SYST35300
Hybrid Mobile App Development

Presentation created for Sheridan College by A. Pak

Agenda

- Mobile Development Options
- Angular Review

Mobile Development Options

- Web Apps
- Progressive Web Apps
- Native Apps
- Hybrid Apps

Mobile Development Options

- Web Apps
  - Use standard web technologies – typically HTML5, JavaScript and CSS.
  - Launch from servers.
  - Certain limitations such as push notifications, secure offline storage; and access to native device functionalities such as camera, calendar, contact, geolocation, etc.

Mobile Development Options

- Progressive Web Apps (PWA)
  - Web Apps that use emerging web browser APIs and features along with traditional progressive enhancement strategy to bring a native app-like user experience to cross-platform web applications.
  - Can be "installed" on device giving the web apps the same user experience as native apps.
  - Able to work "offline".
  - Not able to access native device functionalities.

https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

Mobile Development Options

- Native Apps
  - Specific to a given mobile platform (iOS, Android).
  - Use the development tools and language for the respective platforms.
    - iOS – Swift / Objective-C
    - Android – Java / Kotlin
  - Look and perform the best.

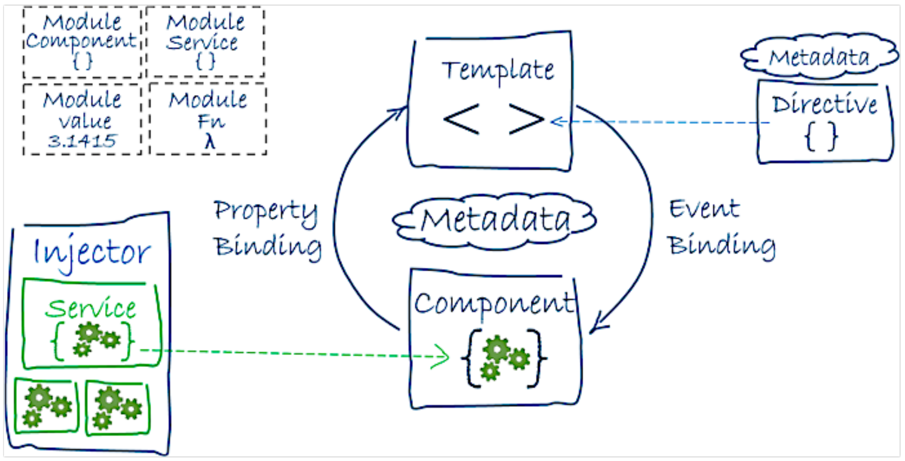
Mobile Development Options

- Hybrid Apps
  - To embed HTML5 apps inside a thin native container.
  - Combine the advantages of native and HTML5 apps.
  - Hybrid Mobile app frameworks – Ionic, Framework7, Kendo UI, NativeScript, Onsen UI, React Native, etc.

Angular Review

Angular Concept

- Angular builds applications in HTML and either JavaScript or TypeScript that compiles to JavaScript.
- Angular applications are composed of *HTML templates* with Angularized markup; component classes to manage those templates; *services* to apply application logic; and *modules* that bind them together.



Angular Component

NgModule-based vs Standalone Components:

- The NgModule-based component system consolidates components, directives, and pipes into cohesive blocks of functionality.
- Angular introduced standalone components in version 14, which are components that don't belong to any specific Angular module.
- By using standalone components, developers gain more flexibility and control over their codebase. They can create small, focused components that are easier to understand, test, and maintain, with dependencies imported directly into the component.
- Starting with CLI version 17, applications will be generated using standalone components by default.

In this course, we still use the Angular's standalone system.

Angular Component

- Components are the main building block of an Angular application, and an application may have any number of Components.
- A Component controls a particular view of the application with its own logic and data inside a class.
- A Component consists of a template, class and metadata data defined with the @*Component* decorator. Components, directives, and pipes must be included in the 'imports' array to be available to the component.
- Components use *services*, which provide specific functionality not directly related to views.
- The @*Component* decorator is imported from the @angular/core library.

Angular Component

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { InputComponent } from './custom-input.component';

@Component({
  standalone: true,
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css'],
  imports: [CommonModule, FormsModule, InputComponent],
})
export class ExampleComponent {}
```

The **imports** property of the @Component decorator is used to explicitly declare the **standalone components, directives, and pipes** that the component depends on.

The CommonModule provides commonly used directives, pipes, and services used in Angular applications.

Angular Service

- A *Service* is used to define any function, feature or data for your application.
- A Service is defined as a class with the @*Injectable* decorator.
- Typically a Service is defined with a narrow, well-defined purpose to do something specific.

Angular Module

NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

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

@NgModule ({
            declarations:[ ],
            imports:[ ],
            providers:[ ],
            exports:  [ ]
})
export class xxxModule{  }
```

Angular Module

- The *@NgModule* decorator is used to define the imports, declarations, exports, and other options.
  - declarations - components, directives, and pipes that belong to this module.
  - imports - classes from other modules that are needed in *this* module.
  - providers - creators of services that are accessible in all parts of the module.
  - exports - components, directive, and pipes to be shared with other modules.

Installations

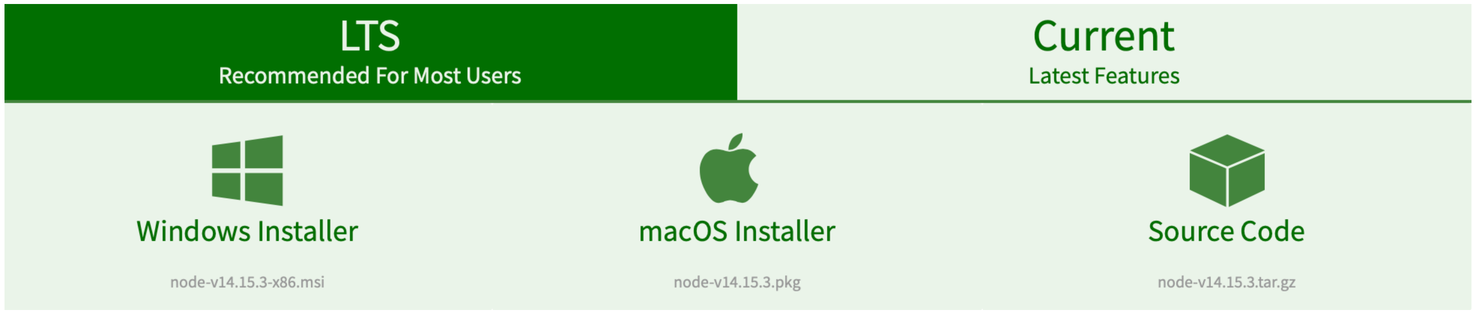
Node.js / Node Package Manager (NPM)

Angular CLI / Angular Command Line Interface

IDE / Visual Studio Code

Images taken from freebiesupply.com

Node Install



Node / NPM

Go to https://nodejs.org/en/

Download and install **"Recommended for Most Users"** version

To check Node version:

```
node --version
npm --version
```

Angular CLI Install

Install Angular CLI

- Open Command Prompt as an ADMINISTRATOR

```
npm install -g @angular/cli@latest
```

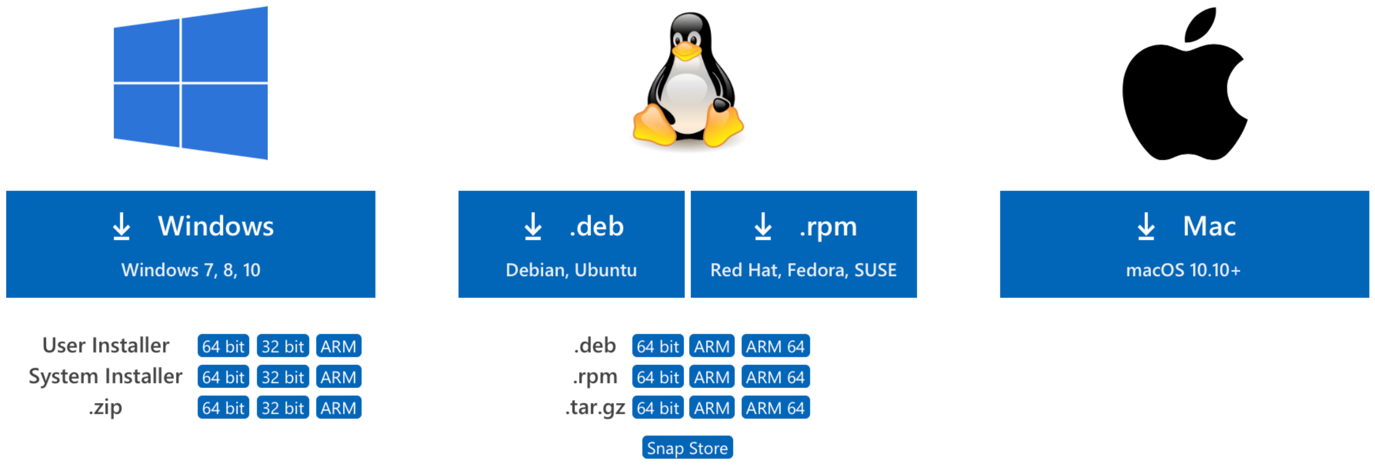
If error during install:

- `npm uninstall -g angular-cli @angular/cli`
- `npm cache verify`
- `npm cache clean --force`
- `npm config rm proxy`
- `npm config rm https-proxy`
- `npm install -g @angular/cli@latest`

To check version of Angular:

```
ng version
```

Visual Studio Code Install



Windows
Windows 7, 8, 10

.deb
Debian, Ubuntu

.rpm
Red Hat, Fedora, SUSE

Mac
macOS 10.10+

User Installer	64 bit	32 bit	ARM
System Installer	64 bit	32 bit	ARM
.zip	64 bit	32 bit	ARM

.deb	64 bit	ARM	ARM 64
.rpm	64 bit	ARM	ARM 64
.tar.gz	64 bit	ARM	ARM 64

Snap Store

Visual Studio Code

Go to code.visualstudio.com/Download

Download and install

New Angular Project

To create a new Angular project using the command line (as an Admin) , use the following command:

```
ng new newAppName
```

ng	Angular CLI
new	Command to create a new project
newAppName	Name of the project

- `Which stylesheet format would you like to use? CSS`
- `Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? Y`
- `Would you like to use the Server Routing and App Engine APIs (Developer Preview) for this server application? N`

This command creates a new Angular project with standalone components.

- May take several minutes to set up ... *be patient*
- *NOTE: Ctrl + C will terminate Node*

Run Angular Project

```
ng serve --open
```

NOTE: This step can be perfromed from the Terminal window either inside or outside of Visual Studio Code.

By default, the Chrome browser will open automatically at http://localhost:4200.

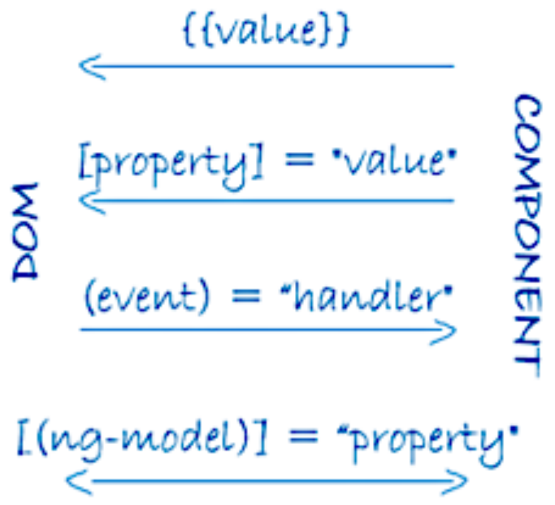
Module / Component / Service

From the command line (Admin), navigate to your Angular project folder:

- To generate a **module:**
  ```
  ng generate module xxxx
  ```
- To generate a **component:**
  ```
  ng generate component xxxx --skip-tests
  ```
- To generate a **service:**
  ```
  ng generate service xxxx --skip-tests
  ```

Data Binding

- Data binding is one of Angular's important features to connect properties in Angular class to html tags
- There are four forms of data binding syntax. Each form has a direction — to the DOM, from the DOM, or in both directions.



1. String Interpolation
2. Property binding
3. Event binding
4. Two-way data binding

Data Binding

One-Way Data Binding from the **property to the DOM.**

Changes made to the property values are automatically assigned to the HTML elements, attributes specified through the data binding notation.

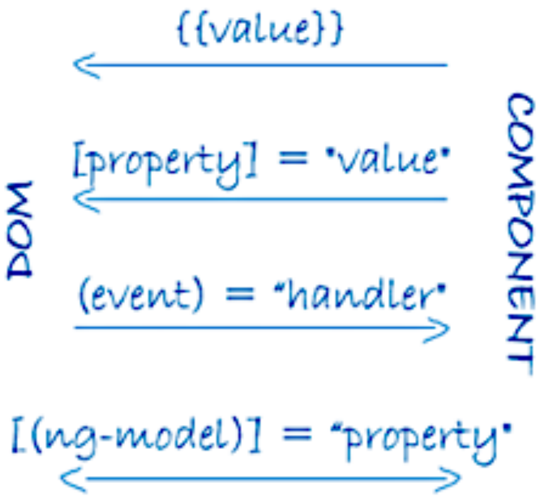
- **Interpolation**

```
{{ propertyName }}
```

- **Property binding** allows you to bind the property of a class to an element, attribute, component, or directive.

```
[ target ] = "propertyName"

bind-target = "propertyName"
```

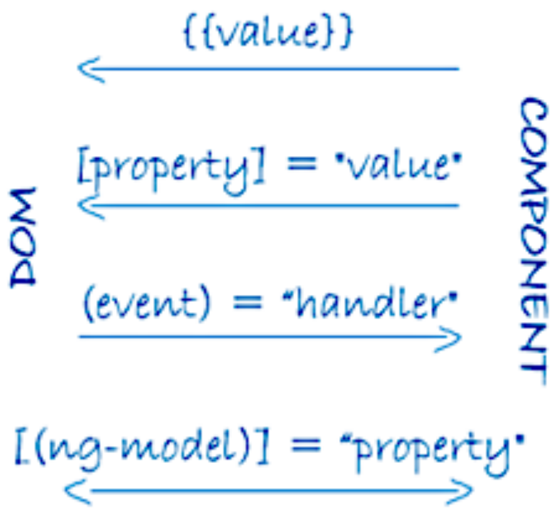


Data Binding

One-Way Data Binding from the **DOM to the property.**

Event binding allows you to define events that occur in the template (user-initiated), and communicate to the component class.

```
( event ) = "method()"    OR
on-event  = "method()"
```

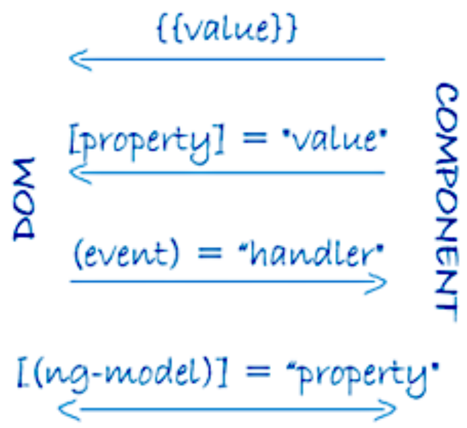


Data Binding

Two-way data binding, using the ngModel directive, binds a property to an HTML textbox, enabling data to flow ***in both directions***: from the property to the textbox, and from the textbox back to the property.

The **`ngModel`** directive creates a FormControl instance that tracks the value, user interaction, and validation status of the control, ensuring the view synced with the model.

```
[( ngModel )] =  "propertyName"
bindon-ngModel = "propertyName"
(ngModelChange) = "method()"
```



Needs to import *FormsModule* from *@angular/forms*

Into the component's ts file.

String Interpolation

String Interpolation:

- {{ }} MUST return a string.
- *Examples:*
  - A hard-coded string: **`{{"John Doe"}}.`**
  - A string from a variable: **`{{sname}}`**, where **`sname`** is initialized in the .ts file **`sname="John Doe"`**;
  - A string returned from a method: **`{{getStudentName()}}.`**

String Interpolation

- Create a new Angular project named **w1-inclass1.**
- Generate a component named **header.**
- Update header.component.html:

```
<!-- myFirstApp: header component / change -->
<h1>Angular App / By: {{ sname }}</h1>
```

- Update header.component.ts:

```
<-- Add string data declaration before constructor -->
sname = 'John Doe';
```

- Import the header component into the root component.
- Include the header selector in the root component's HTML file.
- Go to browser to see results.

String Interpolation

- Create a class or an interface named **student.ts** in the app folder to hold multiple properties and make it accessible across all components:

```
export class (or interface) Student {
        sid!: number;
        sname!: string;
        scampus: string | undefined;
        slogin: string | undefined;
}
```

String Interpolation

- Edit header.component.ts:
  - Import the Student class / interface:

```
import { Student } from '../student';
```

  - Create a local property in the component using the class / interface as the type:

```
currStudent: Student =
  { sid: 991000000, sname: 'John Doe',
    scampus: 'Davis', slogin: 'doej' };
```

- Edit header.component.html:

```
<!-- myFirstApp: header component -->
<h1>Angular App / By: {{currStudent.sname}}</h1>
```

String Interpolation

- Generate a component named **content.**
- Import the content component into the root component.
- Include the content selector in the root component's HTML file.
- Edit the content.component.ts file:
  - Import the **Student** class / interface.
  - Create a local property named ***currStudent*** with the same values as in the Header component.
- Render the content of ***currStudent.name*** in the content.component.html file and format it to uppercase using a pipe, such as:

```
{{currStudent.sname | uppercase}}
```

- Remember to include **CommonModule** in the imports array in the content.component.ts file.

Property Binding

Bind the element's properties using property binding:

- Add the following to content.component.html.

```
<div class="inputArea">
   <h3>Data Binding Demonstration</h3>
   <p>{{city}}</p>
   <label><input [placeholder] = "city">
   </label><br>
   <label><input bind-placeholder = "city">
   </label><br><br>
</div>
```

- Add to content.component.ts.

```
city = 'Brampton';
```

Property Binding

- More property binding examples:

```
<img [src]="image" width="150px">

<p [innerHTML]="details"></p>
```

- Component's .ts file:

```
details = 'New paragraph';
image = 'xxx';
```

Note: Place the image in the '**public**' folder.

Event Binding

- Create a **click** event on a button to call a function in the HTML file:

```
<button [disabled]="city === ''"
        (click)="clearCity()">Reset City
</button>
```

- Add the function to the .ts file:

```
clearCity() { this.city = ''; }
```

Event Binding

More event bindings:

- Edit the HTML file:

```
<h3 on-mouseover="mouse()"
   (mouseout)="mouse2()">Data Binding Demonstration
</h3>
```

- Add to the .ts:

```
mouse() { this.city = 'Mississauga'; }
mouse2() { this.city = 'Toronto'; }
```

2-way Data Binding

- Data flows from the model (component data) to the view, and then back from the view to the model:

```
                      <input [placeholder] = "city"
Property binding →          [value] = "city"
Event binding    →          (input) = "city = $any($event.target).value">
```

- Using the **ngModel** directive:

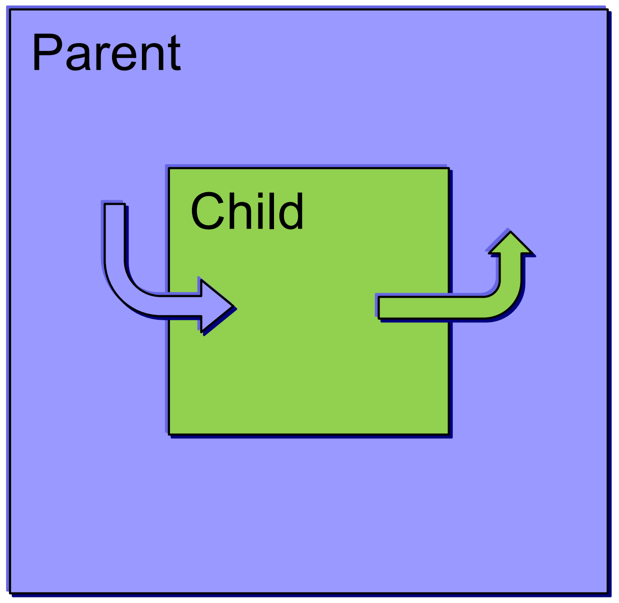
```
<input bind-placeholder = "city"
      [(ngModel)] = "city">
```

- Import FormsModule into the component's ts file:

```
import { FormsModule } from '@angular/forms';
imports: […, FormsModule]
```

Data Sharing

Data sharing between components with **Parent / Child Relationship**



Data Sharing

From Parent to Child Components

Uses the @Input() decorator in the child component to receive data from the parent component.

Child component.ts:

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

@Input() childMsg:any;
```

Child component.html:

```
{{childMsg}}
```

Parent component.html:

```
<app-child [childMsg]="'This is the message'"></app-child>
```

Data Sharing

From Child to Parent Components

Child component:

Uses the @Output() decorator in the child component to pass data to the parent component by emitting it via EventEmitter.

Child component.html:

```
<button (click)='sendMsg()'>Send Message</button>
```

Child component.ts:

```
import { EventEmitter, Output } from '@angular/core';

@Output() childMsg = new EventEmitter();

sendMsg() {
   this.childMsg.emit('Hello!');
}
```

Data Sharing

From Child to Parent Components (cont.)

Parent component:

Uses a function to bind to the child's event to get the data.

Parent component.html:

```
<app-child (childMsg)='getMsg($event)'></app-child>
{{myMsg}}
```

Contains the data from the child component

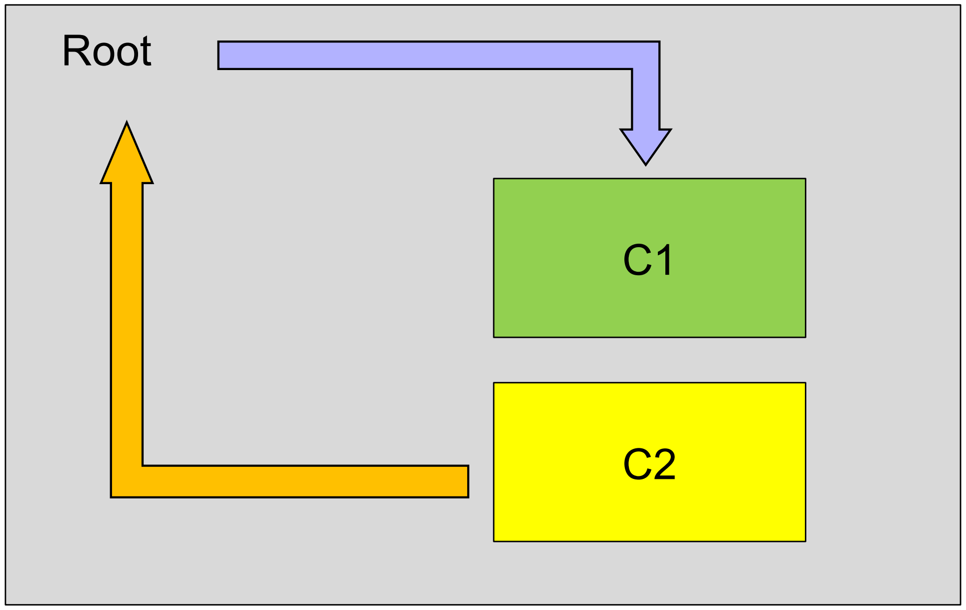
Parent component.ts:

```
myMsg: any;

getMsg($event:any) {
    this.myMsg = $event;
}
```

In-Class Exercise

1. Create a standalone project "w1-inclass2"
2. Create two components "c1" and "c2"
3. Pass data from root component to "c1"
4. Pass data from "c2" to root component



In-Class Exercise Cont…

1. Create a standalone project "w1-inclass2"
2. Create two components "c1" and "c2"
3. Pass data from root component to "c1"
4. Pass data from "c2" to root component

1. `ng new w1-inclass2`
2. `ng g component components/c1   ng g component components/c2`
3. Pass data from root component to "c1"
   - Update c1.component.ts

     ```
     import { Input } from '@angular/core';
     @Input() c1Data: any;
     ```
   - Update c1.component.html

     ```
     {{c1Data.class}} - {{c1Data.desc}}
     ```
   - Update app.component.ts

     Import c1 and c2 components.

     ```
     homeData = {class: 'SYST35300', desc: 'Hybrid Mobile Dev'};
     ```
   - Update app.component.html

     ```
     <app-c1 [c1Data]="homeData"></app-c1>
     ```

In-Class Exercise Cont…

1. Create a standalone project "w1-inclass2"
2. Create two components "c1" and "c2"
3. Pass data from root component to "c1"
4. Pass data from "c2" to root component

4. Pass data from "c2" to root component
   - Update c2.component.ts

```
import { EventEmitter, Output } from '@angular/core';

@Output() childMsg = new EventEmitter();
c2Data = {fName: 'Andy', lName: 'Pak'};
sendMsg() { this.childMsg.emit(this.c2Data); }
```

   - Update c2.component.html

```
<button (click)='sendMsg()'>Send Message</button>
```

In-Class Exercise Cont…

1. Create a standalone project "w1-inclass2"
2. Create two components "c1" and "c2"
3. Pass data from root component to "c1"
4. Pass data from "c2" to root component

4. Pass data from "c2" to root component (cont.)
   - Update app.component.ts

```
myMsg = {fName: '', lName: ''};
getMsg($event: any) {
    this.myMsg = $event;
}
```

   - Update app.component.html

```
<app-c2 (childMsg)='getMsg($event)'></app-c2><br>
{{myMsg.fName}} {{myMsg.lName}}
```