# Angular
# Lab 1
# Data Binding



## 1 Lab setup

Make sure you have the following items installed

- Latest version of Node and NPM (note: labs are tested with Node v20.13 and NPM v10.7 but should work with later versions with no or minimal changes)
- Latest version of TypeScript (note: labs are tested with TypeScript v5.4. but should work with later versions with no or minimal changes)
- Latest version of Angular (note: labs are tested with Angular v17. Angular 16 and 17 incorporate significant feature changes that distinguish them from Angular 15 and earlier, so the code samples for these labs will not run in projects generated with Angular 15 or earlier).
- Visual Studio Code (or a suitable alternative IDE for Angular/TypeScript)
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`.

- The `changes` subfolder holds the source code and other related files for the lab. The creation of the Angular app will proceed in a step-wise fashion in order to clearly demonstrate the various features of the framework. The various project source files in this folder are numbered in a way to reflect this gradual construction. For e.g. `xxx.v1.html, xxx.v2.html`, etc represents successive changes that are to be made to the `xxxx.html`

- The `final` subfolder holds the complete Angular project. If there was some issue in completing the step-wise construction of the app using the files from the `changes` subfolder, you can use this to run the final working version.

The project folder in `final` is however missing the crucial `node_modules` subfolder which contains all the dependencies (JavaScript libraries) that your Angular app needs in order to be built properly. These dependencies are specified in the `package.json` file in the root project folder. They are omitted in the project commit to GitHub because the number and size of the files are prohibitively large.

In order to run the app via the Angular development server (`ng serve`), you will need to create this subfolder containing the required dependencies. There are two ways to accomplish this:

a) Copy the project root folder to a suitable location on your local drive and run `npm install` in a shell prompt in the root project folder. NPM will then create the `node_modules` subfolder and download the required dependencies as specified in `package.json` to populate it. This process can take quite a while to complete (depending on your broadband connection speed)

b) Alternatively, you can reuse the dependencies of an existing complete project as most of the projects for our workshop use the same dependencies. In this case, copy the `node_modules` subfolder from any existing project, and you should be able to immediately start the app. If you encounter any errors, this probably means that there are some missing dependencies or Angular version mismatch, in which case you will then have to resort to `npm install`.

The lab instructions here assume you are using the Chrome browser. If you are using a different browser (Edge, Firefox), do a Google search to find the equivalent functionality.

# 2 Generating a new Angular project

We will be creating many Angular projects during this workshop. Create a new folder on your machine to hold the source code for the Angular projects that you will generate during the workshop. This folder can be on your Desktop or any suitable location (for e.g. `C:\angularlabs\myprojects`). However, please **DO NOT** generate your projects directly in your root drive C:\

You will generate new Angular projects using the Angular CLI from the command prompt or shell terminal.

You can open a DOS command prompt in this new folder by navigating to it using File Explorer and then launching the command prompt directly in that folder:
https://johnwargo.com/posts/2024/launch-windows-terminal/

At the terminal / command prompt, type:

```
ng new bindingdemo
```

Press enter to accept the default values for all the question prompts that follow regarding stylesheet format (CSS) and enabling Server side rendering (No)

```
G:\temp\workshoplabs>ng new bindingdemo
? Which stylesheet format would you like to use? CSS             [
https://developer.mozilla.org/docs/Web/CSS                      ]
? Do you want to enable Server-Side Rendering (SSR) and Static Site
Generation (SSG/Prerendering)? (y/N)
```

The process of creating a project will take a while (depending on the speed of your broadband connection) as NPM will need to download and install all the JavaScript module dependencies that the Angular project needs to run properly.

Navigate into the root project folder from the terminal by typing:

```
cd bindingdemo
```

Build the app and serve it via Angular's live development server by typing:

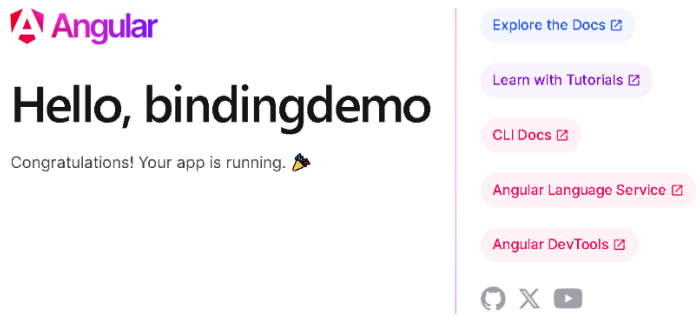
```
ng serve
```

The final output line from this should indicate that the compilation process was successful and identify the port that the live development server is currently serving up the Angular app dynamically from (by default this will be port 4200)

```
Initial chunk files | Names         | Raw size
polyfills.js        | polyfills     | 85.81 kB |
main.js             | main          |  4.56 kB |
styles.css          | styles        | 95 bytes |

                    | Initial total | 90.46 kB

Application bundle generation complete. [1.405 seconds]

Watch mode enabled. Watching for file changes...
  ➜  Local:   http://localhost:4200/
  ➜  press h + enter to show help
```

Open a new browser tab at:
http://localhost:4200/

You should be able to see the default autogenerated landing page for all new Angular projects.

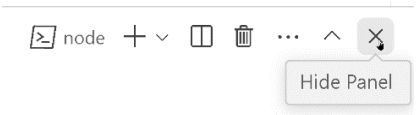


To stop the development server, type Ctrl + C in the terminal window that it is running in. You can restart again anytime by typing `ng serve` in the project root folder.

An alternative to using the DOS command prompt to run the development server is to use the embedded terminal in Visual Studio Code.
Open the root folder of this Angular project using VS Code (File -> Open Folder).
Select Terminal -> New Terminal from the main menu
In the newly open terminal at the bottom, start the development server with the standard command:

```
ng serve
```

Once the server is started, you can subsequently hide the terminal panel:

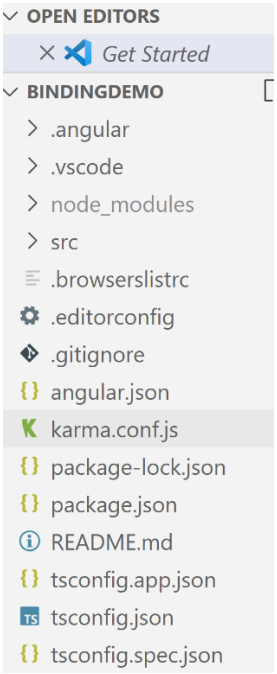



You can access it again to view console output error messages and so on through View -> Terminal from the main menu.

Additional references:
https://ccbill.com/kb/install-angular-on-windows

# 3 General structure of an Angular app

Open the root folder of this Angular project using VS Code (File -> Open Folder).
You should see a variety of files listed in the Explorer tab.




Most of these files are in JSON format and provide configuration for the Angular app as well the TypeScript compiler used to compile it. We will examine these in more detail in a subsequent lab.

Navigate in the Explorer into `src/app`. You will see the core files that constitute a minimal Angular app. Open `app.component.ts`

This is the root component of the Angular app, which is always a TypeScript class and has the default name of AppComponent. It typically has a single property title which by default will be the name of the project (`bindingdemo` in this case).

It has a @Component decorator which identifies the class as a proper Angular component (rather than just an ordinary TypeScript class). The decorator specifies several important metadata items:

- `selector`: This is a CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML of any other component in the app (or `index.html` for the case of the root component). While we normally use element selectors, any valid CSS selector can be used as well. Note that there can only be one component instantiated for a given element selector in a template. When the component is instantiated, its corresponding template is then substituted into the place of its selector tag in the target template HTML and Angular then renders this as a view.

- `standalone` : This determines whether the component is a standalone component or included as part of an existing module. This is a new property introduced in Angular 17 to simplify construction of apps, by making all components in an application to be standalone by default

- `imports`: Specifies specific user-defined components or Angular library components, directives, and packages that will be subsequently used in the template HTML. This is necessary for standalone components. The default package is RouterOutlet which is necessary to be used in routing (which we will see in another lab)

- `templateUrl`: The path and file name of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the *template* property (which we will see later in another lab)

- `styleUrl`: Specifies an external CSS stylesheet which contains style rules that will apply to the component's template. Alternatively, you can use the *styles* array property to define CSS rules as strings directly (which we will see later in another lab)

There are other metadata items that can be specified here, which we will examine in a later lab.

Open `app.component.html`

This contains the template HTML corresponding to the root component (notice that the name of this file is specified in the `templateUrl` property of the @Component decorator metadata). This contains the HTML and styling that produces the default landing page for the Angular app that we saw earlier.

Open `app.component.css`

This is the stylesheet where we can specify CSS styling rules for the template HTML.
Notice that is currently empty. All the styling for the landing page is currently achieved via embedded style rules in the HTML within the <style> tags.

Open `app.component.spec.ts`

This contains the code to perform basic unit testing on the Angular app, which we can expand upon if we wish. Angular uses the Jasmine test framework by default for creating tests and uses the Karma test runner to run the tests.

Open `app.config.ts`

This provides the configuration logic for the app. Currently we only have minimal basic configuration which just includes routing configuration information. We will study routing in more detail in an upcoming lab.

Navigate back out to `src`
Open `src/main.ts`

This file is responsible for bootstrapping (creating or launching) the application. Notice that is references the root component (AppComponent) as well the configuration file (app.config.ts).

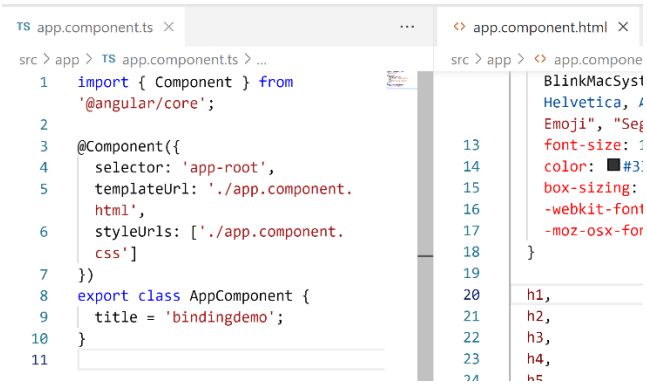
Open `src/styles.css`

The style rules in `app.component.css` apply only to the HTML in `app.component.html`. Later on, when you create new components, each of these new components will have their own corresponding template (*.component.html) and stylesheet ( *.component.css ), whose style rules are only specific to that corresponding template.
On the other hand, `src/styles.css` holds global style rules, i.e. they apply to ALL the HTML templates of all the components in the app

Open `src/index.html`

This is the HTML file (the single page of a SPA) that will be served up by the Angular live development server. Notice that the <body> element of this file contains the CSS selector `(<app-root>)` for the root component AppComponent. As explained earlier, Angular will replace this selector with the HTML from the template when it builds and serves up the app.

You can use View -> Editor layout to split the editors for the different files in order to lay them out in different orientations to simplify working with them. It is often useful to have both the component and its template viewable simultaneously in two different editors because the content in both these files are tightly bound with each other, as we will see in a later lab.



You can also use View -> Word Wrap to toggle word wrap if necessary to avoid scrolling across editor views for long lines of code.

Additional references:
https://angular.io/guide/architecture-components#introduction-to-components-and-templates
https://angular.io/guide/architecture-components#component-metadata
https://angular.io/api/core/Component#description

# 4 Template expressions and interpolations

The source code for this lab can be found in the `changes` subfolder of `Binding-Demo`

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v1.ts`

`app.component-v1.html`

`app.component-v1.css`

Notice that after you have made these changes, the Angular development server detects these changes, automatically rebuilds the app and serves it at the port that is currently listening on (default 4200). There is no need to restart the server again.

Make changes to the values of the properties and methods in AppComponent and notice that the changes are updated immediately to the app landing page. The Angular development server recompiles and updates the application automatically so that the new content is displayed in the browser view without the need to reload.

Template expressions are used primarily in data binding. The most common form is to use them in interpolations within the double curly braces `{{ }}`

When used in interpolation, template expressions will always produce a string value that can be incorporated into the portion of HTML elements that accept text (for e.g. `<p>`, `<span>`, `<break>`,)

The expression context refers to the location where the particular variable names used in a template expression can be resolved; this is typically the component that the template is directly associated with. Thus, the template expression will typically reference component properties and methods.

Component properties must be public (this is the default access modifier if none is specified) in order to be accessible in the template expression

References:

https://angular.dev/guide/templates/interpolation
https://angular.io/guide/interpolation#displaying-values-with-interpolation
https://angular.io/guide/interpolation#template-expressions

Issues to keep in mind regarding template expressions:

https://angular.io/guide/interpolation#syntax
https://angular.io/guide/interpolation#expression-best-practices

A listing of good sites with comprehensive collection of royalty free images that you can use for your projects:
https://buffer.com/library/free-images/

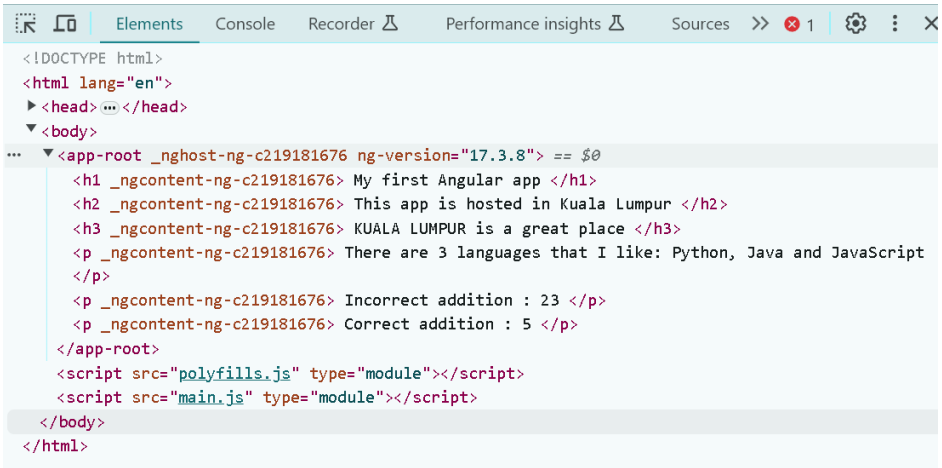
# 5 Angular app scripts and the DOM

On the main landing page of the Angular app, right click anywhere in the page and select View Page Source from the context menu. You will be able to see the actual HTML file served up from the Angular development server. It has a single <app-root> element and references 2 additional JavaScript scripts referenced from it at the bottom. These scripts together implement the functionality of the Angular app that will generate a live DOM from the static HTML to create the view that you currently see in your browser:

- `polyfills.js` –provides the polyfill features that are not necessary for app compatibility with older browsers, particularly those that do not support newer features such as ES6.
- `main.js` – contains the application codebase; including key items such as components (ts, html and css), pipes, directives, services and all other imported modules

These two scripts are bundled using a new build system based on esbuild and Vite (previewed in Angular 16 and enabled by default in Angular 17). You should be able to see a script referencing this Vite module at the top of the HTML file.
You can click on any of these script references to view their contents. For e.g. click on `main.js` to view the transpiled Javascript (ES5) for your AppComponent Typescript class.

In the browser tab where the Angular app view is active, select More Tools -> Developer Tools, and click on the Elements tab to switch to the Element view which shows the DOM tree that the current view in the browser is rendered from. You will notice that the <app-root> element is now populated with content through the operation of these 2 scripts:

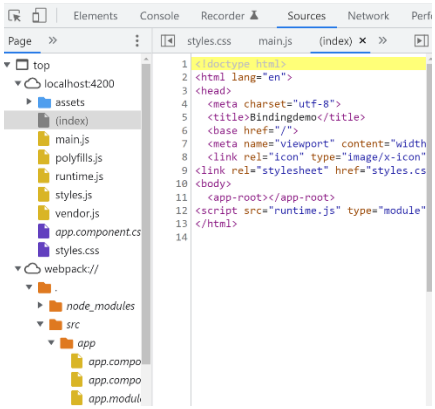



Notice that there are the standard HTML elements such as <h1>, <h2> and <p> present; but these are augmented with Angular-specific attributes such as `_ngcontent-xxxx` and `_nghost-xxx`, which

Angular uses isolate component styles from one another (to be covered in a later lab). You can also the current Angular version specified in the Angular-specific attribute `ng-version`.

The style sheet for this component (`app.component.css`) is also embedded directly in the page via `<style>` tags in the `<head>` element.

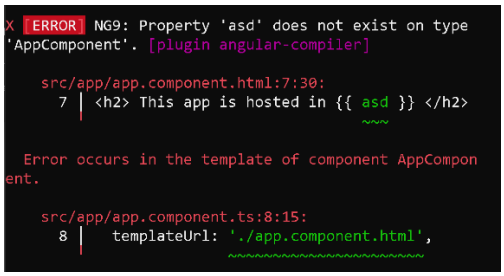
If you navigate to the Sources view, you can view the 2 Javascript files mentioned earlier. Further at the bottom, if you navigate to `src/app`, you will also be able to view the source code for your app (`app.component.html, app.component.ts`, etc)

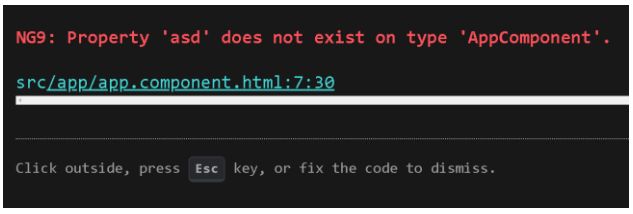



Purposely introduce an error into the component or the template and save it. For e.g. in `app.component.html`, try to reference a property that does not exist in the component.

```
<h2> This app is hosted in {{ asd }} </h2>
```

Notice that a compilation error is flagged in the console running the Angular development server, with more specific information on the error. The compilation error is also shown on the landing page of the Angular app.




Correct the error and save. The app should render normally again.

# 6 Property binding

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v2.ts`

`app.component-v2.html`

Place the following files from `changes` into `src/assets` (navigate to this folder in the Explorer View in VS Code and drag and drop the images)

```
cat.jpg
dog.jpg
horse.jpg
```

NOTE: If you are not able to see the specified animal image render correctly on the landing page of the app, stop the development server (Ctrl + C) and restart it again.

Make changes to the values of the properties and methods in AppComponent and notice that the changes are updated immediately to the app landing page. The Angular development server recompiles and updates the application automatically so that the new content is displayed in the browser view without the need to reload.

Notice that the style rule specified in `app.component.css` is applied to `app.component.html`, as discussed earlier

Template expressions are also used in property binding. In this case, they will appear between double quotes (unlike in interpolations, where they appear within {{ }}).

When template expressions are used, the value from the template expression is assigned to the property of a binding target. This target could be a HTML element, a component or a directive. It is important to keep in mind that when it is a HTML element, we are assigning the value to a property of the DOM object of that element and `*NOT*` a HTML tag attribute.

Attributes initialize DOM properties and you can configure them to modify an element's behavior. Properties are features of DOM nodes.

- A few HTML attributes have 1:1 mapping to DOM properties; for example, `id`.
- Some HTML attributes don't have corresponding DOM properties; for example, `aria-*`.
- Some DOM properties don't have corresponding attributes; for example, `textContent`.

In this example, we are binding to the various specific properties of the `h3, button, img` and `a` elements.

All HTML DOM elements have a set of common properties and methods:

In addition, each DOM element will have a further subset of properties and methods that are common to it, for e.g.

DOM for the <button> element

DOM for the <img> element

DOM for the <a> element

## 6.1 Interpolation vs property binding and their results

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v2-2.html`

`app.component-v2-2.ts`

In some situations, you can achieve equivalent results using interpolations and property binding.

Note that when you change the value of `isDisabled` in `app.component.ts`, there is no effect on the button where interpolation is used on the `disabled` attribute.

In general, if you are placing string values in HTML elements that accept text, then the best is use interpolation instead of property binding.
References:

https://angular.io/guide/binding-syntax#types-of-data-binding
https://angular.io/guide/binding-syntax#binding-types-and-targets
https://angular.io/guide/binding-syntax#data-binding-and-html
https://angular.io/guide/binding-syntax#html-attributes-and-dom-properties

# 7 Class binding

Class binding can be used for binding to a single CSS class value or to multiple class values

## 7.1 Single class binding

To create a single class binding, use the prefix class followed by a dot and the name of the CSS class—for example, `[class.sale]="onSale"`. Angular adds the class when the bound expression, `onSale` is truthy, and it removes the class when the expression is `falsy`—with the exception of undefined. Single class binding does not remove any existing classes on that element unless those classes have the same name as the one used in the binding.

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v3.ts`

`app.component-v3.html`

`app.component-v3.css`

The CSS stylesheet defines a variety of style rules based on the `class` selector

Change the values of `isThisForReal` and `isItDangerous` properties in the AppComponent and verify that the classes on the 2 paragraphs change accordingly.

For binding to multiple CSS classes, use `[class]="classExpression"`. The expression can be one of:

- A space-delimited string of class names.
- An object with class names as the keys and truthy or falsy expressions as the values.
- An array of class names.

## 7.2 Multiple class binding

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v3-2.ts`

`app.component-v3-2.html`

Verify that the classes on the 3 paragraphs change accordingly. Notice that for the 3rd case where there are two classes with conflicting style rules now applied to the <p> element simultaneously (`normal` and `medium`), the style rule for the class that appears last in the list of class names takes effect.

You can change the truthy/falsy values of the object used in the 2nd approach and verify that the correct classes appear on the affected element.

Note that class binding is not technically property binding per se, since for a HTML DOM element, there is no `class` property per se, only `className` and `classList`.

In addition to class binding, we can also use the NgClass built-in directive to add or remove multiple classes (to be covered in a later lab).

References:

https://angular.io/guide/attribute-binding#binding-to-the-class-attribute
https://www.netjstech.com/2020/04/angular-class-binding-with-examples.html

# 8 Style binding

Style binding is used to set the `style` attribute. This is one of the 3 different ways to apply CSS style rules to elements within a HTML document:

To create a single style binding, use the prefix style followed by a dot and the name of the CSS style property—for example, `[style.width]="width"`. Angular sets the property to the value of the bound expression, which is usually a string. Optionally, you can add a unit extension like `em` or `%`, which requires a number type.

For binding to multiple styles simultaneously, use `[style]="styleExpression"`. The `styleExpression` can be one of:

- A string list of styles such as `"width: 100px; height: 100px; background-color: cornflowerblue;"`.
- An object with style names as the keys and style values as the values, such as `{width: '100px', height: '100px', backgroundColor: 'cornflowerblue'}`.

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v4.ts`

`app.component-v4.html`

Verify that the style rules on the various elements involved change accordingly. You can change the value of the properties in the component referenced in the template and verify the changes are reflected in the DOM.

References:

https://angular.io/guide/attribute-binding#binding-to-the-style-attribute
https://www.tektutorialshub.com/angular/angular-style-binding/

# 9 Event binding

Event bindings use template statements which are executed when the target event occurs. In an event binding, Angular sets up an event handler for the target event. When the event is raised, the handler executes the template statement. The template statement typically invokes a component method which performs an action in response to the event, for e.g. storing a value from a HTML control element into a component property.

There are several common events for event binding when this is used on a standard HTML element (such as button, text, etc):

The binding conveys information about the event (the event payload) through an event object named `$event`. The type of the event object is determined by the target event. The `$event` object can be passed as a parameter to a component method in the template expression of the event binding. The parameter type will be identical to the event type used in the binding (for e.g. `click`, `input`, `submit`, `keydown`, etc) shown in:

We can use the base Event type for the parameter
https://developer.mozilla.org/en-US/docs/Web/API/Event
and then cast this to the appropriate type, for e.g.
https://developer.mozilla.org/en-US/docs/Web/API/InputEvent
https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent
https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent
and then access properties and methods for these types where appropriate.

## 9.1 Using event.target to access the element triggering the event

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v6.ts`

`app.component-v6.html`

Most of the time, we are interested in information regarding the element that triggered the event, rather than the event itself. All standard DOM event objects have a `target` property, a reference to the element that raised the event.

We need to cast `event.target` to the correct `HTMLxxxElement` type in order to extract the relevant properties of that particular element ( for e.g. the content for the case of most <input> elements).
For a standard input type element ( `<input type="text">` ), we will need to cast to HTMLInputElement:
For a button element (<button>), we will need to cast to HTMLButtonElement:
Since all these specific elements inherit from the generic HTMLElement, we can also access any of the properties here that might be useful for our purposes:

Switch to the Console view in DevTools to be able to see the console log output from the various methods that are invoked when the typing into the text fields, checking the checkbox and clicking on the button.

Note that most use cases for event binding for button will not pass the `$event` object since all that is required typically is just to know that a specific button has been clicked and to execute some logic to handle that event in the component method.

References:

https://angular.io/guide/event-binding#binding-to-events
https://angular.io/guide/event-binding-concepts#how-event-binding-works
https://angular.io/guide/event-binding-concepts#handling-events

https://www.tektutorialshub.com/angular/event-binding-in-angular

## 9.2 Template reference variables for event binding

The template reference variable can be used to reference any DOM element, component or a directive directly. It can then be subsequently accessed elsewhere in the template as well as being passed onto a method in the component.

It is often used as an alternative to the `$event` object in order to access the event content as it eliminates the need for knowledge and typing of the `$event` object prior to use (as is the case in event binding). The template variable when passed as a parameter to the component method through event binding is already of the correct `HTMLxxxElement` type and no further casting is required as is the case with the `$event` object

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v7.ts`

`app.component-v7.html`

References:

https://angular.io/guide/template-reference-variables#template-variables
https://angular.io/guide/template-reference-variables#syntax

### 9.3 Combining interpolation, property, class and event binding

We can combine the different types of binding we have covered so far to perform dynamic changes to the template DOM in response to events that occur. This will follow the basic pattern of:

- Using event binding to change some component property / template variable in response to a particular event
- Using property / class / style binding or interpolation to transfer the updated value from this property back to the appropriate element in the template

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v8.ts`

`app.component-v8.html`

`app.component-v8.css`

Note that an event binding is required for Angular to detect an event and perform its change detection process, where by Angular traverses the view hierarchy and updates the view. Sometimes, you may wish to just access the updated `value` property of the template variable in the template itself without actually calling a component method or updating a component property. In that case, you will need to include a "dummy" event binding statement ( `(keyup)="0"` ) :

## 10 Two way binding with ngModel directive

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v9.ts`

`app.component-v9.html`

In order to use the `ngModel` directive, you will have to import the `FormsModule` and add it to the `imports` list in the root component AppComponent

Two-way binding is useful for situations where the value of a property in a component is synchronized to the value of a HTML element (typically a form control like a text field), where by changes in the property are reflected in the value and vice versa.

This is usually accomplished by using the ngModel directive directly with the banana-in-the-box syntax to accomplish exactly the same functionality. This is the most common form of two-way binding syntax.

A more advanced form of this is to use ngModel and ngModelChange event and property binding separately if we do not want complete synchronization between the element value and the component property. For e.g. we might want to perform a preliminary operation on the element value (for e.g. changing its case) before storing it in the component property or vice versa. In this situation, the `$event` object directly refers to the latest value for the <input> element, so we can have an parameter of the appropriate type in the component method involved in the event binding without any need for additional preliminary type casting that needs to be done for the conventional event binding

ngModel is primarily used in template-driven forms. However, most of the time, we will be using the reactive approach to constructing forms as they are more comprehensive ability and flexibility compared to template driven forms.

References:

https://angular.io/guide/built-in-directives#displaying-and-updating-properties-with-ngmodel
https://www.netjstech.com/2020/04/angular-two-way-data-binding-with-example.html

# 11 NgModule based projects (Angular 15 and earlier)

In your main projects folder, generate a new project using the `--standalone` flag to create a project structure that is based on NgModule by default. This is the older project structure generated in Angular 15 and earlier.

```
ng new oldStyleDemo --standalone=false
```

Press enter to accept all the default values as usual.

Angular 16 and later versions introduce the concept of standalone components. This is now the default setting for newly generated Angular apps in Angular 17 and later.

Standalone components were introduced provide a simplified way to build Angular applications. Standalone components, directives, and pipes aim to streamline the development process by eliminating or reducing the need for NgModules. NgModule was originally introduced in Angular as a way to implement modularity in apps. Modules provide a way to group together related components.

In Angular 15 and earlier, newly generated Angular projects have at least one NgModule class, the root module, which is conventionally named AppModule. The root component (AppComponent) is also automatically included into this module. Furthermore, any new components that are generated (we will see how to do this in a later lab) are also included into AppModule by default.

In Angular 17 and later, AppModule is no longer present by default, and the root component (AppComponent) is a standalone component. Furthermore, any new components that are generated (we will see how to do this in a later lab) are also standalone components by default.

Create a new instance of VS Code (File -> New Window). The existing instance should still have the previous project folder open.
Using this new instance, open the root folder of the newly generated project `oldStyleDemo` (File -> Open Folder)

Check in `src/app`. Notice that we now have two files that are not present in the project folder of the previous project that we were working with (`bindingdemo`)

- `app.module.ts` - This is the root module which contains a reference to the root component (AppComponent). In the @NgModule decorator, it declares the various components that will be active in the module (currently only AppComponent), specifies the

modules / packages that are imported into the current module, the providers and also specifies AppComponent as the component to bootstrap the application.

- `app-routing.module.ts` - This is the routing module that specifies the routes that will be used by the app. This is generated in all new projects and imported by default into `app.module.ts`.

The functionality of the two modules above is provided in a slightly simpler form in the newer project structure (Angular 17 and later) via the files `app.routes.ts` and `app.config.ts`

In the older project structure (`oldStyleDemo`) in `src`, we also have a `main.ts` which specifies the bootstrapping sequence of the application.

Compare the `src/main.ts` of the older project structure (`oldStyleDemo`) with the `src/main.ts` of the current project structure (`bindingDemo` )

- Older project structure `main.ts` - bootstraps the root module, which in turn loads the routing module and the root component
- Newer project structure `main.ts` - bootstraps the root component and app configuration (`app.config.ts`) directly. The app configuration in turn loads the routing information.

We will now start a development server for the `oldStyleDemo` project. The development server will need to run on a different port from 4200, as we will leave the development server for the `bindingDemo` project running.

Open a DOS command prompt in the root folder of the `oldStyleDemo` project and type

```
ng serve --port 4400
```

and check that the app is served up at `localhost:4400` in a new browser tab.

NOTE: If using port 4400 results in an error on your machine, try another random port number (e.g. 4600, 4800, etc) that does not.

We are now going to replicate the changes that we made in the previous lab for the `bindingdemo` project to incorporate two-way binding using ngModel into the `oldStyleDemo` project

Modify the following files in `src/app` of the `oldStyleDemo` project with the latest update from `changes`:

```
app.component-v10.ts
```

```
app.component-v10.html
```

```
app.module-v10.ts
```

Check that the functionality is identical to that of the `bindingdemo` project in the previous lab.

Notice the key differences between the new project structure (`bindingdemo`) Angular 17 onwards and old project structure (`oldStyleDemo`) Angular 15 and earlier.

In new project structure (Angular 17 onwards),
in `app.component.ts`

- The FormsModule (which is required for the NgModel directive) is directly imported to be used
- The `standalone` property exists and is set to true in the @Component decorator

In old project structure (Angular 15 earlier),
in `app.component.ts`

- There is no imports or `standalone` property

in `app.module.ts`

- The import of the FormsModule (which is required for the NgModel directive) as well as AppRoutingModule occurs here.

## 11.1 Migrating older project structure (Angular 15 earlier) to current project structure

Angular CLI provides a way to facilitate migrating an older project structure (Angular 15 earlier) to the current project structure (Angular 17 onwards):

Stop the development server for the `oldstyle` project and close the VS code editor instance which has this project open in it.

In the command prompt of the root folder of the `oldstyle` project, type:

a) `ng g @angular/core:standalone` and select "Convert all components, directives and pipes to standalone"
b) Repeat the same command again (`ng g @angular/core:standalone`) but this time select "Remove unnecessary NgModule classes"
c) Repeat the same command again (`ng g @angular/core:standalone`) but this time select "Bootstrap the project using standalone APIs"

Open the `oldstyle` project folder in a new VS code editor instance, and now verify that the project structure and source code file content is identical to that of the new project structure of `bindingdemo`.