Discovering Lifecycle Hooks

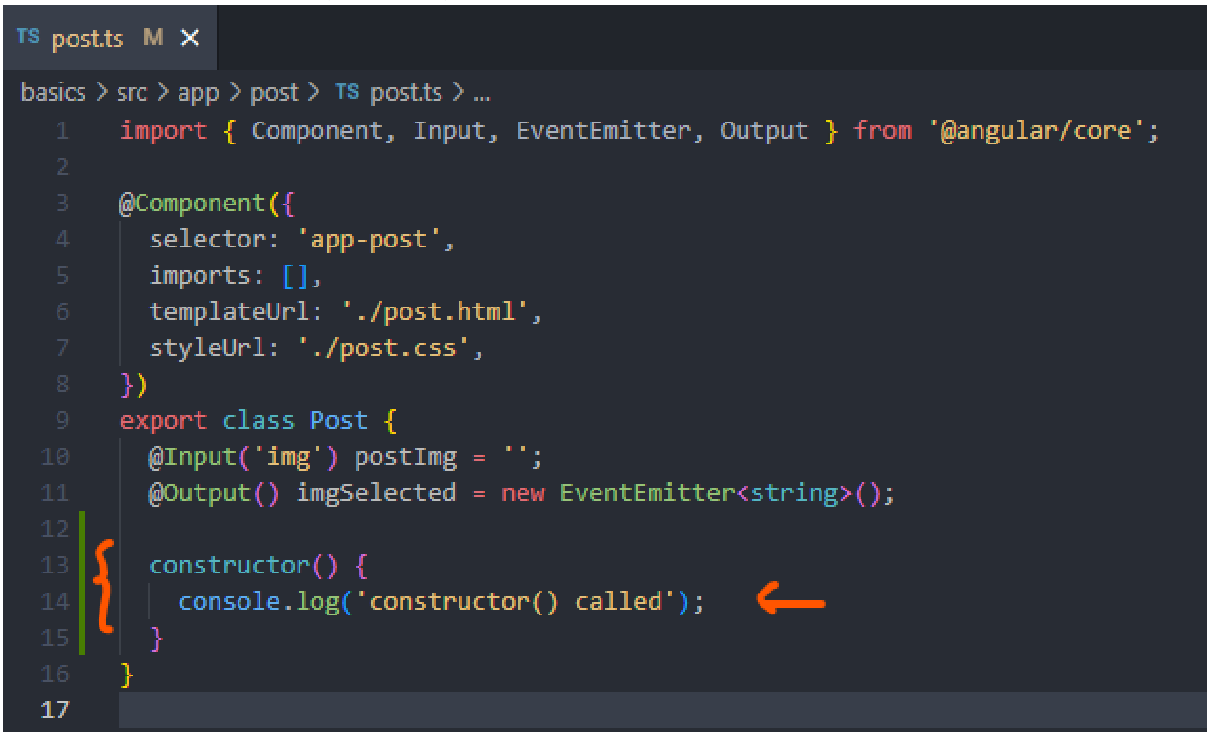
In this lecture, we're going to learn about lifecycle hooks.

Throughout this section, we've seen how we can interact with components. Angular gives us the option of having refined control over the changes in our components through lifecycle hooks. They are functions that run during events in our components. Specifically, they run when components are experiencing changes.

There are a lot of lifecycle hooks in Angular. Some are more common than others.

Let's start with the simpler hooks.

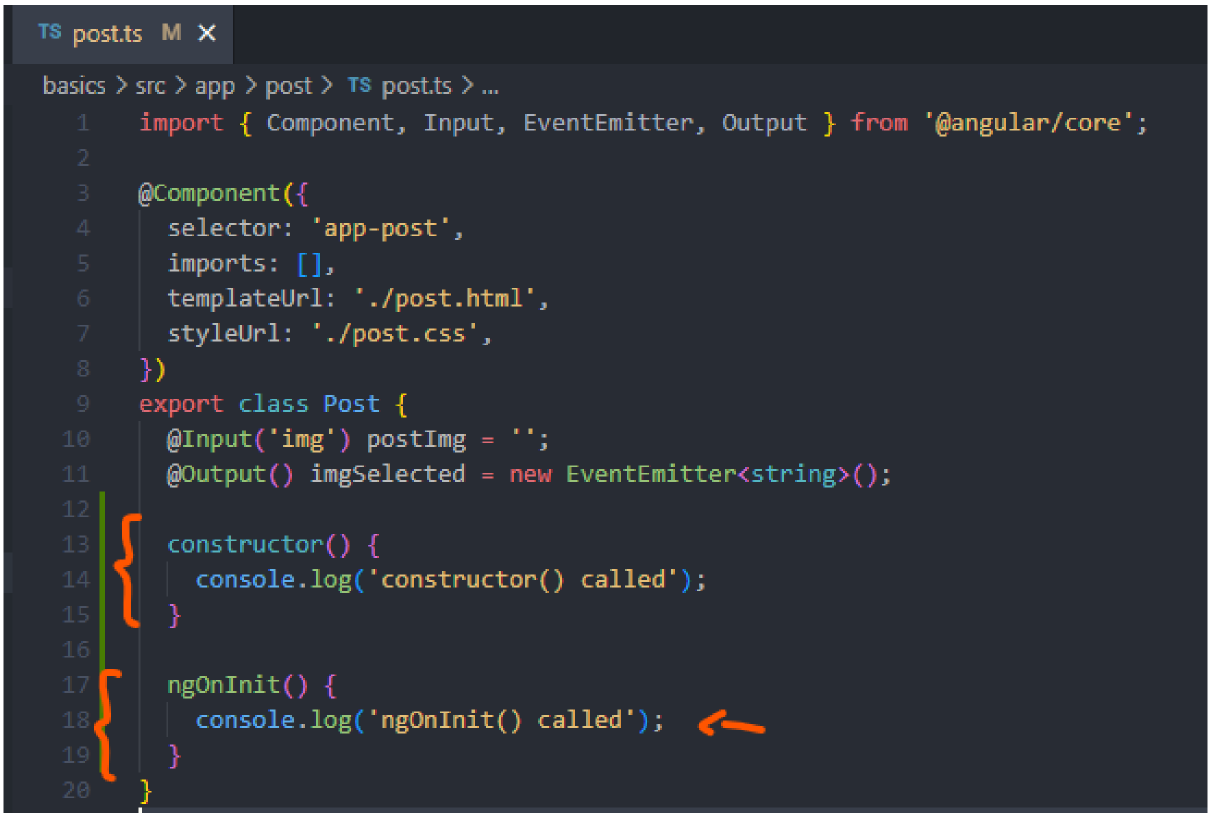
1. The first hook is the **constructor function**. We will be working inside the **post component class**. Inside this class add **constructor function**. Technically, this is not a hook run by Angular. Instead, it's a default hook in the JavaScript language. **It gets called during the initialization of our component. It is the first function that will run whenever we use our component.** Inside this function, let's log a message:



post.ts M ×
basics > src > app > post > post.ts > ...

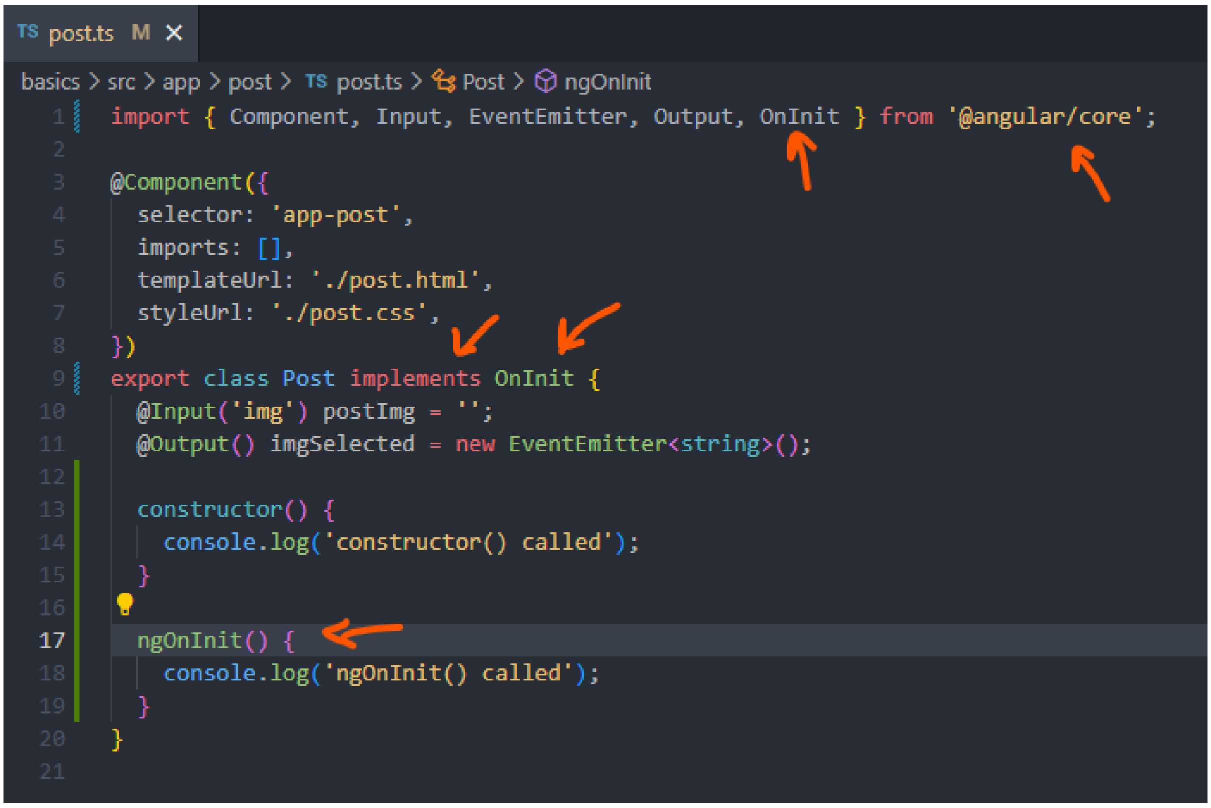
```
import { Component, Input, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'app-post',
  imports: [],
  templateUrl: './post.html',
  styleUrl: './post.css',
})
export class Post {
  @Input('img') postImg = '';
  @Output() imgSelected = new EventEmitter<string>();

  constructor() {
    console.log('constructor() called');
  }
}
```

2. There's another hook for when our component is initialized. It's called the **ngOnInit function inside our class.** We will define this function. Like before, we will log a message:



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

@Component({
  selector: 'app-post',
  imports: [],
  templateUrl: './post.html',
  styleUrl: './post.css',
})
export class Post {
  @Input('img') postImg = '';
  @Output() imgSelected = new EventEmitter<string>();

  constructor() {
    console.log('constructor() called');
  }

  ngOnInit() {
    console.log('ngOnInit() called');
  }
}
```

We can write the **ngOnInit() function** in our component without a problem. **However, it's not considered good practice. There is a better way to write this function. We can implement an interface.** We learned about interfaces in the TypeScript section of this course. They are blueprint for objects to help the compiler understand the properties and methods inside an object. Angular defines a set of interfaces for life cycle functions in a component. There are two benefits to adding them to our component. Let's look at what those are. At the top, we can update the **@angular/core package** to include an interface called **OnInit**. Previously, we learned how to apply interfaces on objects with curly brackets syntax. We can apply interfaces to classes by adding the **implements keyword after the class name followed by the name of the interface.** Let's implement the **onInit interface to the class:**



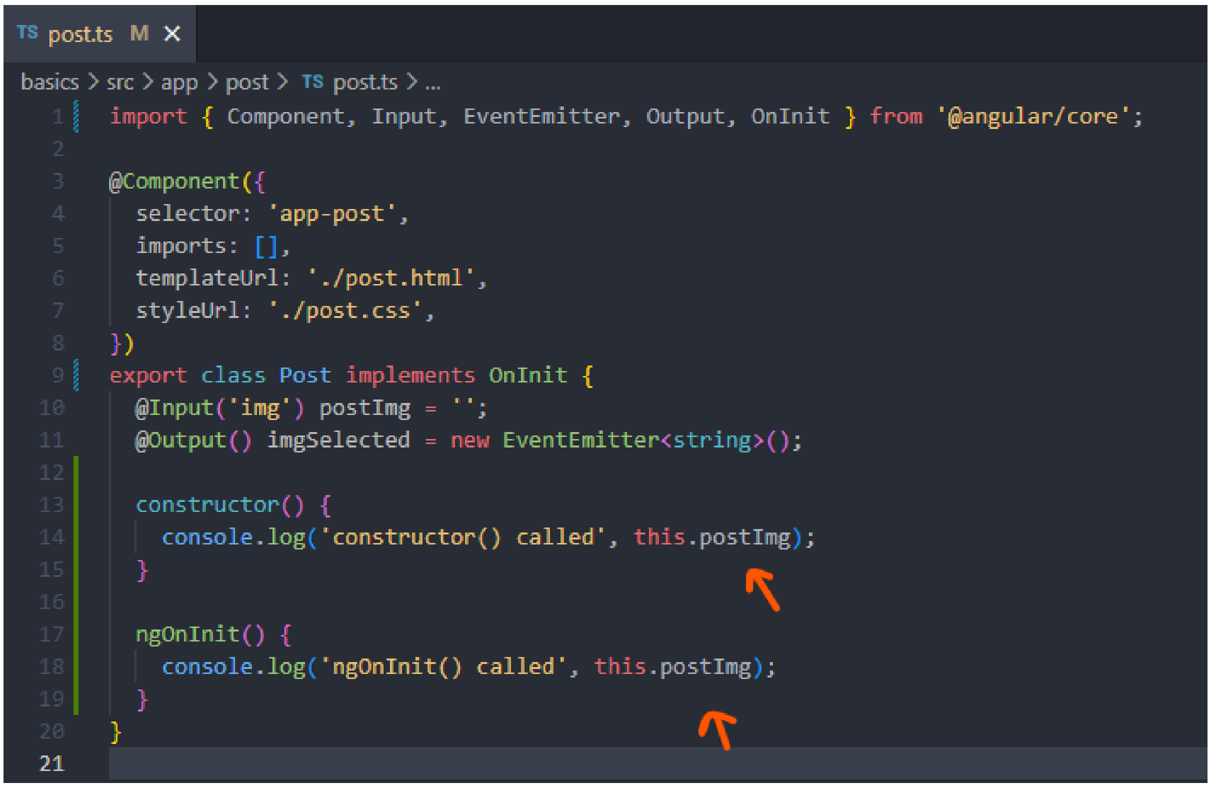
post.ts M

basics > src > app > post > post.ts > Post > ngOnInit

```typescript
import { Component, Input, EventEmitter, Output, OnInit } from '@angular/core';

@Component({
  selector: 'app-post',
  imports: [],
  templateUrl: './post.html',
  styleUrl: './post.css',
})
export class Post implements OnInit {
  @Input('img') postImg = '';
  @Output() imgSelected = new EventEmitter<string>();

  constructor() {
    console.log('constructor() called');
  }

  ngOnInit() {
    console.log('ngOnInit() called');
  }
}

```

It's always considered good practice to implement an interface for lifecycle hooks. The **ngOnInit()** function will run when the component is initialized with the data. It's completely different from the **constructor function**. The data will have changed from both functions.

To better understand, let's update our log statements to include the post image property, what do you thing will happen? Do you think we will see the same values?

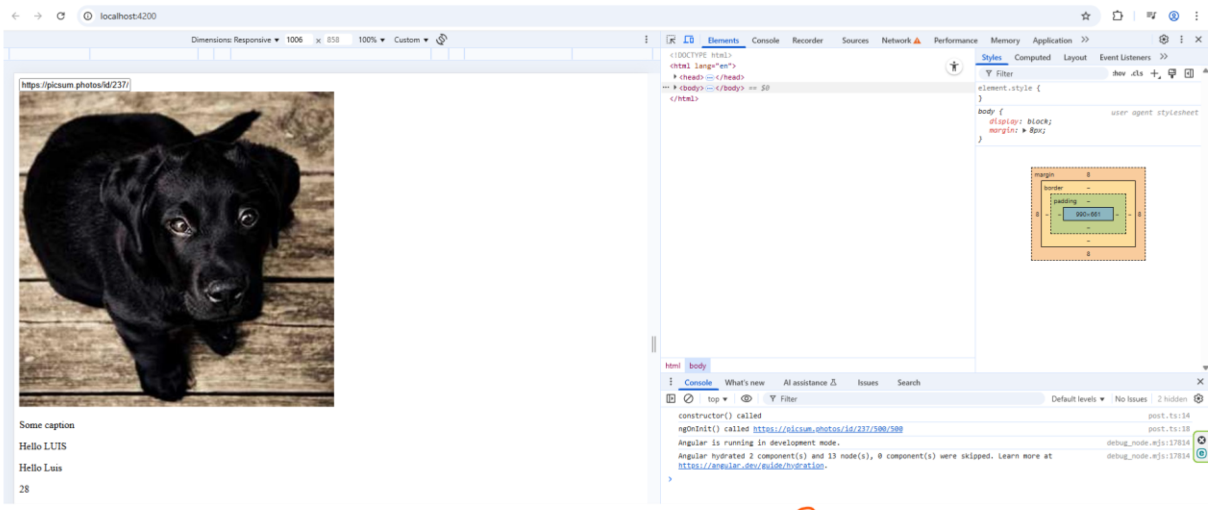


post.ts M

basics > src > app > post > post.ts > ...

```typescript
import { Component, Input, EventEmitter, Output, OnInit } from '@angular/core';

@Component({
  selector: 'app-post',
  imports: [],
  templateUrl: './post.html',
  styleUrl: './post.css',
})
export class Post implements OnInit {
  @Input('img') postImg = '';
  @Output() imgSelected = new EventEmitter<string>();

  constructor() {
    console.log('constructor() called', this.postImg);
  }

  ngOnInit() {
    console.log('ngOnInit() called', this.postImg);
  }
}

```

Let’s find out, refresh the browser with the console opened:



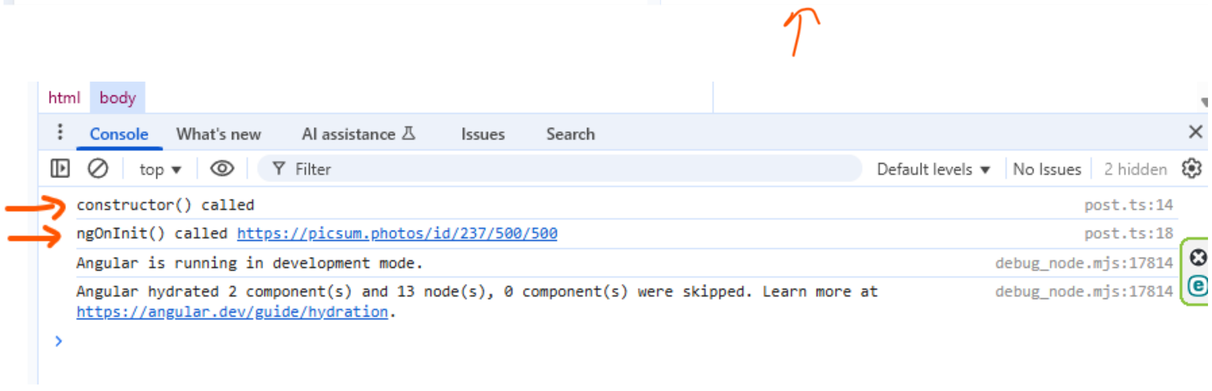
https://picsum.photos/id/237/

Some caption

Hello LUIS

Hello Luis

28



```
constructor() called                                                   post.ts:14
ngOnInit() called https://picsum.photos/id/237/500/500                 post.ts:18
Angular is running in development mode.                                debug_node.mjs:17814
Angular hydrated 2 component(s) and 13 node(s), 0 component(s) were skipped. Learn more at   debug_node.mjs:17814
https://angular.dev/guide/hydration.
```

The post image property will be empty in the **constructor function.** However, the property will contain the URL to the image in the **ngOnInit function**.

During the constructor function, the data passed down from the parent to the child component hasn’t occurred yet. Therefore, we won’t have access to input data.

The ngOnInit function runs after binding has occurred. If we need the data from the parent to be available when the component is initialized, we should use the ngOnInit function. Just like the constructor function, this hook runs once. It’s a great hook for making the changes to our component after the data has been sent down.

Documentation: https://v17.angular.io/guide/lifecycle-hooks

Let’s explore the other hooks in the next lecture.