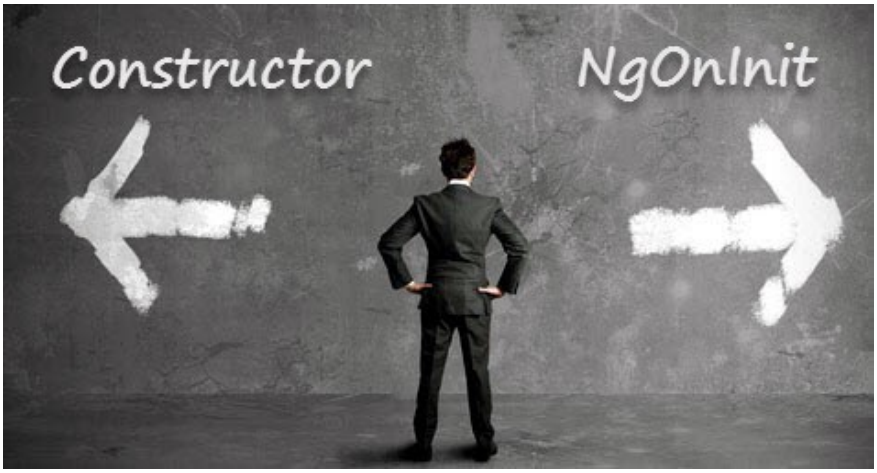
The essential difference between Constructor and ngOnInit in Angular

Max Koretskyi aka Wizard Follow
Sep 7, 2017 · 6 min read



> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

One of the most popular Angular questions on stackoverflow is Difference between Constructor and ngOnInit with over 100k views. I gave my answer to this question there but also decided to expand on it in this article. While most answers in the thread and articles on the web focus on the difference between the usage of the two here I'd like to give a more comprehensive comparison that taps into components initialization process.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular**

> **grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

. . .

Difference related to JS/TS language

Let's start with a most obvious difference that is related to the language itself. `ngOnInit` is just a method on a class which structurally is not different to any other method on a class. It's just that Angular team decided to name it that way but it could have been any other name:

```
class MyComponent {
  ngOnInit() { }
  otherNameForNgOnInit() { }
}
```

And it's completely up to you if you want to implement that method or not on a component class. During compilation Angular compiler checks whether a component has this method implemented and marks the class with an appropriate flag:

```
export const enum NodeFlags {
  ...
  OnInit = 1 << 16,
```

This flag is then used to decide whether to call the method on a component class instance or not during change detection:

```
if (def.flags & NodeFlags.OnInit && ...) {
  componentClassInstance.ngOnInit();
}
```

A constructor in turn is a different thing. Regardless whether you implement it or not in TypeScript class it's still will be called when

creating an instance of a class. This is because a typescript class constructor is transpiled into a JavaScript constructor function:

```
class MyComponent {
  constructor() {
    console.log('Hello');
  }
}
```

transpiled into

```
function MyComponent() {
  console.log('Hello');
}
```

To create a class instance this function is called with the `new` operator:

```
const componentInstance = new MyComponent()
```

So if you omit the constructor in a class, it's transpiled into an empty function:

```
class MyComponent { }
```

transpiles into an empty function

```
function MyComponent() {}
```

That is why I'm saying that a constructor is called regardless whether you implement it or not on a class.

• • •

Difference related to the component initialization process

There's a huge difference between the two from the perspective of the component initialization phase. Angular bootstrap process consists of the two major stages:

- constructing components tree
- running change detection

And the constructor of the component is called when Angular constructs components tree. All lifecycle hooks including `ngOnInit` are called as part of the following change detection phase. Usually a component initialization logic requires either some DI providers or available input bindings or rendered DOM. And these are available at different stages of Angular bootstrap process.

When Angular constructs a components tree the root module injector is already configured so you can inject any global dependencies. Also, when Angular instantiates a child component class the injector for the parent component is also already set up so you can inject providers defined on the parent component including the parent component itself. A component constructor is the only method that is called in the context of the injector so if you need any dependency that's the only place to get those dependencies. The `@Input` communication mechanism is processed as part of following change detection phase so input bindings are not available in constructor.

When Angular starts change detection the components tree is constructed and the constructors for all components in the tree have been called. Also at this point every component's template nodes are added to the DOM. Here you have available all the data you may need to initialize a component—DI providers, DOM and input bindings.

You can read more about change detection in Everything you need to know about change detection in Angular and how Angular processes inputs in The mechanics of property bindings update in Angular.

Let's demonstrate these phases with a quick example. Suppose you have the following template:

```
<my-app>
   <child-comp [i]='prop'>
```

So Angular starts bootstrapping the application. As described above it first creates classes for each component. So it calls `MyAppComponent` constructor. When executing a component constructor Angular resolves all dependencies that are injected into `MyAppComponent` constructor and provides them as parameters. It also creates a DOM node which is the host element of the `my-app` component. Then it proceeds to creating a host element for the `child-comp` and calling `ChildComponent` constructor. At this stage Angular is not concerned with the `i` input binding and any lifecycle hooks. So when this process is finished Angular ends up with the following tree of component views:

```
MyAppView
  - MyApp component instance
  - my-app host element data
      ChildComponentView
        - ChildComponent component instance
        - child-comp host element data
```

Only then Angular runs change detection and updates bindings for the `my-app` and calls `ngOnInit` on the `MyAppComponent` instance. Then it proceeds to updating the bindings for the `child-comp` and calls `ngOnInit` on the `ChildComponent` class.

You can learn more about a view that I'm referring to above in the Here is why you will not find components inside Angular.

• • •

Difference related to the usage

And now let's see the difference from the usage perspective.

Constructor

A class constructor in Angular is mostly used to inject dependencies. Angular calls this *constructor injection pattern* which is explained in details here. For a more in-depth architectural insights you can read Constructor Injection vs. Setter Injection by Miško Hevery.

However, the usage of a constructor is not limited to DI. For example, `router-outlet` directive of `@angular/router` module uses it to register itself and its location (viewContainerRef) within the router ecosystem. I described that approach in Here is how to get ViewContainerRef before @ViewChild query is evaluated.

Yet the common practice is put as little logic into constructor as possible.

NgOnInit

As we learnt above when Angular calls `ngOnInit` it has finished creating a component DOM, injected all required dependencies through constructor and processed input bindings. So here you have all the required information available which makes it a good place to perform initialization logic.

It's a common practice to use `ngOnInit` to perform initialization logic even if this logic doesn't depend on DI, DOM or input bindings.

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.