Promises and Observables

Async actions

- Some actions in JS do not return an answer right away and will return the answer after a period of time
  - request data from rest server
  - setTimeout
  - setInterval
  - web workers
- We need a way to subscribe an event that will run when the response is back

What are promises?

- promise is an object which implements an action that returns data in an async way
- subscribers can choose to listen for the promise when the data arrives
- promise is built in in JS since ES6
- promise can also send fail event for the subscribers
- To use promises with TypeScript you have to include the **es6** lib
- promise in typescript:
  - **const myStringPromise: Promise<string> = new Promise((resolve, reject) => {...})**
- Let’s create a simple promise which returns a string data

Promise behaviour

- The constructor for the promise get a function
- the constructor function is called right away with 2 arguments: resolve and reject which are both functions
- we call the resolve when we want to return the data to the listeners
- we call the reject to inform listeners of a fail event
- that function will run right away, no matter if there are any subscribers
- the function will run once no matter if there are multiple subscribers
- the promise is not cancelable

then - promise listener

- we subscribe a listener to a promise by calling the instance method **then**
- the method **then** can optionally get a success method that will run when we call resolve (1st argument) **onsuccess**
- the second argument is a method that will run when we call reject **onreject**
- the **onsuccess** and **onreject** can return a value or a promise with a value
- **then** will return a promise with the data from what the **onsuccess** and **onreject** returned
- if a promise is already resolved when we attach **then** the **then** function will be called after script tasks are finished (similar to setTimeout with 0)

Promise Chaining

- **then** returns a promise with altered data
- common use case:
  - **promise A** requests a rest server for a json data
  - a listener is attached to **promise A** and gets the json data
  - the listener is turning the json data to a class model and return it to make **promise B** with classes instead of json
  - listeners can subscribe to **promise B** and deal with classes and not raw json

Fetch

- with fetch we can send an ajax request to a server
- fetch returns a promise
- fetch gets url as first argument
- second argument is optional options for the request
- let’s try and use fetch to grab all the tasks from the todo rest server
- url: **https://nztodo.herokuapp.com/api/task/?format=json**
- after we grab the json data from the server we will use promise chaining to create another promise with array of task class

Problem with promises

- async tasks repeats often in JS apps
- async tasks are often prone to errors
- difficult to manage exceptions
- data in async tasks need to be customizable and easy to alter and manage

What is RXJS?

- open source library actively developed by Microsoft and open source community
- contains a set of libraries for async and event based actions
- implementation of the observable design pattern

Data Streams

- To understand rxjs you need to understand what a data stream is
- you can look at a data stream as a quiet stream which async will transmit data to whomever is listening
- Let’s try and understand the data stream we have when a user types in the search in youtube

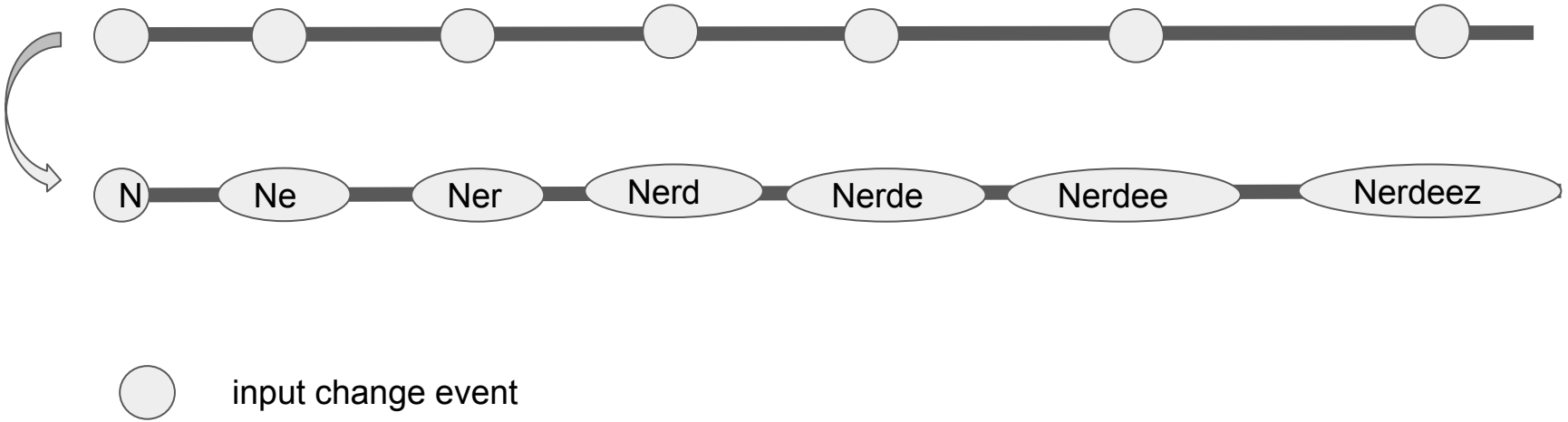
Data Streams - youtube search

User Types the search string: **Nerdeez**

input change event

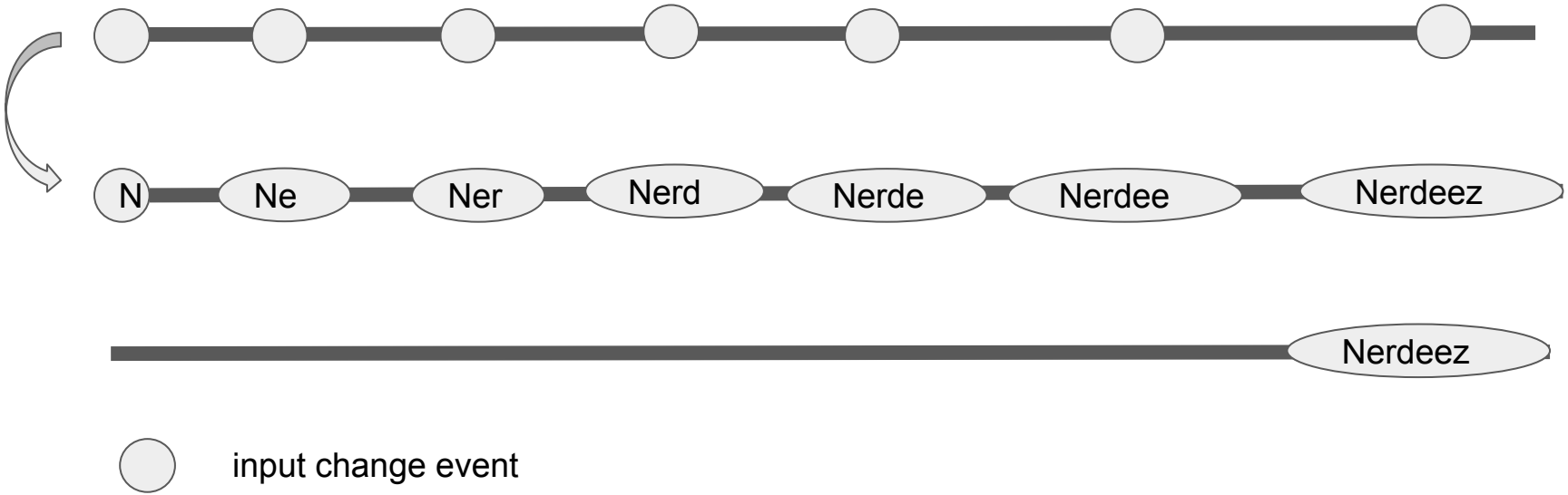
Data Streams - youtube search

- From the event we want to grab the current value of the input
  - **event.target.value**



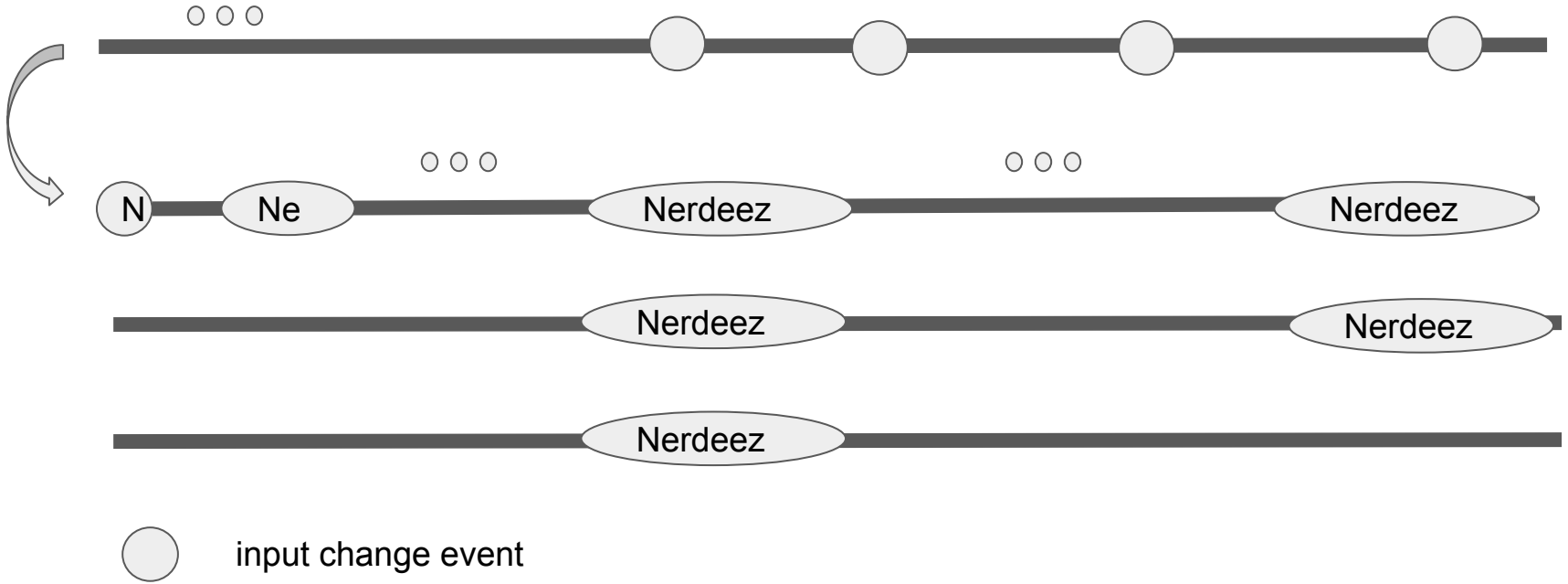
Data Streams - youtube search

- We don’t want a request to be sent everytime a user type text, rather we would like a period of time after he stopped typing to pass.
  - This way for a search term only one request is sent



Data Streams - youtube search

- What if the user types **Nerdeez** than types a lot of other characters but change is mind and delete them to return to the original string
  - we want to send the request only if the value is different then the previous one



Implementation with promises

1. create a promise for every change event
2. chain another promise which will return the value from the event
3. collect all the promises in array and after a period of time send the last one
4. save the last promise and create new promise only if it’s different value from the last

If you try to do it by code it will be very complicated and good chance there will be errors on the implementation

RXJS to the rescue

- RXJS is a library which helps us deal with data streams so complex async event become much simpler.
- RXJS is currently at stage 1 will probably be apart of JS in the future
- We use npm to install it
- in **tsconfig.json** you need to add: **compilerOptions.skipLibCheck**

The main players of the library are:

- Observables - control the data stream
- Observers - control the subscribers to the data stream
- Subject - similar to observables but can connect to multiple listeners

Observers

- an observer is subscribing to a data stream (listener)
- an observer is an object that implements: **next, error, complete**
  - **next -** function which is called when the observable sends a value, the function will have the value in the argument
  - **error -** function which is called when the observable sends an error, the observable can send an argument to the error function
  - **complete -** function which is called when the observable is completed.

Observable

- represents a data stream
- We create it by calling the **Observable.create**
- in TypeScript we can use generic types and define the type of the Observable according to the data it sends
- **Observable.create** will get a function with an observer
- when we want to send data we call the **observer.next**
- there is no limit to the amount **Observable** can call **next**
- if the observable failed he can send **observer.error** once
- for closing the observable we can call the complete
- the function in **Observable.create** can optionally return a function which will be called after we call the complete for cleanup
- The observer won’t activate until there is an observer listening to it
- let’s create an observable which counts every second

Subscription

- when we call subscribe on the observable we are creating a subscription
- a subscription contains **unsubscribe** method
- calling **unsubscribe** won’t call the observer complete
- calling **unsubscribe** will call the **Observable TearDownFunc**
- creating more than one subscription will call the observable create function for each one and basically create a data stream for every subscriber
- Let’s try and create a subscription and try to cancel it after a period of time

Subject

- subject behaviour is more similar to promises
- you have a shouter which can emit multiple values (not like promises)
- you can attach multiple listeners to a subject
- one subject will emit to all listeners (like promise not like observables)
- subject has a behaviour similar to observers and observables
- we can still use operators on a subject
- **BehaviorSubject** starts with an initial value
- We already stumbled in angular2 on a wrapper of angular around subject, Can you remember what the wrapper is? and what did we use it for?

Operators

- operators manipulate the data stream and usually create a new data stream
- there is a lot of operators that cover a lot of common use cases
- since most of the operators return observable it allows you to chain the operators
- Operators can also be used as a function (**OperatorFunction**) using **Observable.pipe**
- **OperatorFunction** get's and observable with type T and returns observable with type R
- The operators are categorized
    - **Creating Observables -** operators that create new observables like: **Range, Interval, Create...**
    - **Transforming Observables -** transform items emitted by observables: **Map, Buffer, GroupBy …**
    - **Filtering Observables -** selectively emits item from source observable: **Debounce, Distinct**
    - **Combining Observables -** from multiple observables create one: **Merge, When**
    - **Error handling -** operator that helps recover from error notification: **Catch, Retry**
    - **Utility -** tollbox of useful operators for working with observables: **Delay, Timeout**
    - **Conditional and Boolean -** evaluate observables or items emitted by observables
    - **Mathematical -** operates on the entire sequence of items emitted by operators

Operators example

- Let’s try and demonstrate operators with a simple example
- we have our timer observable which emits counter value every second
- let’s create another observable which emits a value only if the counter is dividing by 2
- there is so many operators out there, the best way to know what to use is to use the operator decision tree: http://reactivex.io/documentation/operators.html
- we will use the **filter** operator, when using an operator you have to import it

Operators advanced example 1

- let’s try and create the data stream of user typing search string.
- we want a delay to collect all the search string
- we want distinct values
- We will have to include the rxjs bundle in our html
- the operators that we will use
  - **fromEvent**
  - **debounceTime**
  - **distinctUntilChanged**

Operators advanced example 2

- let’s try and continue the previous example
- now when you get a search string you need to query our rest todo server according to that string
  - **https://nztodo.herokuapp.com/api/task/?search=<search-string>**
- we need to transform what we get from the server to class representing tasks
- operators we will use:
  - **flatMap**
  - **ajax**

Operators tips

- don’t try and remember by heart all the operators
- try to convert your async actions to data stream
- think about how you would like to get that data stream and how you need to transform it
- use the operators action tree to understand which operators can assist you in transforming your data tree

Student EX.

a list of the common operators you will use:

- **combineLatest, merge, concat, from, of, throw, catchError, filter, debounceTime, distinctUntilChanged, map, mergeMap, pluck, tap, delay, finalize,**
- for each one of those operators try to use them and see how it transforms you data stream
- you can use those operators on **Observable** or **Subject**

Summary

- the first step to using rxjs is transforming the way you think about async actions
- we need to transform async actions to data streams
- The strength of the library is in the many operators it has which allows us to manipulate the data streams
- ReactiveX programming is available in must of the popular languages like:
  - .NET
  - python
  - Java
- so it’s a useful library to almost every project