INTRODUCTION

LESSON 01

SWAFE-01

WHAT IS ANGULAR?

- An open source JavaScript client side MVC framework
- Developed and maintained by Google
- Angular is a complete rewrite of it's popular predecessor AngularJS
  - Initial released in 2016 (Angular) and 2010 (AngularJS)
  - Current versions 11.2.2 (Angular) and 1.8.2 (AngularJS)
- Angular applications are implemented with TypeScript

WHY USE ANGULAR?

- Modern web applications have lots of client-side JavaScript
  - Need a framework to structure the code
- Why select Angular, there are many client-side frameworks and libraries?
  - Angular is a complete framework
  - Component-based (as React)
  - Complies with upcoming Web Component standards
- Why Angular and not AngularJS?
  - Better performance
  - Easier to learn

ANGULAR IS A COMPLETE FRAMEWORK

- Organize code through modules
- UI widgets through components
- Manipulate DOM through directives
- Format data with pipes
- Access to APIs through the HTTP module
- Enhances HTML forms
- Application routing with routing modules
- Execellent tooling support

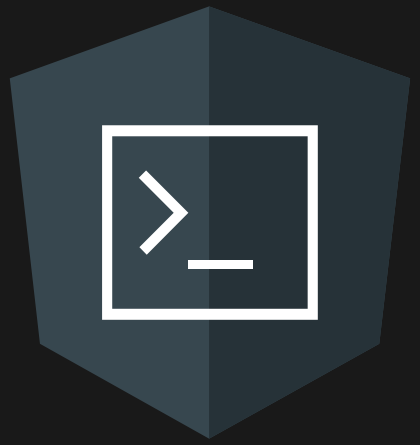
ANGULAR PHILOSOPHY

- Declarative programming should be used to create user interfaces and connect software components
- Imperative programming is better suited to defining business logic

DECLARATIVE VS. IMPERATIVE

- Declarative programming focuses on what the program should accomplish without specifying how the program should achieve the result.
  - Declarative code avoids mutation of state and variables
- Imperative programming focuses on describing how a program operates.
  - State changes and order of execution is important

ANGULAR CLI



OVERVIEW

- Installation
- Generate workspaces
  - Create and initialize new Angular workspaces and applications
- Generate project files
  - Generate and/or modify files based on scematics
- Build code for distribution
  - Compile an Angular application and build a version optimized for distribution
- Package management
  - Update applications and keep dependencies up to date

INSTALLATION

WORKSPACE

- A collection of Angular projects (this is, applications and libraries
- Typically co-located in a single source-control repository
- Commands that generate or operate on applications and libraries must be executed from within a workspace folder
- Workspace configuration is located in `angular.json` in the root directory

CREATE WORKSPACE

WORKSPACE CONFIGURATION

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  ...
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "my-first-application": {
      "projectType": "application",
      ...
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "configurations": {
            "production": {
              "fileReplacements": [
                {
                  "replace": "src/environments/environment.ts",
```

GENERATE FILES

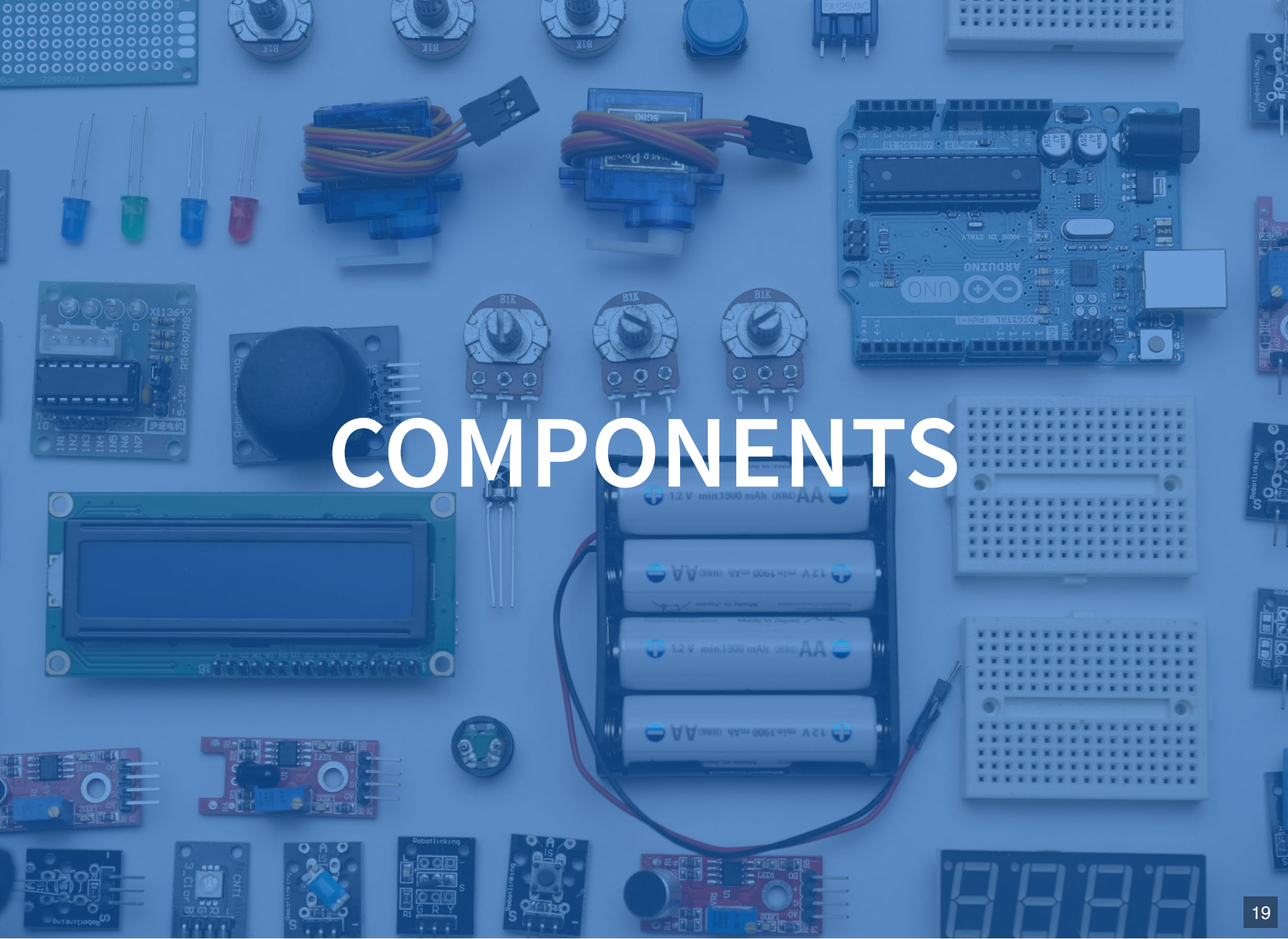
- Create angular artifacts with `ng generate`
  - `ng g c` generates a new component scaffold
  - `ng g m` generates a new module scaffold, add option `--routing` to generate a routing module as well
  - `ng g s` generates a new service
- These commands must be run from within a project folder
- Check out CLI Command Reference for all options (and there are a lot)

STRICT MODE

- Improves maintainability
- Helps you catch bugs ahead of time:
  - Enable `strict` mode in TypeScript
  - Turns on strict Angular compiler flags
- Reduces bundle size budgets by 75%
- Can be applied at workspace and project level
- Enabled by default for projects/workspaces created with Angular CLI

THE NUTS AND BOLTS

- Components
- Services
- Modules
- Directives, Pipes and Bindings


COMPONENTS

OVERVIEW

- The main building block for Angular applications
- Each component consists of:
  - HTML template
  - A TypeScript class that defines behavior
  - A CSS selector that defines how the component is used a template
  - CSS styles applied to the template (optional)
- Runtime behavior can be controlled by lifecycle hooks

TYPESCRIPT

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

@Component({
  selector: 'app-one',
  templateUrl: './one.component.html',
  styleUrls: ['./one.component.scss']
})
export class OneComponent implements OnInit {

  constructor() { }

  ngOnInit(): void { }
}
```

examples/lesson01-introduction/projects/components/src/app/one/one.component.ts

HTML TEMPLATE

```
1 <h1>Test</h1>
```

examples/lesson01-introduction/projects/components/src/app/one/one.component.html

STYLES

```scss
h1 {
  background-color: lime;
}

.container {
  display: flex;
}
```

examples/lesson01-introduction/projects/components/src/app/one/one.component.scss

LIFECYCLE HOOKS

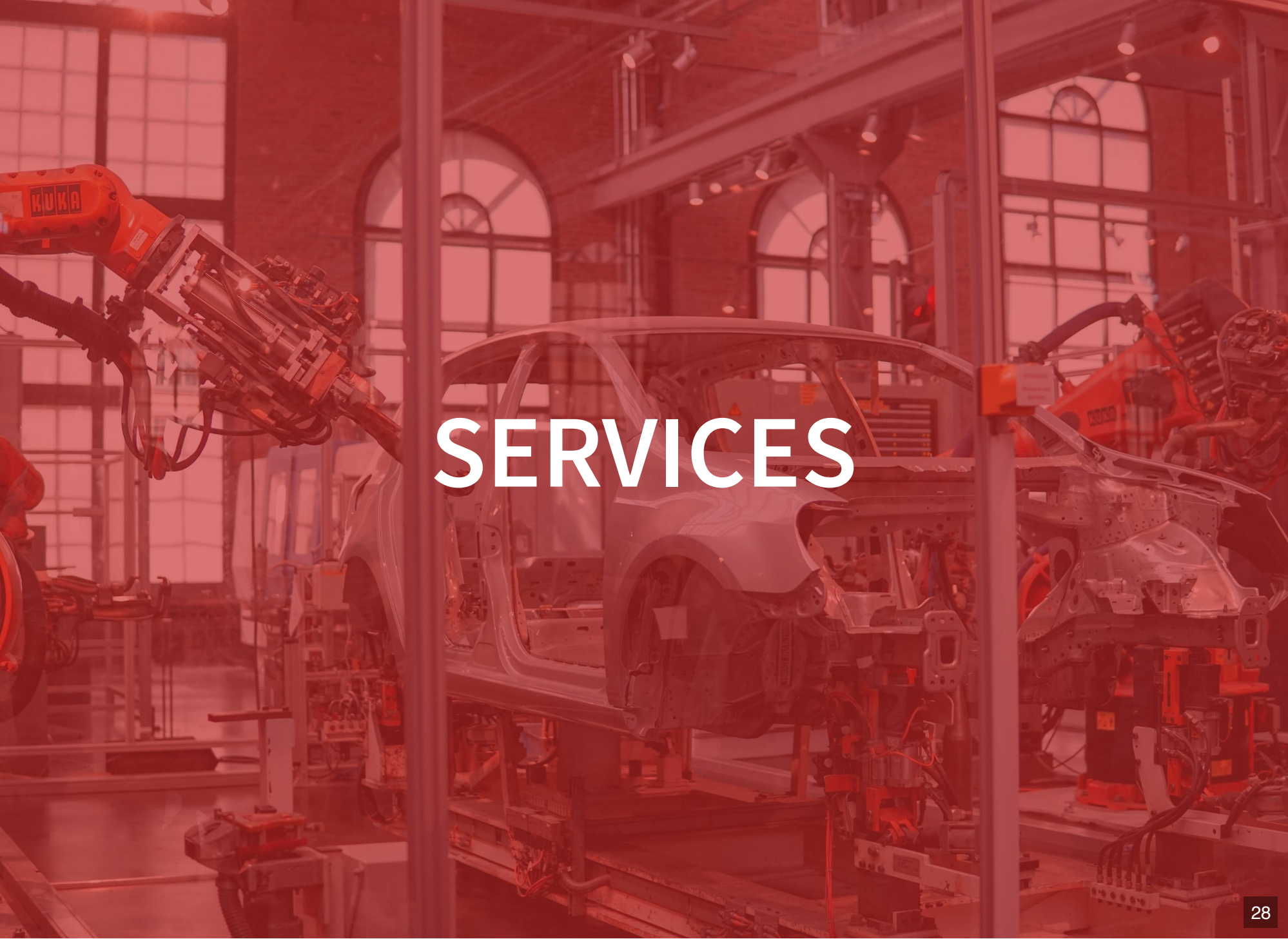
- Component instance has a lifecycle when Angular instantiates the class and renders the view
- Lifecycle handles change detection and updates view and instance as needed
- The lifecycle ends when Angular destroys the component instance and removes the template from the DOM
- Directive has a similar lifecycle

LIFECYCLE HOOKS

- Available lifehook methods
  1. `ngOnChanges()` *
  2. `ngOnInit()`
  3. `ngOnDoCheck()` *
  4. `ngAfterContentInit()`
  5. `ngAfterContentChecked()` *
  6. `ngAfterViewInit()`
  7. `ngAfterViewChecked()` *
  8. `ngOnDestroy()`
- Methods marked with "*" are called multiple times

LIFECYCLE HOOKS

- You only need to implement the hooks you need, if any
- You will most likely never implement all hooks
- The most commonly used are:
  - `ngOnInit()`
    - Perform complex initialization outside of the constructor
    - A good place for a component to fetch its initial data
  - `ngOnDestroy()`
    - Unsubscribe from Observables and DOM events
    - Unregister callbacks that the directive registered with global or application services
    - Notify other parts of the application that the component is going away
- Write lean hook methods to avoid performance issues
  - Be very careful about how much logic or computation you put into lifecycle methods
  - Some are called multiple times in quick succession



SERVICES

OVERVIEW

- Fetching and manipulating data
- Sharing data between unrelated components
- A class with a narrow, well-defined purpose
- Used with dependency injection

DEFINING A SERVICE

```ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class MyFirstService {

  constructor() { }

  now(): string {
    return new Date(Date.now()).toISOString()
  }
}
```

examples/lesson01-introduction/projects/services/src/app/my-first.service.ts

DEFINING A SERVICE

- The `@Injectable()` decorator specifies that Angular can use the class in the DI system
- Making a class with `@Injectable()` ensures that the compiler will generate the necessary metadata to create the class's dependencies when the class is injected
- The metadata `providedIn` determines which injectors will provide the injectable

INJECTING A SERVICE INTO A COMPONENT

```
1  import { Component, OnInit } from '@angular/core';
2  import { MyFirstService } from './my-first.service';
3
4  @Component({
5    selector: 'app-root',
6    templateUrl: './app.component.html',
7    styleUrls: ['./app.component.scss']
8  })
9  export class AppComponent implements OnInit {
10   title = 'services';
11   now: string | null = null
12
13   constructor(private service: MyFirstService) { }
14
15   ngOnInit() {
16     this.now = this.service.now()
17   }
18 }
```

examples/lesson01-introduction/projects/services/src/app/app.component.ts

DEPENDENCY INJECTION (DI)

- DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need
- DI elements in Angular
  - The injector is the main mechanism. Angular creates a framework-wide injector during the bootstrap process
  - Injectors create dependencies and maintains containers of instances that is reused if possible
  - A provider is an object that tells an injector how to create or obtain a dependency
- A dependency does not have to be a service–it could be a function, for example, or a value

PROVIDERS

- Dependency providers make services available to the application
- A service can be provided in multiple ways
  - At application-level where the service is available to the whole application
  - At component-level where each component has it's own instance of the service (sandboxing)

```
@Component({
  selector: 'selector',
  template: ...,
  providers: [MyFirstService]
})
```

DEFINING PROVIDERS

```
// Provider syntax
providers: [Logger]

// Expands into full provider object
[{ provide: Logger, useClass: Logger}]
```

- Property `provide` holds the token that serves as key for dependency value
- The next property holds a Provider definition object:
  - `useClass` is a provider definition object
  - `useExisting` – the provider is an instance of an existing object
  - `useValue` – the provider is an object or primitive value
  - `useFactory` – the provider is a factory method


MODULES

OVERVIEW

- A container for a group of related components, services, directives, and pipes
- A package that contains certain business domain functionality
- All apps must have at least one module (root module)
- Modules can be loaded eagerly when the application starts or lazy loaded asynchrously by the router

ROOT MODULE

```
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3
4  import { AppComponent } from './app.component';
5  import { FeatureOneModule } from './feature-one/feature-one.module';
6  import { FeatureTwoModule } from './feature-two/feature-two.module';
7
8  @NgModule({
9    declarations: [
10     AppComponent
11   ],
12   imports: [
13     BrowserModule,
14     FeatureOneModule,
15     FeatureTwoModule,
16   ],
17   providers: [],
18   bootstrap: [AppComponent]
19 })
```

examples/lesson01-introduction/projects/modules/src/app/app.module.ts

ANGULAR MODULES

- Angular itself is split into modules
  - `BrowserModule` – Exports infrastructure for all Angular apps
  - `FormsModule` and `ReactiveFormsModule` – Exports the required providers and directives for template-driven and reactive forms
  - `RouterModule` — Adds directives and providers for in-app navigation among views defined in an application
  - `HttpClientModule` – Configures the dependency injector for `HttpClient` service

DECLARE MODULE

```
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3  import { FeatureThreeModule } from '../feature-three/feature-three.module';
4  import { FeatureTwoComponent } from './feature-two/feature-two.component';
5
6  @NgModule({
7    declarations: [
8      FeatureTwoComponent
9    ],
10   imports: [
11     CommonModule,
12     FeatureThreeModule,
13   ],
14   exports: [
15     FeatureTwoComponent
16   ]
17 })
18 export class FeatureTwoModule { }
```

examples/lesson01-introduction/projects/modules/src/app/feature-two/feature-two.module.ts

FEATURE MODULES

- Feature modules are `NgModule` instances for the purpose of organizing code
- Helps to apply clear boundaries for features
  - Keep code related to a specific functionality or feature separate from other code
- Helps with collaboration between developers and teams
- Managing the size of the root module
  - Applications are split up in smaller chunks that gets delivered when needed
- Only the root module should import `BrowserModule`, use `CommmonModule` instead in feature modules


DIRECTIVES

DIRECTIVES

- Directives are classes that add additional behavior to elements
- Many modules, such as `RouterModule`, `FormsModule`, and `ReactiveFormsModule` define their own attribute directives
- Three different types of directives
  - Components
    - A component is simply a directive with a template
    - Use the `Component` decoratorwhen declaring components
  - Attribute directives
    - Built-in attribute such as `NgClass` and `NgModel`
    - Custom directives
  - Structural directives
    - Change the DOM layout by adding and removing DOM elements
    - Examples are: `NgIf`, `NgForOf`, and `NgSwitch`

ATTRIBUTE DIRECTIVES

- `NgClass` – adds and removes a set of CSS classes
- `NgStyle` – adds and removes a set of HTML styles
- `NgModel` – adds two-way data binding to an HTML form element

See examples/lesson01-introduction/projects/directives for implementation details

STRUCTURAL DIRECTIVES

- `NgIf` – conditionally includes a template based on a boolean expression
- `NgForOf` – renders a template for each item in a collection
- `NgNgSwitch` – specifies an expression to match against and show views in container

See examples/lesson01-introduction/projects/directives for implementation details

CUSTOM DIRECTIVES

- You can implement custom directives (both structural and attribute)
- Generate custom directive code with CLI command `ng generate directive ...`

CUSTOM ATTRIBUTE DIRECTIVE

```typescript
import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({
  selector: '[appHover]'
})
export class HoverDirective {

  constructor(private element: ElementRef) { }

  @HostListener('mouseenter')
  onMouseEnter() {
    this.highlight('lime')
  }
  @HostListener('mouseleave')
  onMouseLeave() {
    this.highlight('')
  }

  highlight(color: string) {
```

examples/lesson01-introduction/projects/directives/src/app/attribute/hover.directive.ts

CUSTOM STRUCTURAL DIRECTIVE

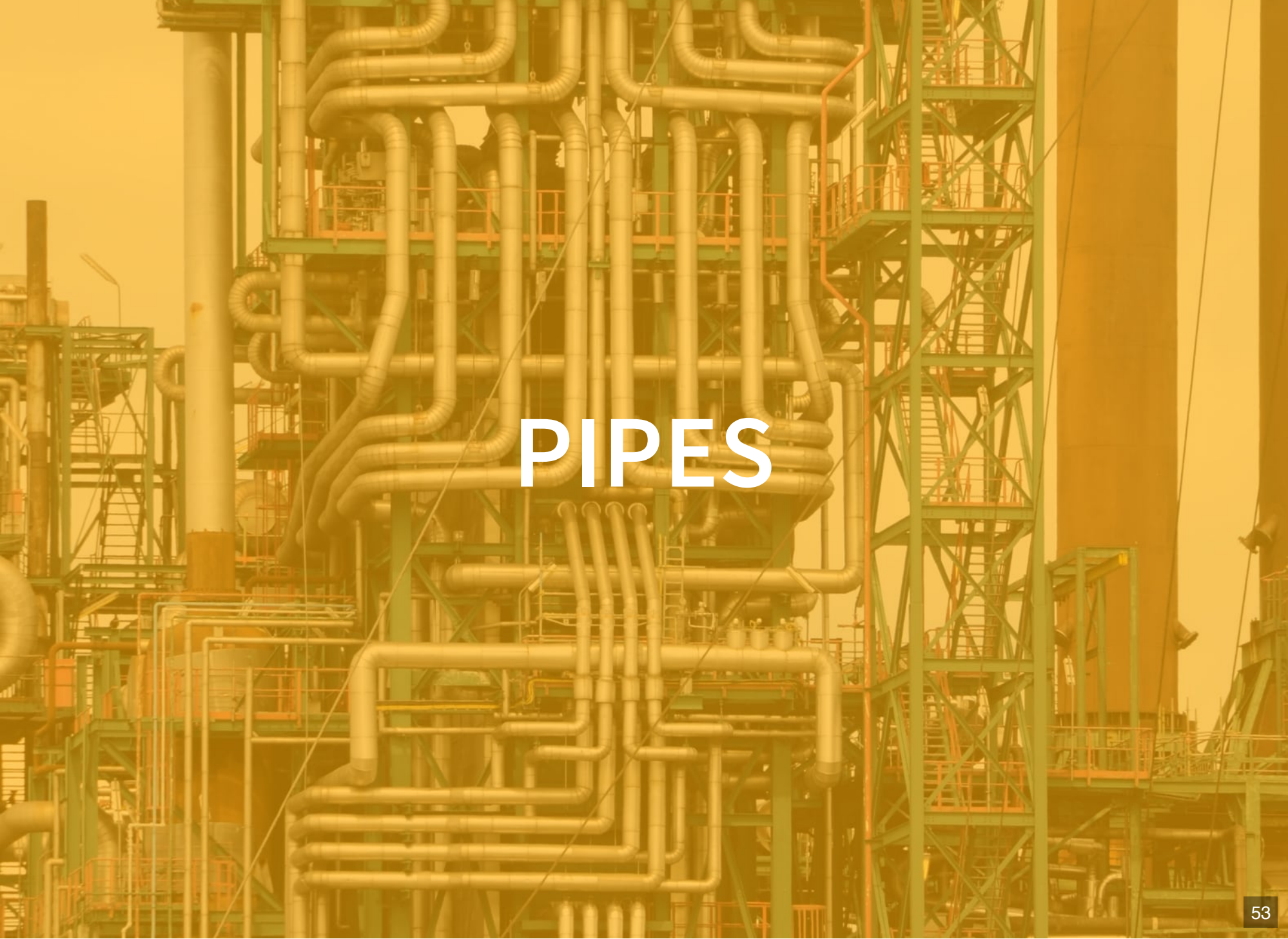
```
1  import { Directive, OnDestroy, OnInit, TemplateRef, ViewContainerRef } from '@
2  import { Subscription } from 'rxjs';
3  import { AuthService } from './auth.service';
4
5  @Directive({
6    selector: '[appIsAuthenticated]'
7  })
8  export class IsAuthenticatedDirective implements OnInit, OnDestroy {
9
10   subscription!: Subscription;
11
12   constructor(
13     private container: ViewContainerRef,
14     private template: TemplateRef<any>,
15     private auth: AuthService) { }
16
17     ngOnInit() {
18       this.subscription = this.auth.isAuthenticated().subscribe(authenticated
19         if(authenticated) {
```

examples/lesson01-introduction/projects/directives/src/app/custom/is-authenticated.directive.ts

CUSTOM STRUCTURAL DIRECTIVE

```
<app-navigation></app-navigation>
<div class="content">
  <div class="wrapper">
    <app-restricted *appIsAuthenticated></app-restricted>
    <app-public></app-public>
  </div>
</div>
```

examples/lesson01-introduction/projects/directives/src/app/custom/app.component.html


PIPES

OVERVIEW

- Pipes are used to transform data for display
- Pipes are declared once and used throughout application
- Angular provides some built-in pipes
  - `DatePipe` formats a date value according to locale rules
  - `DecimalPipe` formats a value to digit options and locale rules
  - `CurrencyPipe` transforms a number to a currency string, formatted according to locale rules
  - `JsonPipe` converts a value into its JSON-format representation
  - `AsyncPipe` Unwraps a value from an asynchronous primitive
- You can build your own pipes for custom data types

DATEPIPE

```
@Component({
  selector: 'date-pipe',
  template: `<div>
    <p>Today is {{ today | date }}</p>
    <p>Or if you prefer, {{ today | date:'fullDate' }}</p>
    <p>The time is {{ today | date:'h:mm a z' }}</p>
  </div>`
 })
 // Get the current date and time as a date-time value.
 export class DatePipeComponent {
   today: number = Date.now();
 }
 // Prints:
 // Today is Aug 20, 2021
 // Or if you prefer, Friday, August 20, 2021
 // The time is 2:19 PM GMT+2
```

examples/lesson01-introduction/projects/pipes/src/app/app.component.html

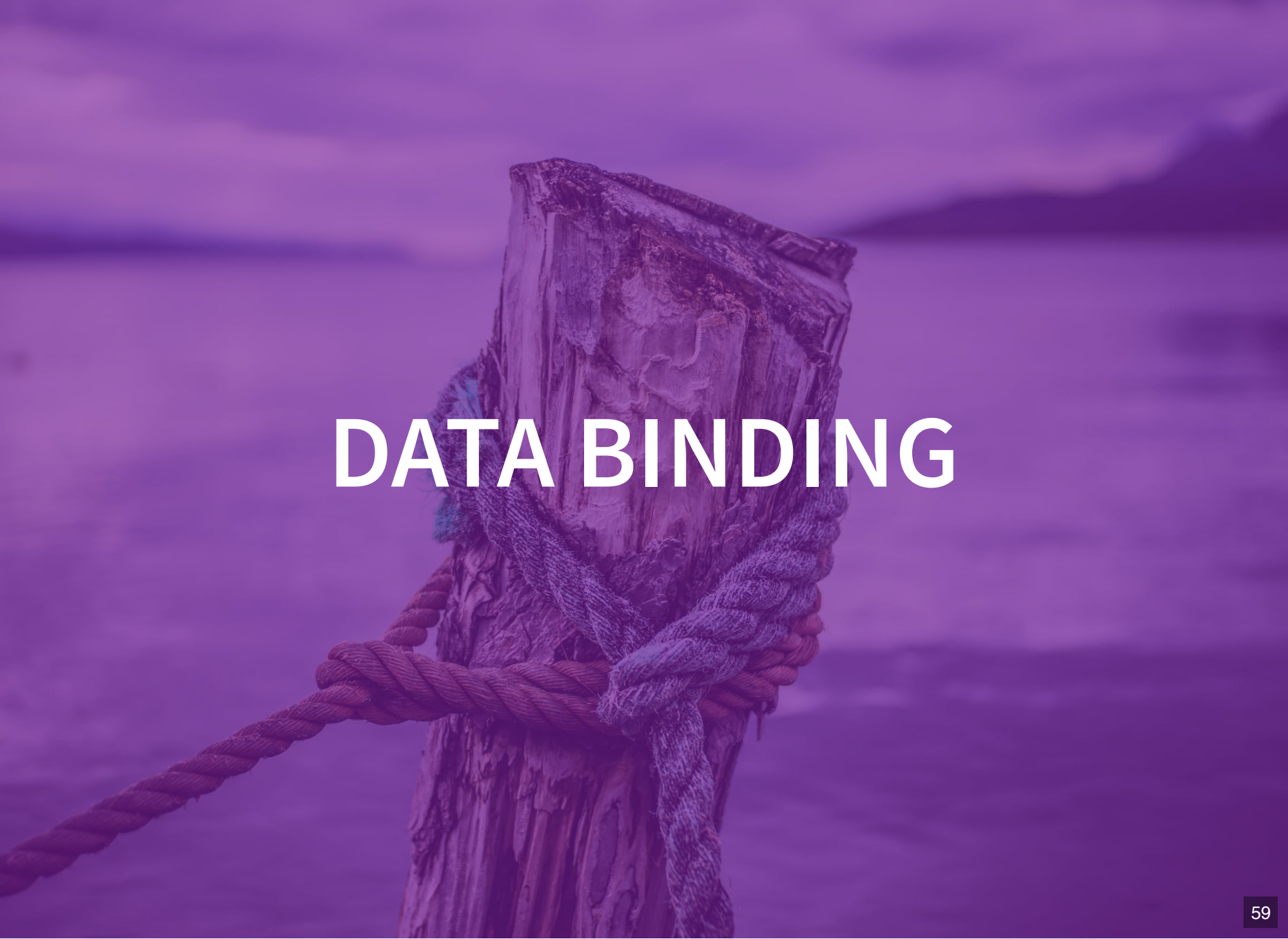
PURE VS. IMPURE PIPES

- Pipes are *pure* by default
- A pipe must use a pure function
  - A function that processes the input and returns an output without side effects
  - Given the same input, the function always returns the same output
- Angular only executes pipes when it detects pure change to the input value
  - A change to primitive value, such as `String`, `Number`, `Boolean`, or `Symbol`
  - A changed object reference, such as `Date`, `Array`, or `Object`
  - Angular does not execute pipes when *mutating* data, only when the references *change*
- Do not do heavy or long-running computation in impure pipes, it can dramatically slow down application execution

CUSTOM PIPES

```
1  import { Pipe, PipeTransform } from '@angular/core';
2  import { User } from './user.model';
3
4  @Pipe({
5    name: 'user'
6  })
7  export class UserPipe implements PipeTransform {
8
9    transform(user: User): string {
10     return `${user.firstName} ${user.lastName}`;
11   }
12 }
```

examples/lesson01-introduction/projects/pipes/src/app/user.pipe.ts


DATA BINDING

OVERVIEW

- Property binding
  - One-way binding from a component's property to a target element
- Event binding
  - Listen for and respond to user actions
- Two-way binding
  - Listen for events and update values simultaneously between parent and child components

PROPERTY BINDING

- Property binding set values for properties of HTML elements or directives
- Property binding moves a value en one direction, from a component's property into a target element property
- Use squared brackets (`[ ]`) to declare property as a target property
- Property binding best practices
  - Avoid side effects - Evaluation of a template expression should have no visible effect
  - Return proper type – A template expression should evaluate to the type of value that the target property expects

THEM BRACKETS

```
<!-- Evaluate right-hand side of the assignment as a dynamic expression -->
<img [src]="itemImageUrl" alt="image">

<!-- Renders the string 'parentItem' -->
<app-item-detail childItem="parentItem"></app-item-detail>

<!-- Set model property of a custom component -->
<app-item-detail [childItem]="parentItem"></app-item-detail>

<!-- Set property of a directive -->
<p [ngClass]="classes">[ngClass] binding to the classes property making this b
```

EVENT BINDING

- Listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches

BINDING TO EVENTS

- This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right

```
1 <!-- Bind to click event -->
2 <button (click)="onSave()">Save</button>
```

- The event binding listens for the button's click events and calls the component's `onSave()` method whenever a click occurs

TWO-WAY BINDING

- Combines property binding with event binding
  - Property binding sets a specific element property
  - Event binding listens for en element change event
- Use a combination of square brackets and parentheses `[()]` to create a two-way binding in a template

CREATE A TWO-WAY BINDING

- A two-way-binding must follow a specific naming convention
  - The `@Output()` must end with `Change` prefixed with the `@Input()` property name
  - For example, if `@Input() size` then `@Output() sizeChange`

CREATE A TWO-WAY BINDING

```
export class SizerComponent {

  @Input() size!: number | string;
  @Output() sizeChange = new EventEmitter<number>();

  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }
}
```

examples/lesson01-introduction/projects/bindings/src/app/sizer/sizer.component.ts

CREATE A TWO-WAY-BINDING

```
<div>
  <button (click)="dec()" title="smaller">-</button>
  <button (click)="inc()" title="bigger">+</button>
  <label [style.font-size.px]="size">FontSize: {{size}}px</label>
</div>
```

examples/lesson01-introduction/projects/bindings/src/app/sizer/sizer.component.html

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
<div [style.font-size.px]="fontSizePx">Resizable Text</div>
```

examples/lesson01-introduction/projects/bindings/src/app/app.component.ts

WRAP-UP

- What is Angular?
  - A component-based framework for building scalable web applications
  - A suite of developer tools to help build, test and update code
- Angular CLI
  - A command-line interface used to build Angular applications
- Angular artifacts
  - Directives additional behavior to elements
  - Services contains business logic
  - Modules organizes code