# Angular 2

Gopalakrishnan Subramani
gs@nodesen.se
+91 9886991146

www.nodesen.se

https://github.com/nodesense/ngapp

## Agenda

- Angular 2-Introduction
- SPA-Introduction
- ES 6
- TypeScript
- Tooling
- Components
- Templates
- Services
- Pipes

## Single Page Application

# What is SPA?

- SPA is an  architectural pattern
- Single-Page Applications (SPAs) are Web apps that loads..
- A single HTML page
- Dynamically update that page as the user interacts with the app without leaving page
- SPA uses a lot of AJAX (Asynchronous JavaScript and XML) for downloading partial webpages, templates, data
- SPA often uses RESTful APIs, Web Services
- JSON is most used for data exchange

## Advantage over Server Side MVC Architecture

- Server Side Web pages generated by server frameworks like ASP.NET MVC, Spring MVC, Express JS

- A Web Page consists of HTML elements, a lot of JavaScripts, Images, CSS, they are downloaded, parsed, executed by browser for every page

- Downloads a lots of content for every page load

- Drain battery so fast

# Advantages of SPA

- Parse HTML, create DOM hierarchy only once
- Parse CSS, create CSS structure only once
- Interpret/Compile JavaScript only one
- Create a web layout only once
- Download static assets, master page layout only once
- Dynamic contents, data downloaded on need basics using AJAX,
- Consume less battery, CPU resources, mobile friendly

## Angular 2

# What is Angular 2?

Angular 2.0 is a **Client** side, **Modular**, **Dynamic** Web Application Development **Framework** written entirely in **TypeScript, ES6** to build **Single Page Rich Internet Application** using popular Web Technologies **HTML5**, **JavaScript, CSS** and **Component** based architecture

## Open Source

- Angular 2 is developed by Google
- Open Source, MIT License
- Community supported, well documented
- Finally released in September 2016

# Angular Branding

- **AngularJS** refers to Angular 1.x series
- **Angular** refers to Angular 2.x, Angular 4.0 onwards
- **Angular** is completely **different** than **AngularJS, not a version upgrade, entirely new framework**
- **No backward/forward compatibility between Angular and AngularJS**

## Why ‘Angular’ Name?

- ‘Angular’ refers to HTML angle brackets ‘<‘, ‘/>’
- Uses HTML for both template and markup
- Angular **extends** HTML at runtime

# HTML Extension?

```
<p highlight> Mouse over me, I glow</p>

<contact-address address="homeAddress" >

</contact-address>

<ul>

    <li *ngFor="let fruit of fruits">

     {{fruit}}

   </li>

</ul>
```

**highlight** - Angular directive
**contact-address** - Angular component
***ngFor** - Angular for each loop

- No jQuery
- No template language
- Just HTML

# Why Angular 2?

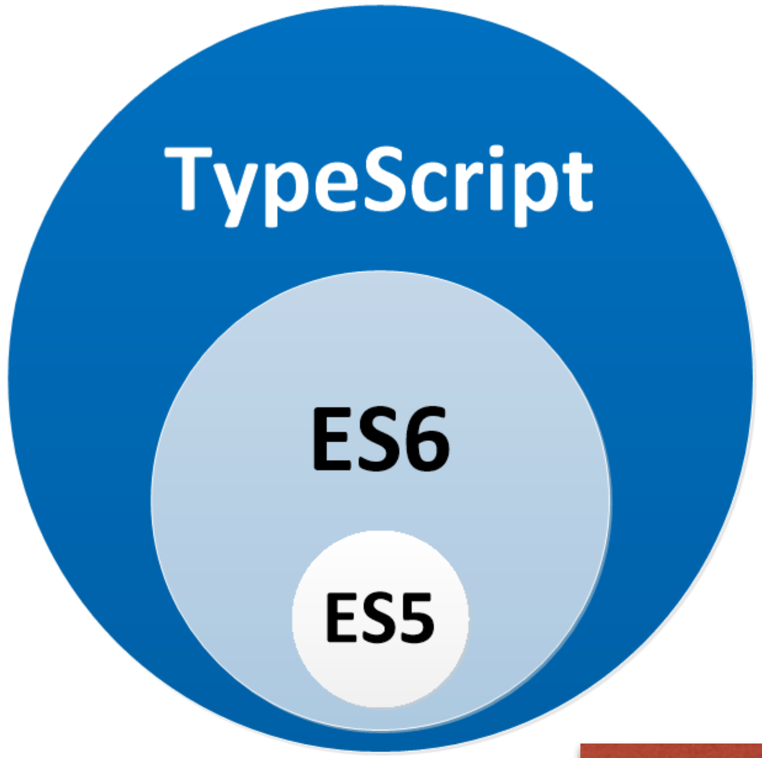
- Component based
- Best Practices, Modules, In-Built libraries
- Clean and Modular design
- Very fast change detection with focus on speed and performance
- Reactive mechanisms baked into the framework
- Support for Model View Controller patterns
- A lot of improvements over Angular 1.x, 5 to 100 times* faster
- Written in TypeScript, ES6, a new generation JavaScript

# Notable Features of ES6

- let, const for block level variable declaration over ‘var’
- module export, import
- Classes, Constructor, Inheritance, Polymorphism - Object Oriented JavaScript
- Getter, Setter Properties
- Lambda (Fat Arrow) functions
- Default Function Parameters
- for..of loop   (for each)

# TypeScript

- Typed Super set of  JavaScript
- Backward compatible with ES6, ES5
- Promised Forward compatibility with ES7 onwards



TypeScript, Microsoft implementation with static typing, interfaces, decorators

JS/ES6

```
let i;
let product;
```

$\longrightarrow$

TypeScript

```
let i:number = 100;
let product:Product;
```

# Languages

- ES 5 - 2009 (old, popular JavaScript)
- ES 6-ECMAScript 2015 - June, 2015
- ES 7 - June 2016

**JavaScript**

**TypeScript 2.x** - Microsoft, by Anders Hejlsberg, lead designer of Turbo Pascal, Delphi, C# Languages

# Scoping in JavaScript

ES5

- Global Scope
- Function Level Scope

ES6

- Global Scope
- Function Level Scope
- Block Level Scope
- Module/File level Scope

CommonJS spec

- Module/File level scope

# let for block scope

```
function getName(player) {
   if (player.winner) {
    var prefix = 'Champian';
   }
    //prefix is accessing here
    return prefix + player.name;
}
```

ES5
Hoisting Bug

---

```
function getName(player) {
   if (player.winner) {
    let prefix = 'Champian';
   }
    //prefix is NOT accessing here, maintain scoping
    return prefix + player.name;
}
```

**ES6**
**let, const for block scope**

# const

```
const PI = 3.14;
PI = 2.14; //Error

const center = {x:0, y:0};
// Error, can’t change reference
center = {x:20, y:20}

center.x = 100; //OK, no error
```

# for..of loop

ES6

```
let points = [10, 20, 30, 40, 50];
for (let point of points) {
    console.log(point)
}
```

Output

```
10
20
30
40
50
```

**For Each Loop**

## Rest Operator ...

```
function addPoints(...points) {
    var dataPoints = [];

    for (let point of points) {
        dataPointsInput.push(point);
    }
}

addPoints(10, 20, 30, 40, 50);
```

```
class Shape {
    constructor(name) {
        this.name = name;
    }
}

class Circle extends Shape {
    constructor(point) {
        super(Circle.getType())
        this.point = point;
    }

    getName() { return this.name}

    static getType() {
        return "Circle";
    }
}

let circle = new Circle({ x: 10, y: 20 });
circle.getName(); //circle.name
```

Class
Constructor
Static
Inheritance

No access specifier

# Getter/Setter Function

```
class Framework {
    constructor(name) {
        this._name = name;
    }

    get name() {
        return this._name;
    }

    set name(newName) {
      if (newName) {
        this._name = newName;
      }
    }
}
```

```
var framework = new
        Framework("AngularJS");

//calls get name()
if (framework.name) {
}

// calls set name()
framework.name = "Angular";
```

# Fat Arrow

```
function add(a, b) {
    return a + b;
}
```

```
let add = (a, b) => a + b
```

```
getProduct(id)
    .then(function (product) {
        return getRating(product);
    })
    .catch(function (error) {
        logError(error);
        showAlert(error);
    })
```

## Fat Arrow Lambda

```
getProduct(id)
  .then(product => getRating(product))
  .catch(error => {
          showError(error);
          showAlert(error)
})
```

# Template Literals

```
let firstname = ‘Nila’;
let lastname = ‘Krish’;

//ES 5
var fullname = 'Miss ' + firstname + ' ' + lastname;

//ES6 way
let fullname = `Miss ${firstname} ${lastname}`;

//ES 6 Way Multiline string, no need for escape \
let template = `
            Hello,
                    ${fullname}
            `
```

backquote

# Modules

```
//product.js
export class Product {

}

export class Service {
 }

//local, file scope
class InnerClass {

}

//local, file scope
let name = “product”
```

```
//product-edit.component.js

import {Product} from "./product"

var product = new Product()


//alias to prevent collision
import {Product as ProductModel}
from "./product"

var product = new ProductModel()


//or import all with alias

import {* as pm} from "./product"
let name =“hello”;
var product = new pm.Product()
 var ps = new pm.Service();
```

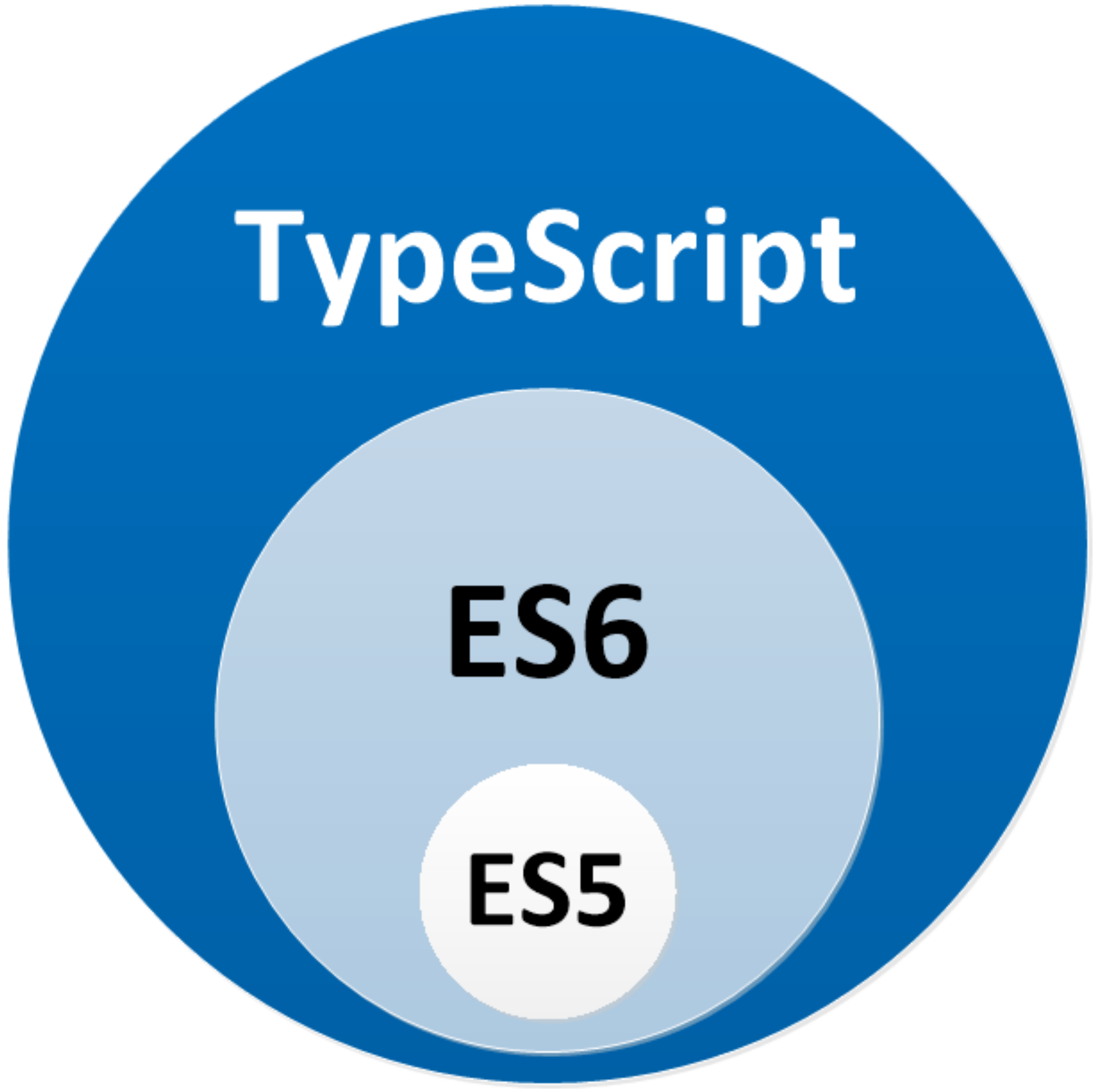
# A lot more..References

- http://es6-features.org/
- http://www.ecma-international.org/ecma-262/6.0/

## TypeScript

# Introduction

- Is a typed superset of Javascript
- Is a compiled language, so it catches errors before runtime
- Includes the features of ES6 but with types, as well as better tooling support
- TypeScript allows us to decorate our classes via
- @<Decorator> the syntax
- Classes, modules, types, interfaces and @decorators



TypeScript, Microsoft implementation with static typing, interfaces, decorators

# Just Types

```
JavaScript + types

let i:number = 10;
i = “text”; //error

let k:number | string;
k = 10;
k = “hello”;
k = false;//Error

let result:boolean;
result = true;

let name:string = ‘Krish’;
```

```
let product:Product = new Product()

//Array
let fruits:string[]=[“Apple”];
let numbers:number[] = [1, 2]


//any
let v : any;
v = 1;
v = “text”
v = [“one”, 1]

let names:any[];
```

**TypeScript is statically typed JavaScript**

# Functions

```
 function getProduct(id): Product {
     Product p = new Product()
     return p;
 }

//void
function setName(name: string): void {
    this.name = name;
}

//Multi type arguments, return type
function add(a : string | number, b: string | number) :
number | string | any{
  return a + b;
}

add(1, 2); //3
add("hello ", "world"); //hello world
add(true, false); //ERROR
```

# Interface

```
interface Shape {
 area(r: number):number;
}


class Circle implements Shape {
  static PI:number = 3.14;

  area(r: number):number { return Circle.PI * r * r;}
}
```

## Access Specifier

```
class Square {
    private size: number;

    public name: string;

    color: string; //by default public

    protected internalAreaCalculation(): void {
    }
}
```

# Constructor With Access Specifier

```
class Product {
  private name:string;
  private year: number;

  constructor(name: string,
              year:number) {
    this.name = name;
    this.year = year;
  }
}
```

```
class Product {

  constructor(private name: string,
              private year:number) {

  }
}
```

Syntactic Sugar to declare variable and assign them in constructor

## Decorator

```
@Component({ selector: 'home' })
export class HomeComponent {
      constructor(@Inject(“apiEndPoint”)
                  string apiEndPoint) {
           …
       }

       @Input()
       product:Product;


 }
```

# Generics

```
//Generic Types
function identity<T> (arg: T): T {
      return arg;
}
// type of output will be 'string'
let output = identity<string>("Krish");

//Generic Arrays

function identityTests<T>(arg: T[]): T[] {
    return arg;
}

identityTests<string>(["Steve"])
```

## Generic Class

```
class GenericNumber<T> {
       zeroValue: T;
       add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
```

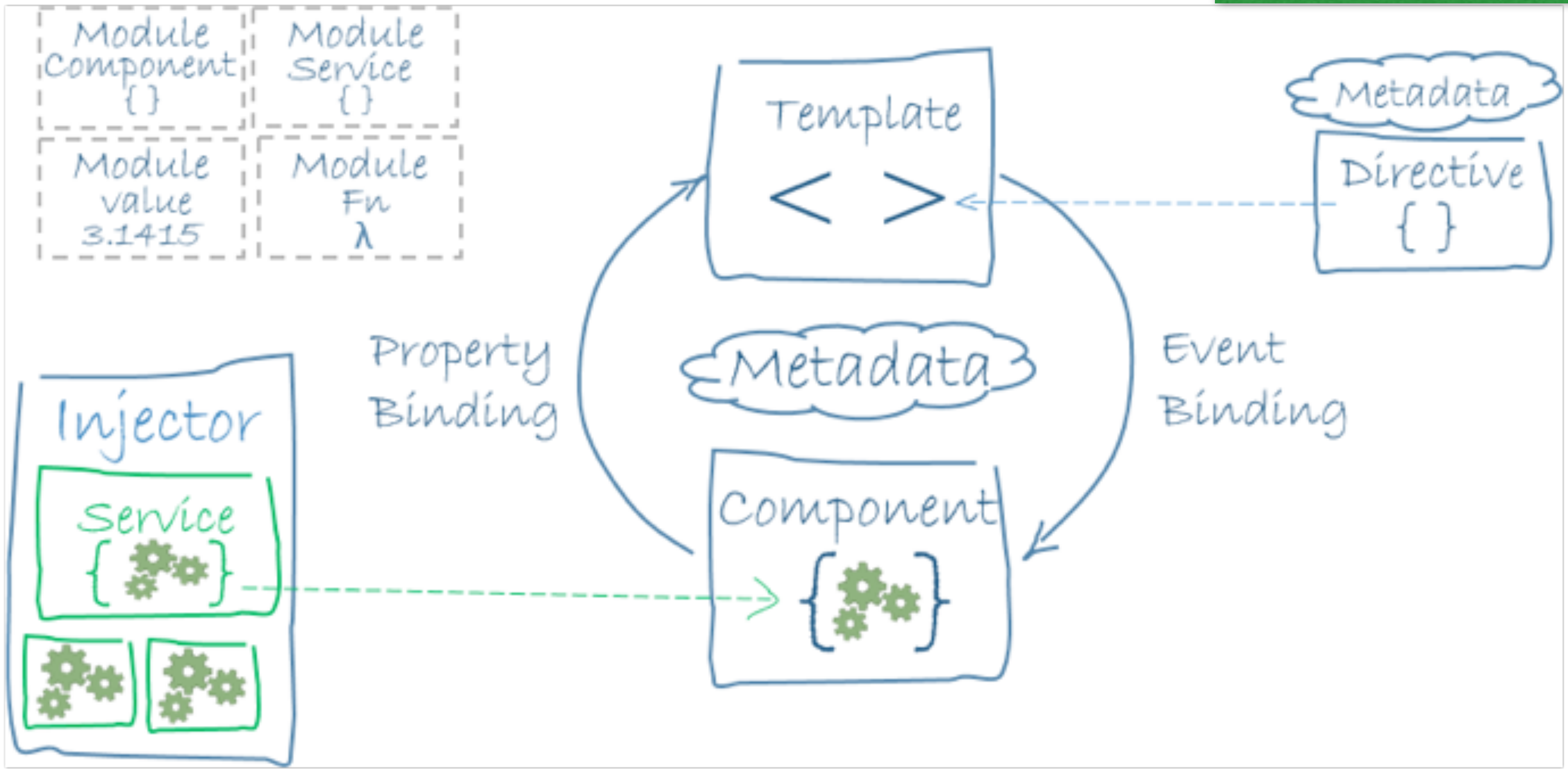
# TypeScript

https://www.typescriptlang.org/docs/tutorial.html

## Angular
## Big-Picture

Building Blocks

Your Angular Application



Angular Core Framework

Router | Http | Forms | Compiler | Platform

Core & Common

3rd Party Core Modules

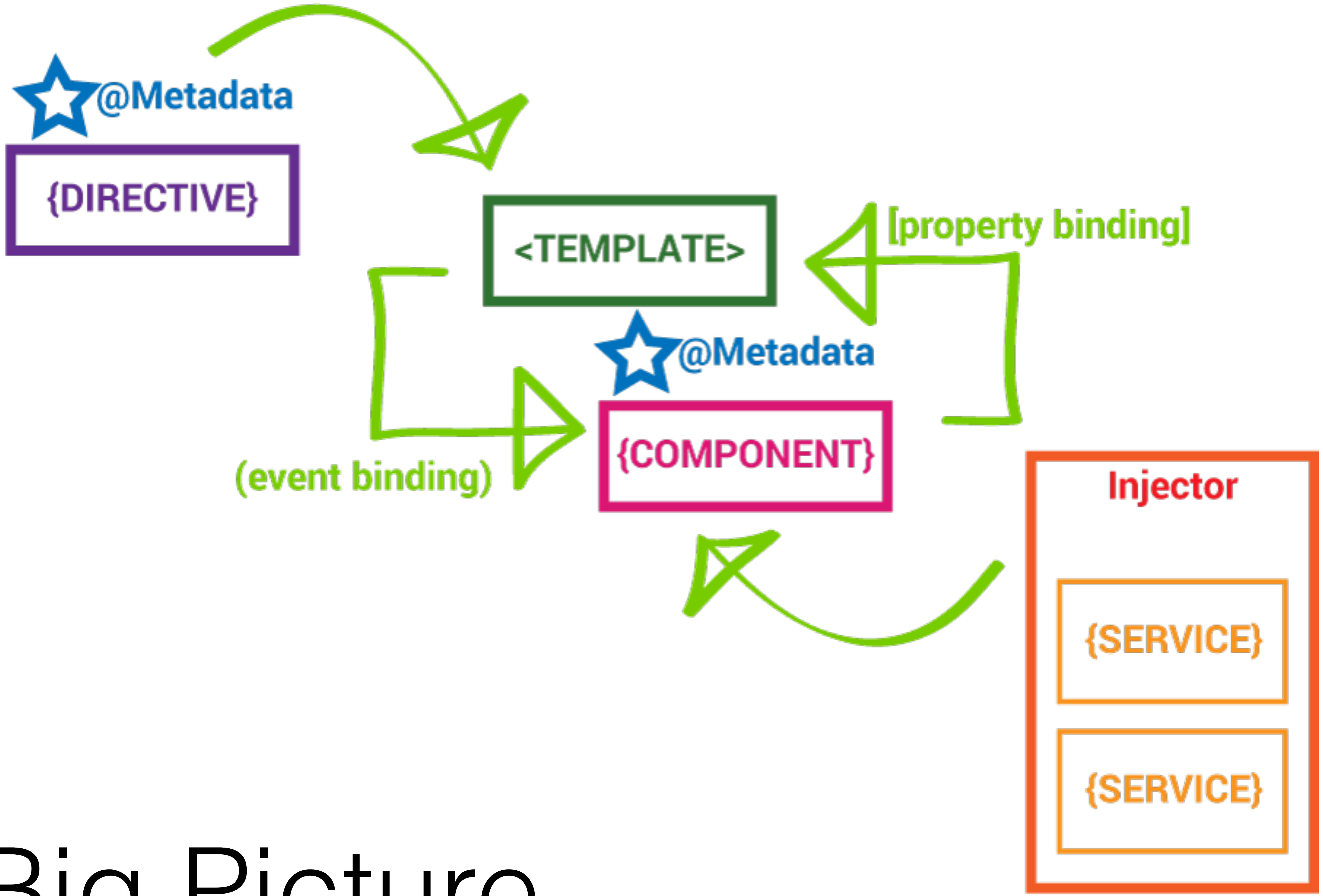
RxJS | Reflect | Zone | SystemJS

Shim

TS

ES6

| | |
|---|---|
| RxJS | Microsoft Reactive Extension implements Observer/ Subscriber design pattern for data change notification |
| Zone | Zone.js for trigging change detection. Zone.js overwrites browser events, setInterval, setTimeout, XHR Ajax Request |
| Reflect<br>Shim | Polyfills, implemented by core-js library, for ES6 specific polyfills (cross-browser compatibility) to support Map, Set, Promise, Array.filter, Array.map, etc…<br>Reflect to enable reflection in JavaScript |
| SystemJS | Manages File Dependencies, download files to browsers |


@Metadata
{DIRECTIVE}
<TEMPLATE>
[property binding]
@Metadata
{COMPONENT}
(event binding)
Injector
{SERVICE}
{SERVICE}

## Big Picture

# Building Blocks

- Module
- Component
- Metadata
- Template
- Data Binding
- Service
- Directive
- Dependency Injection

## ES 6-Module

- Module is sort of namespace,

- Uses ES6 module syntax

- Angular 2 applications use modules as the core mechanism for composition

- Modules export things that other modules can import

```
  //product.model.ts
  export class Product {
      constructor() {
          this.name = '';
          this.year = 2016;
      }
  }
```

---

```
//product.component.ts
import {Product} from "./product.model";
```

## Module

## Angular NgModule

- @NgModule Decorator
- declarations - the view classes: components, directives, and pipes.
- exports - Components/Directives that can be used in other modules.
- import - other dependent modules.
- providers - creators of services, accessible in all parts of the app.
- bootstrap - the main, root application view - Root component

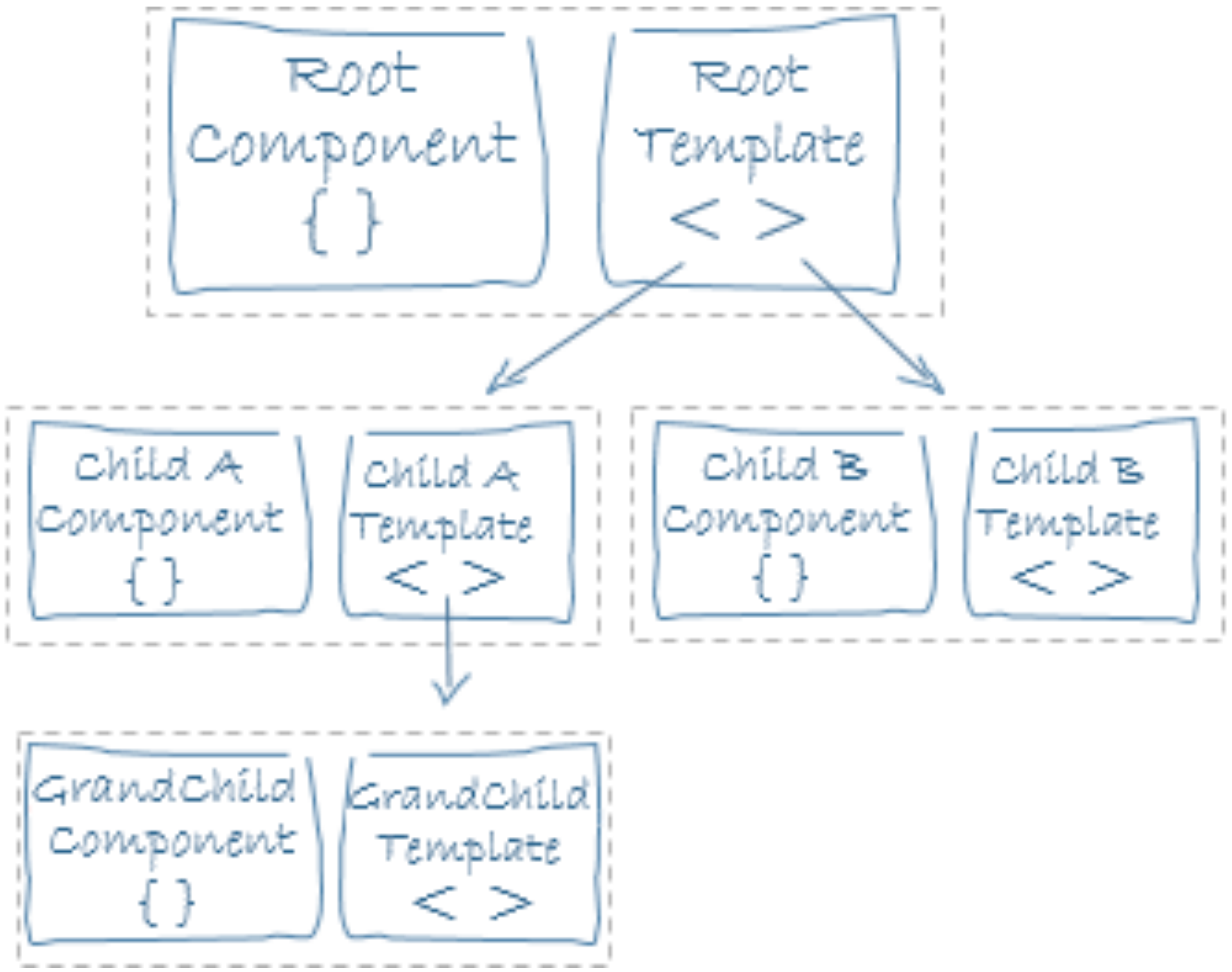
# NgModule

```
import { NgModule }       from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ DataService ],
  declarations: [ AppComponent ],
  exports:      [ ProductListComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

## Component

- Components are just ES6 classes
- Component represent “C” in MV**C** (aka Controller)
- Component class has Properties and methods that are available to the template/view i.e. “V” in MVC
- Models are Properties of components
- Component shall have children component, nesting possible

# Component Hierarchy



```
@Component(…)
export class ProductDetailComponent {
    product: Product;

    constructor() {

    }

    saveProduct() {
        this.dataService
        .saveProduct(this.product)
    }
}
```

## Component

# Metadata

- Metadata is used for reflection
- Metadata attaches more information to classes, attributes, parameters
- Metadata helps Angular framework to load component, template, styles, inject arguments
- We can attach metadata with TypeScript using  decorators
- Decorators are just functions
- Most common is the @Component() decorator

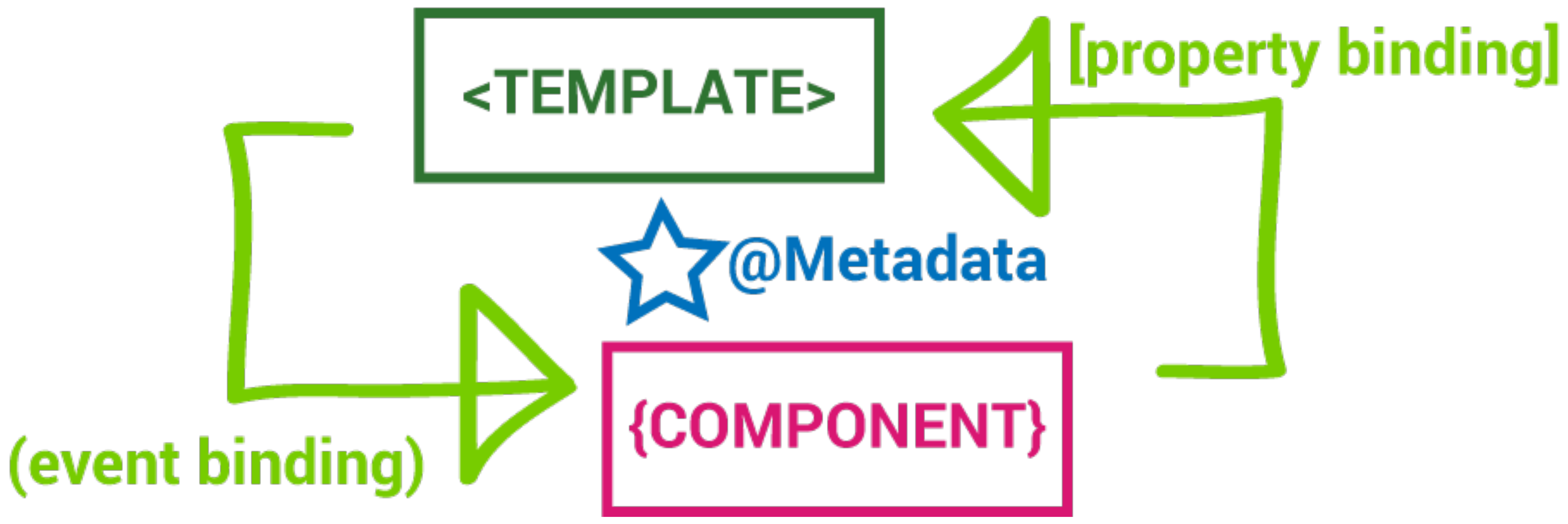
# Metadata

```
@Component({
    selector: 'product-detail'
    templateUrl: 'app/product/product-
detail.component.html'
})
export class ProductDetailComponent {
}
```

## Template

- A template is HTML that tells Angular how to render a component
- Templates include data bindings as well as other components and directives
- Angular 2 leverages native DOM events and properties  which dramatically reduces the need for a ton of built-in directives
- Angular 2 leverages shadow DOM to do some really interesting things with view encapsulation

# Template



## Template

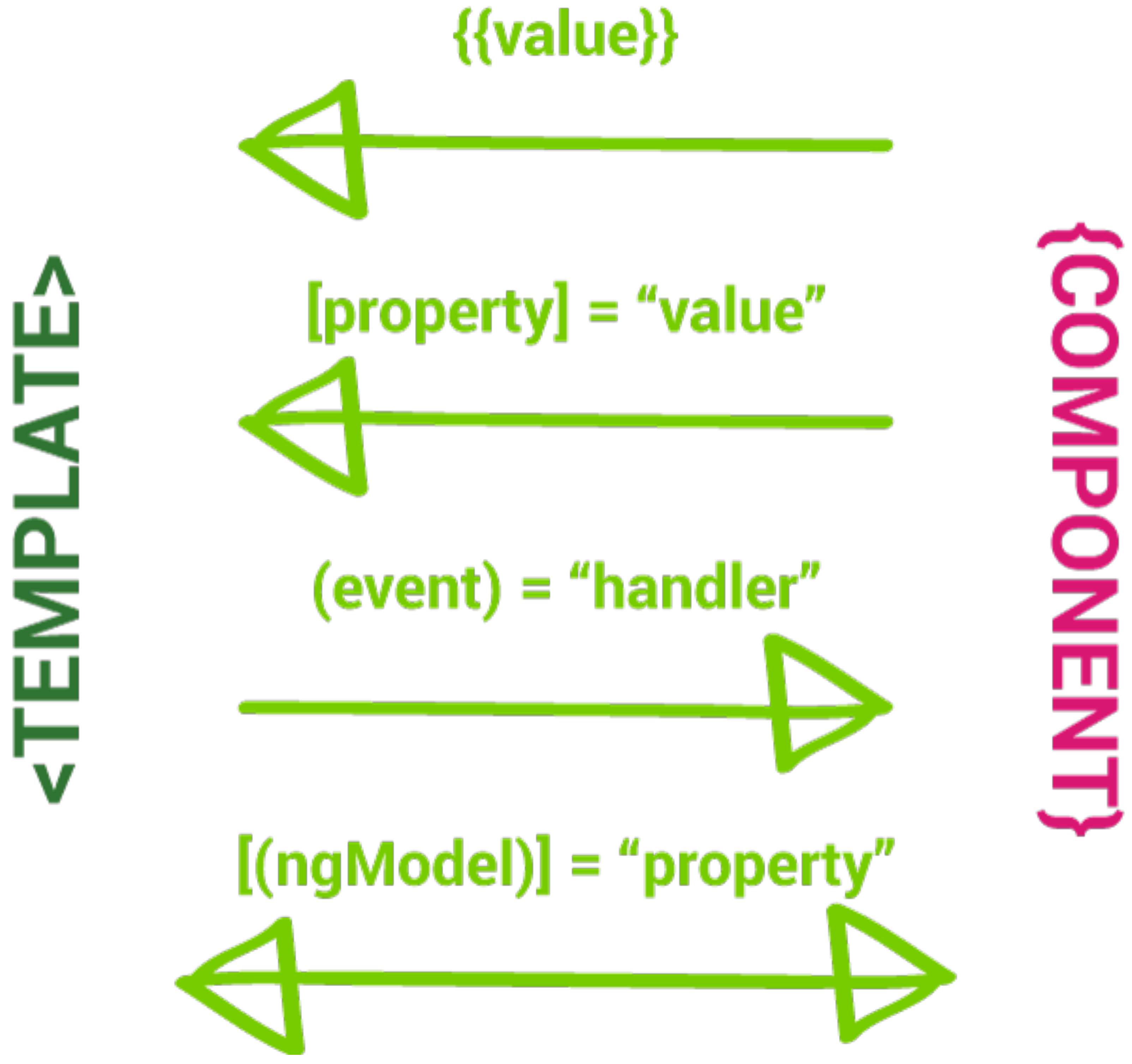
```
@Component({
    selector: 'product-edit'
    template: `
     <div class="product-details" (click)="editSpec()">
        <h2>{{product.name}}</h2>
        <p>{{product.description}}</p>
     </div>
    `,
    styles: [`
        .product-detail {
        outline: 1px lightgray  padding:   5px;
        margin: 5px;
        }
     `]
})
export class ProductEditComponent {
    product: Product;
    ...
}
```

## Data Binding

- Enables data to flow from the component to template  and vice-versa

- Includes interpolation, property binding, event binding,  and two-way binding (property binding and event  binding combined)

- The binding syntax has expanded but the result is a  much smaller framework footprint



# Data Binding

# Data Binding

```
<h1>{{product.name}}</h1>

<brand-detail [brand]=“brandInfo”>
</brand-detail>
<form name="productForm"
      (ngSubmit)="saveProduct()" >

     <input name="name"
      [(ngModel)]="product.name" >

</form>
```

# Service

- A service is just a class
- Should only do one specific thing
- Good for Business logic, Communication with service implementation
- Decorate with @Injectable when we need to inject dependencies into other services, components

## Service

```
import {Injectable} from "@angular/core";

import {Http} from "@angular/http";

@Injectable()
export class ProductService {
    constructor(private http: Http) {
     }

    getProducts() {
        return this.http
        .get("https://example.com/api/products")
    }
}
```

## Directive

- A directive is a class decorated with @Directive
- A component is just a directive with added template features
- Built-in directives include structural directives and attribute directives
- ngFor, ngIf are few of built-in directives

```
import { Directive, ElementRef,  Renderer, HostListener } from
'@angular/core';

@Directive({ selector: '[highlight]' })
export class HighlightDirective {
    constructor(private el: ElementRef, private renderer: Renderer) {
    }

    @HostListener('mouseenter') onMouseEnter() {
        this.renderer.setElementStyle(this.el.nativeElement,
                'backgroundColor', "green");
    }
}
```

<p **highlight** >I shall be green on mouse enter</p>

# Dependency Injection

- Supplies instance of a class with fully-formed dependencies

- Maintains a container of previously created service instances

- To use DI for a service, we register it as a provider in one of two ways: With in NgModule, or Component metadata

# Injection

```
import {Component, Inject} from "@angular/core";
import {ProductService} from "./product.services";
@Component({
     ..
    providers: [
        ProductService
    ]
})
export class ProductEditComponent {
    constructor(private productService:
ProductService
            ) {
   }
}
```

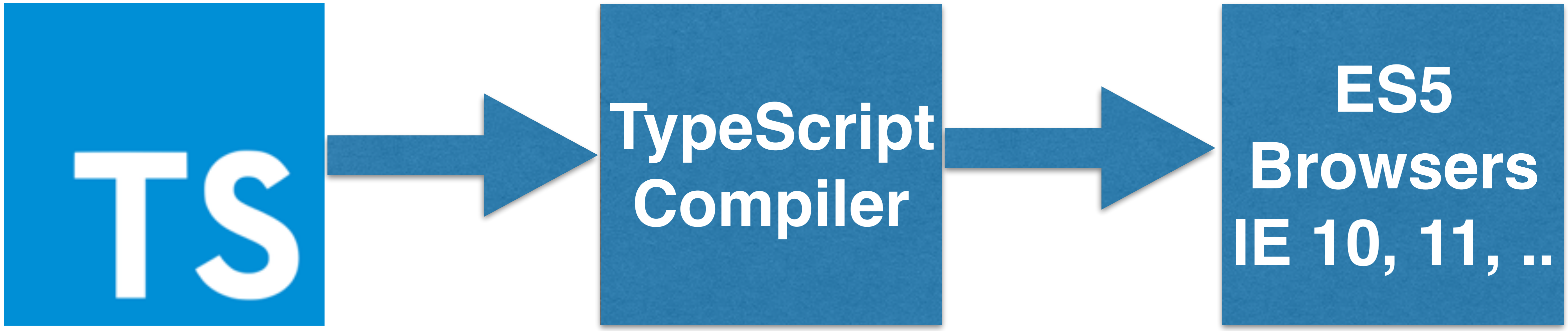
## But..

- Old Browsers like IE 10, IE 11, Android 4.0 Browsers doesn’t support ES 6
- Even current browsers doesn’t fully support ES6
- Node.js 6.x has support for 93% of ES6 features
- Browsers **never** support TypeScript anytime

**Solution?** →

## Tooling

# Transpiling



**Transpiler**: A **source-to-source compiler**, transcompiler or transpiler is a type of compiler that takes the source code of a program written in **one programming language** as its input and **produces** the equivalent **source code in another programming** language.

We need TypeScript Compiler to Build Angular 2 Application

# Node.js

- Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.
- Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.
- With node.js, you can write standalone Javascript program, execute it like
- C:\> node factorial.js (this execute JavaScript code in factorial.js file)

# Node.js, must because

- TypeScript compiler is written in JavaScript
- Running Unit Tests for your application
- Running E2E (Protractor) Test cases
- Development Environment
- Creating Production Bundle
- Managing dependent packages for all the above

# npm

- Node Package Manager (npm)
- Installed along with Node.js
- Manages packages for Node.js, Angular 2 projects for build time
- Similar to Maven, NuGet

## Install Node Modules Globally

install a module, which can be used in command line, global level. installed as part of %APPDATA%/npm folder on windows (may vary on Mac/Linux)

ex: install typescript compiler at system level

```
> npm install typescript -g
```

to uninstall global module

```
> npm uninstall typescript -g
```

## npm cli

//to create package.json, a project file similar to pom.xml for maven, packages.xml for nuget.net

```
> npm init
```

Dependencies shall be installed on “node_modules”

# npm cli

install typescript compiler at local path, files stored in node_modules

```
> npm install typescript
```

uninstall typescript from local path

```
> npm uninstall typescript
```

## npm cli

Install jasmine, update “dependencies” of package.json file

```
> npm install jasmine  --save
```

install karma, update devDependencies of package.json

```
> npm install karma  --save-dev
```

# TypeScript

install typescript at user account level, %APPDATA%/npm folder

```
> npm install typescript -g
```

compile a single file

```
> tsc input.ts
```

Compile all files in directory, sub-directory, exit after compilation

```
> tsc
```

Run on watch mode, automatically compile when file changed

```
> tsc  --watch
```

# tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": “node",
```

Compile TypeScript to ES5 for old browser

```
    "sourceMap": true,
```

Generate map file: debug

```
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
```

Support @Decorator Meta Data

```
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  },
  "exclude": [
    "node_modules/*"
  ]
}
```

Exclude folders from compilation

## ES6/TS

### product.ts

```
export class Product {

}
```

### list-component.ts

```
import Product
       from "./product"


let product:Product = new Product();
```

## ES5 with CommonJS

### product.js

```
    var Product = (function () {
        function Product() {
        }
        return Product;
    }());

    exports.Product = Product;
```

### list-component.js

```
var product_1 = require('./product');


var product = new product_1.Product()
```

## CommonJS

- Independent Group, published Module spec for Node.js projects
- Module/File level scoping in JavaScript
- Avoids Global variable issues in JavaScript
- Promote JavaScript Files are module, filling the gap of missing namespaces in JavaScript
- Defines “require(“..”)” and module.exports for JavaScripts
- Not part of ES5 Specs.

# CommonJS Modules

- All TypeScripts Files are converted into **CommonJS, ES5** Modules by TypeScript Compiler
- CommonJS files **cannot** be loaded into browser by <**script** src="..."> </script> Tag
- Because Browsers don't implement CommonJS "require" or "exports" or "module.exports"
- Using Script tag for loading 100+ files is **not efficient** for larger applications

# Module Loading

- Modular code is not required to develop with Angular, but it is recommended

- Allows us to easily use specific parts of a library

- Erases collisions between two different libraries

- We **don’t** have to include **script** tags for everything

- Because modules are not supported in Browser, we have to translate a module (file) to a pseudo module (wrapped function)

# Popular Module Loaders

- SystemJS
- WebPack

# SystemJS

**Primary part of workshop**

- Allow us to use single package manager, supports CommonJS, AMD Module loaders, called as Universal Module Loader
- Based on ES5 Standards
- Handles Dependencies correctly
- Can do client side transpiling => compile TypeScript into JavaScript (beware of performance, not recommended this way)
- Handle Large JS Application, but not recommended for production, since it downloads a lot of files, delay loading
- Advanced Bundler for production (minification etc)

## WebPack

- Most advanced Module Bundler
- Handles large applications
- Support JS files, CommonJS, AMD modules
- Support HTML, CSS, Images
- Does a lot with loaders and plug-ins

# Using Non-TypeScript Libraries

- DefinitelyTyped is a most famous open source projects, provides type declaration for most famous JavaScript libraries which are not written in TypeScript, ex: jQuery, Lodash etc

- https://github.com/DefinitelyTyped/DefinitelyTyped

You can install them using @types, special  package option

```
> npm install @types/jquery --save
> npm install @types/lodash --save
```

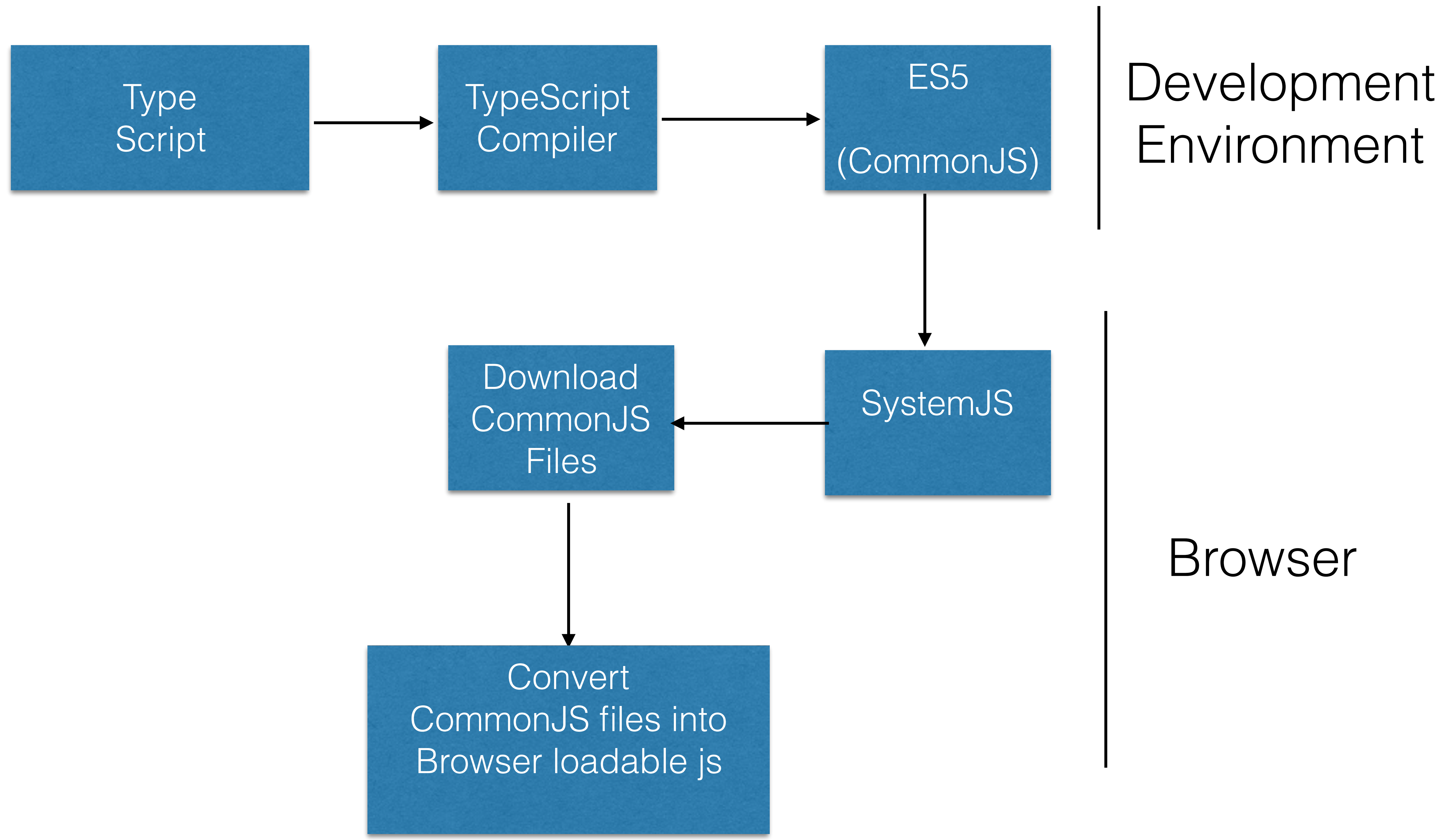
## Visual Studio Code

- Light-weight Development IDE developed by Microsoft by Using Node.js (Yes, node.js)
- Uses Github’s Electron framework, HTML5 and CSS, WebKit
- Free for Commercial or Open Source project.
- Has better support for TypeScript than any other editor in market now

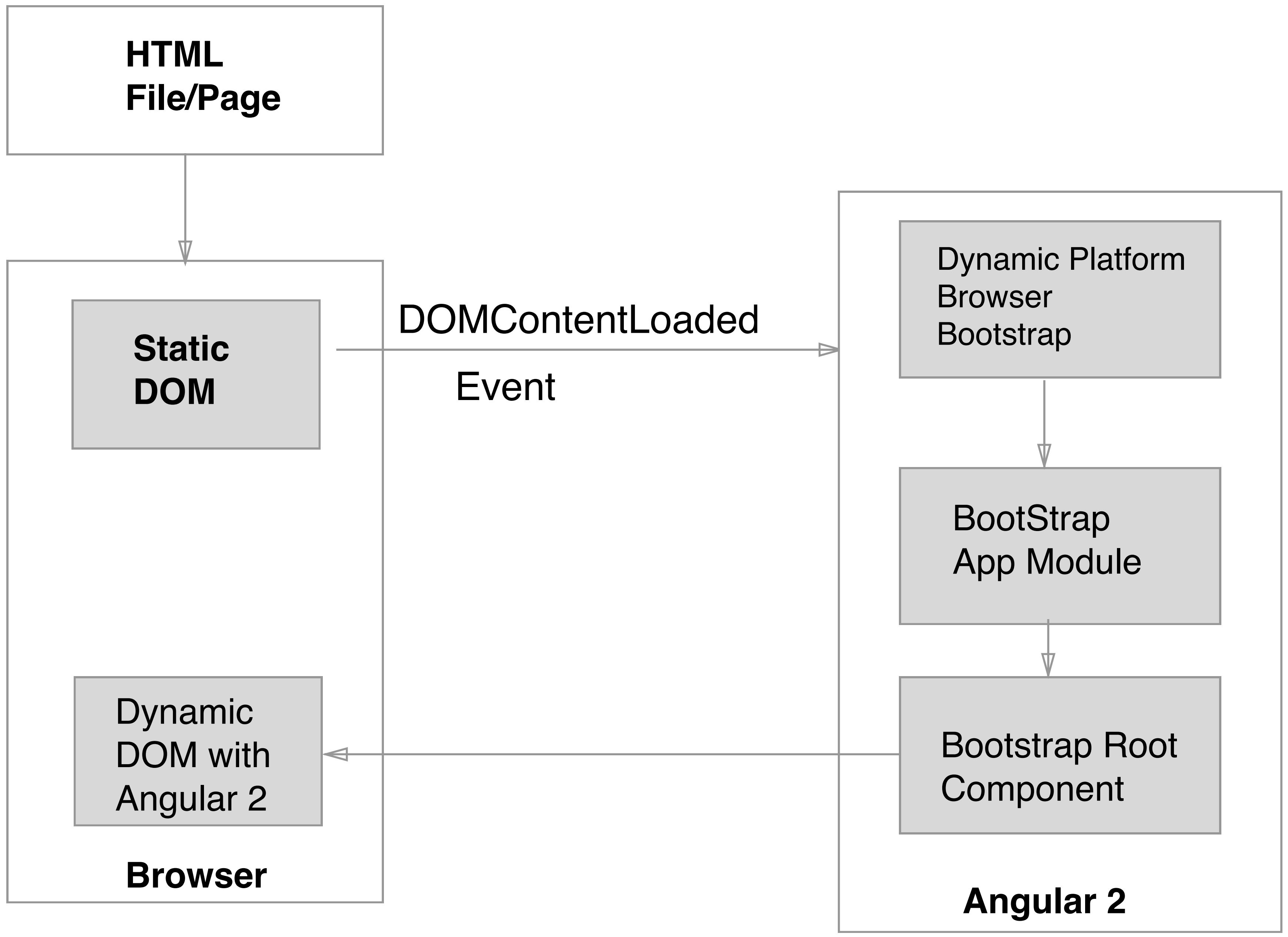
## Tooling not needed for Production

- Angular  runs in client side browser, you **don’t need Node.js, npm, node_modules, TypeScript** compilers, Module Loaders like **SystemJS,** or **WebPack for production environment**. Just take the bundle, assets, put them on Tomcat, IIS whatever servers you host.

# Tool Chain


Type Script
TypeScript Compiler
ES5 (CommonJS)
Development Environment
SystemJS
Download CommonJS Files
Convert CommonJS files into Browser loadable js
Browser

## BootStrapping


HTML File/Page
Static DOM
DOMContentLoaded Event
Dynamic Platform Browser Bootstrap
BootStrap App Module
Bootstrap Root Component
Dynamic DOM with Angular 2
Browser
Angular 2

## Page Load

- Download the HTML Page
- Download the System JS
- Download the System JS Configuration

- Wait for DOMContentLoaded event

- Bootstrap the Module
- Find the references to Components mentioned in bootstrap Module
- Mentioned as part of bootstrap list
- Start loading the  AppComponent mentioned as part of Module
- Search for HTML element based on selector name mentioned as part of App Component
- Load the component in HTML when match found

## Dynamic Bootstrapping

- Uses Just-In-Time (JIT) compiler
- Angular compile compiles the application in the browser and then launches the app

-
  ```
  // Compile and launch the module
  platformBrowserDynamic().bootstrapModule(App
  Module);
  ```

# platformBrowserDynamic?

- Bootstrap the AppModule
- Load all the dependent modules declared as part of “imports” section of AppModule
- Converts all the Templates to JavaScript module
- Download templateUrl files, convert them to JavaScript

## Static Bootstrapping

- Uses Ahead-Of-Time (AOT) compiler
- Launch faster
- Angular compiler runs ahead of time as part of the build process, producing a collection of class factories in their own files
- During this process, it produces AppModuleNgFactory
- The application code downloaded to the browser is much smaller than the dynamic equivalent
- ngc compiler

```
// Launch with the app module factory.
platformBrowser().bootstrapModuleFactory(AppModuleNgF
actory);
```

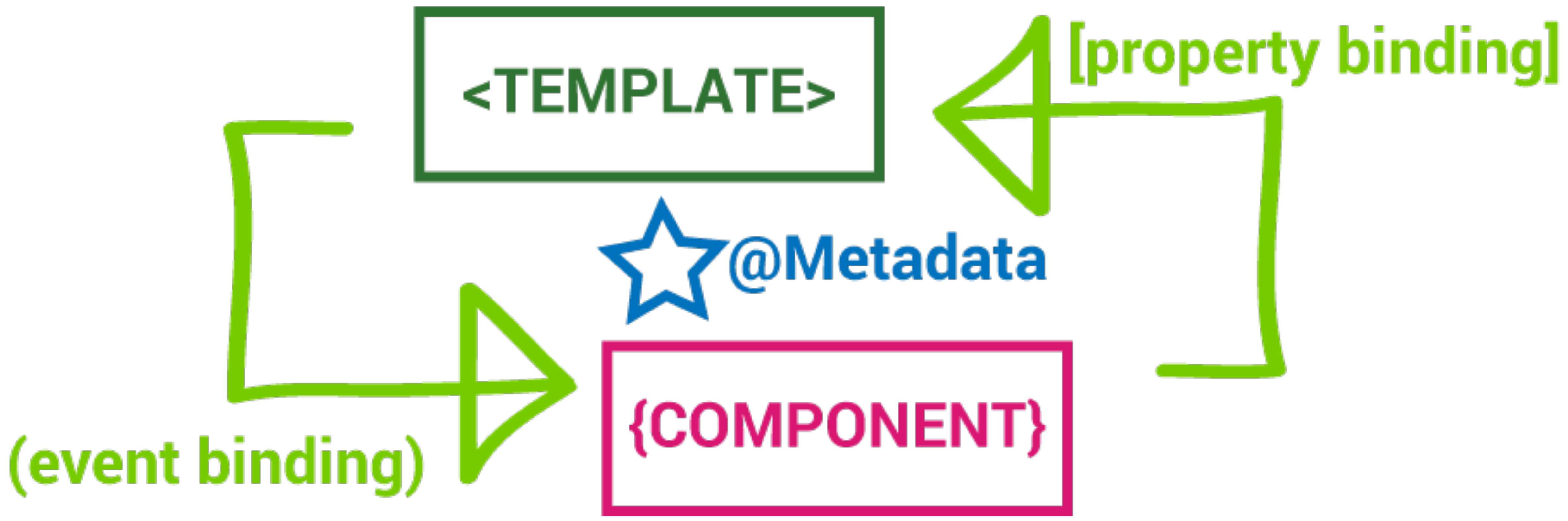
## Bootstrapping Summary

- Both the JIT and AOT compilers generate an AppModuleNgFactory class from the same AppModule source code
- The JIT compiler creates that factory class on the fly, in memory, in the browser.
- The AOT compiler outputs the factory to a physical file that we're importing here in the static version of main.ts

## Components

## Component Fundamentals

- Class
- Import
- Decorate
- Lifecycle Hooks



# Component

# Class

- Create the component as an ES6 class
- Properties and methods on our component class will be available for binding in our template
- Import the core Angular dependencies
- Import 3rd party dependencies
- Import your custom dependencies
- This approach gives us a more control over the managing our dependencies

# Decorate

- We turn our class into something Angular 2 can use by decorating it with a Angular specific metadata
- Use the @<decorator> syntax to decorate your classes
- The most common class decorators are @Component,
- @Injectable, @Directive and @Pipe
- You can also decorate properties and methods within your class
- The two most common member decorators are @Input and @Output

```
import {Component, OnInit} from "@angular/core";
@Component({
    selector: ‘contact’
    templateUrl: ‘./contact.component.html',
    styles: [],
    styleUrls: ['./contact.component.css']
})
export class ContactComponent implements OnInit {
    //inject dependent services
    constructor(private dataService:DataService) {
    }
    ngOnInit() {
         //put initialization code here
    }
    ngOnDestroy() {
        //cleanup resources used in component
    }
}
```

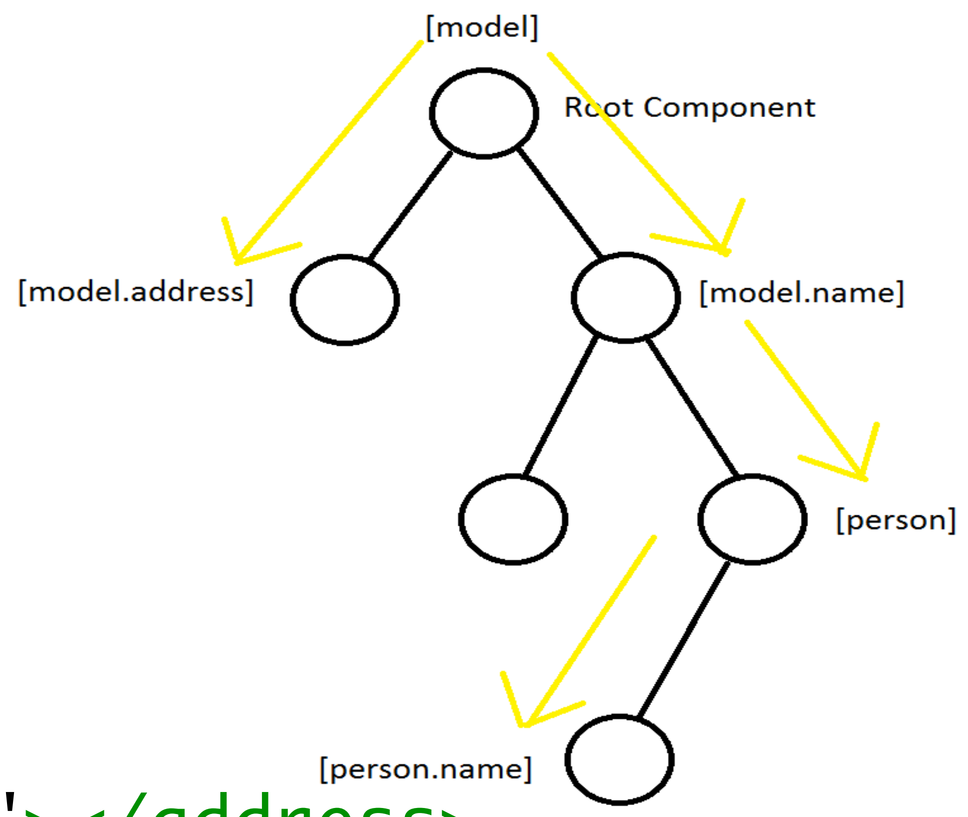
<contact> </contact>

## Change Detection

- Each component has its own change detector
- Input and Output bindings are unidirectional
- @Input() —> Downwards
- @Output()  —> Upwards
- Acyclic Directional Graph leads to more efficient change detection
- One digest cycle to detect changes

# @Input Binding

```
import { Component, Input } from '@angular/core';
@Component({
    selector: 'address',
    template: `<p>{{address.street}}</p>`
})
export class AddressComponent {
    //passed as argument to component
    @Input() addressInfo:AddressInfo;
    constructor() { }
}
```



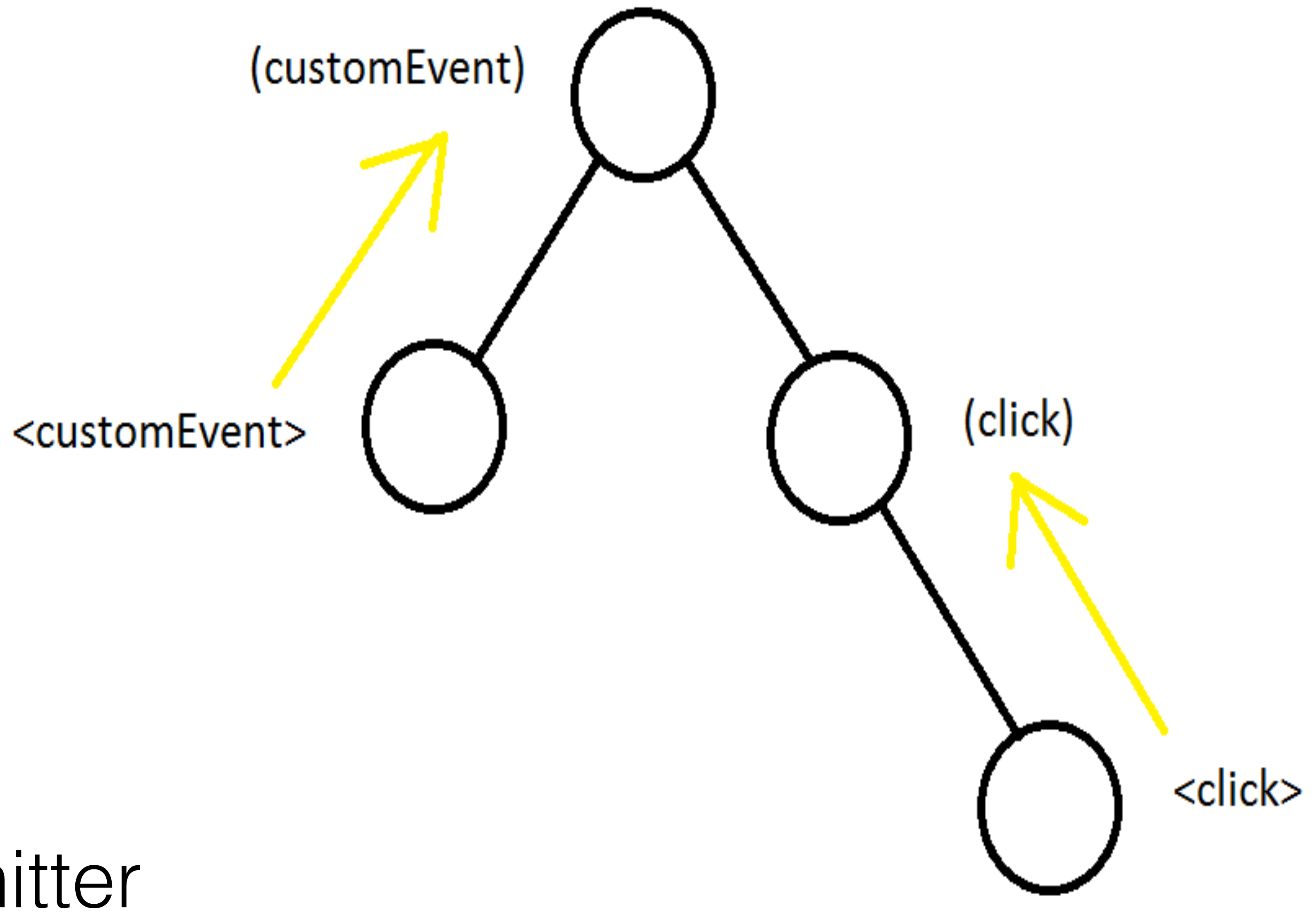
```
from contact.html

<address [addressInfo]="billingAddress"></address>
```

Input Bindings  Data flows from parent to child
Pass data into child component from parent

# @Output Binding

Output Bindings —> Events flows from child to parent



EventEmitter

## @Output

```
@Component({
    selector: 'contact',
    template: `<address [addressInfo]='mainAddress'
                (contactUsEvent) = 'contactUs($event)'>
                </address>`
})
export class ContactComponent {
    mainAddress: AddressInfo;
    contactUs(address: AddressInfo) {
    }
}
```

```
@Component({
    selector: 'address',
    template: `<p>{{addressInfo.street}}</p>
    <button (click)="contactClicked()">Contact</button>
    `
})
export class ContactAddress {
    @Input() addressInfo: AddressInfo;
    @Output() contactUsEvent = new EventEmitter();
    contactClicked() {
        this.contactUsEvent.emit(this.addressInfo);
    }
}
```

# Binding: Summary

- Input Bindings
- Native DOM Properties (style, width, height, etc)
- Custom Component Properties
- Output Bindings
- Native DOM Events (i.e. click, keyup, mousemove, etc)
- Custom Component Events (i.e. EventEmitter)

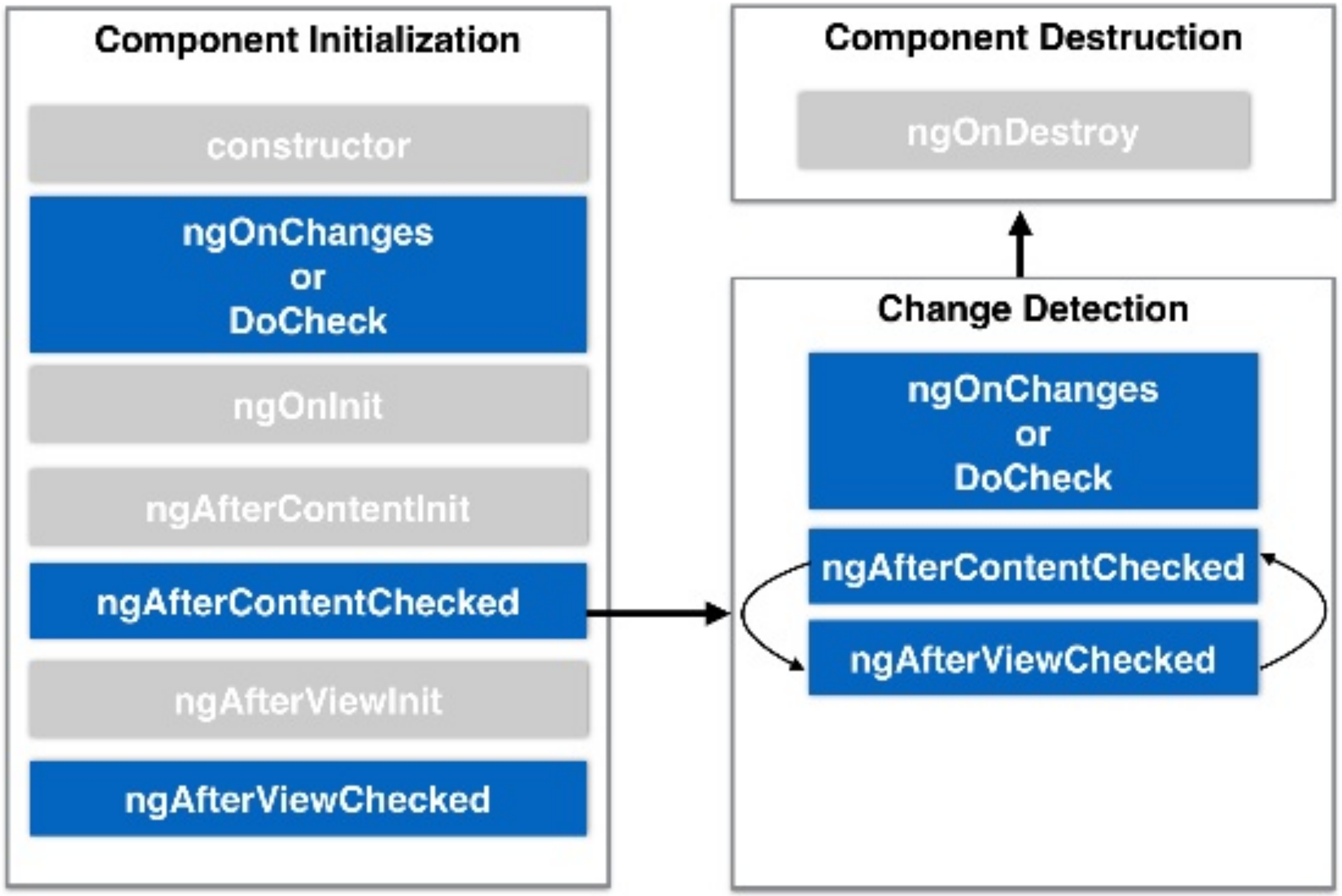
# Lifecycle Hooks

- Allow us to perform custom logic at various stages of a component's life
- Data isn't always immediately available in the constructor
- Only available in TypeScript
- The lifecycle interfaces are optional. We recommend adding them to benefit from TypeScript's strong typing and editor tooling
- Implemented as class methods on the component class

## Lifecycle Hooks (cont.)

- ngOnChanges - called when an input or output binding value changes
- ngOnInit - after the first ngOnChanges
- ngDoCheck - developer's custom change detection
- ngAfterContentInit - after component content initialized
- ngAfterContentChecked - after every check of component content
- ngAfterViewInit - after component's view(s) are initialized
- ngAfterViewChecked - after every check of a component's view(s)
- ngOnDestroy - just before the directive is destroyed.

# Component Lifecycle



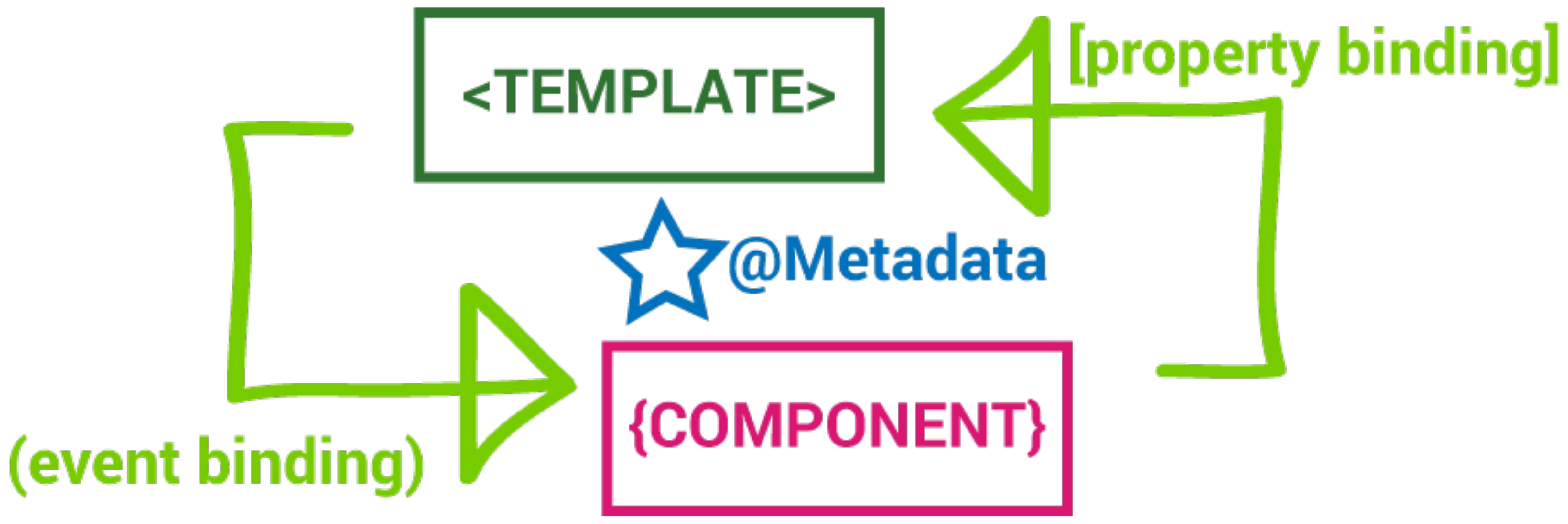
Grey boxes called only once

Blue boxes called when there is change

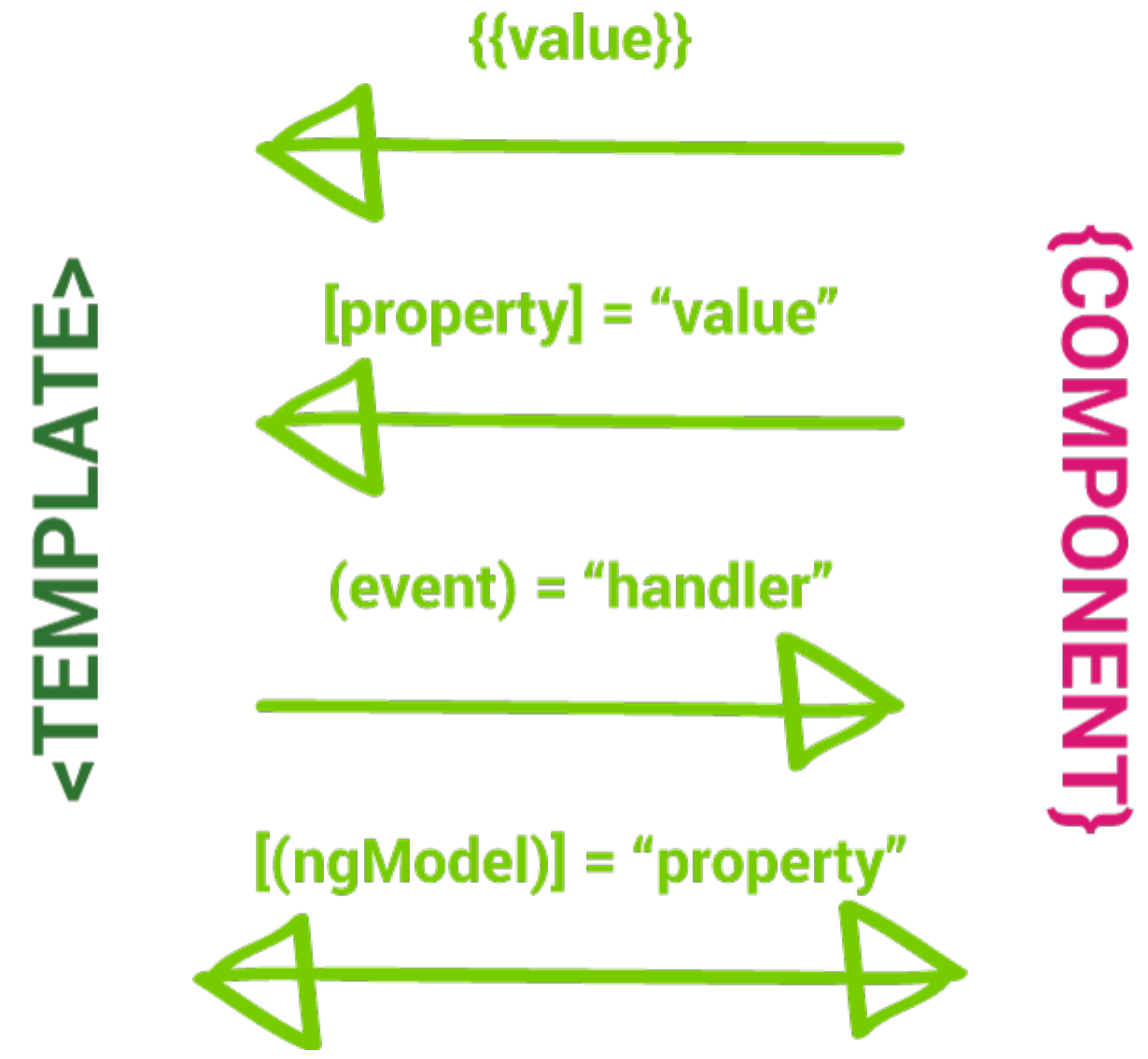
## Templates

# Templates

- Interpolation
- Method Binding
- Property Binding
- Two Way Binding
- Hashtag Operator
- Asterisk Operator
- Elvis Operator (?.)



# Template



# Data Binding

# Interpolation

- Allows us to bind to component properties in out template
- Defined with the double curly brace syntax:
- {{ propertyValue }}
- We can bind to methods as well
- Angular converts interpolation to property binding

# Interpolation

```
<span>{{result}}<span>

<span>{{1 + 2 }}<span>

<span>{{ a  * b }}<span>


<span>{{ getResult() }}<span>
```

## Property Bindings []

- Flows data from the component to an element
- Created with brackets <img [src]=”image.src” />
- Canonical form is bind-attribute e.g.
- <img bind-src=”image.src” />

```
<span [style.color]="color">Green Text!</span>

<address [addressInfo]=“branchAddress"> </address>
```

# Property Bindings (cont.)

- Don’t use the brackets if:
- the target property accepts a string value
- the **string** is a **fixed value** that we can bake into the template
- this initial value never changes

```
<a routerLink="/about">About</a>
```

# Event Bindings ()

- Flows data from an element to the component
- Created with **parentheses** <button (click)=contactUs()”></ button>
- Canonical form is on-event e.g. <button **on-click**=contactUs()”></button>
- Get access to the event object inside the method via
- $event e.g. <button (click)=contactUs($event)”></ button>

# Two-way Bindings [()]

- Really just a combination of property and event bindings
- Used in conjunction with ngModel
- Referred to as "hotdog in a box"
- Two-way data binding combines the @input and @output binding into a single notation

# Two-Way Binding

```html
<!-- syntactic sugar -->
<input [(ngModel)]="name">

<!-- Angular way -->
<input [ngModel]="name" (ngModelChange)="name=$event">
```

## Asterisk Operator *

- Asterisks indicate a directive that modifies the HTML
- It is syntactic sugar to avoid having to use template  elements directly

# *ngIf

ngIf is a **structural** directive

Useful for display or hide elements based on logical condition (true | false)

If the result of the condition is ‘false’, the element is **removed from DOM**

If the result of condition is ‘true’, **the element is added to DOM**

```
<!-- never displayed -->
<div *ngIf="false">
    <span>Not shown</span>
</div>

<!– always displayed -->
<div *ngIf="true">
    <span>Shown Always</span>
</div>

<!-- displayed if a  > than b -->
<div *ngIf="a > b"></div>

<!-- displayed if myFunc returns true
value -->
<div *ngIf="myFunc()"></div>
```

# *ngIf - Syntactic Sugar

```
<div *ngIf=”userIsVisible”>{{user.name}}</div>

<template [ngIf]="userIsVisible">
<div>{{user.name}}</div>
</template>
```

## *ngSwitch

```
<div *ngSwitch="direction">
      <span *ngSwitchCase="'oneway'">Onward journey</span>

      <span *ngSwitchCase="'twoway'">Two way journey</span>

      <span *ngSwitchDefault>Select an Option</span>
</div>
```

Add/Remove DOM Elements based on condition

# ngSwitch

```
<div [ngSwitch]="direction”>
     <template [ngSwitchCase]=“'oneway'">
      <span>Onward journey</span>
     </template>
     <template [ngSwitchCase]=“'twoway'">
       <span>Two way journey</span>
     </template>
     <template ngSwitchDefault>
         <span>Select an Option</span>
      </template>
</div>
```

without * sugar

# *ngFor

- Repeat a DOM element(s) over the collections, apply value of each item in the collection

```
 <ul>
     <li *ngFor="let product of products" >
         {{product.name}}
     </li>
 </ul>


<ul>
    <li *ngFor="let brand of brands; let i = index;
let o = odd;">
        {{i}}.{{brand}} - {{o}}
    </li>
</ul>
```

## ngFor

```
<ul>
    <template ngFor let-product [ngForOf]="products" >
    <li>
        {{product.name}}
    </li>
    </template>
</ul>
```

# trackBy

```
<ul>
     <li *ngFor="let product of products;
trackBy:trackByProduct"  >
           {{product.name}}
      </li>
</ul>


class ProductListComponent {
         ...
         trackByProduct(index: number, product: any) {
             return product.id;
          }
...}
```

Use trackBy to optimize the performance of DOM Rendering in case of list refresh, without trackBy Angular cannot understand object identity, it render the whole list again and again. trackBy helps to render only the specific object when added/removed

## Hashtag Operator #

- The hashtag (#) defines a local variable inside our template

- Template variable is available on the same element,  sibling elements, or child elements of the element on  which it was declared

- To consume, simply use it as a variable without the hashtag

# #Reference Variable

```
<!-- productName refers to the input element;
pass its `value` to an event handler -->

<input #productName placeholder="product name">

<button (click)="updateName(productName.value)">
  Update
</button>
```

```
<!-- productName refers to the input element-->
<input ref-year placeholder="release year">
<button (click)="updateYear(year.value)">Update</button>
```

Both #year and ref-year does same function, # is a syntactic sugar

# Elvis Operator

- Denoted by a question mark immediately followed by a period e.g. ?.
- If you reference a property in your template that does not exist, you will throw an exception.
- The elvis operator is a simple, easy way to guard against null and undefined properties

## Elvis ?.

```
<input  name="name"  #productName >

<button (click)="updateName(productName?.value)">
  Update
</button>
```

Won’t crash when productName is null

# NgStyle

NgStyle is  a directive to set CSS properties for DOM

```
 <!-- Syntax: -> [style.<cssproperty>]="value" -->

   <div [style.background-color]="'yellow'">
        Uses fixed yellow background
   </div>

   <!-- OR -->
   <div [ngStyle]="{color: 'white', 'background-color':
'blue'}">
       Uses fixed white text on blue background
   </div>
```

## Dynamic Style

```
<input type="text" name="color" [(ngModel)]="fontColor"
#colorinput>

<input type="number" name="fontSize" [(ngModel)]="fontSize"
#fontinput>

 <span [ngStyle]="{color: fontColor}" [style.font-
size.px]="fontSize">
    Span Text {{colorinput.value}} {{fontinput.value}}
</span>
```

# NgClass

```
<div [ngClass]="{bordered: false}">This is not bordered</div>
```

Dynamically update css classes, modify “class” attributes of css, add/remove attributes based on condition

```
<div [ngClass]="{bordered: true}">This is bordered</div>

<div [ngClass]="{bordered: isShowBorder, largeText : true}">
    This is bordered with large text
</div>

output

<div class=“borderd largeText”>
</div>

<div [ngClass]="{bordered: isBordered()}">Dynamic Border</div>
```

# NgNonBindable

How not to bind {{ }} expressions or prevent Angular to compile a specific section of HTML?

```
<span class="pre" ngNonBindable>
     This is what {{ content }} rendered
</span>
```

Output

This is what {{ content }} rendered

## Pipes

# Pipes

- Pipes transform displayed values within a template.
- Data Flow: Get data  Transform data   Display on User Interface
- Pipe takes in data as input
- Transform to Output
- Takes optional parameters

# Using Pipe

To use Pipe in template, put pipe symbol ‘|’ next to input/expression and the pipe name

```
<p>{{date | date:'mediumDate' }}</p>

<p>{{price | currency}}</p>

<p>{{title | uppercase}}</p>
```

# Currency

```
<p> {{price | currency}}</p>
<!-- USD99.99 -->

<p>{{price | currency:'INR'}}</p>
<!-- INR99.99 -->
```

```
class ProductDetailComponent {
        price: number = 99.99;
}
```

## Date

```
<p>{{currentDate | date}}</p>
<!--Aug 21, 2016 -->

<p>{{currentDate | date:'fullDate'}}</p>
<!--Sunday, August 21, 2016 -->

<p>{{currentDate | date:'mediumDate'}}</p>
<!--Aug 21, 2016 -->

<p>{{currentDate | date:'shortTime'}}</p>
<!--10:34 AM -->
```

```
class ProductDetailComponent {
        currentDate: Date = new Date();
}
```

## async

- Display results from resolved promise
- Useful for Async Operations
- Wait for Promise.then method and render the data returned from Promise object
- Promise defer the execution, get results later, once available

# Async

```
export class BuildinPipesComponent {
    promise: Promise<string>;

    constructor() {
        this.promise = new Promise(function (resolve, reject) {
            setTimeout(function () {
                resolve("resolved after 3 seconds")
            }, 3000);
        })
    }
}
```

```
<p>promise | async  - {{promise | async}}</p>
<!-- no output for 3 seconds -->
<!-- promise | async -   -->

<!-- after 3 seconds -->
<!-- promise | async - resolved after 3 seconds -->
```

## Chaining Pipe

```
{{ birthday | date | uppercase}}
```

```
<p>promise | async  - {{promise | async | uppercase}}</p>
<!-- promise | async -   -->
<!-- after 3 seconds -->
<!-- promise | async - RESOLVED AFTER 3 SECONDS -->
```

Output of pipe is passed to input of another pipe

async can be chained, anything comes after async, shall be in waiting state till async resolved

# Custom Pipes

```
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({
    name: 'power'
})
export class PowerPipe implements PipeTransform {
    transform(value: number, exponent: string): number {
        let exp = parseInt(exponent);
        return Math.pow(value, exp);
    }
}
```

2 is passed to **value**
3 is passed as **exponent**

```
<p> Power: {{2 | power:3}} </p>
<!--Power: 8  -->
```

## Services

# Services

- Services are just classes, decorated with @Injectable
- A Service object created by Angular framework automatically using Dependency Framework
- Services are useful for writing reusable business logic or communication logic with server

# Service Definition

```
import {Injectable} from '@angular/core';
@Injectable()
export class DataService {
    getProducts() {
        //return list of products from server
    }
    saveProduct(product: Product) {
        //save product to server
    }
}
```

## Injecting into Service

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';

@Injectable()
export class DataService {
    constructor(private http: Http,

        @Inject('apiEndPoint')
        private endPoint: string){

     }
                    ...
                    ...
}
```

# Injecting Service into Component

**Multiple Instances**

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

@Component({
    providers: [DataService]
})
export class ProductDetailComponent {
    constructor(private dataService: DataService) {
    }
}
```

- Defining providers at component level create private service per component
- Not a Singleton Service Object, every component instance shall have its own service instance created

## Singleton Service Provider

```
import {NgModule} from '@angular/core';
@NgModule({
    ...
    providers: [ DataService ]
})
export class ProductModule {}
```

```
import {Component} from '@angular/core';
@Component(…)
export class ProductDetailComponent {
    constructor(private dataService: DataService) {

    }
}
```

- Services defined in Module are singleton instances
- But can be over-written by component level provider

## Dependency Injection

# Dependency Injection

- Helps to instantiate objects on need basis
- Maintain multiple, singleton copies of objects, maintain life time of created objects
- Inversion of Control - Avoid explicit object creation by using Type name like   “new ProductService()”, instead framework creates objects, pass to component, pipes etc

## Interfaces

```
@Injectable()
abstract class ProductService {
   abstract getProducts() ;
}
```

## Product Web Service

```
import {Http, Response} from "@angular/http";
@Injectable()
class ProductWebService extends ProductService {
   constructor(private http: Http) {}

   getProducts() {
       return http.get("/api/products")
       .map( (response : Response) => response.json())
   }
}
```

## Interfaces

```
@Injectable()
abstract class ProductService {
   abstract getProducts()
}
```

## Product Mock  Service for Testing

```
@Injectable()
class ProductMockService extends ProductService {

   getProducts() {
       return JSON.parse(localStorage.getItem("products"))
   }
}
```

## Provider Configuration

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

@NgModule({
    ...
    providers: [
    {
      provide: ProductService,
      useClass: ProductWebService
    }
    ]
})
//For Testing/Mock use
{
    provide: ProductService,
    useClass: ProductMockService
}
```

## Inject Service

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

Component({

})
export class ProductListComponent {
    constructor(private productService:
ProductService) {

    }
}
```

## Injecting Primitive Types

```
providers: [{
    provide: 'apiEndPoint',
    useValue: 'http://example.com'
},
{   provide: 'IS_PROD',
    useValue: false
},
{   provide: 'TIMEOUT',
    useValue: 10000
}]
```

Explicit @Inject decorator needed since injection works based on data types. Primitive cannot be injected

```
constructor(
    @Inject('IS_PROD') is_prod: boolean,
    @Inject('apiEndPoint') apiEndPoint: string,
    @Inject('TIMEOUT') timeout: number
) {}
```