# Spread Operator

**The Spread Operator allows splitting an array to single arguments which are passed to the function as separate arguments**. The spread operator takes either an array or an object and expands it into its set of items. The **spread syntax** allows an iterable such as an array expression or string to be expanded in places where zero or more arguments or elements are expected, or an object expression to be expanded in areas where zero or more key-value pairs are expected.

 Example :

const array = [1, 2];

const combined = [...array, 3, 4];

console.log(combined);

 let obj = { ...object };

 myFunction(...iterableObject);

# Arrow functions

An **arrow function expression** is a syntactically compact alternative to a regular [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function#_blank)

Another significant advantage it offers is the fact that it does not bind its own **this**. In other words, the context inside arrow functions is lexically or statically defined.

The value of this inside arrow functions is not dependent on how they are invoked or how they are defined. It depends only on its enclosing context.

# Directives

Directives add behavior to an existing DOM element or an existing component instance.

## Types of directives in Angular

* Component
* Attribute Directive
* Structural Directive

The basic difference between a component and a directive is that a component has a template, whereas an attribute or structural directive does not have a template

# What Are @HostBinding() and @HostListener() in Angular?

**@HostListener()** function decorator allows you to handle events of the host element in the directive class.

**@HostBinding()** function decorator allows you to set the properties of the host element from the directive class.

# Renderer2

The **Renderer2** class is an abstraction provided by Angular in the form of a service that allows to manipulate elements of your app without having to touch the DOM directly.

Functions:

**addClass** : this.renderer.addClass(this.el.nativeElement, 'wild');

**removeClass** : this.renderer.removeClass(this.el.nativeElement, 'wild');

**createElement** : this.renderer.createElement('div');

**createText** : this.renderer.createText('Hello world!');

**appendChild** : this.renderer.appendChild(div, text);

**setAttribute** : this.renderer.setAttribute(this.el.nativeElement, 'aria-hidden', 'true');

**removeAttribute** : this.renderer.removeAttribute(this.el.nativeElement, 'aria-hidden');

**setStyle** : this.renderer.setStyle(this.el.nativeElement, 'border-left', '2px dashed olive');

**removeStyle** : this.renderer.removeStyle(this.el.nativeElement, 'border-left')

**setProperty** : this.renderer.setProperty(this.el.nativeElement, 'alt', 'Cute alligator');

# Content Projection:

Content projection allows you to insert a shadow DOM in your component. To put it simply, if you want to insert HTML elements or other components in a component, then you do that using the concept of content projection. In Angular, you achieve content projection using < ng-content >< /ng-content >.  You can make reusable components and scalable applications by properly using content projection.

When the data is produced by the Observable itself, we call it a **cold Observable**. When the data is produced outside the Observable, we call it a **hot Observable.**

**Observable                  ┃     BehaviorSubject/Subject**         ┃

┣━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╋━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┫

│ Is just a function, no state        │ Has state. Stores data in memory    │

├─────────────────────────────────────┼─────────────────────────────────────┤

│ Code run for each observer          │ Same code run                       │

│                                     │ only once for all observers         │

├─────────────────────────────────────┼─────────────────────────────────────┤

│ Creates only Observable             │Can create and also listen Observable│

│ ( data producer alone )             │ ( data producer and consumer )      │

├─────────────────────────────────────┼─────────────────────────────────────┤

│ Usage: Simple Observable with only  │ Usage:                              │

│ one Obeserver.                      │ \* Store data and modify frequently  │

│                                     │ \* Multiple observers listen to data │

│                                     │ \* Proxy between Observable  and     │

│                                     │   Observer

# tsconfig.json

The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

 This file is used to give the options about TypeScript used for the Angular JS project.

{

   "compilerOptions": {

      "target": "es5",

      "module": "commonjs",

      "moduleResolution": "node",

      "sourceMap": true,

      "emitDecoratorMetadata": true,

      "experimentalDecorators": true,

      "lib": [ "es2015", "dom" ],

      "noImplicitAny": true,

      "suppressImplicitAnyIndexErrors": true

   }

}

Following are some key points to note about the above code.

* The target for the compilation is es5 and that is because most browsers can only understand ES5 typescript.
* The sourceMap option is used to generate Map files, which are useful when debugging. Hence, during development it is good to keep this option as true.
* The "emitDecoratorMetadata": true and "experimentalDecorators": true is required for Angular JS decorators. If not in place, Angular JS application will not compile.

# package.json

This file contains information about Angular 2 project.

Some key points to note about the above code −

There are two types of dependencies, first is the dependencies and then there are dev dependencies. The dev ones are required during the development process and the others are needed to run the application.

The "build:watch": "tsc -p src/ -w" command is used to compile the typescript in the background by looking for changes in the typescript files.

# Observable.subscribe()

The observable subscribe method is used by angular components to subscribe to messages that are sent to an observable.

# Subject.next()

The subject next method is used to send messages to an observable which are then sent to all angular components that are subscribers (a.k.a. observers) of that observable.

<http://jasonwatmore.com/post/2018/06/25/angular-6-communicating-between-components-with-observable-subject>

<https://www.google.com/amp/s/scotch.io/tutorials/lazy-loading-in-angular-v2/amp>

# Lifecycle Hooks

Angular calls the lifecycle hook methods in the following sequence

1.      **ngOnChanges()** : Respond when Angular (re)sets data-bound input properties. The method receives a SimpleChanges object of current and previous property values.

Called before ngOnInit() and whenever one or more data-bound input properties change.

2.      **ngOnInit**() : Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.

Called once, after the first ngOnChanges().

3.      **ngDoCheck**() : Detect and act upon changes that Angular can't or won't detect on its own.

Called during every change detection run, immediately after ngOnChanges() and ngOnInit().

4.      **ngAfterContentInit**() : Respond after Angular projects external content into the component's view / the view that a directive is in.

Called once after the first ngDoCheck().

5.      **ngAfterContentChecked**() : Respond after Angular checks the content projected into the directive/component.

Called after the ngAfterContentInit() and every subsequent ngDoCheck().

6.      **ngAfterViewInit**() : Respond after Angular initializes the component's views and child views / the view that a directive is in.

Called once after the first ngAfterContentChecked().

7.      **ngAfterViewChecked**() : Respond after Angular checks the component's views and child views / the view that a directive is in.

Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked().

8.      **ngOnDestroy**() : Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.

Called just before Angular destroys the directive/component.

# What are the key components of Angular?

Angular has the below key components,

* **Component:** These are the basic building blocks of angular application to control HTML views.
* **Modules:** An angular module is set of angular basic building blocks like component, directives, services etc. An application is divided into logical pieces and each piece of code is called as "module" which perform a single task.
* **Templates:** This represent the views of an Angular application.
* **Services:** It is used to create components which can be shared across the entire application.
* **Metadata:** This can be used to add more data to an Angular class.

# What is metadata?

Metadata is used to decorate a class so that it can configure the expected behavior of the class. The metadata is represented by decorators

* **Class decorators**, e.g. @Component and @NgModule
* **Property decorators** Used for properties inside classes, e.g. @Input and @Output
* **Method decorators** Used for methods inside classes, e.g. @HostListener
* **Parameter decorators** Used for parameters inside class constructors, e.g. @Inject