

Async JS

Promises, Observables, RXJS

Our Goals

- ▶ **Extend the tools we have for dealing with async code in JS**
- ▶ **Know when our async code is written badly and improve it**
- ▶ **Know who to deals with exceptions in our async code**
- ▶ **Make our async code readable**

What is async code

- ▶ **Code that will run sometime in the future**
- ▶ **Can you give examples of scenarios where you will have to probably write async code?**
- ▶ **Async code is hard for programmers to write**
- ▶ **It's hard to debug**
- ▶ **It's hard to deal with exceptions**

Async code EX

- ▶ **Create a timer that will run in the future and print an hello world message on the screen**

What is Promise

- ▶ As mentioned we have quite a few challenges for dealing with async code
- ▶ To help us deal with those challenges we have the class **Promise**
- ▶ That class is available since ES6 (for older version you have to install a library)
- ▶ To create a promise we need to create an instance of the **Promise class**
- ▶ The constructor will get the async code that we are wrapping
- ▶ The class provides us with an API to easily deal with our async code.
- ▶ The promise will eagerly run once the async function it wraps and it is not cancelable

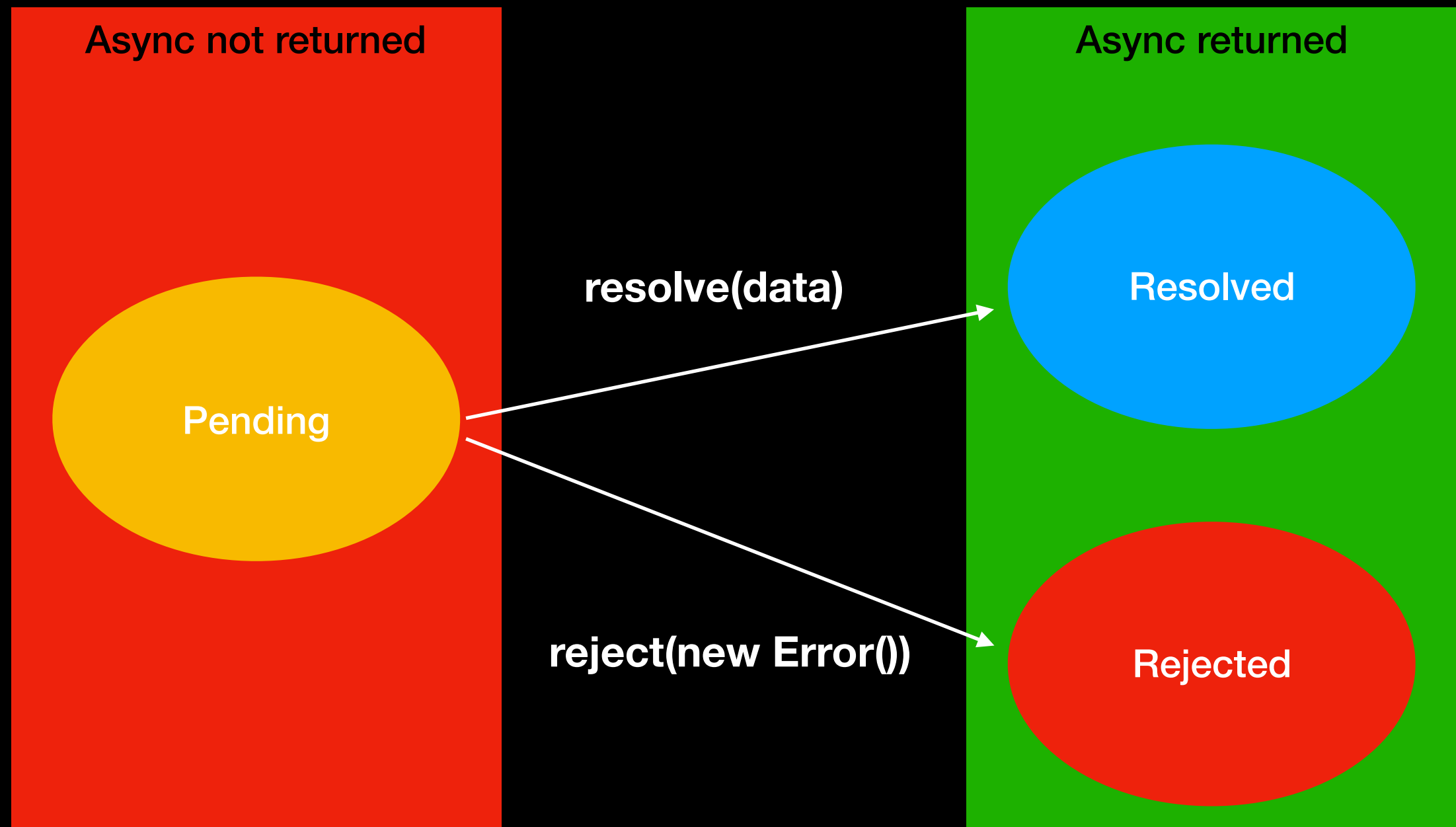
Promise Questions

- ▶ **How can we use promise to wrap our async code and why should we do it?**
- ▶ **How can we manipulate our async code using promises?**
- ▶ **How to deal with errors in our async code using promises**
- ▶ **How can we write our async code so it will be more readable.**

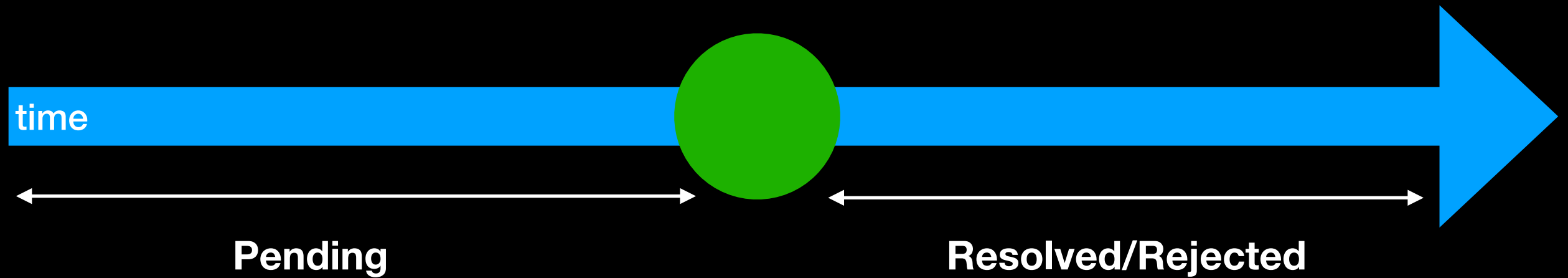
State of a promise

- ▶ **An instance of the promise class can be in one of the following states**
 - **Pending - the async code did not return**
 - **Resolved - The async code returned - success**
 - **Rejected - The async code returned - fail**
- ▶ **When creating the promise we are passing the constructor a function that wraps our async code**
- ▶ **That function is called with 2 arguments of type methods: resolve, reject**
- ▶ **Calling the resolve will transfer the promise to the Resolved state**
- ▶ **Calling the reject will pass the promise to the Rejected state**
- ▶ **We can pass a data argument that will be passed to the listeners**
- ▶ **We can call resolve / reject and pass data once**

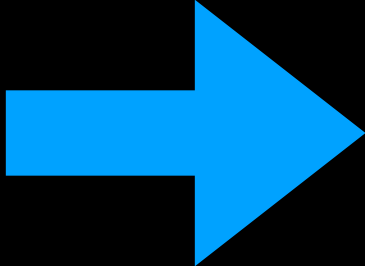
State of a promise



Promise timeline



```
const myStringPromise = new Promise(function(resolve, reject) {  
  ...  
  if (isShouldResolve) {  
    resolve('Some message')  
  } else {  
    reject(new Error('some error'))  
  }  
})
```



myStringPromise : Promise<string>

then - Attaching a listener to the promise

- ▶ We can attach a listener when the async code is back
 - When our promise is resolved
 - When our promise is rejected
- ▶ We attach the listener by calling the **then** method on the promise instance
- ▶ The then method will get an optional two methods
 - The first one will run on the resolved state
 - The second one will run on the rejected state
- ▶ Those methods will get the data we pass when we call the resolve or reject methods we get on the promise async function

Promise - EX

- ▶ **Create 2 timers and wrap each one with a promise**
- ▶ **One promise will be resolved after 1 sec with success**
- ▶ **The second Promise will be rejected after 2 sec with fail**
- ▶ **The success promise will pass a string message to the listeners**
- ▶ **The Fail promise will pass an Error instance with a message**
- ▶ **Attach a listener to both of those promises which will print the message they get on success or the message from the error instance**

catch - attaching a listener to the reject

- ▶ There are times when we only want to attach an error function
- ▶ one way to do it is to use **then** and pass the first argument of the resolve method as **null** or **undefined**
- ▶ A shortcut for doing this is using the **catch** method

what catch/then return

- ▶ **catch and then return a different Promise**
- ▶ **That promise contains data depending on what is returned from the functions inside the then and catch**
- ▶ **This allows you to wait for a promise to get resolved or rejected and then send another promise**
- ▶ **if the return method of the catch and then is a promise, then the next promise will contain the data in the promise returned**
- ▶ **If you throw an error from the method in the then or catch then the returned promise is in rejected state**

what catch/then return - Question

What will be the data in the following promise ? Promise<?>

```
function timer() : Promise<string>
function fetch(url : string) : Promise<Response>
```

```
timer()
  .then(function(urlFromTimer) {
    return fetch(urlFromTimer)
  })
  .then(function(res) {
    return res.json()
  })
  .then(function(json) {
    JSON.stringify(json);
  })
  .then(function(jsonAsString) {
    return jsonAsString.length;
  })
  .catch(function() {
    return 'some error has happened';
  })
```

Promise chaining - EX

- ▶ Install a package called: **node-fetch**
- ▶ Modify the timer exercise we did before
- ▶ The success timer should resolve with a string url
- ▶ we then fetch our todo server based on that url:
 - <https://nztodo.herokuapp.com/api/task/?format=json>
- ▶ We then return the json from the response
- ▶ we then print the objects
- ▶ On the failed timer we are rejecting with a bad url
- ▶ we will fetch from that bad url and get 404
- ▶ if the status is 404 we will throw an error
- ▶ We will print the error message to the console

Promise.all - combining promises

- ▶ **with this method we can run multiple promises simultaneously**
- ▶ **the method gets an array of promises**
- ▶ **the method will return a promise which will be resolved when all the promises in the array are resolved**
- ▶ **if one of the promises is rejected then the combined promise will be rejected as well**
- ▶ **The data in the combined promise will be an array with the data from each promise**

Promise.all - EX

- ▶ Create a timer promise which will be resolved in 1 seconds resolving with a string message
- ▶ Create another timer promise which will be resolved in 2 seconds with a number
- ▶ combine those 2 promises with Promise.all
- ▶ After how much time will the combined promise be resolved?
- ▶ after the combined promise is resolved transform it with promise chaining to a promise with the number of letter of the message from the 1st promise plus the number resolved in the 2nd promise

Dealing with errors in async code

- ▶ If an error happened in our promise or promise chain we have to return a rejected promise
- ▶ a common mistake is to have a promise chain and just let whomever listens to the promise deal with the errors
- ▶ That most important rule of async exceptions is - BE SPECIFIC
- ▶ for example if you have a promise chain of 10 promises and you let the end point deal with the error how will you know which of the 10 had the error
- ▶ Add a catch wherever a promise in the chain can fail
- ▶ create exception classes and return a specific exception class

Exception - EX

- ▶ **Create a custom exception describing 404 server error**
- ▶ **use node-fetch to send a 404 request and make the promise reject with the exception you created.**

async await function

- ▶ the goal of the async function is to write our promise async code in a readable sync way without the need for callbacks
- ▶ you put the async keyword before the function
 - `async function alotOfAsyncCode() { ... }`
- ▶ in async functions you can place an await keyword before your promise
 - `const timerPromiseResult = await timerPromise;`
- ▶ When there is an await the function will step out and continue other code or rest, when the promise is resolved it will step back to the function filling the variable with the data from the promise
- ▶ To catch rejected promises we use try and catch
- ▶ The async function will return a promise with the data returned from the function

async await function - EX

- ▶ **create a regular function that creates a timer promise based on an interval argument**
- ▶ **Create an async function**
 - **In the async function run two timer promises together using Promise.all**
 - **create a timer rejected promise and place it in try and catch**
 - **The async function should return the concat messages of all the timer promises**

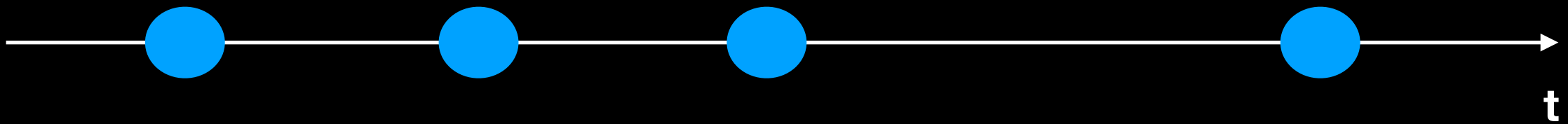
Is promise enough? - Problem

- ▶ **As mentioned a promise runs even if there are no listeners**
- ▶ **A promise is not cancelable**
- ▶ **A promise can call resolve or reject once so what if our async data is not a single async event?**
- ▶ **A promise will run the function it wraps immediately and will run once for all listeners**
- ▶ **We have certain async problems that repeat in different scenarios it will be helpful if we would have additional function to help us solve them**
- ▶ **To extend our async toolset even further we will learn about another library called RXJS**

Data Stream

- ▶ Lets define a concept that will help us understand the purpose for the RXJS library
- ▶ a data stream represents a timeline
- ▶ on that timeline there are async events that emit data, we will call those event pulse.
- ▶ A data stream can contain zero to infinite amount of pulses
- ▶ Few examples
 - interval - we use setInterval to emit a pulse every amount of time
 - event - user press a button, he can press the button zero or more times

Data Stream - diagram



RXJS Questions

- **How does RXJS help us create data stream?**
- **How do we attach listeners to that data stream?**
- **Why do we need it and what is the difference between it and promises**
- **How can we manipulate the data stream?**

What is RXJS

- ▶ **RXJS is a library that helps us solve the data stream problem using the Observable pattern**
- ▶ **pulses are pushed to the listeners who jump into action**
- ▶ **A data stream is represented by the **Observable** type**
- ▶ **We listen to pulses using Observer**
- ▶ **If the pulse sent are of type string then the data stream is **Observable<string>****
- ▶ **RXJS is currently a stage 0 proposal, it's not officially in ECMAScript versions and we need to install it with NPM**
- ▶ **How do we use RXJS? and what differentiate it from promises?**
let's review it now

Install RXJS

- ▶ First let's start by installing the RXJS library.
- ▶ In the terminal type
 - **> npm install rxjs --save**
- ▶ Let's review how to use the library

RXJS - Create datastream - Interval Observable - EX

- ▶ Unlike promise, observable can have multiple pulses pushed to the listeners
- ▶ To demonstrate this we will create a setInterval timer that will send a pulse every second
- ▶ The pulse type will be a number meaning we will create `Observable<number>`
- ▶ the number will count from zero like a second timer and increment by 1 every second
- ▶ We create an observable using the static `Observable.create` method
- ▶ that method will get our async function (similar to promise)
- ▶ The async function will get an object as an argument
- ▶ That object has a next method to emit a pulse

RXJS - Connect a listener to our data - EX

- ▶ A listener to observable is referred as an Observer
- ▶ Listener to Observable<string> is Observer<string>
- ▶ A listener implements 3 methods
 - next - this method is called when the Observable calls next
 - error - called when the Observable is closed with an error
 - complete - called when an Observable is closed with complete
- ▶ Connect a listener to the interval observable we created before
- ▶ That listener will print the pulse data to the console

RXJS - Connect a listener questions - EX

- ▶ **Will the async code in the Observable run if there is no listener? add a console log in the async code to answer the question**
- ▶ **If we have more than one listener, how many times will the async function run?**

RXJS - Closing the data stream EX

- ▶ **We can close the observable from inside the async function by calling complete or error on the argument of the function**
- ▶ **create an interval observable that will complete after 4 seconds**
- ▶ **Create another interval observable that will error after 2 seconds**
- ▶ **connect listeners to those observables**

RXJS - Closing the data stream - Subscription - EX

- ▶ **We can also close the data stream by closing the observable listener connection.**
- ▶ **As mentioned each listener cause a duplication of the data stream**
- ▶ **When we subscribe to the observable we create a listener to the data stream**
- ▶ **the subscribe returns a Subscription object used to encapsulate one or more listener - observable connection**
- ▶ **from the Subscription you can cancel all the connections**
- ▶ **Connect to the interval observable and cancel the connection after a few seconds from the subscription.**

RXJS - cleanup - EX

- ▶ In the EX before we subscribed to the observables and then closed the listening with unsubscribe / complete /error
- ▶ Notice that even after you closed all listeners the script did not end. Why is that? What keeping the script from completing?
- ▶ Our async function in the observable can optionally return a function referred as `tearDownFunction`
- ▶ That function will be called when the data stream is closing and the goal of this function is to clean stuff from the async function
- ▶ Let's use this function to make sure our script will exit after all the listeners unsubscribed

RXJS - Memory leaks

- ▶ **It's important to remember that when dealing with infinite async events rather it is observable, addEventListener, setInterval etc. We need to always think if our code is not causing memory leaks**
- ▶ **those infinite events we have to unsubscribe from and observables as well if you are subscribing to an infinite observable you will have to unsubscribe to avoid memory leaks**
- ▶ **If you observable is connecting to some infinite source like websockets or events you have to remember to clean them in the tearDownFunction**

RXJS - Subject

- ▶ **What are subjects?**
- ▶ **Why do we need them?**
- ▶ **In what cases should I use a Subject?**

RXJS - Subject

- ▶ **Subjects represents a data stream that arrive from a single external source**
- ▶ **The subject does not wrap async function rather it emits pulse from the outside by creating an instance and calling the next**
- ▶ **this means that the Subject implements the next, error, complete methods of the observer and also the subscribe of the observable**
- ▶ **It also implements Subscription method unsubscribe**
- ▶ **Subject extends Observable**
- ▶ **unlike observables that duplicate with each listener and the source depends on the amount of listeners, subject will be one for many listeners**
- ▶ **Consider something that has a single source for example input event, subject will be more suitable to represent that.**

RXJS - Subject - EX

- ▶ **Create an interval subject that will pulse a number every second**
- ▶ **Connect listeners to that subject**
- ▶ **Close that subject after a few seconds**

RXJS - BehaviorSubject - EX

- ▶ Like the previous Subject only with a few differences:
 - This subject can optionally start with an initial value
 - listeners will jump the first time in a sync way with the last value inside
 - Contain a value property with the last value
- ▶ Good for cases where our listeners should always jump first with a certain value
 - for example: I want to connect to a text input change but the text input might contain an initial value
- ▶ Create the interval subject you created before with BehaviorSubject
- ▶ make sure to start it with a value and note that the listener will jump right away with the initial value

RXJS - ReplaySubject

- ▶ **The ReplaySubject is like the regular subject with a few differences**
 - **It will remember X number of latest emits**
 - **It will remember each pulse for a time window T**
 - **You specify X and T on the ReplaySubject constructor**
- ▶ **Useful when you need a listener to get latest pulses.**
- ▶ **EX: create a ReplaySubject that will save the latest 3 items for a window of 3sec**
- ▶ **that subject will pulse every second**
- ▶ **connect to that subject after 4 seconds. which items will be emitted?**

RXJS - AsyncSubject - EX

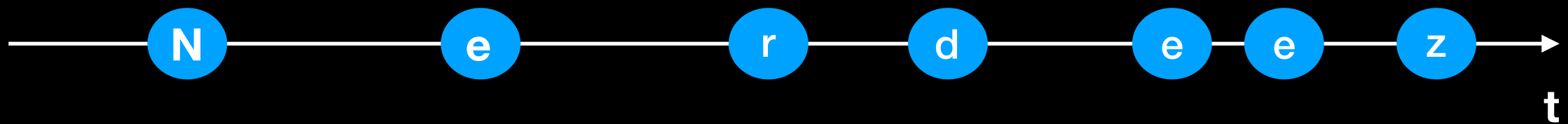
- ▶ Behaves like a subject with the following differences
 - Will only emit the last value
 - Will only emit it after complete
- ▶ Try and create an AsyncSubject that pulse every second a counter and completes after 3 seconds.
- ▶ Attach a listener and make sure his next method only called one time with the last value.

DataStream EX - The autocomplete problem

- ▶ Let's describe a problem that is common in different applications, and let's try to see how rxjs can help us solve this problem
- ▶ Our application consists of a text input where the user types text
- ▶ while he is typing a text we would like to send a request to the server with the text he entered and the server should return a list of available options the user might be aiming for
- ▶ we then display those list of suggestions to the user
- ▶ We can practice solving this problem with our todo rest server located at
 - <https://nztodo.herokuapp.com/api/task/?format=json&search=<search-term>>

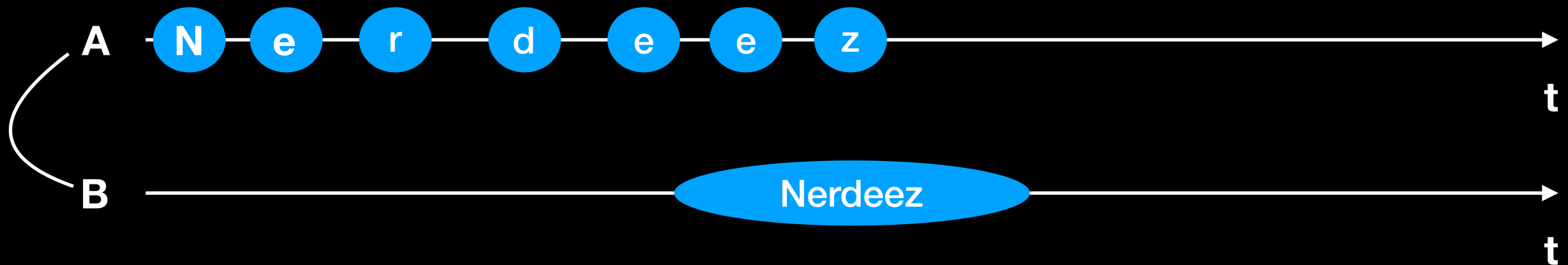
DataStream EX - The autocomplete problem

- ▶ Let's describe the previous problem with a data stream diagram, starting with the first data stream and manipulating it for our result.
- ▶ The user in our diagram types the search term: "Nerdeez"
- ▶ The first data stream will emit a pulse when the user types text in the text field
- ▶ So it might look like so



DataStream EX - The autocomplete problem

- ▶ We can take every pulse from the first data stream and send a request to the server.
- ▶ what would be the problem of doing that?
- ▶ Based on what factor can we filter our data stream and send less requests?
- ▶ So basically we want to transform the previous data stream from A to B
- ▶ Reducing the number of pulses is a filtering transformation



DataStream EX - The autocomplete problem

- ▶ From the user input observable we are taking a full word he types by waiting a period of time with no pulses and we create a new data stream with the full word he typed
- ▶ We can take that word and send the request to the server but let's try and improve it a bit further
- ▶ the user typed Nerdeez we grab the full word and send a request to the server but what happens if after that he start typing again only to return the word back to Nerdeez - in this case we should not sent another request - so we want to transform B to C:



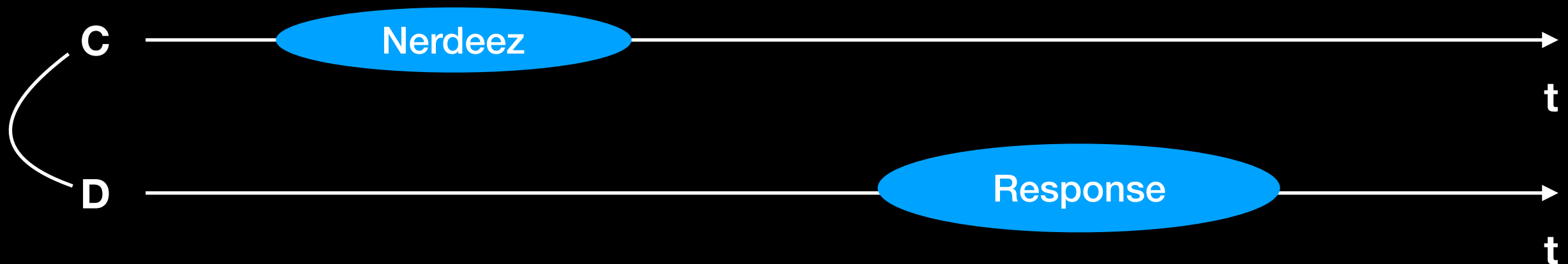
DataStream EX - The autocomplete problem

- ▶ From the user input observable we are taking a full word he types by waiting a period of time with no pulses and we create a new data stream with the full word he typed
- ▶ We can take that word and send the request to the server but let's try and improve it a bit further
- ▶ the user typed Nerdeez we grab the full word and send a request to the server but what happens if after that he start typing again only to return the word back to Nerdeez - in this case we should not sent another request - so we want to do another filtering transformation from B to C:



DataStream EX - The autocomplete problem

- ▶ **C** is a data stream that contains string of the full word the user typed and uniqueness, we need to transform this data stream to a data stream containing request to the server.
- ▶ So **C** will be transformed or mapped to an ajax request:



Operators

- ▶ **Operators are functions that we can use to create, modify observables**
- ▶ **Operators help us solve common data stream problems**
- ▶ **Operators are like toolset that rxjs provide us that gives us more power to solve issues with data stream and observables**
- ▶ **There are a lot of operators and it will be very hard to memorize everyone so let's examine how we can know there is an operator that can help us and how we can find that operator**

Operators - categories

- ▶ **The operators are divided to the following categories**
 - **Creating observables**
 - **Transforming observable**
 - **Filtering observables**
 - **Error handling**
 - **Conditional and Boolean operators**
 - **Mathematical and Aggregate operators**
 - **Backpressure operators**
 - **Connectable observable operators**

Operators - How do I find the operators I need?

- You start by breaking the data stream to the smallest step like we did with the autocomplete problem
- You then look at the transformation between each data stream and decide what category does this match to
- Then you go to reactivex.io and look in the category for the operator most suited for the job
- How do you know the operator exist? if you are creating a transformation that match a category and it's probably a common thing there should be an operator that helps you
- In reactivex.io under the section of the operators document there is a decision tree to help you further if you still can't find your operator

Operators - How to use the operators

- ▶ Now that you found the operator you need to use, how do you actually use that operator
- ▶ Observables have a method called pipe
- ▶ The method will return an Observable
- ▶ The method will get a list of OperatorFunction
- ▶ An operator is a function that return OperatorFunction
- ▶ An OperatorFunction is a function that get's as argument Observable<T> and transforms it to Observable<R>
 - In some cases Observable<T> to Observable<T>
- ▶ So we need to put in the pipe method our list of operator and learn how to use them so they will create the OperatorFunctions we want
- ▶ Each operator we need to view the docs and learn how to use the operator
- ▶ In this lesson we will go over the common operators and when we will use them

Operators - Create observable

- ▶ **First let's cover the common operators that are used to create new observable**
- ▶ **The create observables are not used like the usual operators and are not placed inside the pip.**
- ▶ **They are also not like the usual operators since they do not return OperatorFunction, they simply return the Observable they create**
- ▶ **Those operators are imported from the path 'rxjs'**

Operators - Create observable - of

- ▶ First let's cover the common operators that are used to create new observable
- ▶ The first one we will cover is **of**
- ▶ This operator will create an observable that emits the arguments as the pulses
- ▶ This operator is used to create a final observable with value
- ▶ Very common to use in tests when you need to mock an observable
- ▶ EX: create an observable that emits the pulses: 1,2,3,4,5

Operators - Create observable - throwError

- ▶ **Create an observable that close immediately with an error**
- ▶ **You pass as an argument the instance of the error you want to pass the listeners**
- ▶ **EX: create an error observable with an error message of hello world that will be caught by a listener and printed**

Operators - Create observable - interval

- ▶ emits numbers 0, 1 , 2, 3, ...
- ▶ emits a number every period of time we specify.
- ▶ **EX:** create the interval observable we created in the ex before.

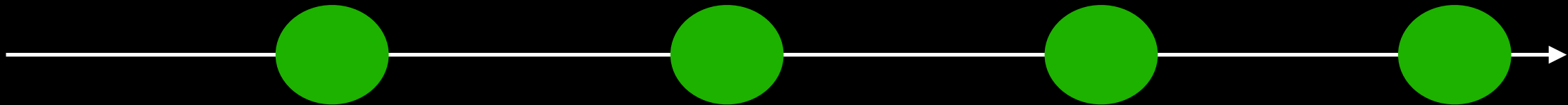
Operators - Create observable - merge

- ▶ merge a list of observables to a single one

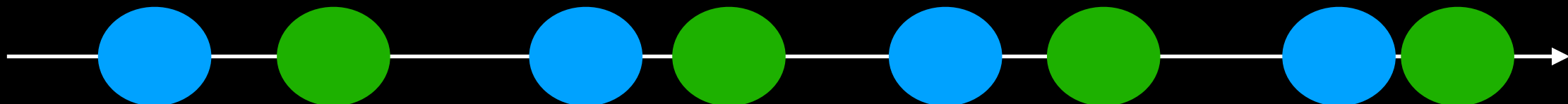
A



B



merge(A, B)



Operators - Create observable - fromEvent

- ▶ Turns an event to data stream
- ▶ Event can be button click or text input, event can also be node EventEmitter and jquery events
- ▶ EX: Let's try and make an EventEmitter to an observable using this operator

Operators - Create observable - from

- ▶ **from** creates an observable from iterable or promise
- ▶ **if iterable it will pulse for every item**
 - **for example for array [1,2,3,4,5] it will have five pulses**
- ▶ **if promise is passed as an argument it will convert it to observable**
- ▶ **EX: try and convert the timer promise we created earlier to an observable**

Operators - Create observable - ajax

- ▶ group of operators for sending a request and getting an Observable that will emit the response
- ▶ This operator is located at 'rxjs/ajax'
- ▶ it is used like this: `ajax.<method>` where method can be:
 - `get(url, headers)`
 - `post(url, body, headers)`
 - `put(url, body, headers)`
 - `patch(url, body, headers)`
 - `delete(url, headers)`
 - `getJSON(url, headers)`
- ▶ or like this: `ajax({...options})`

Operators - Transforming observables

- ▶ Transforming observables are used to change the pulses of an observable from A to B
- ▶ Those operators are applied with the pipe method of Observable
- ▶ it will return a new observable from Observable<A> to Observable
- ▶ Those operators are imported from the path: 'rxjs/operators'
- ▶ Let's go over the most common transforming operators

Operators - Transforming observables - mergeMap

- ▶ **We use this operator where we have an observable the pulses an Observable**
- ▶ **OutObservable is pulsing InnerObservable**
- ▶ **We use this operator if we care about subscribing to the InnerObservable**
- ▶ **EX:**
 - **Mimic a server call with the of operator**
 - **Create an interval observable that will emit the server call observable**
 - **Use the mergeMap to subscribe to the inner value**

Operators - Transforming observables - switchMap

- ▶ **With mergeMap we subscribed to all of the inner observables**
- ▶ **With switchMap we are always subscribing to the last one**
- ▶ **EX: Modify the previous example so we are just interested in the last api call**

Operators - Transforming observables - concatMap

- ▶ **With mergeMap we subscribed to all of the inner observables**
- ▶ **With concatMap the order of the inner observable is important and we will subscribe to the next one only if the previous is finished**

Operators - Filtering observables

- ▶ **The job of these category operators is to reduce the amount of pulses**
- ▶ **These operators are also located at: 'rxjs/operators'**
- ▶ **These operators are also applied in the pipe method of the observable**
- ▶ **They will usually return from Observable<A> to Observable`<A>**
 - **where Observable` will emit less pulses than the original one**

Operators - Filtering observables - filter

- ▶ **The new observable from this operator will emit only if the original observable is emitting and the pulse value is passing a predicate function**
- ▶ **the operator will get a predicate function that will get the value of the pulse and should return a boolean value if this value should be emitted or not**
- ▶ **EX:**
 - **use the interval operator to create an observable that pulse every second**
 - **use the filter operator to make sure only the even numbers are emitted**

Operators - Filtering observables - debounceTime

- ▶ **This operator will create an observable that emits a pulse only if a certain time has passed and no new pulse is emitted**
- ▶ **Perfect for our autocomplete problem**
- ▶ **The operator will get the time in milliseconds to wait until the pulse counts**

Operators - Filtering observables - distinctUntilChanged

- ▶ **this operator will only emit a pulse if that pulse is different from the last pulse**
- ▶ **EX: we can now try and solve autocomplete problem**
 - **Create an event emitter that will pulse n,e,r,d,e,e,z**
 - **add a delay with each pulse**
 - **add a debounceTime to grab the full word**
 - **add a distinctUntilChanged to ignore similar words that the user will print**
 - **from the search word send an ajax request to our todo server**
 - **print the results from the server**

Operators - Filtering observables - take

- ▶ **This operator get's the number of pulses to grab from the original observable before closing**
- ▶ **EX: create an interval observable and take only the first 3 pulses.**

Operators - Dealing with errors

- ▶ **The same rule of being specific with our errors apply to observables as well**
- ▶ **this means that if we have a long data stream transformation we should know exactly based on the Error where the fail was.**
- ▶ **if a throw is place in an operator it will close the observable with an error an alert all error listeners**
- ▶ **If an error was caused and there is no one that listens for the error that an exception will be raised and the script will exit with a non zero value**
 - **unlike promises where currently there is a warning about a promise error not dealt with (this is deprecated and will change in the near future to act like observables)**

Operators - Dealing with errors - catchError

- ▶ When an error happens you have to decide if you want to just pass the dealing of this error to the subscribers in this case you do nothing, or catch the error yourself
- ▶ when catching the error yourself you need to decide if you want to recover from the error, or maybe throw again with a clearer exception
- ▶ catchError will get an error in the observable or operator chain
- ▶ the catchError will be called with a function that will be called with the exception that happened
- ▶ you need to decide if to return observable (usually with the of operator) or to throw again

From promise to Observable and vice versa

- ▶ We saw that we can easily convert a promise to and observable using what operator?
- ▶ We can also convert an Observable to Promise using the method toPromise of the Observable class
- ▶ Few things to note when converting an Observable to Promise
 - Observable the is not completed either by error or complete will result in a promise the is left in the pending state
 - The promise will grab the last pulse before the complete or if there is an error will ignore all the pulses and just be a rejected promise
 - It's best to use the take operator before converting to promise
- ▶ With promises we can use the async await syntax and with observables we can't

Summary

- ▶ **We have quite the toolset for dealing with async js**
- ▶ **First we need to decide if we want to use Promise or Observable**
- ▶ **If we don't require the power of operators and our data stream is made from a single pulse and closes it's best to use Promise**
 - **you can use async await syntax**
 - **you better communicate the data stream structure which other programmers will know and understand if they need to free the memory or not**
- ▶ **Otherwise we will use Observables and choose the right operator based on the steps we want to change the data stream and the operators categories.**