PIEPS

In Angular, pipes are a feature that allows you to transform data within your templates. They are a way to format, filter, and manipulate data before it is displayed to the user. Pipes are used in the HTML template and provide a convenient way to perform common operations on strings, numbers, dates, and other types of data.

Here are some key aspects of pipes in Angular:

1. **Built-in Pipes:**

Angular comes with a set of built-in pipes that cover a wide range of common scenarios. Some examples include:

- **DatePipe:** Formats dates.
- **UpperCasePipe and LowerCasePipe:** Converts text to uppercase or lowercase.
- **CurrencyPipe:** Formats currency values.
- **DecimalPipe and PercentPipe:** Formats numbers and percentages.

2. **Chaining Pipes:**

You can chain multiple pipes together in the template to perform a sequence of transformations. For example:

```
{{ myDate | date | uppercase }}
```

3. **Custom Pipes:**

You can create your own custom pipes to handle specific formatting or transformation requirements. Custom pipes are classes decorated with `@Pipe` and implementing the `PipeTransform` interface.

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

@Pipe({
  name: 'myCustomPipe'
})
export class MyCustomPipe implements PipeTransform {
  transform(value: any, args?: any): any {
    // Implement custom transformation logic
    return transformedValue;
  }
}
```

Custom pipes can then be used in the template just like built-in pipes.

4. **Async Pipe:**

The `async` pipe is a special pipe that unwraps a promise or an observable and returns the resolved value. It's commonly used with asynchronous operations.

```
{{ myPromise | async }}
{{ myObservable | async }}
```

5. **Parametrized Pipes:**

Many built-in pipes can take parameters to customize their behavior. For example, the `date` pipe can take a format string as an argument.

```
{{ myDate | date: 'fullDate' }}
```

6. **Pipes in ngFor:**

Pipes are commonly used in conjunction with `ngFor` to format and display lists of data.

```
<ul>
  <li *ngFor="let item of items">{{ item | myCustomPipe }}</li>
</ul>
```

7. **Pure and Impure Pipes:**

- **Pure Pipes:** By default, pipes are pure, meaning they are stateless and their output only depends on the input parameters. Angular caches the result of a pure pipe and only recalculates it when the input changes.
- **Impure Pipes:** You can make a custom pipe impure by setting the `pure` property to `false` in the `@Pipe` decorator. Impure pipes are recalculated on every change detection cycle.

```
@Pipe({
  name: 'myImpurePipe',
  pure: false
})
```

Example:

Here's an example of using built-in pipes in Angular:

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

@Component({
  selector: 'app-example',
  template: `
    <h2>{{ currentDate | date: 'fullDate' }}</h2>
    <p>{{ amount | currency: 'USD' }}</p>
    <p>{{ percentage | percent: '2.2-2' }}</p>
  `,
})
export class ExampleComponent {
  currentDate = new Date();
  amount = 1234.5678;
  percentage = 0.75;
}
```

In this example, the `date`, `currency`, and `percent` pipes are used to format the `currentDate`, `amount`, and `percentage` values, respectively.

Pipes are a powerful tool in Angular for transforming and presenting data in a user-friendly way within your application's templates. They contribute to the overall declarative and maintainable nature of Angular applications.

Creating a custom pipe

Creating a custom pipe in Angular involves defining a TypeScript class, implementing the `PipeTransform` interface, and using the `@Pipe` decorator. I'll walk you through creating a simple custom pipe that capitalizes the first letter of each word in a string. Let's call it `TitleCasePipe`.

Step 1: Create the Pipe Class

```
// title-case.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'titleCase'
})
export class TitleCasePipe implements PipeTransform {
  transform(value: string): string {
    if (!value) return value; // If the input is null or undefined, return it
as is.

    return value
```

```
      .toLowerCase()
      .split(' ')
      .map(word => word.charAt(0).toUpperCase() + word.slice(1))
      .join(' ');
  }
}
```

Explanation:

- The `TitleCasePipe` class implements the `PipeTransform` interface.
- The `@Pipe` decorator provides metadata for the pipe, including the name by which it will be used in templates (`titleCase`).

Step 2: Declare the Pipe in a Module

To use the custom pipe in your Angular application, you need to declare it in a module.

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { TitleCasePipe } from './title-case.pipe';

@NgModule({
  declarations: [AppComponent, TitleCasePipe],
  imports: [BrowserModule],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Step 3: Use the Custom Pipe in a Component

Now, you can use the `titleCase` pipe in your component's template.

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <h2>Original Text: {{ originalText }}</h2>
    <h2>Transformed Text: {{ originalText | titleCase }}</h2>
  `,
})
export class AppComponent {
  originalText = 'hello world';
}
```

Scenario:

In this example, the `TitleCasePipe` takes a string input, converts it to lowercase, and then capitalizes the first letter of each word. The custom pipe is then used in the `AppComponent` template to demonstrate the transformation.

Run the Application:

When you run your Angular application, you should see the original text and the transformed text displayed in the browser, with the first letter of each word capitalized.

```
Original Text: hello world
Transformed Text: Hello World
```

This scenario demonstrates the creation and usage of a custom pipe for a specific transformation task. Custom pipes are useful when you have specific formatting or transformation needs that aren't covered by built-in pipes. They contribute to code modularity and can be easily reused across different components in your Angular application.