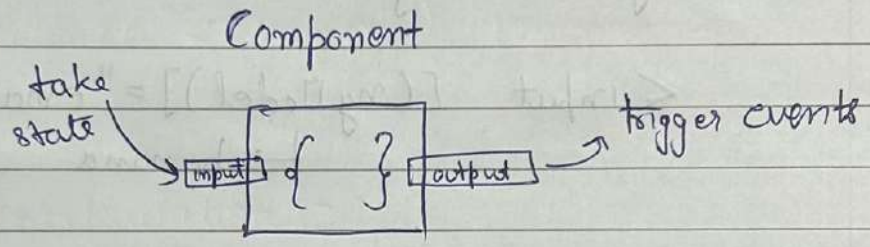
→ Building Reusable Components :-

↳ Pass Data to Components
↳ Raise custom events
↳ Apply styles
↳ shadow DOM
↳ View Encapsulation

* Component API :-



* Input Properties :-

when we want to alter / add data to a component field while calling or mentioning that component .html or DOM, we have to use , input properties

@ Input () decorator

Eg @ Input() <<var>> : <<type>>;
inside the component

→ Aliasing input property :-

@ Input (" Alias name")
↳ this can be used inside dom to pass value.

__/__/__

→ OutPut Properties :-

These are used to raise any custom event we want to map with our component.

< component (<< event name >>) = "fun()" >

↳ this component is defined anywhere

& in component.ts

@ Output () << event name >> = new EventEmitter();

& to emit the event →

this. << event name >>. emit();

* Passing Data while emiting Event :-

→ If we want to pass data or state while emiting

this. << event name >>. emit (data)

& add same parameter in function

&

pass "$ event" in event handler in DOM.

* we can pass any type of data and

__/__/____

Verify data using interfaces

* Aliasing in output property :-

If we want to change variable name of output event in future without changing where its called, we can give alias to the same by

@Output("<<name>>") var name = new EventEmitter();

* Views Encapsulation :-

↳ shadow DOM :-
Allow to apply scoped styles to elements

In component properties

encapsulation : ViewEncapsulation.Emulated

* using this global styles won't affect the component styles.

* ng Content :-
To pass mark up directly while defing component we can send using ng content

__/__/____

and can verify or match using selectors.

Eg. markup (.html)

<ng-content select=" .class name >> " ></ng-content>

&

while calling component

<component>
<h1 class = " << class name >> ">
"content" </h1>
</component>

& this h1 will be replaced or injected for <ng-content tag>

* Same can be done for containers using "ng-container" tag

<ng-container> </ng-container>.