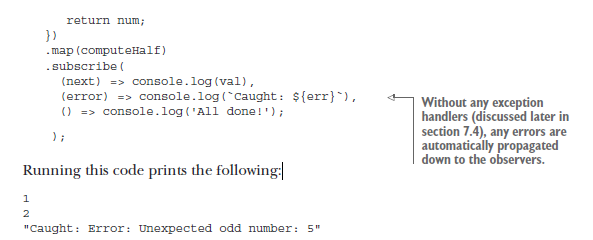


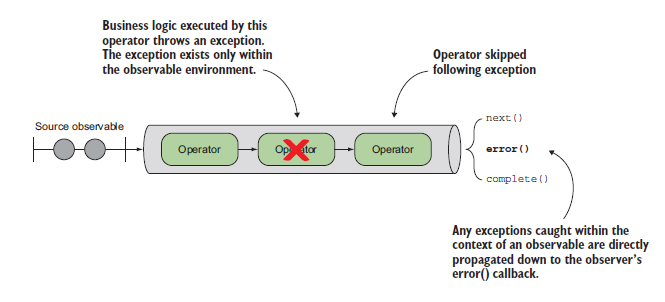
Notice how in this case, data is emitted only 3 times on the result stream because the primary observable only emitted 3 data points. As expected, it combines with the latest value of the secondary stream when emitting data on the result, just like combineLatest does.

## Q) Error Handling

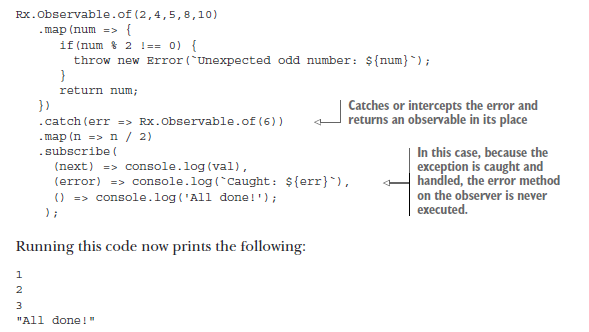
### Q) Errors propagated downstream to observers

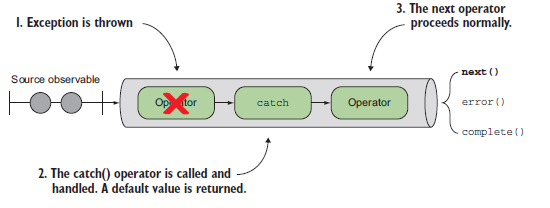




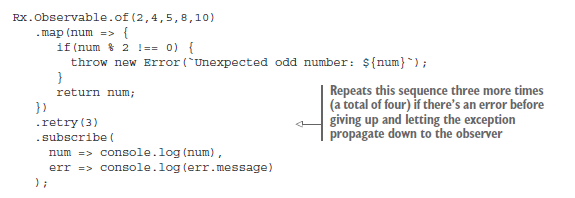


### Q) Catching and reacting to errors



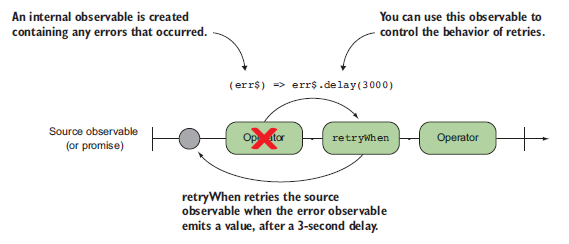


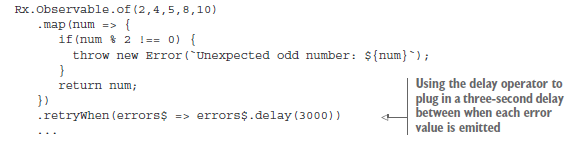
### Q) Retry failed observable



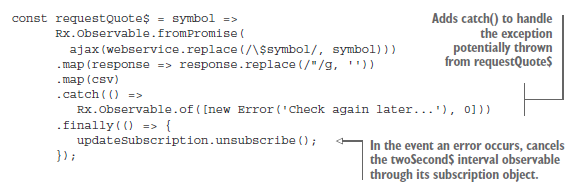
Running this code will print a sequence of numbers 2 and 4 a total of four times before printing “Unexpected odd number: 5.”

Q) Retry when





### Q) finally()



## Q) Subject

### **Q) What is a Subject?**

An RxJS Subject is a special type of Observable that allows values to be multicasted to many Observers. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), Subjects are multicast.

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

**Every Subject is an Observable.** Given a Subject, you can subscribe to it, providing an Observer, which will start receiving values normally. Internally to the Subject, subscribe does not invoke a new execution that delivers values. It simply registers the given Observer in a list of Observers, similarly to how addListener usually works in other libraries and languages.

**Every Subject is an Observer.** It is an object with the methods next(v), error(e), and complete(). To feed a new value to the Subject, just call next(theValue), and it will be multicasted to the Observers registered to listen to the Subject.



A multicasted Observable uses a Subject under the hood to make multiple Observers see the same Observable execution. Under the hood, this is how the multicast operator works: Observers subscribe to an underlying Subject, and the Subject subscribes to the source Observable.

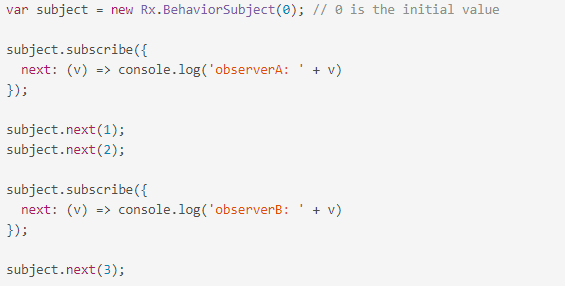


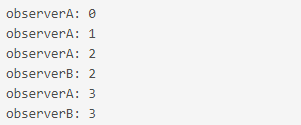
### Q) Subject types

* 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

#### Q) BehaviorSubject

BehaviorSubjects are useful for representing "values over time". For instance, an event stream of birthdays is a Subject, but the stream of a person's age would be a BehaviorSubject.

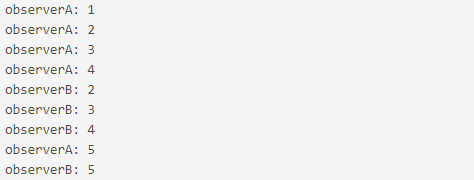




#### Q) ReplaySubject

A ReplaySubject records multiple values from the Observable execution and replays them to new subscribers.





#### Q) AsyncSubject

The AsyncSubject is a variant where only the last value of the Observable execution is sent to its observers, and only when the execution completes.





The AsyncSubject is similar to the [last()](http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-last) operator, in that it waits for the complete notification in order to deliver a single value.

## Q) Schedulers

### Q) What is a Scheduler?

A scheduler controls when a subscription starts and when notifications are delivered. It consists of three components.

* A Scheduler is a data structure. It knows how to store and queue tasks based on priority or other criteria.
* A Scheduler is an execution context. It denotes where and when the task is executed (e.g. immediately, or in another callback mechanism such as setTimeout or process.nextTick, or the animation frame).
* A Scheduler has a (virtual) clock. It provides a notion of "time" by a getter method now() on the scheduler. Tasks being scheduled on a particular scheduler will adhere only to the time denoted by that clock.

# Redux

**Redux attempts to make state mutations predictable** by imposing certain restrictions on how and when updates can happen. These restrictions are reflected in the [three principles](https://redux.js.org/introduction/threeprinciples) of Redux.

## Q) [Three principles](https://redux.js.org/introduction/threeprinciples)

* Single source of truth

**The** [**state**](https://redux.js.org/glossary#state) **of your whole application is stored in an object tree within a single** [**store**](https://redux.js.org/glossary#store)**.**

* State is read-only

**The only way to change the state is to emit an** [**action**](https://redux.js.org/glossary#action)**, an object describing what happened.**

* Changes are made with pure functions

**To specify how the state tree is transformed by actions, you write pure** [**reducers**](https://redux.js.org/glossary#reducer)**.**

## **Q) Action**

Actions are payloads of information that send data from your application to your store. They are the only source of information for the store. You send them to the store using [store.dispatch()](https://redux.js.org/api/store#dispatch).

Actions are plain JavaScript objects. Actions must have a type property that indicates the type of action being performed. Types should typically be defined as string constants.

Other than type, the structure of an action object is really up to you.

## Q) Reducer

**Reducers** specify how the application's state changes in response to [actions](https://redux.js.org/basics/actions) sent to the store. Remember that actions only describe *what happened*, but don't describe how the application's state changes.

## Q) Store

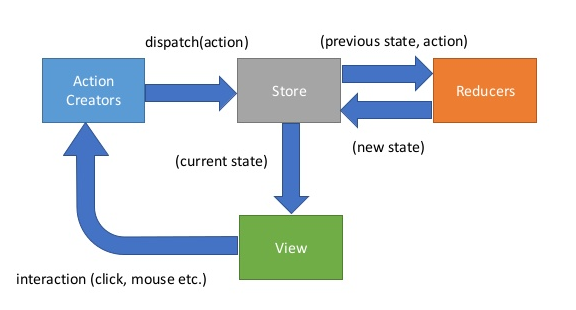
We defined the [actions](https://redux.js.org/basics/actions) that represent the facts about “what happened” and the [reducers](https://redux.js.org/basics/reducers) that update the state according to those actions.

The Store is the object that brings them together. The store has the following responsibilities:

* Holds application state;
* Allows access to state via [getState()](https://redux.js.org/api/store#getState);
* Allows state to be updated via [dispatch(action)](https://redux.js.org/api/store#dispatch);
* Registers listeners via [subscribe(listener)](https://redux.js.org/api/store#subscribe);
* Handles unregistering of listeners via the function returned by [subscribe(listener)](https://redux.js.org/api/store#subscribe).

## Q) Data flow

Redux architecture revolves around a **strict unidirectional data flow.**



# NgRx

## Q) Advantages of Store

Throughout the overview we touched briefly on the advantages of utilizing Store over a typical, Angular 1 style approach but let's take a moment to recap. Why take the time to invest in this particular library, architecture pattern, and learning curve? The primary advantage to a Store-based application are **centralized state**, **performance**, **testability**, and **tooling**.

Centralized, Immutable State

All relevant application state exists in one location. This makes it easier to track down problems, as a snapshot of state at the time of an error can provide important insight and make it easy to recreate issues. This also makes notoriously hard problems such as undo/redo trivial in the context of a Store application and enables powerful tooling.

Performance

Since state is centralized at the top of your application, data updates can flow down through your components relying on slices of store. Angular 2 is built to optimize on such a data-flow arrangement, and can disable change detection in cases where components rely on Observables which have not emitted new values. In an optimal store solution this will be the vast majority of your components.

Testability

All state updates are handled in reducers, which are pure functions. Pure functions are extremely simple to test, as it is simply input in, assert against output. This enables the testing of the most crucial aspects of your application without mocks, spies, or other tricks that can make testing both complex and error prone.

Tooling and Ecosystem

A centralized, immutable state also enables powerful tooling. One such example is ngrx developer tools, which provides a history of actions and state changes, allowing for 8 time travel during development. The patterns provided by Store also allow for a rich ecosystem of easy to implement middleware. Because store provides an entry point both before and after dispatched actions hit application reducers, problems such as syncing slices of state to local storage, advanced logging, and implementing sagas are easily solved with a quick package include and a few lines of code. This ecosystem will only grow over the coming months.

## Q) Actions

Actions are objects that extend the NgRx Action class with a ‘type’ property.  They have an optional ‘payload’ property (naming is up to you but the standard is to name it ‘payload’) for sending in data to the effect/reducer and are dispatched by the store to either run an effect or change state in a reducer.

## Q) Reducers

Reducers are pure functions that are the only ones that can change state.  They aren’t really changing state but making a copy of existing state and changing one or more properties on the new state.

## Q) Effects

Effects allow us to handle asynchronous operations in NgRx.

* Most times this will be calling an API
* The resulting data should be stored in state by returning an action for the reducer
* Effects always return one or more actions (unless you decorate @Effect with {dispatch: false})
* You can inject services into your effects as well so if you need to access those in NgRx, effects are the place to do it

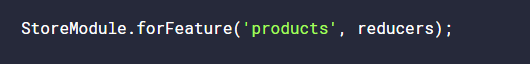
## Q) Selectors

Selectors are pure functions that take slices of state as arguments and return some state data that we can pass to our components.

To better understand what selectors are and what they do, it helps see ngrx state as a data structure - a tree that can be serialised to JSON. Data is added to the state tree by composing state in reducers - that’s the easy part. Now to get data out of the state tree, we have to traverse it to find our property of interest - and return it. That can become more complex, and is where selectors help us out.

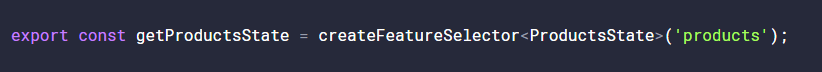
### Q) Top level feature state

We register any feature state into feature modules by importing the StoreModule and invoking it with .forFeature():



The first argument of .forFeature contains a string that represents the name of the feature state, and the second argument supplies our reducers which manage that feature state. The feature name plays a crucial role in creating a state selector for the feature state using a handy function provided by ngrx/store called createFeatureSelector.

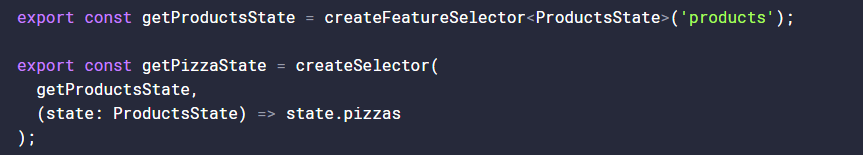
createFeatureSelector allows us to get a top-level feature state property of the state tree simply by calling it out by its feature name:



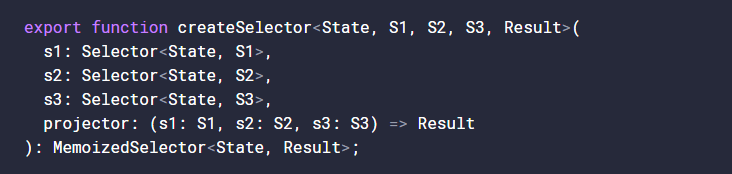
Firstly, we pass it a string that represents the name used to register the feature state in the feature module. It uses this string to look up the feature state from within the root state object, such as state['products'].

It then returns a typed selector function that will return a reference to that specific slice of state.

### Q) State slice selector



The createSelector function takes up to eight selector functions as arguments, each one referencing different slices of state. The last argumet to createSelector can be treated as our “projector function”.



Notice how in the projector we are given s1, s2 and s3 as the function arguments - in the order in which we supplied them. The projector function is passed, as arguments, the returned values of the selectors listed before it in the order in which they were listed.

The role of the projector function is a powerful one. We can ask for various state properties, anywhere within our state tree, we can derive, transform or combine data from the state slices passed to it and return this modified data as a single object - typically for component consumption.

# Karma

# Webpack