Workshop

Angular Basics

Angular

A platform for building mobile and desktop web applications

Why Angular - In general

- Builds on experiences with AngularJS
- Focus on maintainability
- Prevents developers from doing the wrong thing
- Tries staying close to web standards / dom
- Common solutions for common problems
  - i18n
  - Animations

Why Angular - Speed

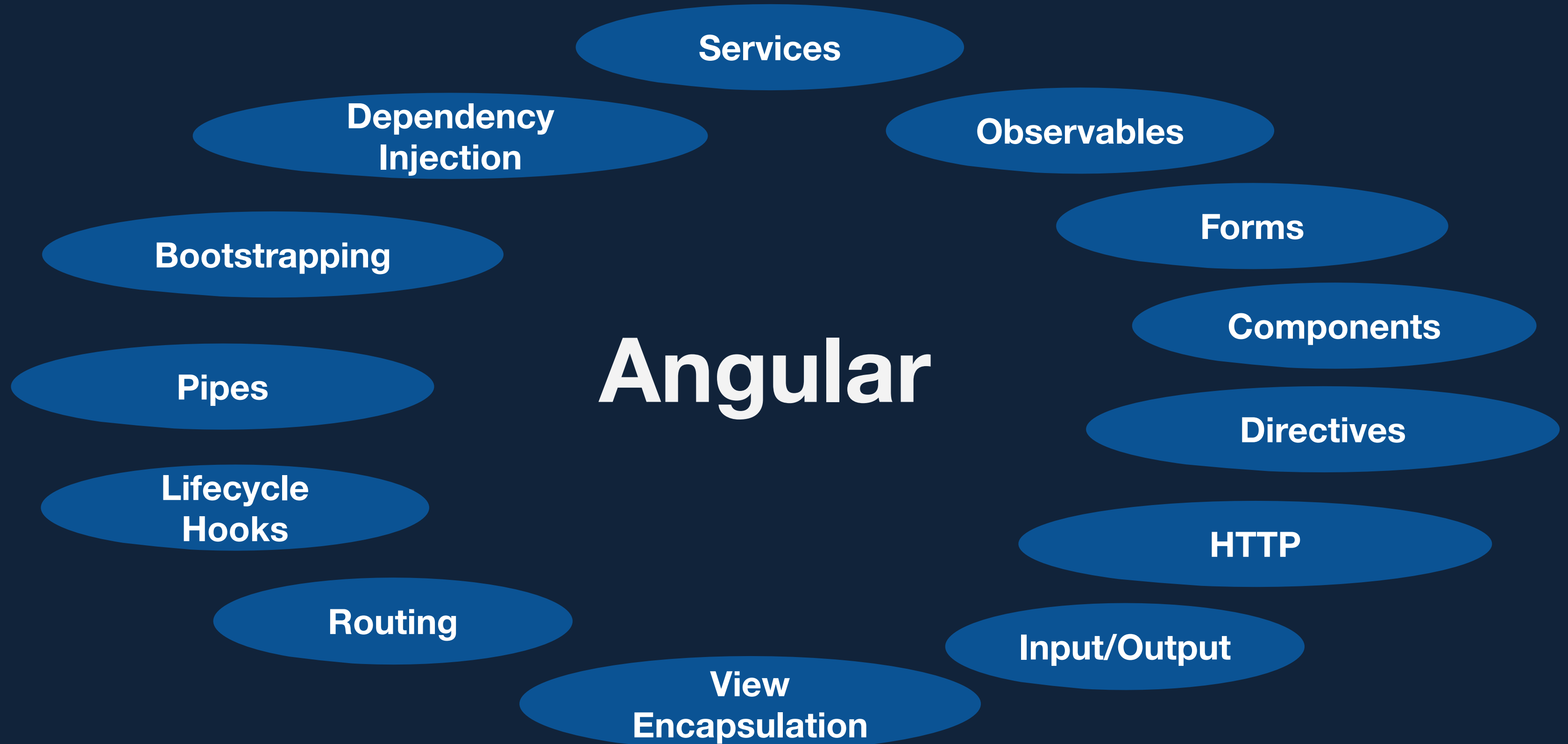
- Pre-rendering (Server Side Rendering)
- Offline compile (Ahead-of-Time compiling)
- View caching
- Web worker

Why Angular - Cross platform

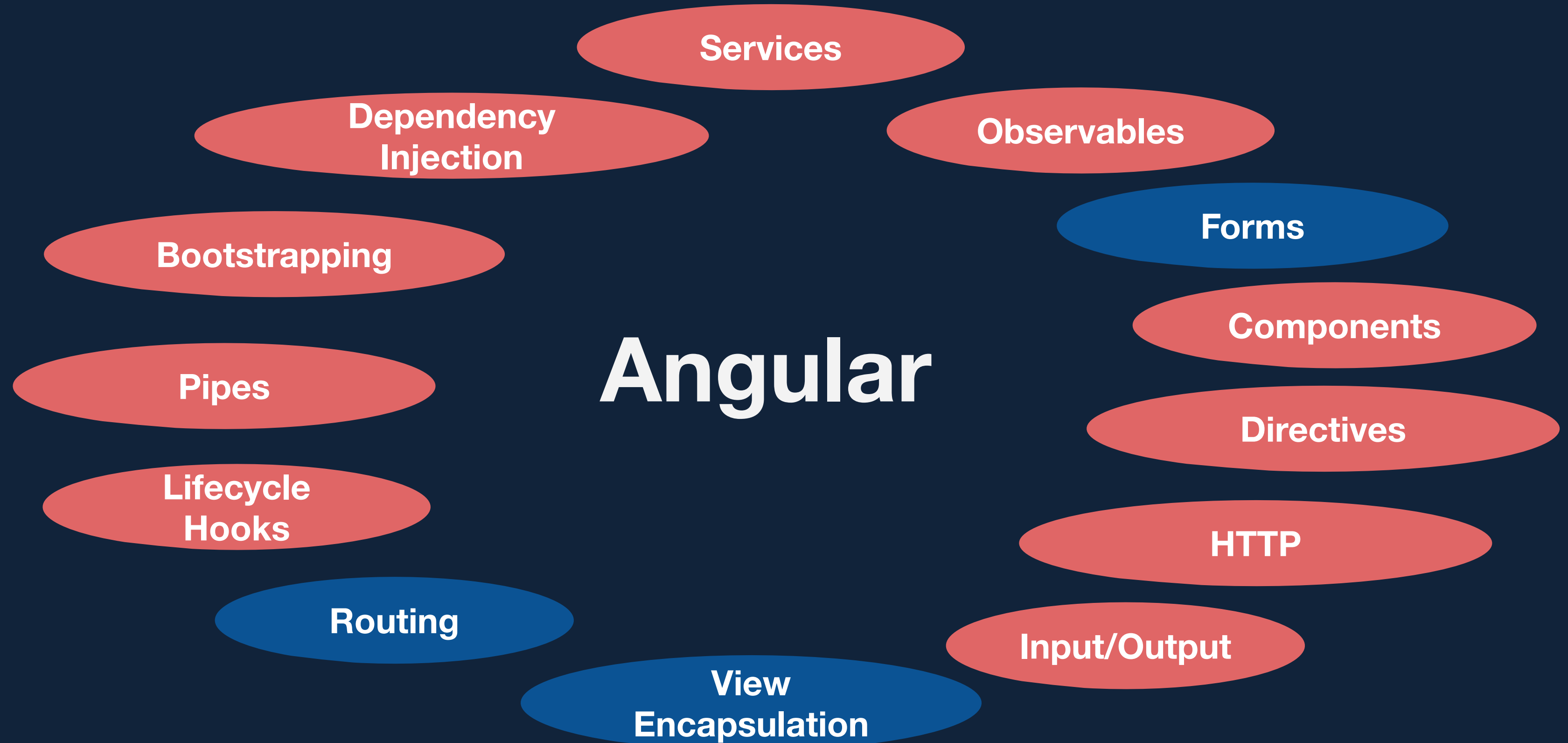
Angular works...

- in the browser
- on the server
- in a mobile web container (Ionic)
- mobile native (NativeScript)

Angular in a Nutshell



Angular in a Nutshell



workshops.de

Angular CLI

Angular CLI

- No more seeds and fragmentation
- No more discussions about style
- Proven directory structure

Based on

- ember-cli
- webpack

Angular CLI - Generator

Type	Usage
Component	ng g component book-list
Directive	ng g directive tooltip
Service	ng g service book-data
Pipe	ng g pipe shout
Interface	ng g interface book
Class	ng g class book

angular-cli

Features

- development web-server
- build process
- testing
- update
- add functionality

```
src
\- main.ts
\- app
   \- book
      \- book.component.css
      \- book.component.html
      \- book.component.ts
      \- book.service.ts
   \- app.module.ts
   \- app.component.ts
   \- index.html
```

```
$ ng --help
```

Validate your CLI Version

- Type `ng --version` in your command line
- Update version to the version your trainer recommends :)

Task

Preparation & Create new Project

@NgModule

The decorator

@NgModule - General

- Groups code and files
- Solves a specific problem/deal with a specific topic
- Shares functionalities between Angular modules

@NgModule - Decorator

- Defines the parts of the module, e.g. components, directives, ...
- Import dependencies
- Export parts to other modules
- Set base component

@NgModule Decorator - Overview

<code>

module decorator

```
@NgModule({
  declarations: [ // pipes, components/directives known in the whole module
    AppComponent,
    BookListComponent // is now known in the whole module
  ],
  imports: [ // depends on other modules
    BrowserModule // imports and re-exports most basic Angular directives
  ],
  providers: [], // list of services
  bootstrap: [AppComponent] // there is one root component
})
export class AppModule {} // in most cases an empty class
```

The bootstrap function

Bootstrap

Your application needs a starting point!

- The `bootstrap` function defines the main module

Bootstrap

Every module can have a bootstrap component!

- e.g. `bootstrap: [AppComponent]`

Bootstrap

<code>

An example bootstrap

```
// app.module.ts
@NgModule({
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})

// main.ts
import { platformBrowserDynamic } from
'@angular/platform-browser-dynamic';
import { AppModule } from './app/';

platformBrowserDynamic().bootstrapModule(AppModule);
```

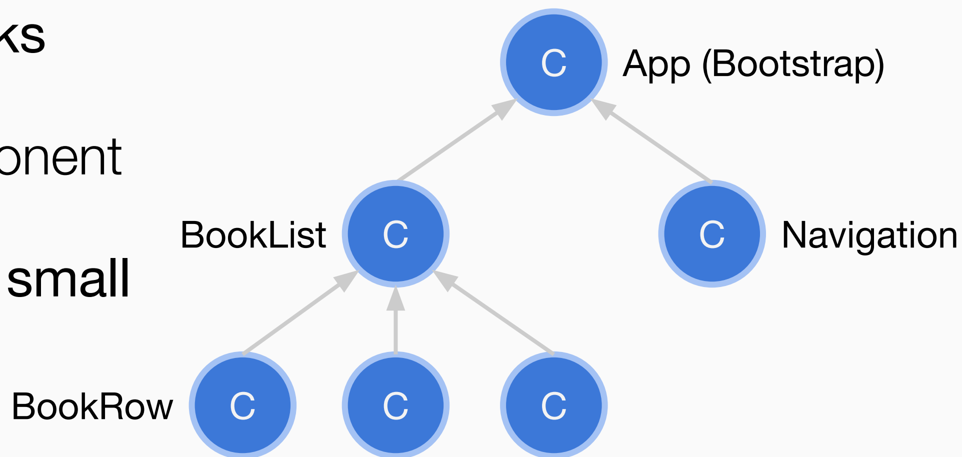
Task

Generate two Submodules

Components

Components

- fundamental **building blocks**
- application itself is a component
- break your application into **small components**



@Component, View, Component Class

```
@Component({
  selector: 'book-list',
  template: `
    <h1>
      Book-Title: {{title}}
    </h1>
  `
})
class BookListComponent {
  title: string;
  constructor() {
    this.title = 'An awesome book';
  }
}
```

@Component Decorator

Component Class

@Component

Decorator

Component Decorator

Component metadata / configuration

Component Decorator - Overview

<code>

Component decorator for a component class

```
@Component({
  selector: 'book-detail',
  templateUrl: './book-detail.html'
})
export class BookDetailComponent {}
```

Component Decorator - Selector

- `element-name`: select by element name. (preferred way)
- `.class`: select by class name.
- `[attribute]`: select by attribute name.
- `[attribute=value]`: select by attribute name and value.
- `:not(sub_selector)`:
  - select only if the element does not match the *sub_selector*.
- `selector1, selector2`:
  - select if either *selector1* or *selector2* matches.

Template Bindings

- `{{ expression }}` Syntax (curly braces)
- Display data inside of the component view
- Possible to execute simple calculations

  `{{ 1 + 2 + book.price }}`
- Function calls

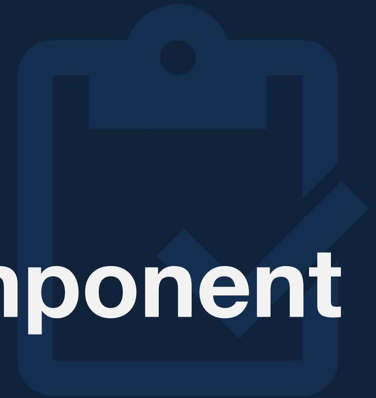
  `{{ showPrice(book) }}`
- Limited set of JavaScript → Not a simple eval('...')

Component Class

- Defines data and behavior of your component
- Possible to inject services or other injectables
- Consists of
  - Methods
  - Objects
  - Arrays
  - Primitive data types (number, boolean, string, etc.)

Task

Generate a Navigation Component



DOM

Property- & Event-Bindings

DOM

- A **DOM node** is an **object**.
- It can store custom **properties** and **methods** like any other object.

DOM

<code>

Example for properties

```
document.body.style = {
  backgroundColor: 'red'
};
alert(document.body.style.backgroundColor); // => red
```

DOM

<code>

Example for events

```
function showCurrentTime() {
  console.log(new Date());
}

document.body.onclick = showCurrentTime;
```

Template Syntax

Event + Property Syntax

- Allows binding to the native DOM **properties** and **events**
- Allows interoperability with other frameworks

Properties

Property-Binding Syntax

- Pass data to the native component object in the DOM
- Defines an attribute binding on the element

Properties - Example 1

<code>

Set the background style of an element.

```
<h2 [style.backgroundColor]="color">Title</h2>
// color is a variable

<h2 [style.backgroundColor]="'red'">Title</h2>
// color is a string
```

Properties - Example 2

<code>

Set the href property of the link

```
<a [href]="book.url">
  {{ book.title }}
</a>
```

Events

Event-Binding Syntax

```
<a (click)="close()"></a>
```

- Used with (event name)
- Defines an event listener on an element
- Listens to native DOM events

Event-Binding Syntax Example

- Executes a function that is defined on the component class
- Executes an expression

```
<button (mouseover)="someFnOnClass()">Execute</button>
<button (click)="isHidden = false">Show</button>
```

Event-Binding Syntax Events

- It's possible to access the event via $event
- $event is the actual native DOM-Event

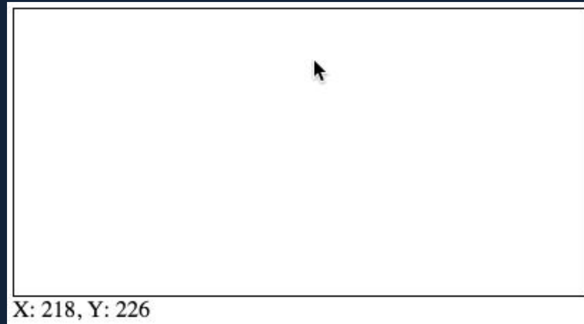
```
<input [value]="name" (input)="handleEvent($event)">
```

Task

Create an Info-Box

Task

Output mouse cursor position



Inputs & Outputs

Inputs & Outputs

- Components are isolated
- Establish communication between components

Inputs

Input-Metadata

@Input decorator declares a data-bound input property

```
export class TitleBoxComponent {
  @Input() headerTitle: string;
}
```

Usage in template

```
<title-box headerTitle="Example Header String As Static String">
<title-box [headerTitle]="localVariableOnComponent">
```

Task

Create a title @Input

Outputs

Output-Metadata

@Output decorator declares an output property.

```
export class TitleBoxComponent {
  @Output() ping;
}
```

Usage in template

```
<title-box (ping)="...">
```

Output-Metadata

<code>

usage of outputs combined with events

```
@Component({})
export class TitleBoxComponent {
  @Output() ping = new EventEmitter<string>();

  sendPing() {
    this.ping.emit('Msg');
  }
}
```

declare ping as EventEmitter

emit an Event

Output - Generics

- `new EventEmitter<string>()` creates an EventEmitter
- The emitted value has to be a string: `this.ping.emit('Msg');`
- **$event** contains the emitted event data → it has not to be the event itself!

Task

Create a (titleClicked) @Output

Structure Syntax

Structure Syntax - * and <ng-template>

```
<div *ngIf="book">
  <span>{{book.title}}</span>
</div>
```

- Structural directives begin with an asterisk (*)
- Syntactic sugar → easier to read/write
- Short form for template elements

Structure Syntax - * and <ng-template>

→ Use *ngFor to iterate over an array of items

```
<div *ngFor="let book of books">
   {{book.title}}
</div>
```

Task

Use *ngFor

Interfaces

Interfaces

- Type-checking of the shape of values
- Interfaces give a type to these shapes

Interfaces - Without an interface

<code>

You can generate interfaces on the fly.

```
const book: { isbn: string, title: string };

book = {
  isbn: '978-1593272821',
  title: 'Eloquent JavaScript'
};
```

Interfaces - With an interface

<code>

Give an interface a name and use it as a type for variables.

```
interface Book {
  isbn: string;
  title: string;
}

const book: Book;

book = {
  isbn: '978-1593272821',
  title: 'Eloquent JavaScript'
};
```

Interfaces - Optional properties

<code>

Properties can be optional.

```
interface Book {
    isbn: string;
    title: string;
    pages?: number;
}
```

Interfaces - Class types

<code>

Forgetting to implement ngOnInit throws a compile error.

```
interface OnInit {
  ngOnInit();
}

class BookListComponent implements OnInit {
  ngOnInit() {
  }
}
```

Services

Services

- “Local Singletons”
- Data-Model-Layer of our application
- May be injected via Dependency Injection (DI)
- Two roles:
  - Provide methods or streams of data to subscribe to
  - Provide operations to modify data

Services - Example

<code>

Services are the Data-Model-Layer of our application

```
@Injectable({
  providedIn: 'root',
})
export class BookDataService {
  private books = [{...}, {...}, {...}];

  getBooks() {
    return this.books;
  }
}
```

Services - Example

<code>

With `providedIn:'root'` the service is registered globally

```
@Injectable({
  providedIn: 'root',
})
export class BookDataService {
  private books = [{...}, {...}, {...}];

  getBooks() {
    return this.books;
  }
}
```

Services - Example

<code>

They define an API to interact with them

```
@Injectable({
  providedIn: 'root',
})
export class BookDataService {
  private books = [{...}, {...}, {...}];

  getBooks() {
    return this.books;
  }
}
```

Services

<code>

Create a service explicit for a module with the providers array

```
@NgModule({
  providers: [
    BookDataService
  ]
})

@Component({ … })
export BookListComponent {
  constructor(private bookData: BookDataService) {}
}
```

Services

<code>

Create a service instance for a component and its children

```
@Component({
  // ...
  providers: [BookDataService]
})
export class BookListComponent {
  constructor(private bookData: BookDataService) {}
}
```

Services

<code>

Create a service instance for a component and its children

```
@Component({
  // ...
})
export class BookListComponent {
  constructor(private bookData: BookDataService) {}
}
```

Dependency Injection

Dependency Injection - Why

- Keep component classes clean
- Better testable code
- Easy replacement of services

Without Dependency Injection

Dependency Injection

<code>

You have to create instances on your own.

```
@Component({ … })
class BookListComponent {
  private bookDataService;
  constructor() {
    this.bookDataService = new BookDataService();
  }
}
```

With Dependency Injection

Dependency Injection

Dependency Injection is also called **Inversion of control**.

The injector has control over service instantiation.

Dependency Injection

<code>

Injector is responsible for creating instances.

```
@NgModule({
  providers: [BookDataService],
})
export class AppModule { }


@Component({})
class BookListComponent {
  constructor(private bookDataService: BookDataService) {}
}
```

Dependency Injection

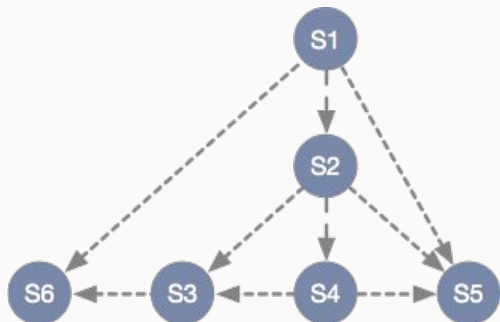
<code>

Injector is responsible for creating instances.

```
@Component({
  providers: [BookDataService]
})
class BookListComponent {
  constructor(private bookDataService: BookDataService) {}
}
```

Dependency Injection

- Services can have dependencies, too
- Injects service instances created in a component, where service is used!
- Watch out for dependency cycles!



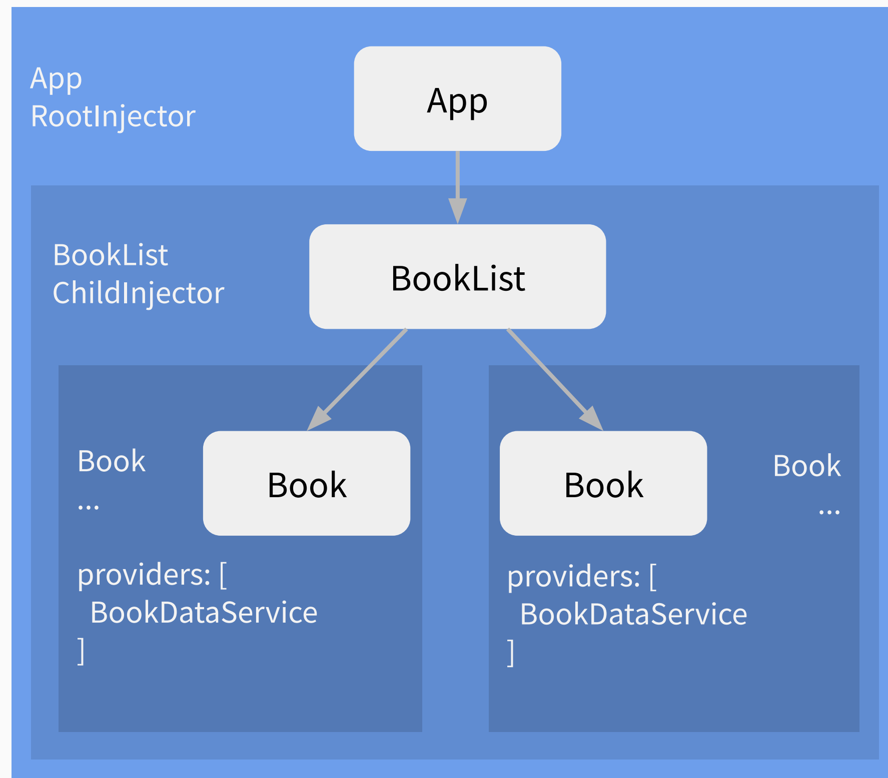

```
service 'S1', (S2, S5, S6)
service 'S2', (S3, S4, S5)
service 'S3', (S6)
service 'S4', (S3, S5)
service 'S5', ()
service 'S6', ()
```

Dependency Injection

- Based on the type of a class
- An instance is available for all child components, too

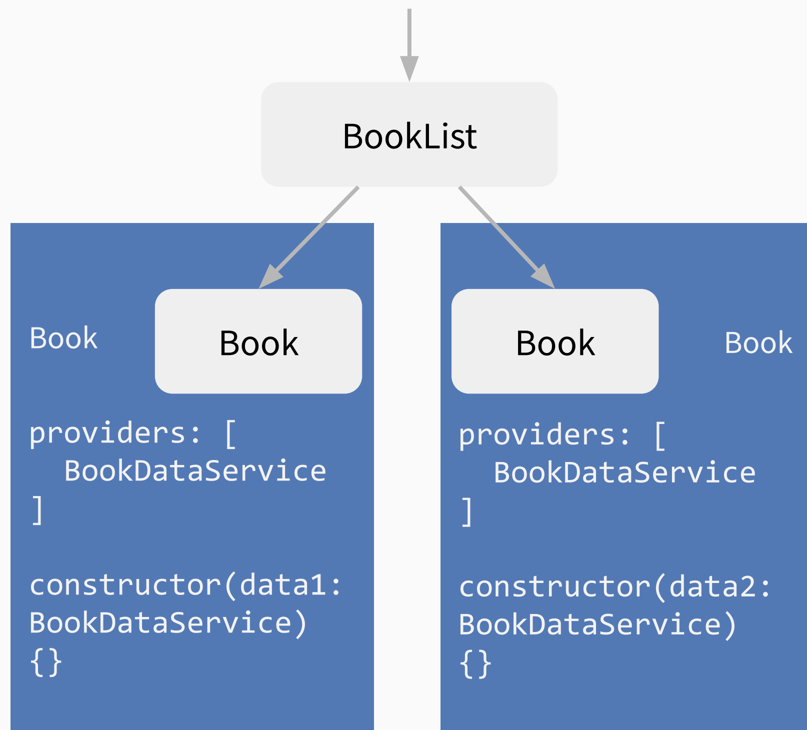
Dependency Injection

- **Injector** per component
- Each component has an own injector
- Base injector = **RootInjector**
- Each nested component has a **ChildInjector**



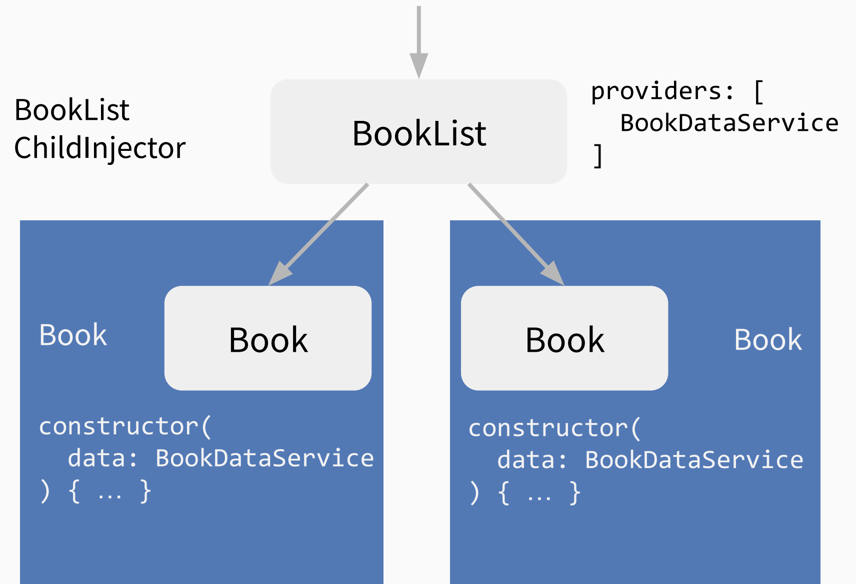
Dependency Injection

- New service instance for each BookComponent



Dependency Injection

- Share one service instance
- Create instance in parent component BookList
- Only inject service in BookComponent → no providers!



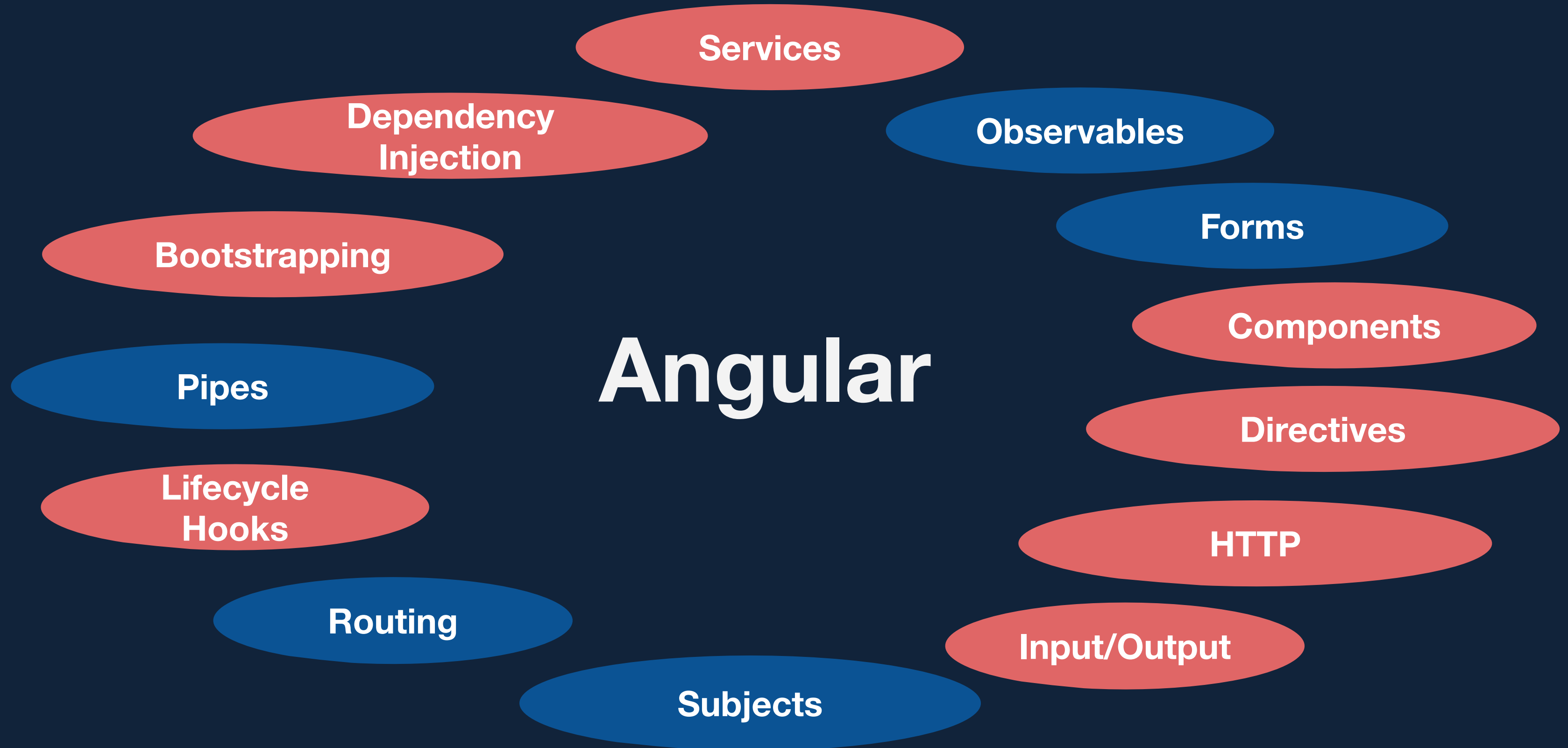
Dependency Injection - @Injectable()

- Annotation of classes that use DI
- Metadata to compile the type-information to the ES5 code
- Without an annotation you lose the type information

Task

Create a BookData service

Angular in a Nutshell



Observables

Why we're talking about it?

Angular is using **RxJS** Observables for **async.**

What is RxJS

- seeing events as collections you can
  - map
  - filter
  - …

Promises vs. Observables

Observables are built

to solve problems around **async.**

(avoid “callback hell”)

Observables

- streams
- any number of things
- Lazy → Only generate values when subscribed to (**cold**)
- can be “unsubscribed” → can be canceled

Observables - subscribing

Without subscribing, an Observable will not emit data

```
.subscribe(nextFn, errorFn, completeFn)
```

Observables Cold vs. Hot

Cold	Hot
• Default • Point to point • Sender per consumer • Sender only starts after subscribe(…)	• Multicast • Sender starts without subscriptions

Operator

- Operators are functions
- allow complex asynchronous code to be easily composed in a declarative manner.

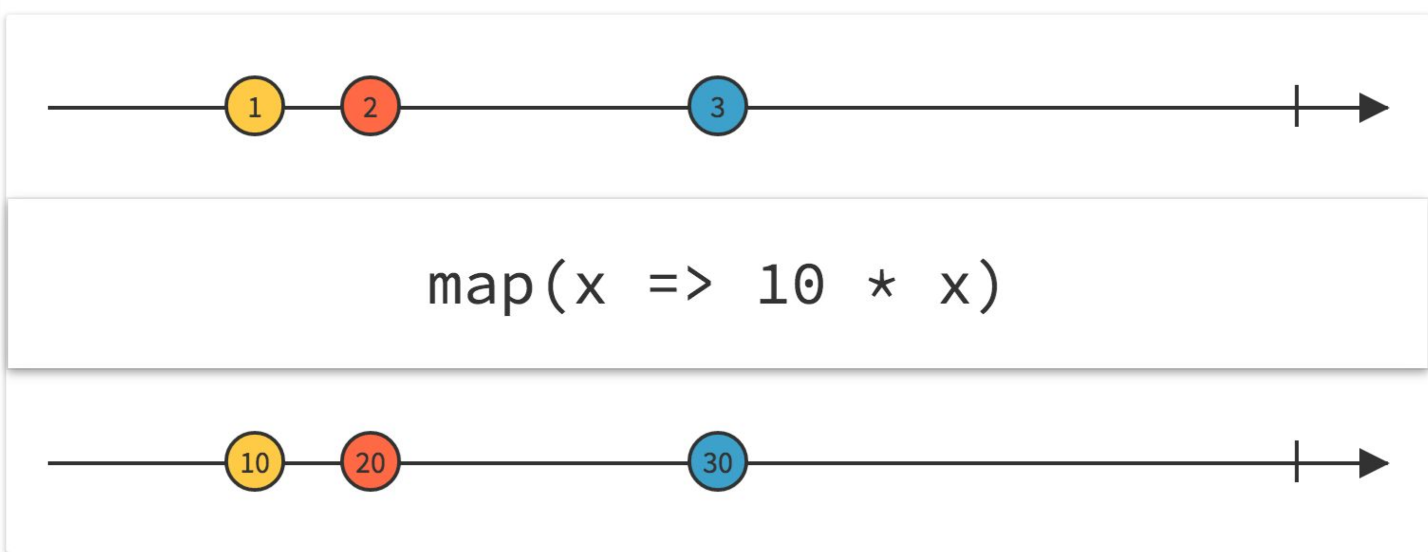
```
observableInstance.pipe(operator1(), operator2(), ...).
```

Observables - Generics

- Functions should return typed data
- Extend type informations of observables with generics
- E.g. `getBooks(): Observable<Book[]> {}`

Transformation Operators

.map()



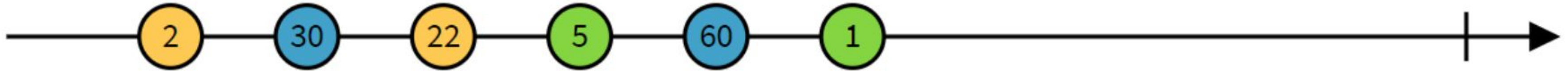
.pluck()

pluck("a")

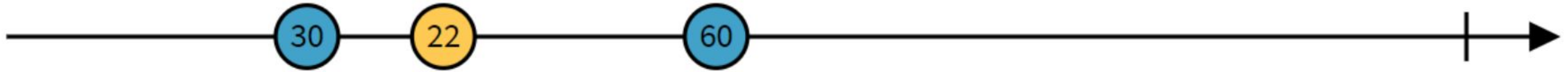
.pairwise()

Filtering Operators

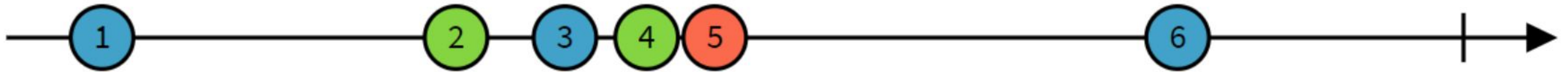
.filter()



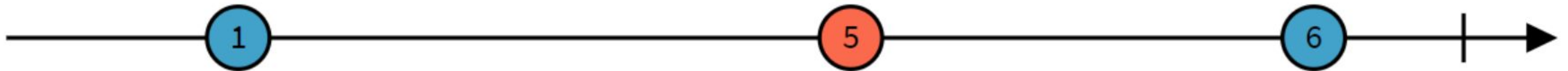

```
filter(x => x > 10)
```



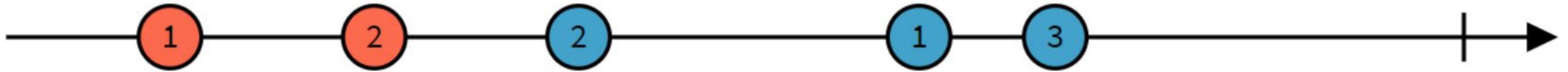
.debounceTime(n: milliseconds)



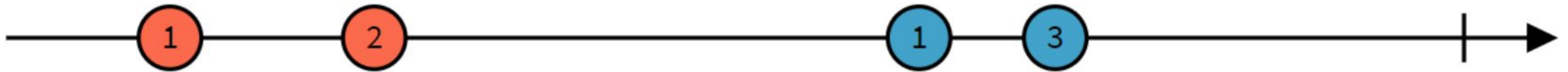
debounceTime(10)



distinctUntilChanged()

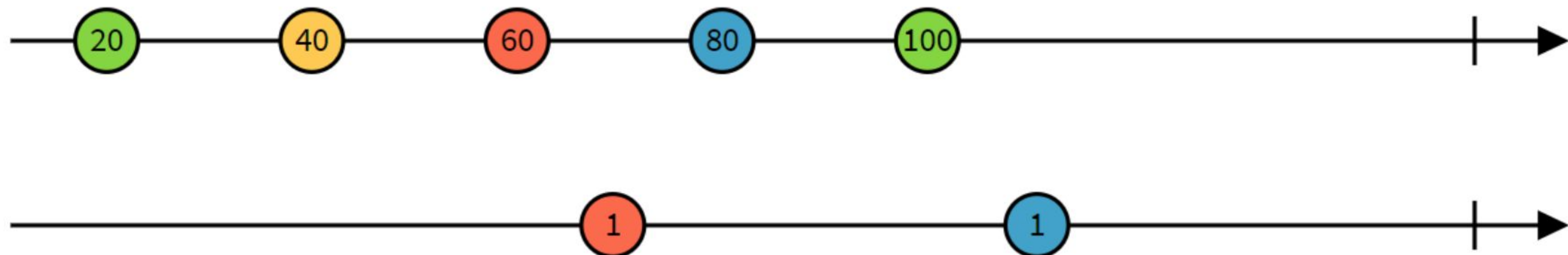


distinctUntilChanged



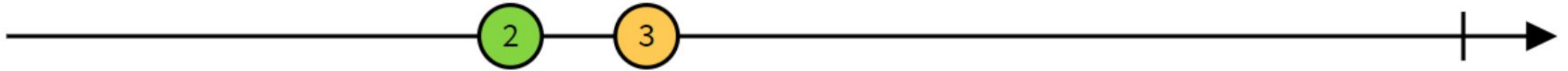
Combination Operators

merge()

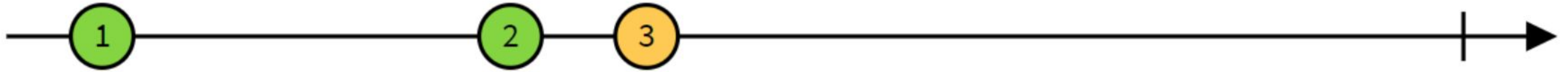


merge

startWith()



startWith(1)



Error Handling

Operators for Error Handling

- catchError
- retry
- retryWhen
- throwError

Observable Creation Operators

You are usually not creating

your own observables!

Observables creation helpers

- of(value1, value2, …)
- from(promise/iterable/observable)
- fromEvent(item, eventName)
- Angular HttpClient
- Many more

Task

Create an Observable

Observables - cancellation

```
const subscription = observable.subscribe(...)

subscription.unsubscribe()
```

Need to unsubscribe!

<code>

```
BookComponent implements OnInit, OnDestroy {
    private subscription: Subscription;
    ngOnInit() {
        this.subscription = this.bookData
         .getBooks()
         .subscribe(books => this.books = books);
    }

    ngOnDestroy() {
        this.subscription?.unsubscribe()
    }
}
```

HttpClient

Using the HttpClient

- Basic HTTP handling
- `import {HttpClientModule} from '@angular/common/http'`
- `import {HttpClient} from '@angular/common/http'`
- Provides methods for
  - GET
  - PUT
  - POST
  - DELETE

Http service

<code>

HttpClientModule has to be imported

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  ...
})
```

HttpClient Interface

Name	Parameter	Returnvalue
get	url, options?	Observable<TPayload>
post	url, body, options?	Observable<TPayload>
put	url, body, options?	Observable<TPayload>
delete	url, options?	Observable<TPayload>
patch	url, body, options?	Observable<TPayload>
head	url, options?	Observable<TPayload>
request	Request, options?	Observable<TPayload>

HttpClient usage

<code>

HttpClient functions return response observables

```
import { HttpClient } from '@angular/common/http';
...
  constructor(private http: HttpClient) {}

  getBooks() {
    return this.http.get<Book[]>(this.baseUrl)
  }
...
```

HttpClient

- Returns an observable
- Expects data in JSON format

HttpClient - Full response

<code>

Use observe: 'response' to get the full response

```
http
  .get<Book[]>('/books', {observe: 'response'})
  .subscribe(resp => {
    console.log(resp.headers.get('X-Custom-Header'));
    console.log(resp.body);
  });
```

HttpClient service

<code>

Subscribe to service observable in a component

```
constructor(private bookData: BookDataService) {
  this.bookData
      .getBooks()
      .subscribe(books => this.books = books);
}
```

Task

Load data from local API

Component Lifecycle Hooks

Component Lifecycle Hooks

- Components and Directives have a lifecycle managed by Angular
- Visibility of key moments and way to act when they occur
- Classes can implement one or more interfaces with hooks

Component Lifecycle Hooks - Interfaces

```
import {Component, OnInit} from '@angular/core'

@Component({...})
class BookListComponent implements OnInit {
  ngOnInit(): void {
    console.log('onInit');
  }
}
```

Lifecycle interfaces are optional but recommended

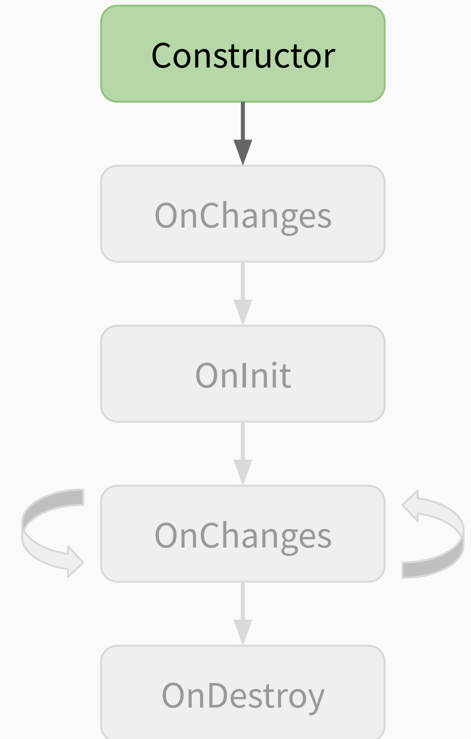
Each Lifecycle hook has an interface without the leading ng

Most important hooks

Component Lifecycle Hooks - Execution

→ Injector instantiates component with new

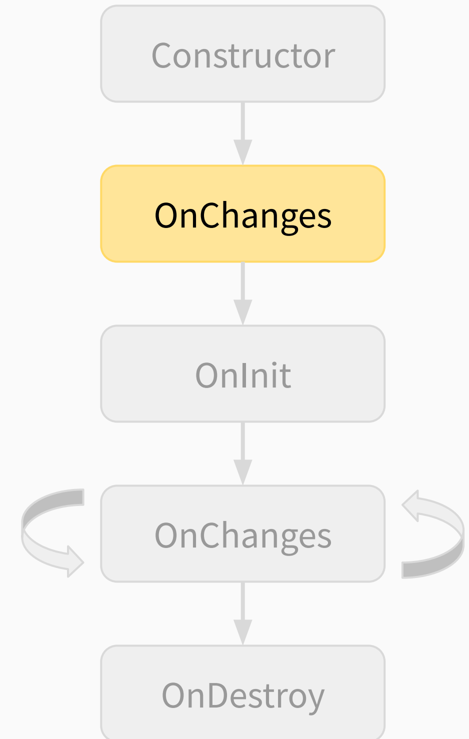
```
@Component({
  selector: 'my-component',
  …
})
class MyComponent {
  constructor () {}
  …
}
```



Component Lifecycle Hooks - Execution

- Check for initial values on @Inputs

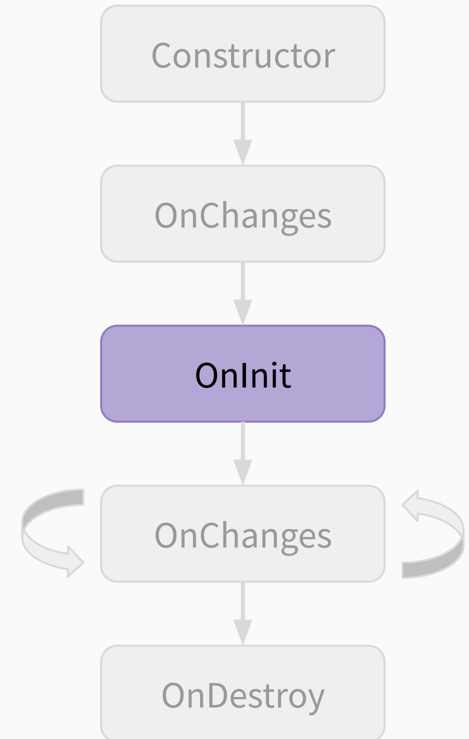
```
<my-component [anInput]="value">
</my-component>
...
class MyComponent {
  @Input() anInput;
  ...
}
```



Component Lifecycle Hooks - Execution

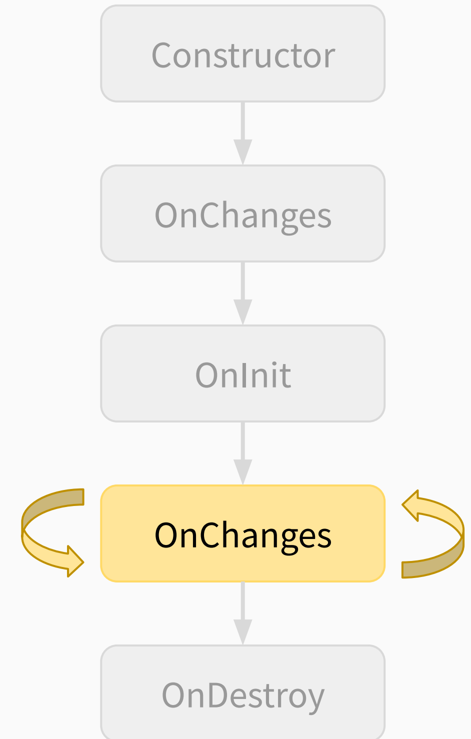
- Initial values are set → called only once
- For heavy or async work

```
class MyComponent implements OnInit {
  @Input() anInput;

  constructor() { // this.anInput = undefined }

  ngOnInit() { // this.anInput is set }
}
```



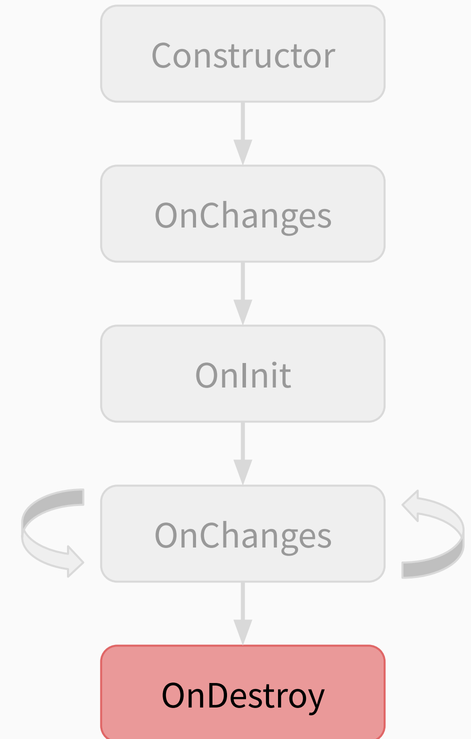
Component Lifecycle Hooks - Execution

→ Every time an Input binding changes



Component Lifecycle Hooks - Execution

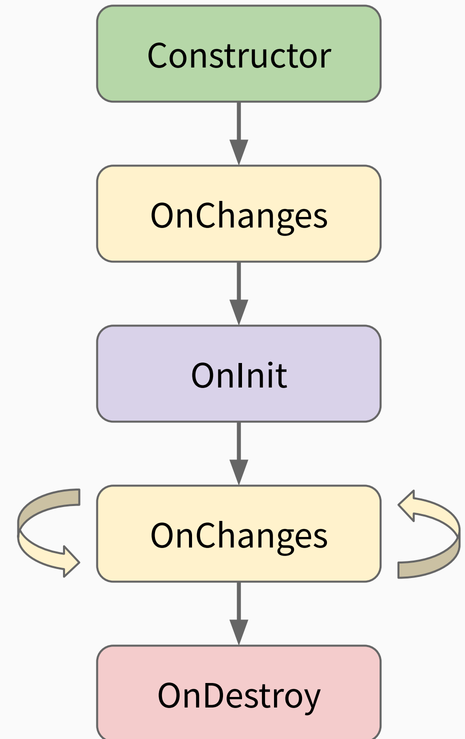
- Cleanup before component is removed
  - Remove event listeners
  - Remove observable subscribers
  - Clean up intervals and timeout
  - Inform other program parts
- Called only once



Component Lifecycle Hooks - Execution

Simplified

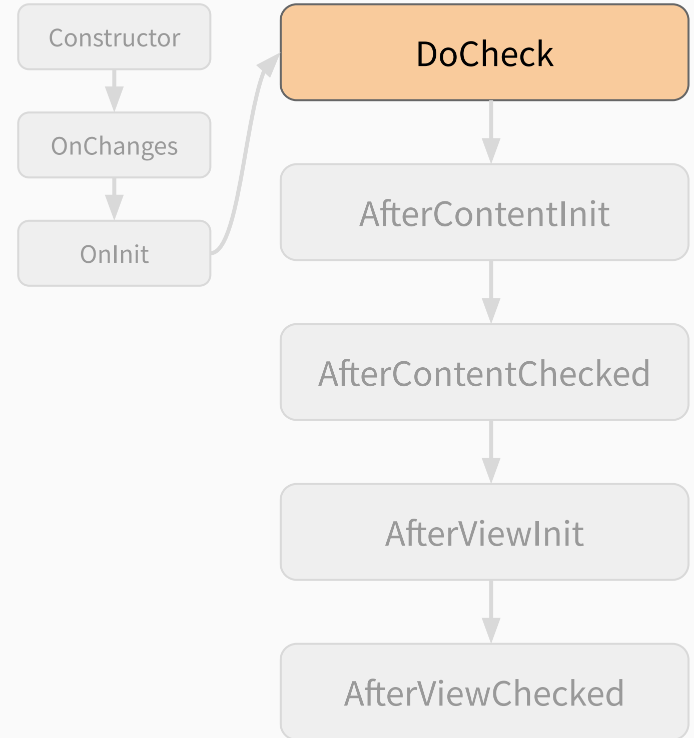
1. Component is instantiated
2. OnChanges: initial @Input values
3. OnInit: once after first OnChanges
4. OnChanges: get changed @Input
5. OnDestroy: component is destroyed



After component creation

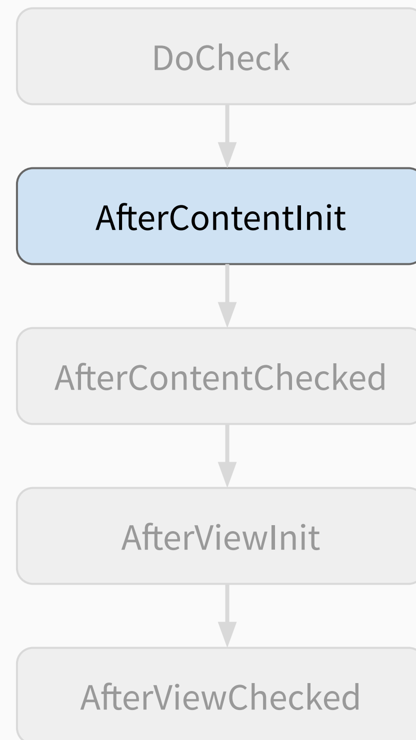
Component Lifecycle Hooks - Execution

- Every time change detection runs
- Custom change detection function



Component Lifecycle Hooks - Execution

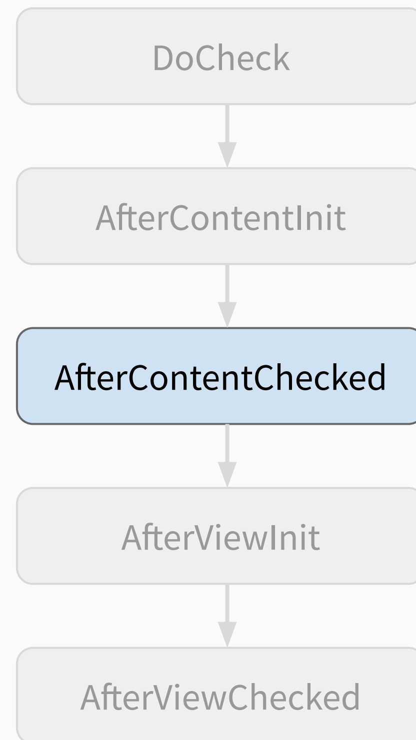
- Content = everything between component tags
- *ngContent* projects content to view after creation
- Hook called after projection finished → only once




```
@Component({
  selector: 'my-component',
  template: '...<ng-content></ng-content>...'
})
...
<my-component><p>Hello</p></my-component>
```

Component Lifecycle Hooks - Execution

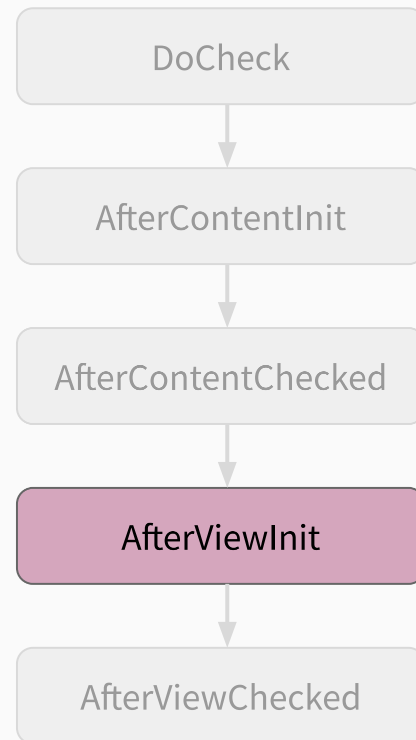
- After change detection → content is checked
- Called every time after DoCheck hook
- Initial check after content is initialised



Component Lifecycle Hooks - Execution

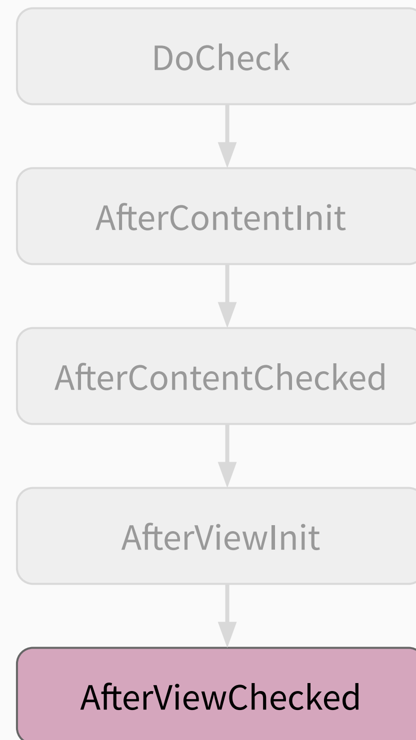
- View = template + bindings
- Called after view and subviews are initialised → only once

```
@Component({
  selector: 'my-component',
  template: `
    <h1>Hello</h1>
    <another-component></another-component>
  `
})
```



Component Lifecycle Hooks - Execution

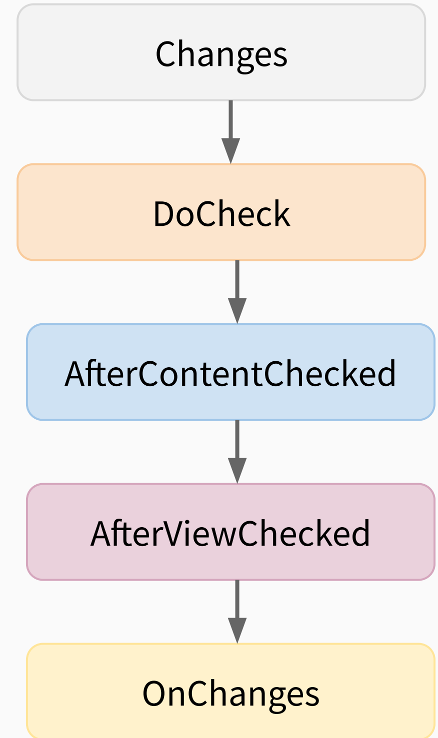
- After change detection → view is checked
- Called every time after DoCheck hook and after AfterContentChecked
- Initial call after the view is initialised



After change detection

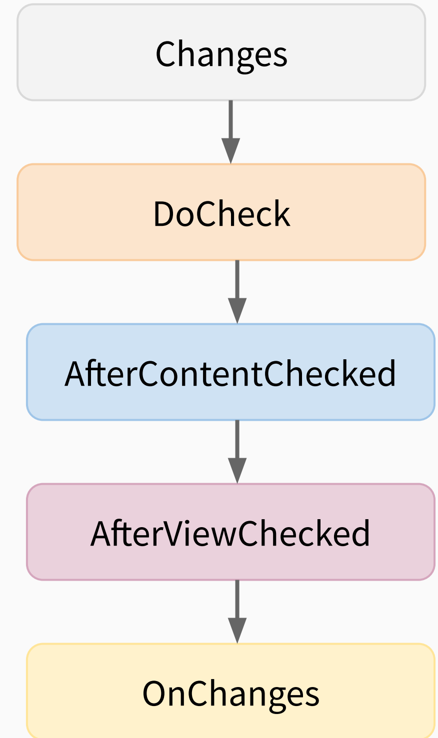
Component Lifecycle Hooks - Execution

- After a change
- Change detection calls custom DoCheck function
- Content and view are checked
- Inform about possible changes



Component Lifecycle Hooks - Execution

- After the *check* hooks are called once → change detection runs again to check for unexpected changes → triggers the *check* hooks again!
- If Angular notices changes after first change detection run → error in JavaScript console (not in production mode)



Task

Component LifeCycle Basic

Pipes

Angular pipes are a way to write display-value transformations that you can declare in your HTML

Angular Pipes

<code>

Use the Pipe operator | to use pipes in your templates

```
<div>
  {{ someName | uppercase }}
  {{ someName | lowercase }}
  {{ someDate | date:"MM/dd/yy" }}
</div>
```

Built-in Pipes

- AsyncPipe
- UpperCasePipe
- LowerCasePipe
- JsonPipe
- SlicePipe
- DecimalPipe
- PercentPipe
- CurrencyPipe
- DatePipe

Async Pipe

The AsyncPipe accepts a Promise or Observable as input and subscribes to the input automatically, eventually returning the emitted values.

Async Pipe

- Subscribe to Observable
- UnSubscribe on component destruction
- Built-In Pipe
- Simple use: {{ books$ | async }}

Async Pipe

<code>

For every async a new subscription is made. Try to minimize use of async

Two subscriptions created.
Could cause performance issues

```
<li *ngFor="let book of books$ | async">
  {{book.title}}
</li>

<span>{{ (books$ | async).length }}
```

Async Pipe

<code>

Finnish Notation. Naming observables with an $ suffix

```
<li *ngFor="let book of books$ | async">
  {{book.title}}
</li>
```

Task

Use the async pipe

Cheat Sheets

Types

<table>
<tr><td>Booleans</td><td>boolean</td></tr>
<tr><td>Numbers</td><td>number</td></tr>
<tr><td>Strings</td><td>string</td></tr>
<tr><td>Lists</td><td>number[] | Array<number></td></tr>
<tr><td>Maps</td><td>interface /* separated defined and named */ | {…} /* inline */</td></tr>
<tr><td>Enumeration</td><td>enum Employees {Miriam, Matthias}</td></tr>
<tr><td>Any</td><td>any</td></tr>
<tr><td>Void</td><td>void // only as return type for functions/methods</td></tr>
<tr><td>Type Casting</td><td><type> | varName as type</td></tr>
</table>

ES2015/TS Classes

`class`	nicer way to define prototypes
`public`	the **default** for attributes and methods
`private`	only accessible within their declaring class
`protected`	accessible from within their declaring class and classes derived from their declaring class
`static`	methods or attributes can be called or get and set without an instance
`extends`	class gets extended by another class

Interfaces - The new keywords

- `interface`

  create a shape with types

- `implements`

  `class`es can implement an `interface`

Component Decorator - Interface

Name	Description	Default
selector	Define CSS Selector to match the element	undefined
template	View-Template as string	' ' (Empty String)
templateUrl	View-Template via URL	undefined
styleUrls[ ]	Reference to styles via URL	[ ]
directives[ ]	Inject other directives	[ ]
pipes[ ]	Inject other pipes	[ ]
providers[ ]	Define the injectable services	[ ]

Component Decorator - Interface

Name	Description	Default
encapsulation	Define the scoping of styles	Emulated
changeDetection	specify a custom changeDetection	CheckAlways

- there are more, but this are the most used and important ones

Component Lifecycle Hooks

Hook method	Interface	Description
`ngOnChanges`	OnChanges	Called when an input or output binding value changes
`ngOnInit`	OnInit	After the first `ngOnChanges`
`ngDoCheck`	DoCheck	Developer's custom change detection
`ngAfterContentInit`	AfterContentInit	After component content initialized
`ngAfterContentChecked`	AfterContentChecked	After every check of component content
`ngAfterViewInit`	AfterViewInit	After component's view(s) are initialized
`ngAfterViewChecked`	AfterViewChecked	After every check of a component's view(s)
`ngOnDestroy`	OnDestroy	Just before the directive is destroyed

View Encapsulation

Mode	Description
`ViewEncapsulation.`**`None`**	No encapsulation, styles in head
`ViewEncapsulation.`**`Emulated`**	Styles in head with attribute suffix (scoped styles)
`ViewEncapsulation.`**`ShadowDom`**	Use the Shadow DOM

symetics