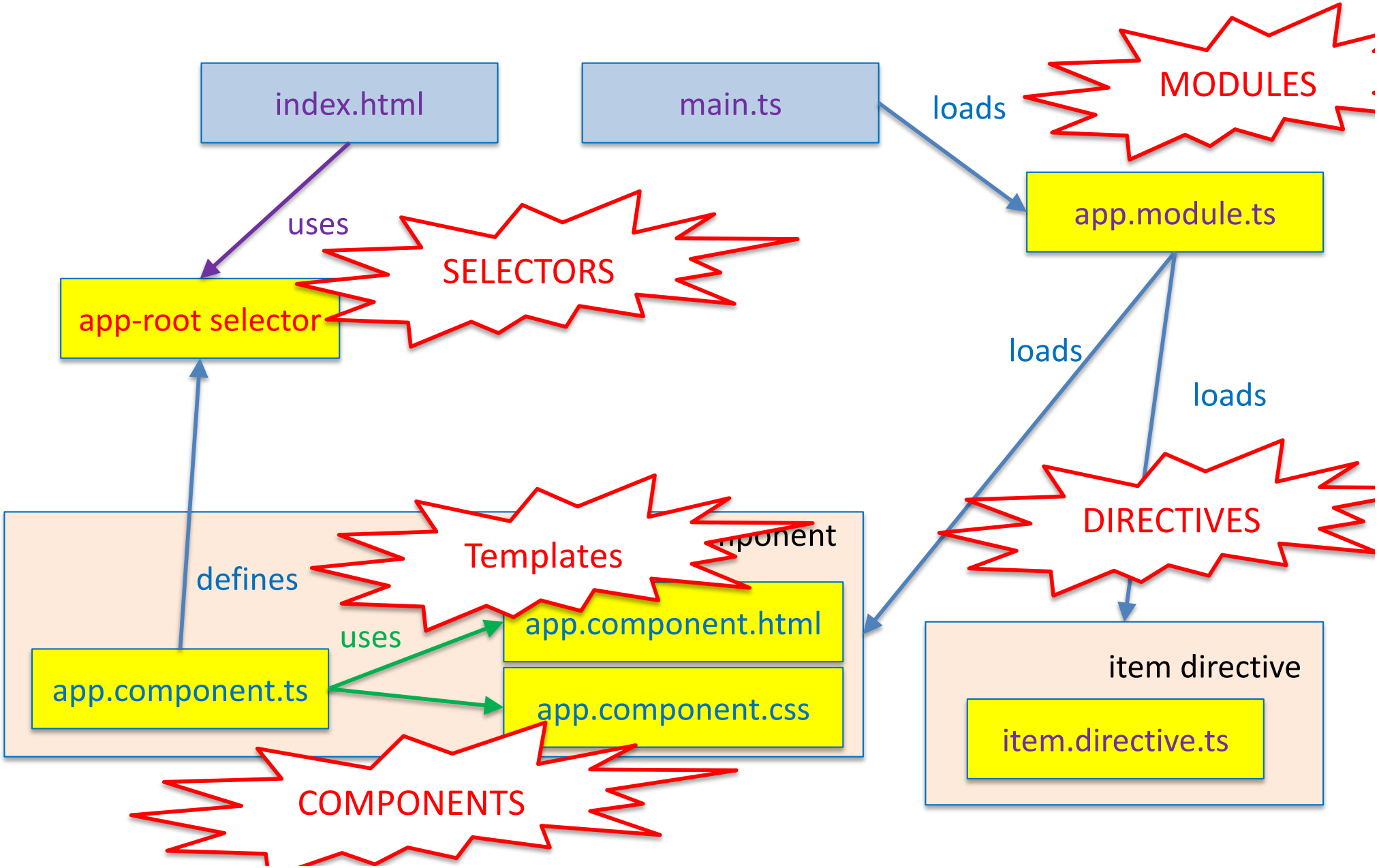
Q

Which are the architectural elements?

Putting the pieces together



Components and templates

A component controls a patch of screen. It contains

- the data
- user interaction logic that defines how the View looks and behaves.

A Component consists of three main building blocks

- Template
- Class
- MetaData

The component has a representation (view), which is defined by a template. A template is portion of HTML that tells Angular how to render the component.

a component:

needs an import

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

needs a decorator, with binding to a template

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

defines a class, with properties and behaviours

```
export class AppComponent {
  aProperty = 'app works!';
  doSomething();
}
```

ng secondProject

CREATE secondProject/README.md
CREATE secondProject/.editorconfig
CREATE secondProject/.gitignore
CREATE secondProject/angular.json
CREATE secondProject/package.json
CREATE secondProject/tsconfig.json
CREATE secondProject/tslint.json
CREATE secondProject/.browserslistrc
CREATE secondProject/karma.conf.js
CREATE secondProject/tsconfig.app.json
CREATE secondProject/tsconfig.spec.json

CREATE secondProject/src/favicon.ico (948 bytes)
CREATE secondProject/src/index.html (299 bytes)
CREATE secondProject/src/main.ts (372 bytes)
CREATE secondProject/src/polyfills.ts (2826 bytes)
CREATE secondProject/src/styles.css (80 bytes)
CREATE secondProject/src/test.ts (753 bytes)
CREATE secondProject/src/assets/.gitkeep (0 bytes)
CREATE secondProject/src/environments/environment.prod.ts
CREATE secondProject/src/environments/environment.ts
CREATE secondProject/src/app/app-routing.module.ts
CREATE secondProject/src/app/app.module.ts
CREATE secondProject/src/app/app.component.css
CREATE secondProject/src/app/app.component.html
CREATE secondProject/src/app/app.component.spec.ts
CREATE secondProject/src/app/app.component.ts
CREATE secondProject/e2e/protractor.conf.js
CREATE secondProject/e2e/tsconfig.json
CREATE secondProject/e2e/src/app.e2e-spec.ts (664 bytes)
CREATE secondProject/e2e/src/app.po.ts (274 bytes)

By Convention, the file name starts with the **feature name** (app) and then followed by the **type of class** (component),separated by a dot. The extension used is **ts** indicating that this is a typescript module file.

Q

Where is a component loaded?

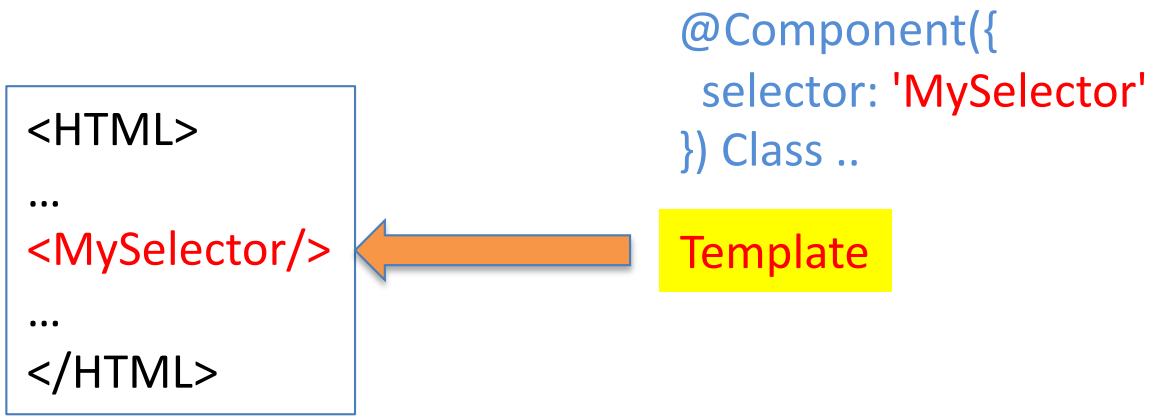
a component defines a selector:

When we build Angular Components, we are actually building new HTML elements.

We specify the name of the HTML element in the selector property of the component metadata.

Then we use it in our HTML: Angular,

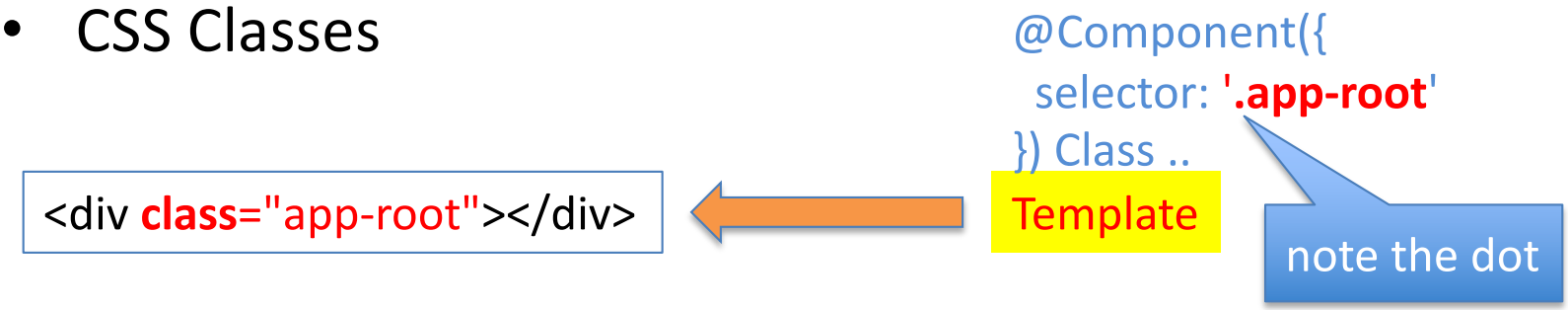
- searches for the selector in the HTML file
- instantiates the component
- renders the Template associated with the component inside the selector.



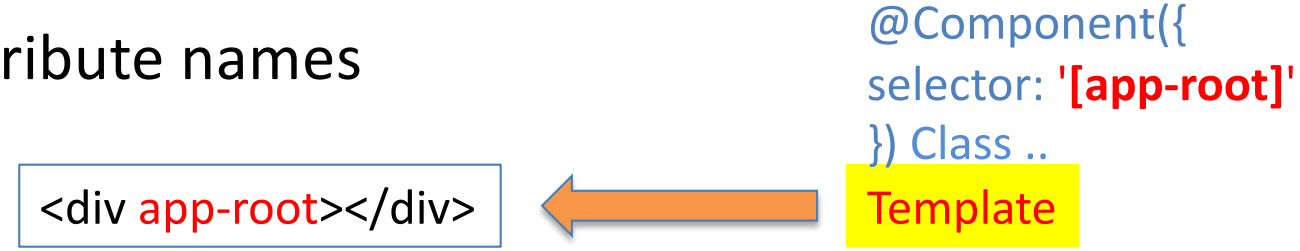
Other types of associations for a selector:

Selectors can be used for:

- CSS Classes



- Attribute names



- Attribute name and value

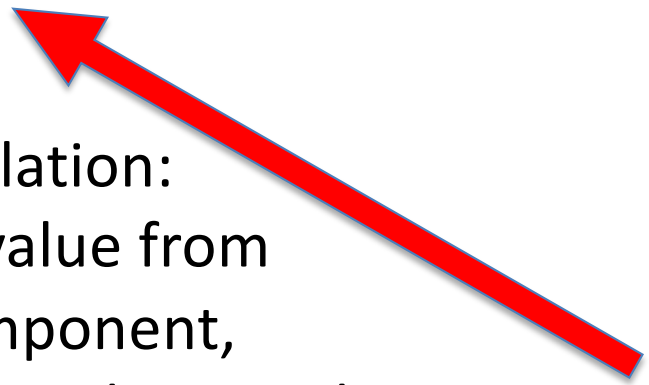
Q

How do we define a template?

Template

it is a standard HTML file, with binding to the Component and Angular directives

```
<tr *ngFor="let customer of customers;">
   <td>{{customer.name}}</td>
</tr>
```



Interpolation:
take a value from
the component,
put it into the template

Component

…
Customer x;
x.name="Pippo"

Angular directives

Structural directives can change the DOM layout by adding and removing DOM elements.

they are: **ngIf, ngFor, ngSwitch**

All structural Directives are preceded by Asterix symbol.

Attribute directives or **Style** directives can change the appearance or behavior of an element.

they are: **ngModel, ngClass, ngStyle**

Structural directives in templates: ngIf

in component:

```
export class AppComponent {
  showMe: boolean;
}
```

in template:

```
<p *ngIf="showMe">
  ShowMe is checked
</p>
```

also:

- ngIf else
- ngIf then else

see details and examples in
https://www.tektutorialshub.com/angular/angular-ngif-directive/

Structural directives in templates: ngSwitch

in component:

```
export class AppComponent {
  num: number= 0;
}
```

in template:

```
<div [ngSwitch]="num">
     <div *ngSwitchCase="'1'">One</div>
     <div *ngSwitchCase="'2'">Two</div>
     <div *ngSwitchCase="'3'">Three</div>
     <div *ngSwitchCase="'4'">Four</div>
     <div *ngSwitchCase="'5'">Five</div>
     <div *ngSwitchDefault>This is Default</div>
   </div>
```

see details and examples in
https://www.tektutorialshub.com/angular/angular-ngswitch-directive/

Structural directives in templates: ngFor

in component: define customer, and customers as an array of customer items

in template:

```
<tr *ngFor="let customer of customers;">
   <td>{{customer.customerNo}}</td>
   <td>{{customer.name}}</td>
   <td>{{customer.address}}</td>
   <td>{{customer.city}}</td>
   <td>{{customer.state}}</td>
</tr>
```

see details and examples in
https://www.tektutorialshub.com/angular/angular-ngfor-directive/

Q

How does the class pass values to the template?

interpolation

The simplest form of binding is interpolation (Template Expression).

Interpolation carries values from the class to the template.

Changes in property values are reflected in the interpolation.

Values must be strings, and are substituted to the interpolation.

```
//Template
{{title}}
{{getTitle()}}

//Component
title = 'Angular Interpolation Example';
getTitle(): string {
    return this.title;
}
```

See https://www.tektutorialshub.com/angular/interpolation-in-angular/

interpolation

Interpolation can also evaluate simple expressions, such as e.g. string concatenation or arithmetic operations. They can bind to any property that accepts a string.

```
<span> {{3*8}} </span>
<p>Show me <span class = "{{giveMeRed}}">red</span></p>
<p style.color={{giveMeRed}}>This is red</p>
```

Interpolation cannot cause effects in the state of the Component: no assignments, no instantiation of classes, no side effects.

They are the simplest form of one-way binding:

Component  Template

interpolation

Interpolation can be applied also to bind to a value of a property of an HTML element. We can bind to any property that accepts a string.

```
<p>Show me <span class = "{{giveMeRed}}">red</span></p>
<p style.color={{giveMeRed}}>This is red</p>


<a href="/product/{{productID}}">{{productName}}</a>
```

https://www.telerik.com/blogs/understanding-angular-property-binding-and-interpolation

property binding

To bind a property of an HTML element there is also a different syntax, e.g.

```
<img src="{{ myProperty }}">
<img [src]="myProperty">
```

```
<h1 [innerText]="title"></h1>
<button [disabled]="isDisabled">I am disabled</button>

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title="Angular Property Binding Example"
  isDisabled= true;
}
```

property binding

interpolation "injects" the value into the html, so when you say value="{{ hello }}" Angular is *inserting* your variable between the brackets. It can only be applied to strings.

property binding allows Angular to directly access the elements property in the html. this is a deeper access. When you say [value]="hello" Angular is grabbing the value property of the element, and *setting* your variable as that property's value.
It can be applied to any data type (e.g. objects).

special cases of property binding: class

Normal html:

```
<div class="red">red</div>
```

Property binding:

NOTE the double + single quote!

```
<div [className]="'red'">Test</div>
<div [className]="'red size20'">Test</div>
<div [ngClass]="'red size20'">
<div ngClass='red size20'>
```

NOTE the single quote only!

The ngClass attribute directive is more powerful, you can use arrays, objects and conditional assigments.

see https://www.tektutorialshub.com/angular/angular-ngclass-directive/

and https://www.tektutorialshub.com/angular/property-binding-in-angular/

special cases of property binding: style

Normal html:

```
<body style="background-color:grey;">
```

Property binding:

NOTE the double + single quote!

```
<p [style.background-color]="'grey'">
<p [style.color]="getColor()"
   [style.font-size.px]="'20'"
   [style.background-color]="status=='error' ? 'red': 'blue'">
   paragraph with multiple styles
</p>
```

.

see https://www.tektutorialshub.com/angular/angular-style-binding/

special cases of property binding: ngStyle

The Angular ngStyle directive allows us to set the many inline style of a HTML element using an expression. The expression can be evaluated at run time allowing us to dynamically change the style of our HTML element

```
<div [ngStyle]="{'color': color}">
```

for details, see
https://www.tektutorialshub.com/angular/angular-ngstyle-directive/

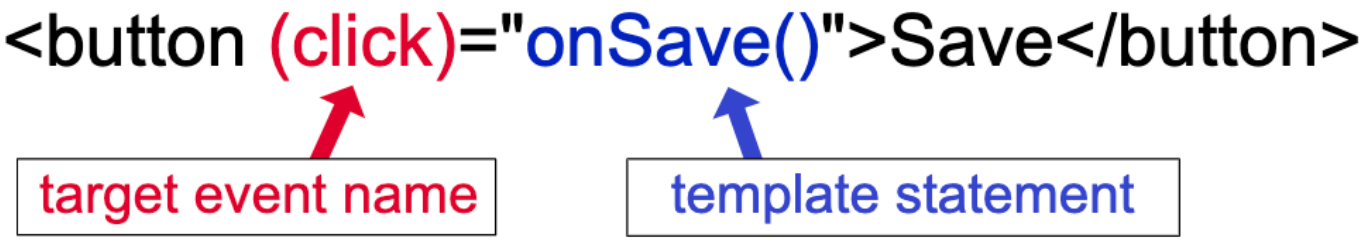
Q

How does the view pass values to the class?

Event binding

We can bind events such as keystroke, clicks, hover, touch, etc to a method in component.
It is one way from view to component.

For Example, when the user changes a input in a text box, we can update the model in the component, run some validations, etc. When the user submits the button, we can then save the model to the backend server.



Event binding

DOM Events carries the event payload. I.e the information about the event. We can access the event payload by using $event as an argument to the handler function.

template

```
<input (input)="handleInput($event)">
<p>You have entered {{value}}</p>
```

component

```
value=""
handleInput(event) {
  this.value=event.target.value
;
}
```

Event binding - example

app.component.ts

```
@Component({
    selector: 'app-root',
    templateUrl:
'./app.component.html',
    styleUrls:
['./app.component.css']
})
export class AppComponent {
    title = 'app works!';
    value=""
        handleInput(event) {
            this.value=event.t
arget.value;
        }
}
```

app.component.html

```
<h1>
  {{title}}
</h1>
<input
(input)="handleInput($event)"
>
<p>You have entered
{{value}}</p>
```

Event binding

The events are those of HTML DOM component:
http://www.w3schools.com/jsref/dom_obj_event.asp
by just removing the on prefix. onclick ---> (click)

- (focus)="myMethod()"
- (submit)="myMethod()"
- (cut)="myMethod()"
- (paste)="myMethod()"
- (keypress)="myMethod()"
- (mouseenter)="myMethod()"
- (mouseup)="myMethod()"
- (dblclick)="myMethod()"
- (dragover)="myMethod()"
- (blur)="myMethod()"
- (scroll)="myMethod()"
- (copy)="myMethod()"
- (keydown)="myMethod()"
- (keyup)="myMethod()"
- (mousedown)="myMethod()"
- (click)="myMethod()"
- (drag)="myMethod()"
- (drop)="myMethod()"

There are actually many more. see:

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

Event binding

Instead of parentheses, you can also use the on- syntax:

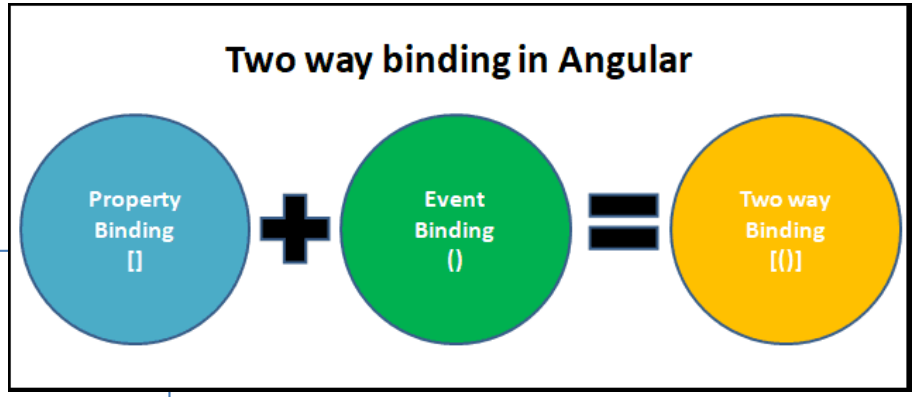
```
<button on-click="clickMe()">Click Me</button>
```

for details, see
https://www.tektutorialshub.com/angular/event-binding-in-angular/

Q

Can we have data passing in the two directions (view to class/class to view)?

two-way binding



Component:

```
value="";
clearValue() {
  this.value="";
}
```

Template:

```
<input type="text"
[(ngModel)]="value">
<p> You entered
{{value}}</p>
<button
(click)="clearValue()">
Clear</button>
```

1) when you type in the input, value is propagated from Template to Component (thanks to ngModel)

2) {{value}} is one-way binding from Component to Template

3) a click on the button triggers the call of clearValue(), which changes value, which is reflected both in <input> (thanks to the two-way binding of ngModel) and on <p> (thanks to one way binding given from interpolation).

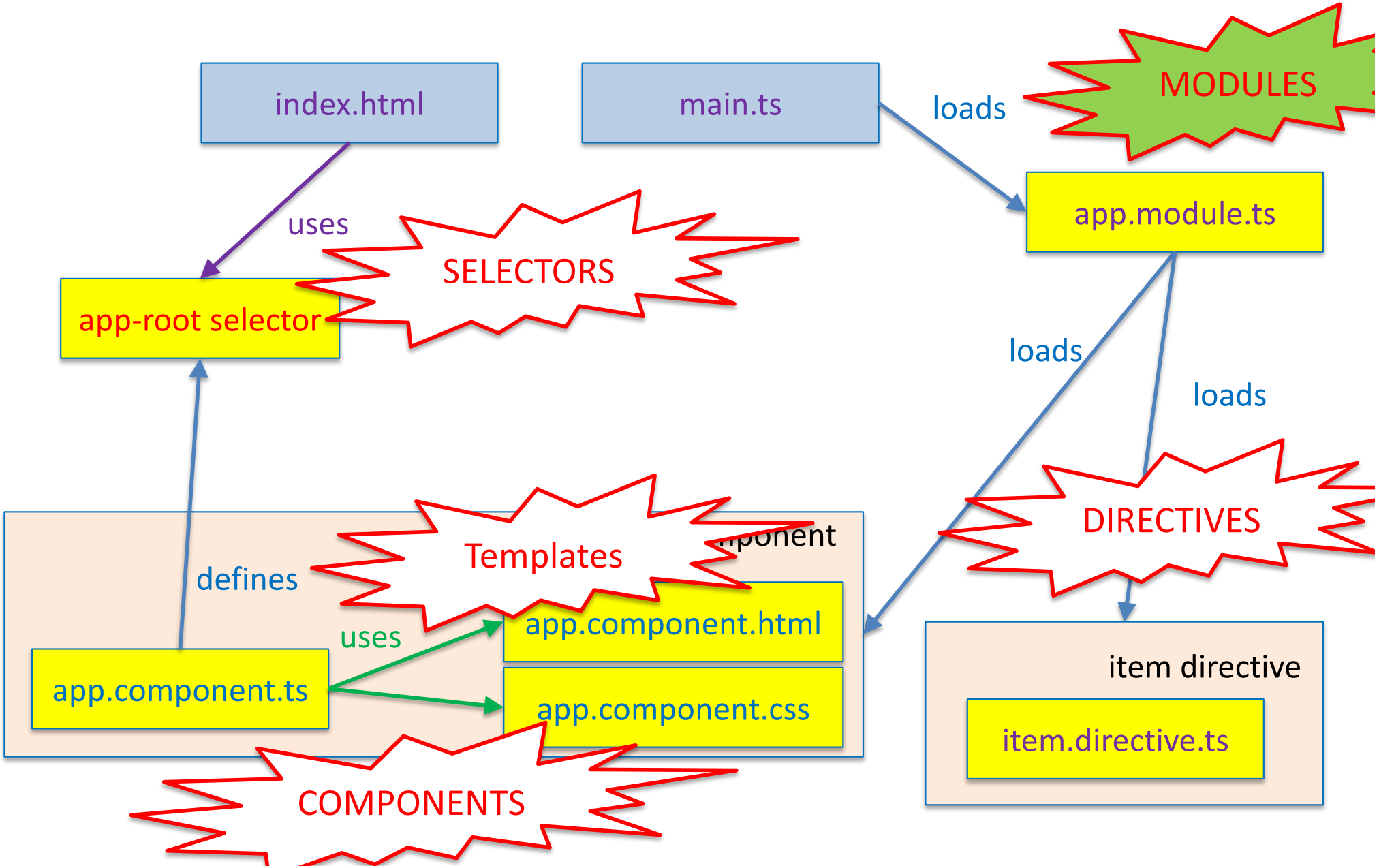
See also https://www.tektutorialshub.com/angular/ngmodel-two-way-data-binding-in-angular/

Q

So far we have seen only a pair template+component.

How do I compose a full app?

Putting the pieces together



Modules: NgModule

The Angular Modules (or NgModules) are Angular ways of group together functional units: related components (but also directives, pipes and services, etc).

Every Component must belong to an Angular Module and cannot be part of more than one module. A component is registered in a Module by declaring it in the Module's metadata.

Every Angular app has a *root module*, conventionally named AppModule, which provides the bootstrap mechanism that launches the application.

Modules: NgModule

An app typically contains many functional modules.

NgModules can import and export functionality from/to other NgModules.

Organizing code into distinct functional modules helps in managing development of complex applications, and in designing for reusability.

lazy-loading loads modules on demand—to minimize the amount of code that needs to be loaded at startup.

Adding a component to a module: ng g – ng generate

To create a new module:

`ng g module newModule`

Assuming you already have a module:

`cd newModule` to change directory into the newModule folder

`ng g component newComponent` to create a component as a child of the module.

or

`ng g component myModule/new-component`

(specifying the path to the module you want to insert the component into)

For a full example, see

https://www.tektutorialshub.com/angular/angular-adding-child-component/

Q

What are parent and child components?

Child components

In general, Angular applications will contain many components.
The the root component usually to just host these child components.

These child components, in turn, can host the more child components creating a Tree-like structure called Component Tree.

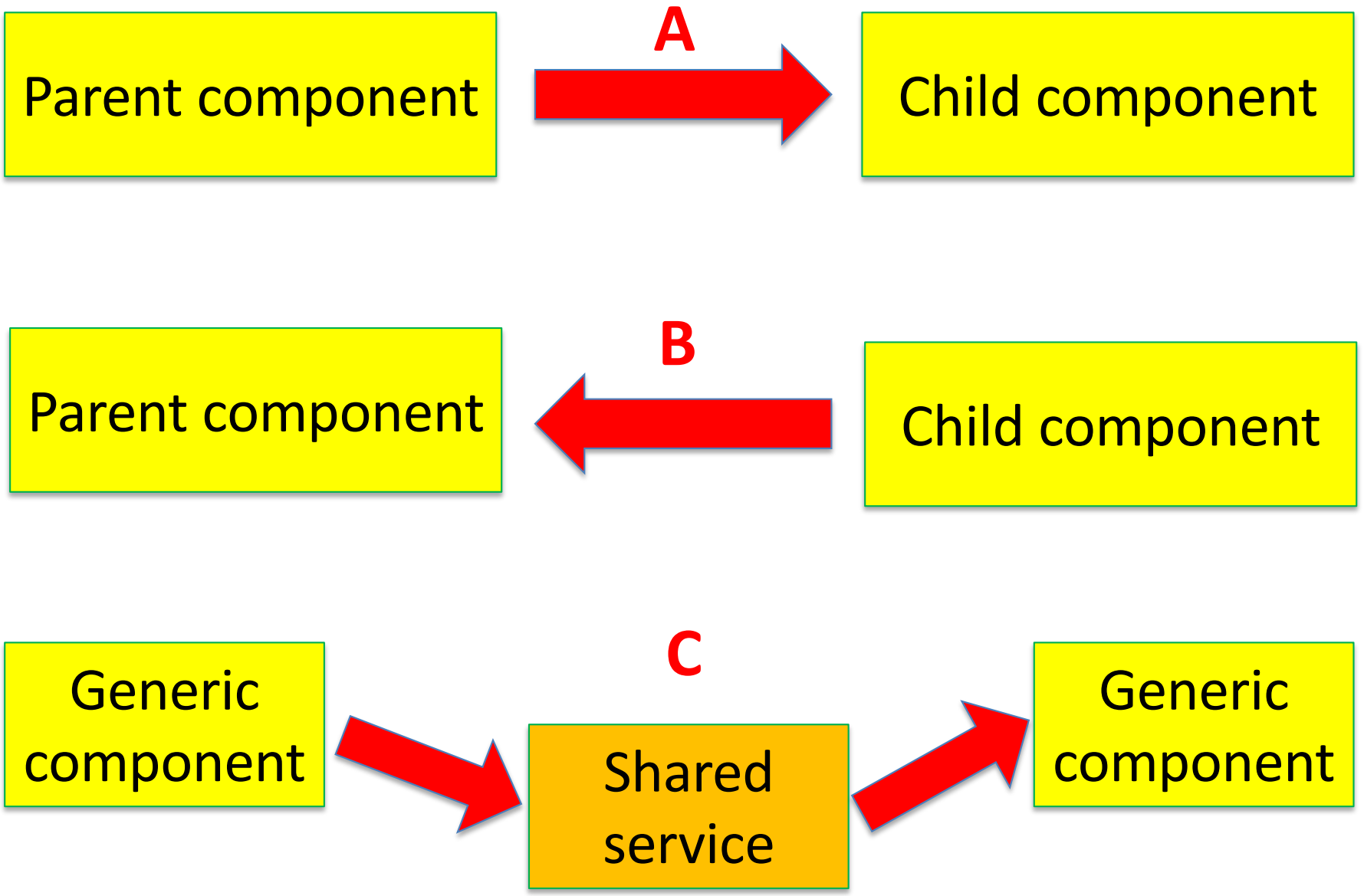
The relation between Parent component and Child component is NOT an IS-A, but a CONTAINMENT (HAS-A)

For an example, see
https://www.tektutorialshub.com/angular/angular-adding-child-component/

Q

How can components communicate?

3 scenarios:


Parent component
A
Child component
Parent component
B
Child component
Generic component
C
Shared service
Generic component

A - Pass data to a child component - 1

In the Child Component:

1. Import the @Input module from @angular/Core Library
2. Mark those property, which you need data from parent as input property using @Input decorator.

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

@Component({
    selector: 'child-component',
    template: `<h2>Child Component</h2>
               current count is {{ count }}
    `
})
export class ChildComponent {
    @Input() count: number;
}
```

in-line template

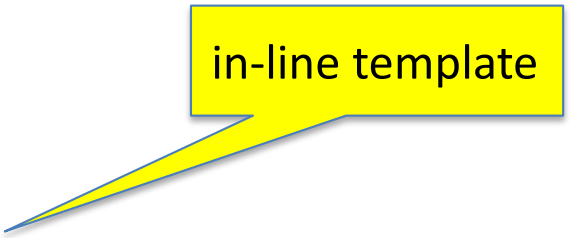
A -Pass data to a child component - 2

In the Parent Component:

1. Bind to Child Property

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

@Component({
  selector: 'app-root',
  template: `
        <h1>Welcome to {{title}}!</h1>
        <button (click)="increment()">Increment</button>
        <button (click)="decrement()">decrement</button>
        <child-component [count]=Counter></child-component>
` , styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Component Interaction';
  Counter = 5;
  increment() { this.Counter++; }
  decrement() { this.Counter--; }
}
```



B - Pass data to a parent component - 1

There are three ways to let parent

1. Parent Listens to Child Event
2. Parent uses Local Variable to access the child
3. Parent uses a @ViewChild to get reference to the child component

For options 1 and 3, see:
https://www.tektutorialshub.com/angular/angular-pass-data-to-parent-component/

B - Pass data to a parent component - 2

Child:

```
import { Component} from '@angular/core';
@Component({
    selector: 'child-component',
    template: `<h2>Child Component</h2>
               current count is {{ count }}
    `})
export class ChildComponent {
    count = 0;
    increment() {this.count++;}
    decrement() {this.count--;}
}
```

in-line template

B - Pass data to a parent component - 3

Parent:

```
import { Component} from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}!</h1>
    <p> current count is {{child.count}} </p>
    <button (click)="child.increment()">Increment</button>
    <button (click)="child.decrement()">decrement</button>
    <child-component #child></child-component>` ,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Parent interacts with child via local variable';
}
```

B - Pass data to a parent component - 3

Child:

```
import { Component, Input, EventEmitter, Output } from
'@angular/core';
@Component({
    selector: 'app-child',
    template: `<button (click)="handleclick()">Click
me</button>`
})
export class AppChildComponent {
    handleclick() {
        console.log('hey I am  clicked in child');
    }
}
```

in-line template

B - Pass data to a parent component - 3

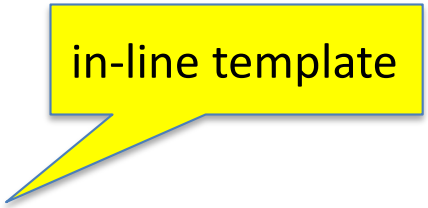
Child: - appchild.component.ts

```
import { Component, Input, EventEmitter, Output } from
'@angular/core';
@Component({
    selector: 'app-child',
    template: `<button (click)="handleclick()">Click
me</button>`
})
export class AppChildComponent {
    handleclick() {
        console.log('hey I am  clicked in child');
    }
}
```

B - Pass data to a parent component - 3

Child: - appcomponent.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `<app-child></app-child>`
})
export class AppComponent implements OnInit {
  ngOnInit() {}
}
```



C - Using components vs services from other modules

The most common way to get a hold of shared services is through Angular dependency injection, rather than through the module system (importing a module will result in a new service instance, which is not a typical usage).
To read about sharing services, see Providers.

Q

Can we generate events ?

What is reactive programming ?

Reactive programming

Reactive programming is programming with asynchronous data streams.

A stream is a sequence of ongoing events ordered in time. It can emit three different things: a value (of some type), an error, or a "completed" signal.

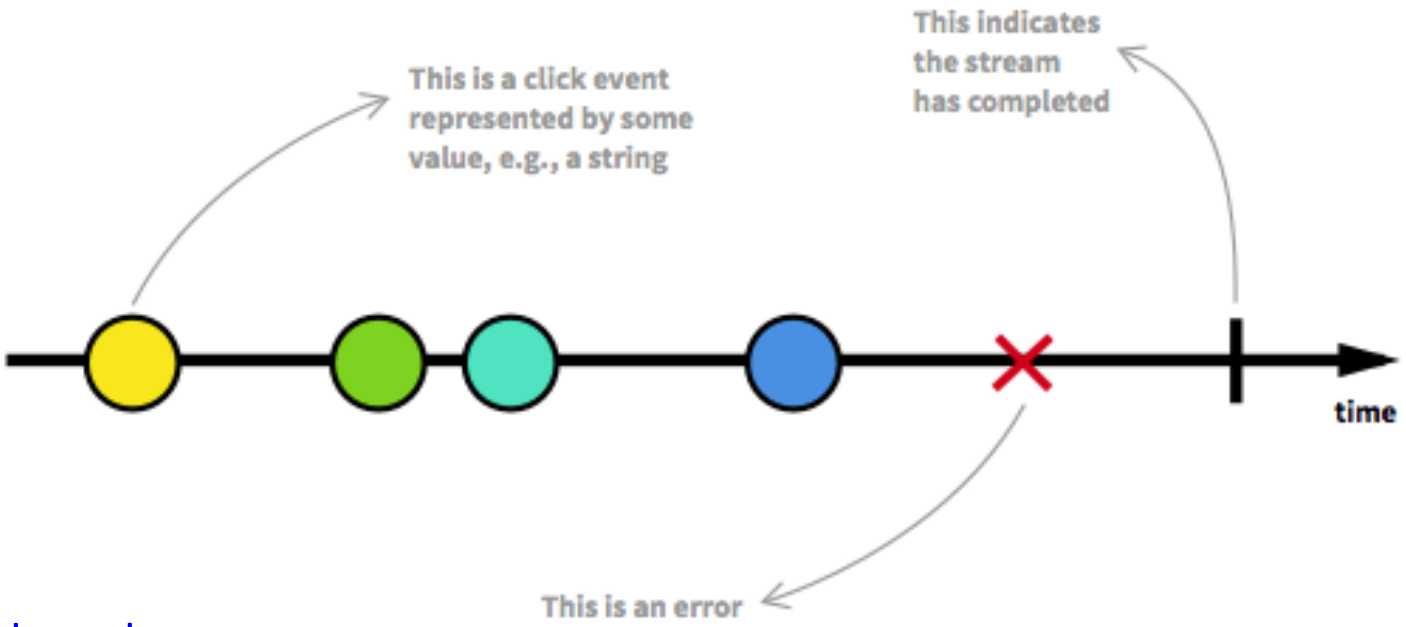


image by [andrestaltz]

Reactive programming

We capture these emitted events asynchronously, by defining:

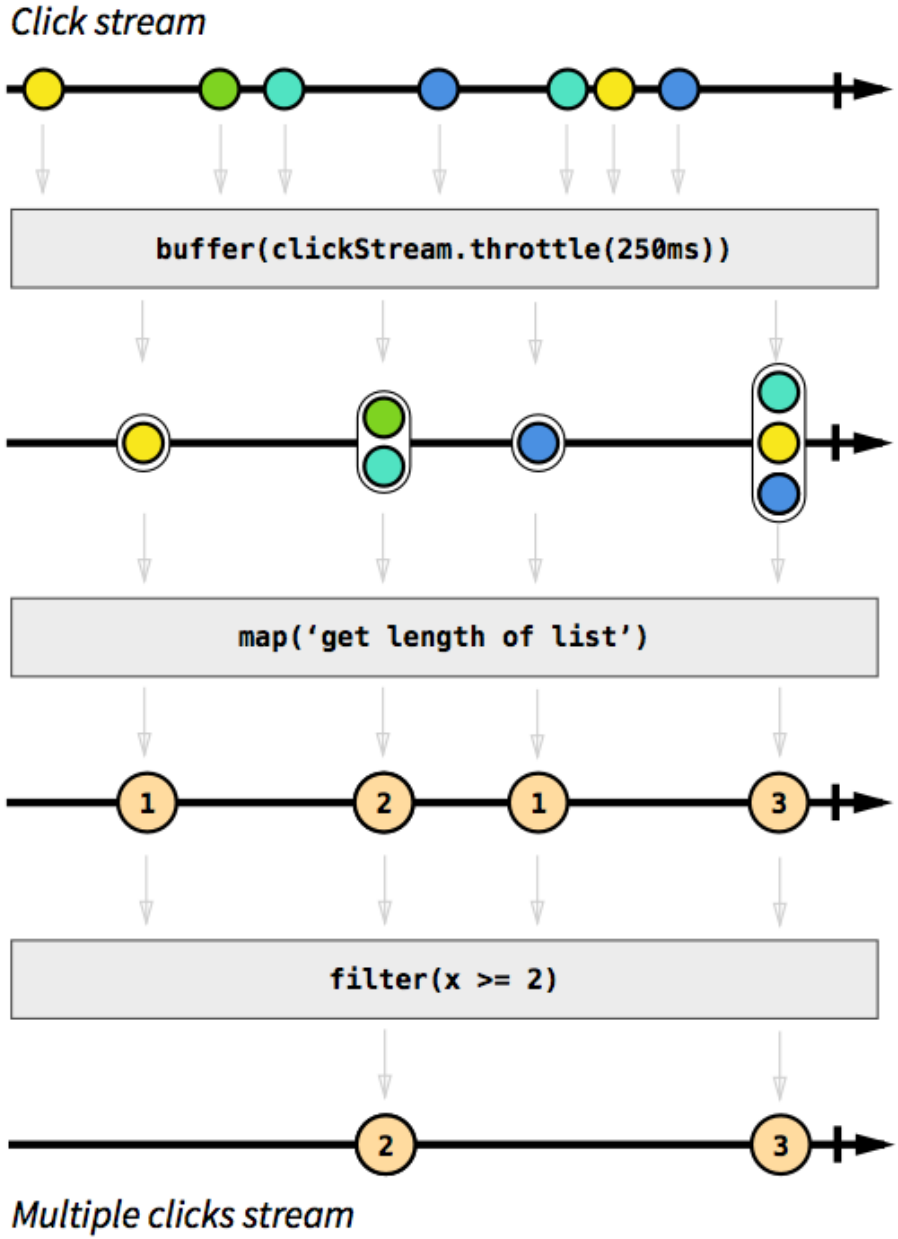
- a function that will execute when a value is emitted,
- a function when an error is emitted,
- a function when 'completed' is emitted.

According to the Observer Design Pattern:

- The stream "observable" being observed.
- The "listening" to the stream is called subscribing.
- The functions we are defining are "observers".

Reactive programming

In a reactive programming environment, you are generally given a toolbox of functions to combine, create and filter any of those streams.



RxJS

RxJS is a library for composing asynchronous and event-based programs by using observable sequences.
It provides:

- one core type, the Observable,
- satellite types (Observer, Schedulers, Subjects)
- operators (map, filter, reduce, every, etc) to allow handling asynchronous events as collections.

see https://rxjs-dev.firebaseapp.com/guide/overview

RxJS - operators

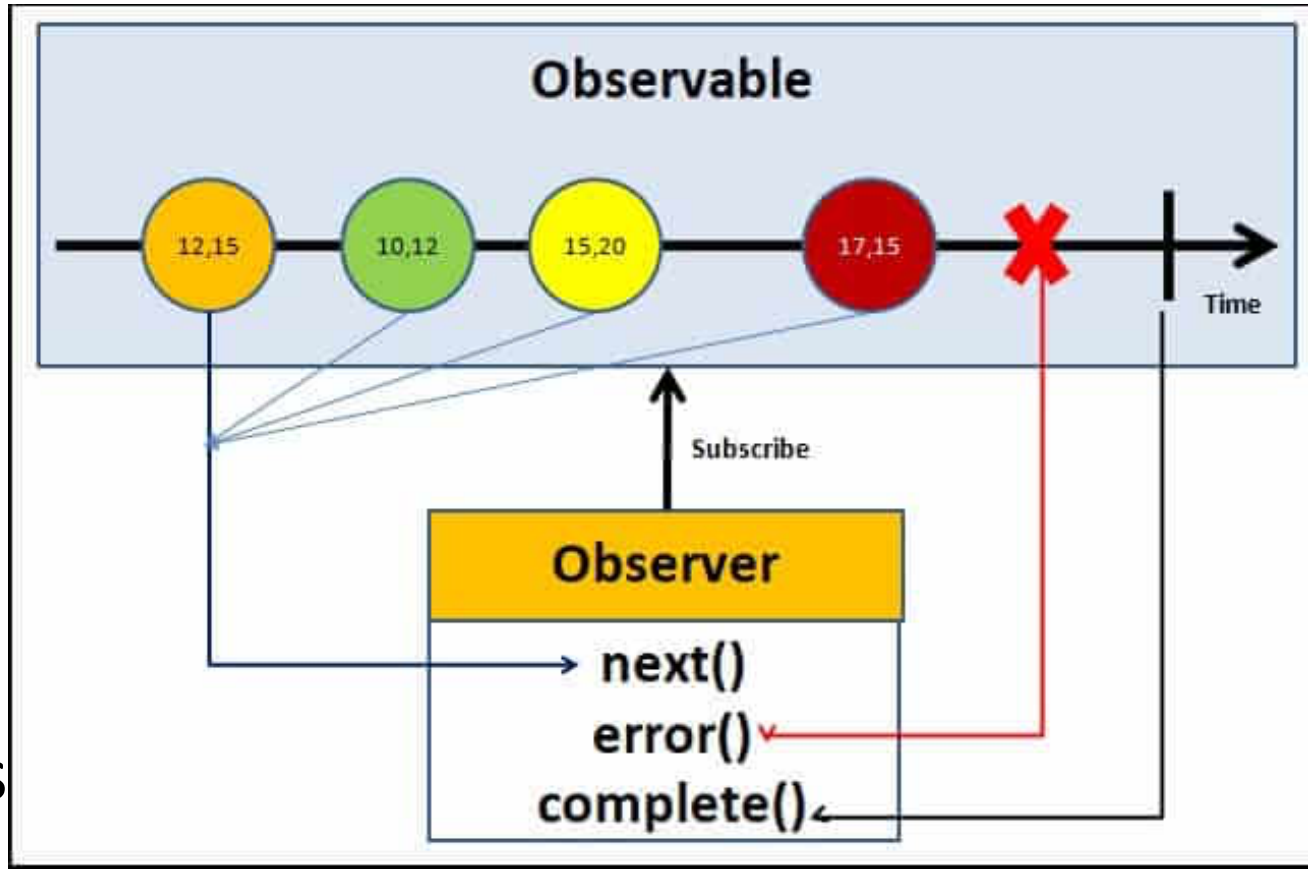
AREA	OPERATORS
Creation	`from`, `fromEvent`, `of`
Combination	`combineLatest`, `concat`, `merge`, `startWith` , `withLatestFrom`, `zip`
Filtering	`debounceTime`, `distinctUntilChanged`, `filter`, `take`, `takeUntil`
Transformation	`bufferTime`, `concatMap`, `map`, `mergeMap`, `scan`, `switchMap`
Utility	`tap`
Multicasting	`share`

see https://angular.io/guide/rx-library#operators

RxJS

The observable:

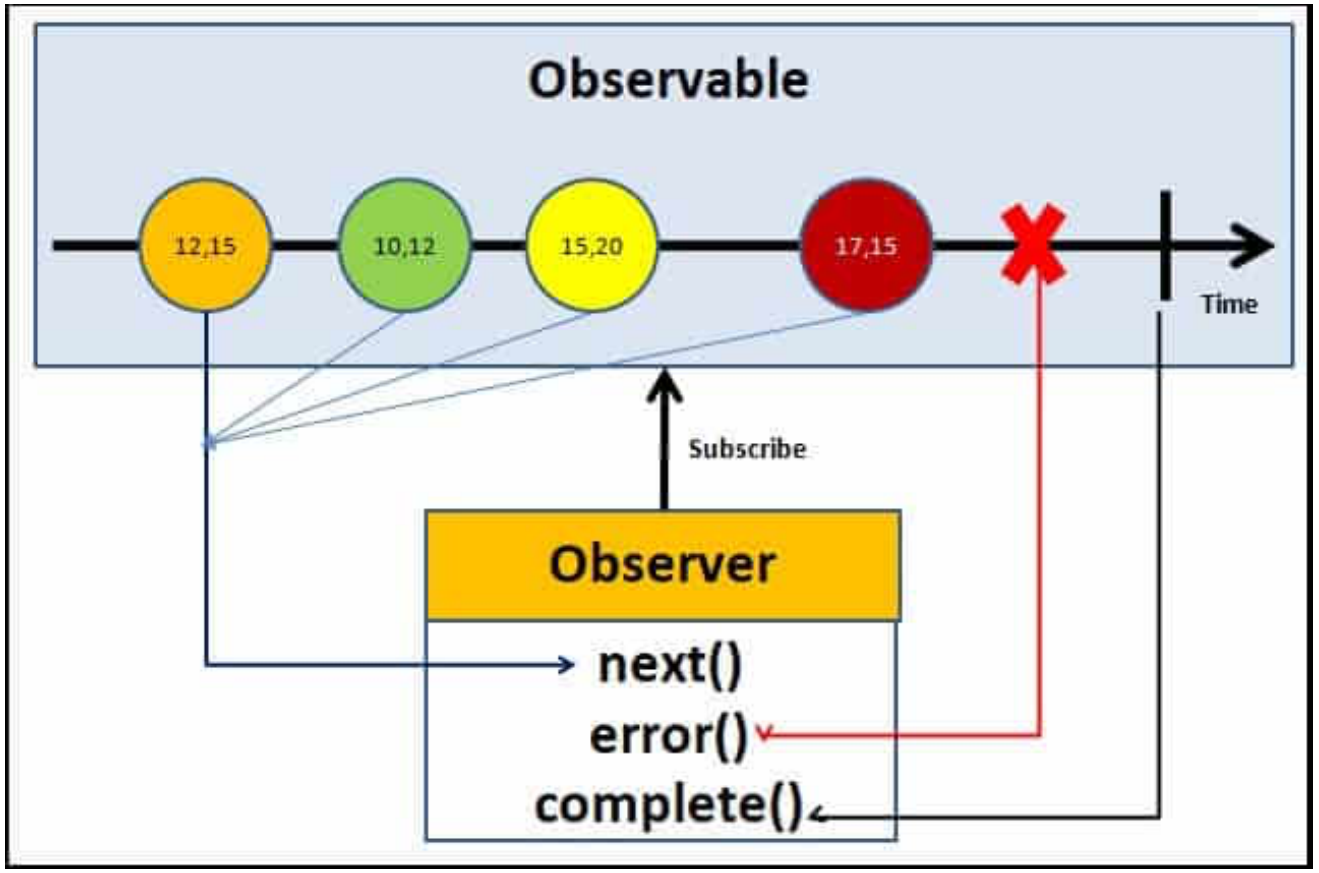
- invokes the next() callback whenever a value arrives in the stream. It passes the value as the argument to the next callback.



- If the error occurs, then the error() callback is invoked.
- It invokes the complete() callback when the stream completes.

RxJS

Observers subscribe to Observables.



Observer registers three callbacks with the observable at the time of subscribing:

- next(),
- error()
- complete()

All three callbacks are optional.

The observer receives the data from the observer via the next() callback

They also receive the errors and completion events via the error() & complete() callbacks

Example – app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'Angular Observable using RxJs - Getting
Started';
  obs = new Observable((observer) => {
    console.log("Observable starts")
    setTimeout(() => { observer.next("1") }, 1000);
    setTimeout(() => { observer.next("2") }, 2000);
    setTimeout(() => { observer.next("3") }, 3000);
    setTimeout(() => { observer.error("error emitted")
}, 3500);
    //this code is never called
    setTimeout(() => { observer.next("5") }, 5000);  })
```

Example – app.component.ts

```
  data=[];

  ngOnInit() {

    this.obs.subscribe(
      val => { console.log(val) },
      error => { console.log("error")},
      () => {console.log("Completed")}
    )
  }
}
```

For a full example see
https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/

Observable typing

In the end, an **Observable** is a (dynamic, asyncronous collection of "things": like Java collections it can be typed.

So an **Observable** is, by default, an ensamble of anything:

```
Observable<Any>
```

but it could be restricted

```
Observable<SomeType>
```

Most often **`SomeType`** is an interface

Observable typing

Although the Angular framework does not enforce a naming convention for observables, you will often see observables named with a trailing “$” sign.

example:

```
export class StopwatchComponent {
  stopwatchValue: number;
  stopwatchValue$: Observable<number>;
  start() {
    this.stopwatchValue$.subscribe(
        num => this.stopwatchValue = num
    );
  }
}
```

Emitting events and custom events

Components can also raise events with EventEmitter.
Using EventEmitter you can create a property and raise it using the EventEmitter.emit(payload).

The Parent component can listen to these events using the event binding and read the payload using
the $event argument.

A full, well elaborated example of custom events is here:
https://www.concretepage.com/angular-2/angular-2-custom-event-binding-eventemitter-example