**Angular 6:**

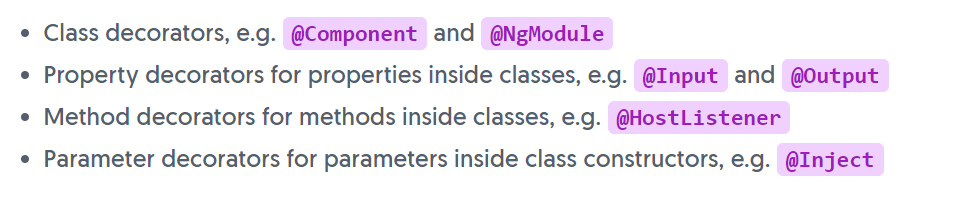
* Angular Element
* Service worker
* Internationalization (i18n)
* Bazel Compiler
* ng-add / ng-update
* ngModelChange
* TypeScript 2.7 support
* Improved decorator error messages
* <ng-template> updated to <template>

**Angular 9:**

* Default Ivy in v9
* Phantom Template Variable Menace
* Dependency Injection Changes in Core
* Service Worker Updates
* i18n Improvements
* More reliable ng update
* API Extractor Updates
* Typescript 3.7 support
* Component Harness
* ModuleWithProviders Support

**1.Decorator?**

**Decorators** are a design pattern that is used to separate modification or decoration of a class without modifying the original source code



**2.Types of Dirctives in Angular?**

Directives:

Directives are classes that add additional behavior to elements in your Angular applications.

A directive is a class in Angular that is declared with a **@Directive** decorator.  
Every directive has its own behaviour and can be imported into various components of an application

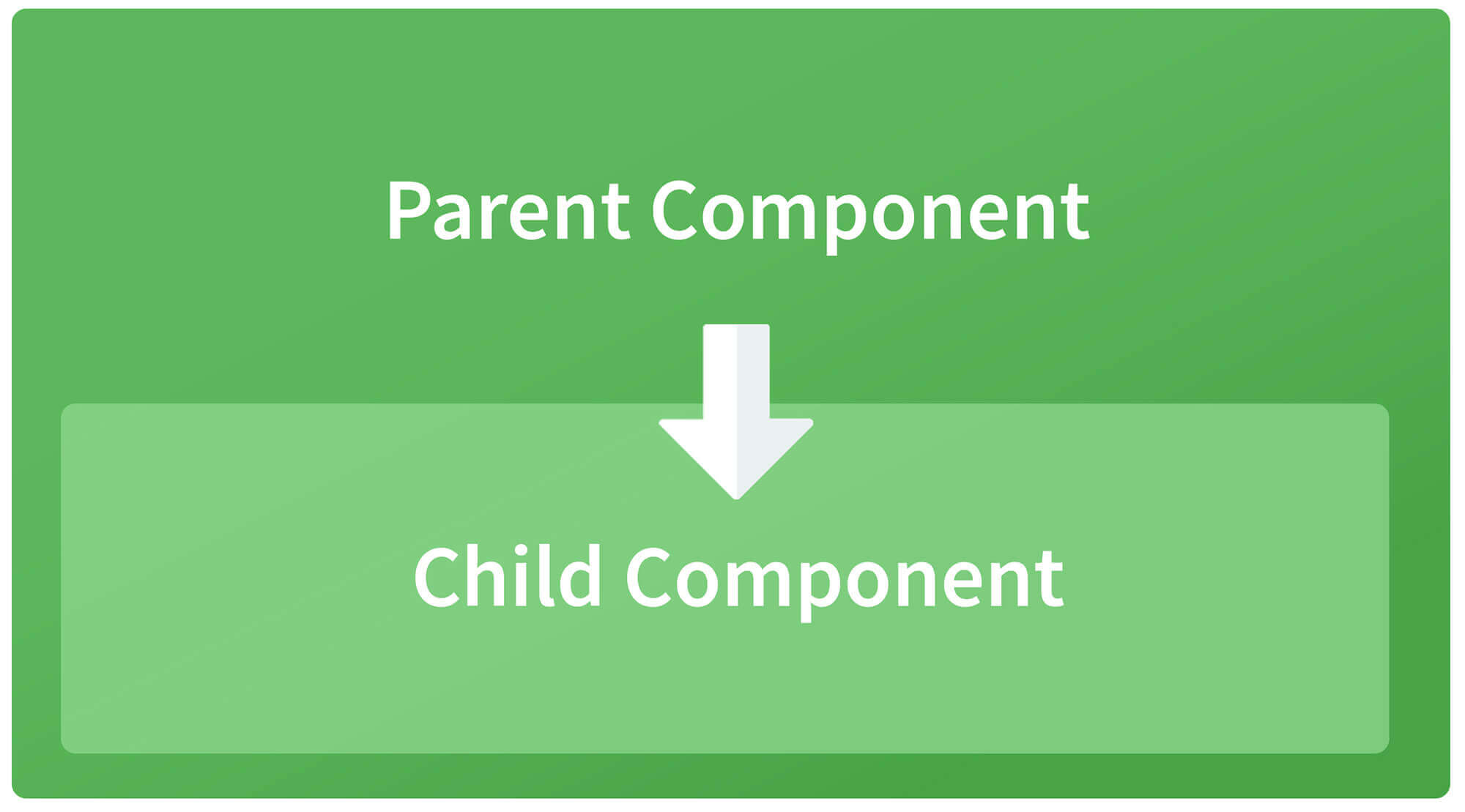
Directives can change the appearance or behavior of DOM elements and Angular components.

**There are four types of directives in Angular,**

* Components **directives**.
* Structural **directives**.
* Attribute **directives**.
* Custom **Directive**.

### 13. How does one share data between components in Angular?

Following are the commonly used methods by which one can pass data between components in angular:



**Parent to child using @Input decorator**  
  
Consider the following parent component:

@Component({

selector: 'app-parent',

template: `

<app-child [data]=data></app-child>

` ,

styleUrls: ['./parent.component.css']

})

export class ParentComponent{

data:string = "Message from parent";

constructor() { }

}

In the above parent component, we are passing “data” property to the following child component:

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

@Component({

selector: 'app-child',

template:`

<p>{{data}}</p>

`,

styleUrls: ['./child.component.css']

})

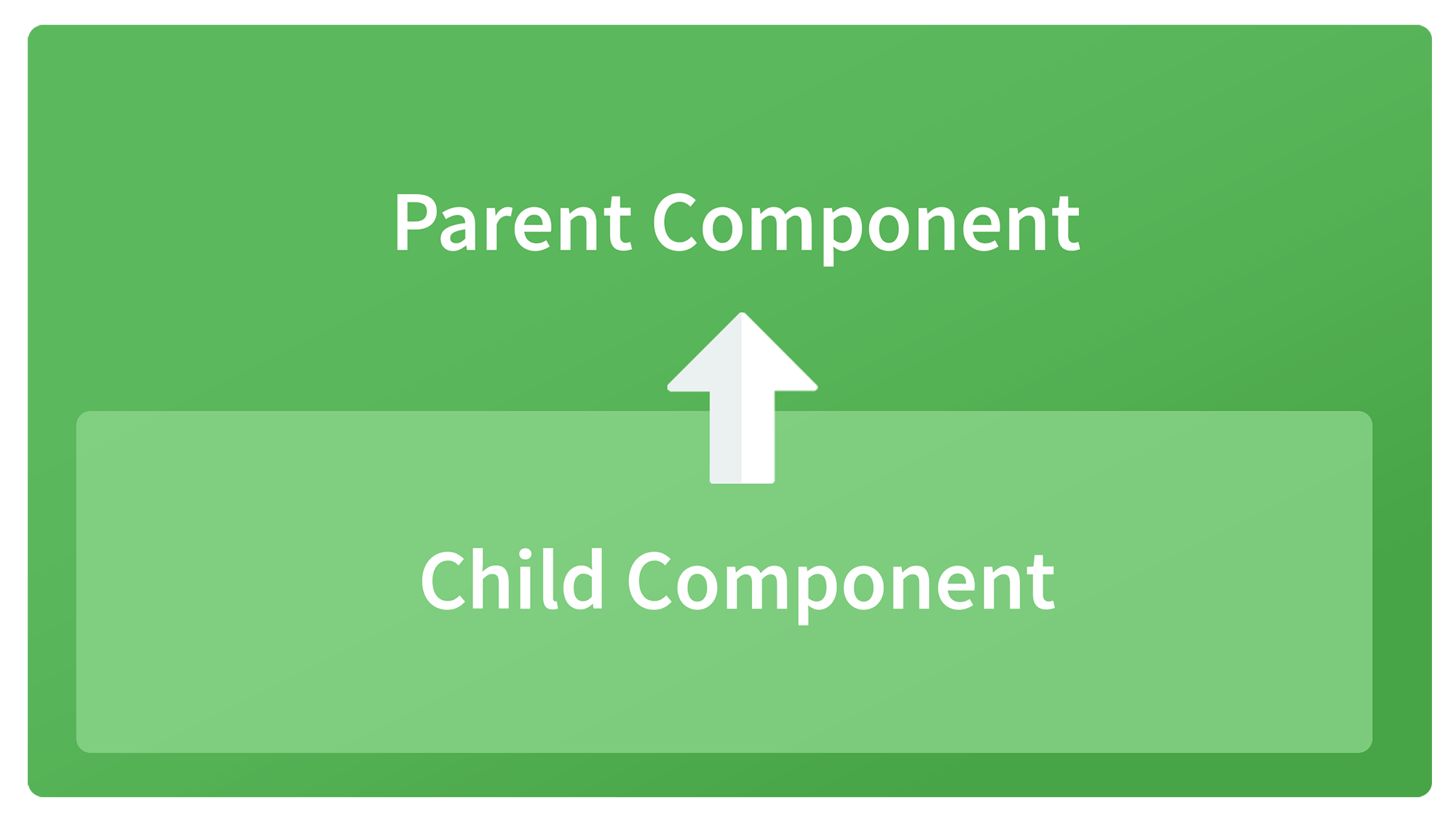
export class ChildComponent {

@Input() data:string

constructor() { }

}

In the child component, we are using @Input decorator to capture data coming from a parent component and using it inside the child component’s template.



**Child to parent using @ViewChild decorator**  
  
Child component:

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

@Component({

selector: 'app-child',

template:`

<p>{{data}}</p>

`,

styleUrls: ['./child.component.css']

})

export class ChildComponent {

data:string = "Message from child to parent";

constructor() { }

}

Parent Component

import { Component,ViewChild, AfterViewInit} from '@angular/core';

import { ChildComponent } from './../child/child.component';

@Component({

selector: 'app-parent',

template: `

<p>{{dataFromChild}}</p>

` ,

styleUrls: ['./parent.component.css']

})

export class ParentComponent implements AfterViewInit {

dataFromChild: string;

@ViewChild(ChildComponent,{static:false}) child;

ngAfterViewInit(){

this.dataFromChild = this.child.data;

}

constructor() { }

}

In the above example, a property named “data” is passed from the child component to the parent component.  
**@ViewChild** decorator is used to reference the child component as “child” property.  
Using the **ngAfterViewInit** hook, we assign the child’s data property to the messageFromChild property and use it in the parent component’s template.  
  
**Child to parent using @Output and EventEmitter**  
  
In this method, we bind a DOM element inside the child component, to an event ( **click** event for example ) and using this event we emit data that will captured by the parent component:  
  
Child Component:

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

@Component({

selector: 'app-child',

template:`

<button (click)="emitData()">Click to emit data</button>

`,

styleUrls: ['./child.component.css']

})

export class ChildComponent {

data:string = "Message from child to parent";

@Output() dataEvent = new EventEmitter<string>();

constructor() { }

emitData(){

this.dataEvent.emit(this.data);

}

}

As you can see in the child component, we have used **@Output** property to bind an **EventEmitter**. This event emitter emits data when the button in the template is clicked.  
  
In the parent component’s template we can capture the emitted data like this:

<app-child (dataEvent)="receiveData($event)"></app-child>

Then inside the receiveData function we can handle the emitted data:

receiveData($event){

this.dataFromChild = $event;

}

*Q1*:

**What is *Routing Guard* in Angular?**

**Added** **Entry** C:\Users\758530\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\9D83F8F0.tmp

Answer

Angular’s **route guards** are interfaces which can tell the router **whether or not it should allow navigation**to a requested route. They make this decision by looking for a true or false return value from a class which *implements* the given guard interface.

There are five different types of guards and each of them is called in a particular sequence. The router’s behavior is modified differently depending on which guard is used. The guards are:

* CanActivate
* CanActivateChild
* CanDeactivate
* CanLoad
* Resolve
* *Q6*: **What is a *Service*, and when will you use it?**
* **Add to PDF** **Junior** C:\Users\758530\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\4067AE14.tmp
* Answer
* Angular **services** are singleton objects which get instantiated only once during the lifetime of an application. They contain methods that maintain data throughout the life of an application, i.e. data does not get refreshed and is available all the time. The **main objective** of a service is to **organize and share business logic**, models, or data and functions with different components of an Angular application.
* The *separation of concerns* is the main reason why Angular services came into existence. An Angular service is a stateless object and provides some very useful functions.

*Q7*: What is the difference between @Component and @Directive in Angular?

**Add to PDF** **Junior** C:\Users\758530\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\A77A8142.tmp

Answer

* **Directives** add behaviour to an existing DOM element or an existing component instance.
* **A component**, rather than adding/modifying behaviour, actually creates its own view (hierarchy of DOM elements) with attached behaviour.

Write a component when you want to create a reusable set of DOM elements of UI with custom behaviour. Write a directive when you want to write reusable behaviour to supplement existing DOM elements.

*\*\*\*Q26*: **Explain the difference between Promise and Observable in Angular?**

**Promises** deal with one asynchronous event at a time, while **observables** handle a sequence of asynchronous events over a period of time.

#### **Question**: **How do Observables differ from Promises?**

**Answer**: As soon as a [promise](http://andyshora.com/promises-angularjs-explained-as-cartoon.html) is made, the execution takes place. However, this is not the case with observables because they are lazy. This means that nothing happens until a subscription is made. While promises handle a single event, observable is a stream that allows passing of more than one event. A callback is made for each event in an observable.

*Q14*: **What are Observables?**

**Add to PDF** **Junior** C:\Users\758530\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\97D291AE.tmp

Answer

**Observables** are declarative which provide support for passing messages between publishers and subscribers in your application.

They are mainly used for event handling, asynchronous programming, and handling multiple values. In this case, you define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they *unsubscribe*.

*Q16*: **What is an Observer?**

Answer

**Observer** is an interface for a consumer of push-based notifications delivered by an Observable. It has below structure,

interface Observer<T> {

closed?: boolean;

next: (value: T) => void;

error: (err: any) => void;

complete: () => void;

}

A handler that implements the Observer interface for receiving observable notifications will be passed as a parameter for observable as below,

myObservable.subscribe(myObserver);

**Note:** If you don't supply a handler for a notification type, the observer ignores notifications of that type.

*Q19*: **What is the purpose of base href tag?**

Answer:

The routing application should add element to the index.html as the first child in the tag inorder to indicate how to compose navigation URLs. If app folder is the application root then you can set the href value as below

<base href="/">

*Q22*:  **What is *Angular Universal*?**

Angular Universal is the process of **server-side rendering** (SSR) your application to HTML on the **Server** (ie: Node.js).

*Q24*: **Explain the difference between Constructor and ngOnInit?**

* Most of the time we use ngOnInit() for all the initialization/declaration.
* It’s better to avoid writing actual work in the constructor.
* The constructor() should only be used to initialize class members but shouldn't do actual "work".
* So we should use constructor() to setup Dependency Injection, Initialization of class fields, etc.
* ngOnInit() is a better place to write "actual work code" that we need to execute as soon as the class is instantiated.

*\*\*\*Q27*: **Why should ngOnInit be used, if we already have a constructor?**

* Mostly **we** use **ngOnInit for** all the initialization/declaration and avoid stuff to work in the **constructor**. The **constructor should** only be **used** to initialize class members but shouldn't do actual "work". So **you should** use **constructor**() to setup Dependency Injection and not much else.

*\*\*\*Q30*: **What is *Router Outlet*?**

# **RouterOutlet**

Acts as a placeholder that Angular dynamically fills based on the current router state.

*\*\*\*Q45*: **What is the purpose of *Wildcard route*?\*\***

A **Wildcard route** has a **path** consisting of two asterisks (\*\*). It matches every URL, the router will select this **route** if it can't match a **route** earlier in the configuration. A **Wildcard Route** can navigate to a custom component or can redirect to an existing **route**.

*\*\*\*Q46*: **What is *Router State*?**

Represents the state of the router as a tree of activated routes.

**10. How are observables different from promises?**

The first difference is that an Observable is **lazy** whereas a Promise is **eager**.

|  |  |
| --- | --- |
| Promise | Observable |
| Emits a single value | Emits multiple values over a period of time |
| Not Lazy | Lazy. An observable is not called until we subscribe to the observable |
| Cannot be cancelled | Can be cancelled by using the unsubscribe() method |
|  | Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc. |

Consider the following Observable:

const observable = rxjs.Observable.create(observer => {

console.log('Text inside an observable');

observer.next('Hello world!');

observer.complete();

});

console.log('Before subscribing an Observable');

observable.subscribe((message)=> console.log(message));

When you run the above Observable, you can see messages being displayed in the following order:

Before subscribing an Observable  
Text inside an observable  
Hello world!

As you can see, observables are lazy. Observable runs only when someone subscribes to them hence, the message “Before subscribing…” is displayed ahead of the message inside the observable.  
  
Now let’s consider a Promise:

const promise = new Promise((resolve, reject) => {

console.log('Text inside promise');

resolve('Hello world!');

});

console.log('Before calling then method on Promise');

greetingPoster.then(message => console.log(message));

Running the above promise, the messages will be displayed in the following order:

Text inside promise  
Before calling then method on Promise  
Hello world!

As you can see the message inside Promise is displayed first. This means that a promise runs before the **then** method is called. Therefore, promises are **eager**.  
  
The next difference is that Promises are always **asynchronous**. Even when the promise is immediately resolved. Whereas an Observable, can be both **synchronous** and **asynchronous**.  
  
The above example of an observable is the case to show that an observable is synchronous. Let’s see the case where an observable can be asynchronous:

const observable = rxjs.Observable.create(observer => {

setTimeout(()=>{

observer.next('Hello world');

observer.complete();

},3000)

});

console.log('Before calling subscribe on an Observable');

observable.subscribe((data)=> console.log(data));

console.log('After calling subscribe on an Observable');

The messages will be displayed in the following order:

Before calling subscribe on an Observable  
After calling subscribe on an Observable  
Hello world!

You can see in this case, observable runs asynchronously.  
  
The next difference is that Observables can emit **multiple** values whereas Promises can emit only one value.  
  
The biggest feature of using observables is the use of **operators**. We can use multiple operators on an observable whereas, there is no such feature in a promise.

#### **Question**: **Why prioritize TypeScript over JavaScript in Angular?**

**Answer**: TypeScript is a superset of Javascript as it is Javascript + Types or extra features like typecasting for variables, annotations, variable scope and much more. The typescript is designed in a way to overcome Javascript shortcomings like typecasting of variables, classes, decorators, variable scope and many more. Moreover, Typescript is purely object-oriented programming that offers a "Compiler" that can convert to Javascript-equivalent code.

#### **Question: Discuss the lifecycle designed for directive and components in Angular JS especially for the newly introduced version 6.0?**

**Answer:**

Components and directive of Angular JS follow the following typical lifecycle.

* nhOnInit
* ngDoCheck
* ngOnDestroy
* Constructor
* ngOnChanges
* ngAfterContentInit (only for components)
* ngAfterContentChecked (only for components)
* ngAfterViewInit (only for components)
* ngAfterViewChecked (only for components)

#### **Question: Write the features for Angular 6 over Angular 5.**

**Answer:**Following are the features:

**1. Added ng update**

CLI command updates angular project dependencies to their latest versions. The ng update is a normal package manager tool to identify and update in normal package manager tools to identify and update other dependencies.

**2. Uses RxJS6**

This is the third party library (RxJS) and introduces two important changes as compared to RxJS5.

1. Introduces a new internal package structure.
2. Operator concept

To update of RxJS6, run the following command:

npm **install** --save rxjs@6

To update your existing Angular Project, run the following:

npm **install** --save rxjs-compat

**3. The <ng-template>**

Angular 6 uses <ng-template> instead of <template>

**4. Service Level Changes**

If in an earlier version, the user wanted to provide a service to the entire application, the user was required to add it to providers in the AppModule but it is not required in the case of Angular6.

**5. Renamed Operators**

Angular 6 has renamed following operators:

* do() => tap()
* catch() => catchError()
* finally() => finalize()
* switch()=>switchAll()
* throw() => throwError

#### Q2. How absolute, relative, a static and fixed position will differ?

**Answer:**

This is the basic UI Developer Interview Question asked in an interview. Please find below the different tables that UI Developer supports are:  
**Absolute:** It will place the element exactly where a user wants to place it. In general absolute will place relative to the parent. If no parent is available, then it is placed relative to the page itself.

**Relative**: It will place the element relative to itself (if we didn’t give any relative positioning); for example, if we set position relative to an element and given as top: 10px, then it will place the element 10px down from where the actual position of the element to be.

**Static**: It will place the element according to the flow of the document. It uses the default position; if we want to remove any position, then we can use a static position to replace it.

**Fixed**: It will place the element relative to the browser window or viewport as viewport doesn’t change when scrolling. So element will be fixed at that position.

#### Q3. What is a responsive website?

**Answer:**  
Any website is meant to be responsive when it looks good and fits in all types of screen resolution device types. In order to re-arrange the elements, display or hide the elements from the user interface, we can use CSS media queries to get things done.

#### Q4. What is the difference between inline, block and inline-block?

**Answer:**  
**Inline:** In this thing, elements will follow the flow without breaking. Margin/padding will push other elements horizontally, not vertically, and inline elements ignore height and width.

**Block:** It breaks the line and doesn’t fit in the line. It usually has div, p, text, section etc.

**Inline-block:** It is similar to the inline element and follows the page flow. The only difference is that it will consider height and width.

Q: Explain the differences between local storage, session storage, and cookies.

A:  These can be explained as follows:

* Local storage allows data storage with no expiration date and offers the most substantial maximum storage limit.
* Session storage stores the data associated with a session. When the user closes the tab or browser, the data disappears.
* Cookies are reserved mostly for server-side reading, storing data sent back to the server. The data size must be less than 4KB.