Component Styles

- Class and Style Bindings
- Built-In Directives - NgStyle & NgClass
- Using Component Styles
- Special selectors
- Loading Styles into Components
- Controlling View Encapsulation: Emulated, Native, and None
- Appendix 1: Inspecting the generated runtime component styles
- Appendix 2: Loading Styles with Relative URLs

Class and Style Binding

Angular applications are styled with regular CSS. That means we can apply everything we know about CSS stylesheets, selectors, rules, and media queries to our Angular applications directly.

Class Binding:

We can add and remove CSS class names from an element's class attribute with a **class binding**.

Class binding syntax resembles property binding. Instead of an element property between brackets, we start with the prefix class, optionally followed by a dot (.) and the name of a CSS class: [class.class-name].

Add the following to Index.html

```
<style>
  .redBorder
  {
    border: solid 2px red
  }
  .yellowBackground{
    background-color:yellow;
  }
</style>
```

app.component.ts

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

@Component({
 selector: 'my-app',
 template: `
```

```
      <div class="redBorder">Class redBorder applied as in normal HTML</div> <br/>
      <div [class]="applyStyle1">Plain text without styles</div> <br/>
      <div [class]="applyStyle2">Class redBorder is applied</div> <br/>
      <div [class]="applyStyle3">Classes redBorder and yellowBackground are applied</div> <br/>

      <div [class.redBorder]="isRedBorder"
         [class.yellowBackground]="isYellowBackground">
         Class redBorder is applied
      </div> <br/>
/>
`
})

export class AppComponent
{
   applyStyle1 = ""
   applyStyle2 = "redBorder"
   applyStyle3 = "redBorder yellowBackground"

   isRedBorder = true
   isYellowBackground = true
}
```

Style binding

We can set inline styles with a **style binding**.

Style binding syntax resembles property binding. Instead of an element property between brackets, we start with the prefix style, followed by a dot (.) and the name of a CSS style property: [style.style-property].

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

@Component({
   selector: 'my-app',
   template: `
   <button [style.color] = "isSpecial ? 'red': 'green'">Red</button>
   <button [style.background-color]="canSave ? 'cyan': 'grey'" >Save</button>
```

```
  <button [style.font-size.em]="isSpecial ? 3 : 1" >Big</button>
  <button [style.font-size.%]="!isSpecial ? 150 : 50" >Small</button>
`
})

export class AppComponent {
  isSpecial = true;
  canSave = true;
}
```

Note that Some style binding styles have unit extension

While the above two ways are ideal for setting single class and style, we generally prefer **NgClass** and **NgStyle** directives when setting several inline styles and classes at the same time.

Built-In Directive – NgClass and NgStyle

NgClass Directive:

The NgClass directive is a better choice when we want to add or remove *many* CSS classes at the same time.

A good way to apply NgClass is by binding it to a key:value control object. Each key of the object is a CSS class name; its value is true if the class should be added, false if it should be removed.

Add the following to Index.html

```
<style>
  .saveable {
    font-style:italic
  }
  .modified {
    font-weight:bold
  }
  .special {
    font-size: 24px;
  }
</style>
```

app.component.ts

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


@Component({
```

```
  selector: 'my-app',
  template: `
    <div [ngClass]="setClasses()">This div is saveable and special</div>
`
})

export class AppComponent {
  canSave = true;
  isModified = true;
  isSpecial = true;
  setClasses() {
    let classes = {
      saveable: this.canSave,     // true
      modified: this.isModified, // false
      special: this.isSpecial,    // true
    };
    return classes;
  }
}
```

NgClass Directive:

We can set inline styles dynamically, based on the state of the component. Binding to NgStyle lets us set many inline styles simultaneously.

We apply NgStyle by binding it to a key:value control object. Each key of the object is a style name; its value is whatever is appropriate for that style.

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

@Component({
  selector: 'my-app',
  template: `
    <div [ngStyle]="setStyles()">
      This div is italic, normal weight, and extra large (24px).
    </div>
`
```

```
})

export class AppComponent {
  canSave = true;
  isModified = false;
  isSpecial = true;
  setStyles() {
    let styles = {
      // CSS property names
      'font-style': this.canSave ? 'italic' : 'normal',  // italic
      'font-weight': this.isModified ? 'bold' : 'normal',  // normal
      'font-size': this.isSpecial ? '24px' : '8px',    // 24px
    };
    return styles;
  }
}
```

Using Component Styles

For every Angular component we write, we may define not only an HTML template, but also the CSS styles that go with that template, specifying any selectors, rules, and media queries that we need.

One way to do this is to set the styles property in the component metadata. The styles property takes an array of strings that contain CSS code. Usually we give it one string as in this example:

Benefits:

1. We can use the CSS class names and selectors that make the most sense in the context of each component.
2. Class names and selectors are local to the component and won't collide with classes and selectors used elsewhere in the application.
3. Our component's styles *cannot* be changed by changes to styles elsewhere in the application.
4. We can co-locate the CSS code of each component with the TypeScript and HTML code of the component, which leads to a neat and tidy project structure.
5. We can change or remove component CSS code in the future without trawling through the whole application to see where else it may have been used. We just look at the component we're in.

```
@Component({
  selector: 'my-app',
  template: `
```

```
    <div>This is div</div>
    <h1>This is H1 again</h1>
  `,
  styles: ["div {font-weight: bold; color:red} h1 {color:blue}"]
})
```

Special Selectors:

1. Use the **:host** pseudo-class selector to target styles in the element that *hosts* the component (as opposed to targeting elements *inside* the component's template):

   :host { border-width: 3px; }

// In below example we target the host element again, but only when it also has the active CSS class

   :host(.active) {border-width: 3px;}

2. **:host-context()** pseudo-class selector. It looks for a CSS class in *any ancestor* of the component host element, all the way up to the document root. It's useful when combined with another selector.

   In the following example, we apply a background-color style to all <h2> elements *inside* the component, only if some ancestor element has the CSS class theme-light.

   :**host-context**(.theme-light) h2 {

       background-color: #eef;

   }

3. We can use the /deep/ or >>> selector to force a style down through the child component tree into all the child component views. The /deep/ selector works to any depth of nested components, and it applies *both to the view children and the content children* of the component.

   In this example, we target all <h3> elements, from the host element down through this component to all of its child elements in the DOM:

   :host **/deep/** h3 { font-style: italic; }

Loading Styles into Components

1. **Styles in Metadata**

```
@Component({
  selector: 'my-app',
  template: `
    <div>This is div</div>
  `,
  styles: [`
    div {font-weight: bold; color:red}
    h1 {color:blue}
```

```
  `]
})
```

2. Template Inline Styles

```
@Component({
  selector: 'my-app',
  template: `
    <style>
      .div {
        font-style:italic
      }
    </style>
    <div>This is div</div>
  `
})
```

3. Style URLs in Metadata

```
@Component({
  selector: 'my-app',
  template: `
    <div>This is div</div>
  `,
  styleUrls: ['app/demo.component.css']
})
```

Note: The URL is ***relative to the application root*** which is usually the location of the index.html web page that hosts the application.

Loading Styles with Relative URLs

We can change the way Angular calculates the full URL be setting the component metadata's moduleId property to module.id.

```
@Component({
  moduleId: module.id,
  selector: 'my-app',
  template: `
    <div>This is div</div>
  `,
  styleUrls: ['demo.component.css']
```

```
})
```

4. Template Link Tags

```
@Component({
   selector: 'my-app',
   template: `
      <link rel="stylesheet" href="app/demo.component.css">
      <div>This is div</div>
`})
```