Pipes

Every application starts out with what seems like a simple task: get data, transform them, and show them to users. Getting data could be as simple as creating a local variable or as complex as streaming data over a WebSocket.

Once data arrives, you could push their raw `toString` values directly to the view, but that rarely makes for a good user experience. For example, in most use cases, users prefer to see a date in a simple format like `April 15, 1988` rather than the raw string format `Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time)`.

Clearly, some values benefit from a bit of editing. You may notice that you desire many of the same transformations repeatedly, both within and across many applications. You can almost think of them as styles. In fact, you might like to apply them in your HTML templates as you do styles.

Introducing Angular pipes, a way to write display-value transformations that you can declare in your HTML.

You can run the live example / download example in Stackblitz and download the code from there.

Using pipes

A pipe takes in data as input and transforms it to a desired output. In this page, you'll use pipes to transform a component's birthday property into a human-friendly date.

src/app/hero-birthday1.component.ts

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

Focus on the component's template.

src/app/app.component.html

```
<p>The hero's birthday is {{ birthday | date }}</p>
```

Inside the interpolation expression, you flow the component's `birthday` value through the pipe operator (|) to the Date pipe function on the right. All pipes work this way.

Built-in pipes

Angular comes with a stock of pipes such as `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`. They are all available for use in any template.

> Read more about these and many other built-in pipes in the pipes topics of the API Reference; filter for entries that include the word "pipe".
>
> Angular doesn't have a `FilterPipe` or an `OrderByPipe` for reasons explained in the Appendix of this page.

Parameterizing a pipe

A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon (:) and then the parameter value (such as `currency:'EUR'`). If the pipe accepts multiple parameters, separate the values with colons (such as `slice:1:5`)

Modify the birthday template to give the date pipe a format parameter. After formatting the hero's April 15th birthday, it renders as `04/15/88`:

src/app/app.component.html

```
<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>
```

The parameter value can be any valid template expression, (see the Template expressions section of the Template Syntax page) such as a string literal or a component property. In other words, you can control the format through a binding the same way you control the birthday value through a binding.

Write a second component that *binds* the pipe's format parameter to the component's `format` property. Here's the template for that component:

src/app/hero-birthday2.component.ts (template)

```
template: `
  <p>The hero's birthday is {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Toggle Format</button>
`
```

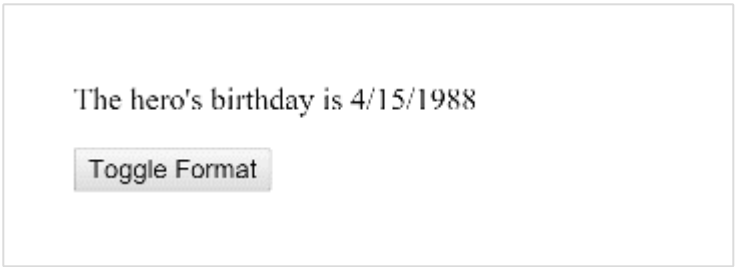
You also added a button to the template and bound its click event to the component's `toggleFormat()` method. That method toggles the component's `format` property between a short form (`'shortDate'`) and a longer form (`'fullDate'`).

src/app/hero-birthday2.component.ts (class)

```
export class HeroBirthday2Component {
  birthday = new Date(1988, 3, 15); // April 15, 1988
  toggle = true; // start with true == shortDate

  get format()   { return this.toggle ? 'shortDate' : 'fullDate'; }
  toggleFormat() { this.toggle = !this.toggle; }
}
```

As you click the button, the displayed date alternates between "`04/15/1988`" and "`Friday, April 15, 1988`".



Read more about the `DatePipe` format options in the Date Pipe API Reference page.

Chaining pipes

You can chain pipes together in potentially useful combinations. In the following example, to display the birthday in uppercase, the birthday is chained to the `DatePipe` and on to the `UpperCasePipe`. The birthday displays as `APR 15, 1988`.

src/app/app.component.html

```
The chained hero's birthday is
{{ birthday | date | uppercase}}
```

This example—which displays `FRIDAY, APRIL 15, 1988`—chains the same pipes as above, but passes in a parameter to `date` as well.

src/app/app.component.html

```
The chained hero's birthday is
{{  birthday | date:'fullDate' | uppercase}}
```

Custom pipes

You can write your own custom pipes. Here's a custom pipe named `ExponentialStrengthPipe` that can boost a hero's powers:

src/app/exponential-strength.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent?: number): number {
    return Math.pow(value, isNaN(exponent) ? 1 : exponent);
  }
}
```

This pipe definition reveals the following key points:

- A pipe is a class decorated with pipe metadata.
- The pipe class implements the `PipeTransform` interface's `transform` method that accepts an input value followed by optional parameters and returns the transformed value.
- There will be one additional argument to the `transform` method for each parameter passed to the pipe. Your pipe has one such parameter: the `exponent`.
- To tell Angular that this is a pipe, you apply the `@Pipe` decorator, which you import from the core Angular library.
- The `@Pipe` decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier. Your pipe's name is `exponentialStrength`.

The *PipeTransform* interface

The `transform` method is essential to a pipe. The `PipeTransform` *interface* defines that method and guides both tooling and the compiler. Technically, it's optional; Angular looks for and executes the `transform` method regardless.

Now you need a component to demonstrate the pipe.

src/app/power-booster.component.ts

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

@Component({
  selector: 'app-power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{2 | exponentialStrength: 10}}</p>
  `
})
export class PowerBoosterComponent { }
```

Note the following:

- You use your custom pipe the same way you use built-in pipes.
- You must include your pipe in the `declarations` array of the `AppModule`
- If you choose to inject your pipe into a class, you must provide it in the `providers` array of your `NgModule`.

REMEMBER THE DECLARATIONS ARRAY

You must register custom pipes. If you don't, Angular reports an error. The Angular CLI's generator registers the pipe automatically.

To probe the behavior in the live example / download example, change the value and optional exponent in the template.

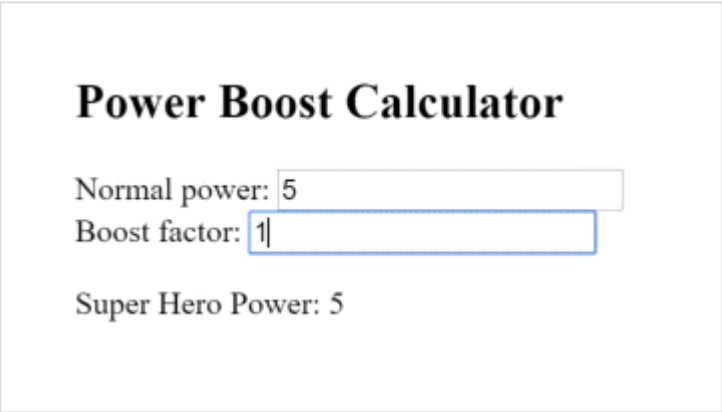
Power Boost Calculator

It's not much fun updating the template to test the custom pipe. Upgrade the example to a "Power Boost Calculator" that combines your pipe and two-way data binding with `ngModel`.

src/app/power-boost-calculator.component.ts

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

@Component({
  selector: 'app-power-boost-calculator',
  template: `
    <h2>Power Boost Calculator</h2>
    <div>Normal power: <input [(ngModel)]="power"></div>
    <div>Boost factor: <input [(ngModel)]="factor"></div>
    <p>
      Super Hero Power: {{power | exponentialStrength: factor}}
    </p>
  `
})
export class PowerBoostCalculatorComponent {
  power = 5;
  factor = 1;
}
```



Pipes and change detection

Angular looks for changes to data-bound values through a *change detection* process that runs after every DOM event: every keystroke, mouse move, timer tick, and server response. This could be expensive. Angular strives to lower the cost whenever possible and appropriate.

Angular picks a simpler, faster change detection algorithm when you use a pipe.

No pipe

In the next example, the component uses the default, aggressive change detection strategy to monitor and update its display of every hero in the `heroes` array. Here's the template:

src/app/flying-heroes.component.html (v1)

```
New hero:
  <input type="text" #box
         (keyup.enter)="addHero(box.value); box.value=''"
         placeholder="hero name">
  <button (click)="reset()">Reset</button>
  <div *ngFor="let hero of heroes">
    {{hero.name}}
  </div>
```

The companion component class provides heroes, adds heroes into the array, and can reset the array.

src/app/flying-heroes.component.ts (v1)

```
export class FlyingHeroesComponent {
  heroes: any[] = [];
  canFly = true;
  constructor() { this.reset(); }

  addHero(name: string) {
    name = name.trim();
    if (!name) { return; }
    let hero = {name, canFly: this.canFly};
    this.heroes.push(hero);
  }

  reset() { this.heroes = HEROES.slice(); }
}
```

You can add heroes and Angular updates the display when you do. If you click the `reset` button, Angular replaces `heroes` with a new array of the original heroes and updates the display. If you added the ability to remove or change a hero, Angular would detect those changes and update the display as well.

FlyingHeroesPipe

Add a `FlyingHeroesPipe` to the `*ngFor` repeater that filters the list of heroes to just those heroes who can fly.

src/app/flying-heroes.component.html (flyers)

```
<div *ngFor="let hero of (heroes | flyingHeroes)">
  {{hero.name}}
</div>
```

Here's the `FlyingHeroesPipe` implementation, which follows the pattern for custom pipes described earlier.

src/app/flying-heroes.pipe.ts

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

import { Flyer } from './heroes';

@Pipe({ name: 'flyingHeroes' })
export class FlyingHeroesPipe implements PipeTransform {
  transform(allHeroes: Flyer[]) {
    return allHeroes.filter(hero => hero.canFly);
  }
}
```

Notice the odd behavior in the live example / download example: when you add flying heroes, none of them are displayed under "Heroes who fly."

Although you're not getting the behavior you want, Angular isn't broken. It's just using a different change-detection algorithm that ignores changes to the list or any of its items.

Notice how a hero is added:

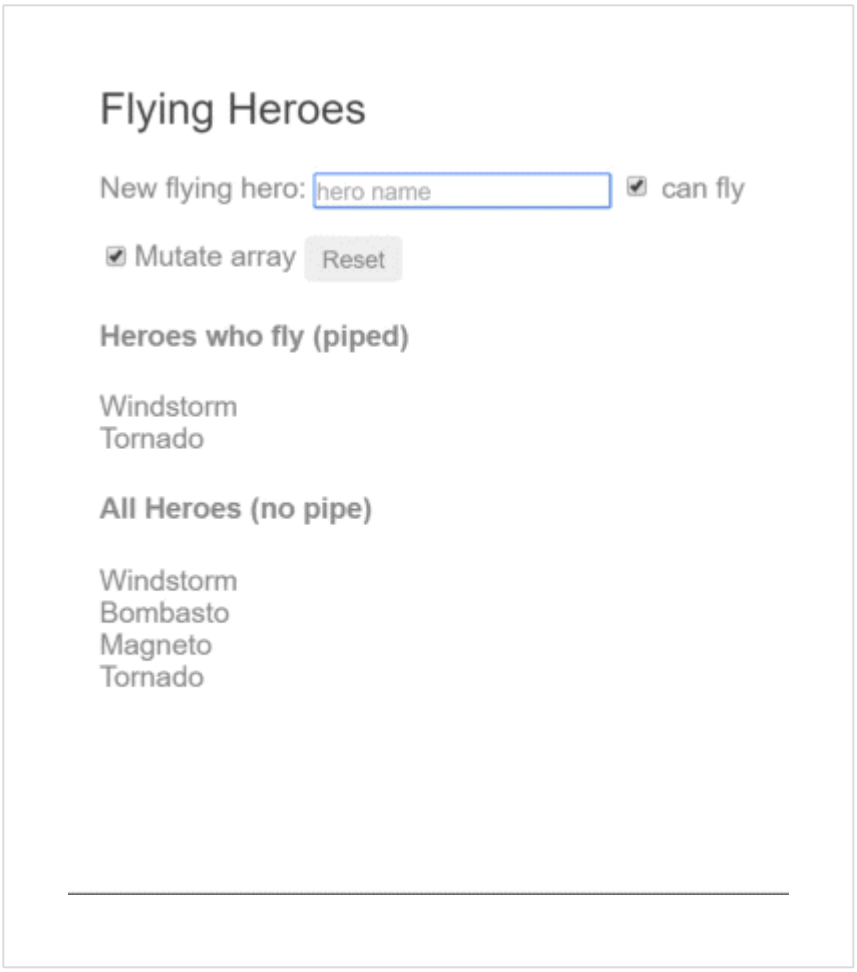
src/app/flying-heroes.component.ts

```
this.heroes.push(hero);
```

You add the hero into the `heroes` array. The reference to the array hasn't changed. It's the same array. That's all Angular cares about. From its perspective, *same array, no change, no display update*.

To fix that, create an array with the new hero appended and assign that to `heroes`. This time Angular detects that the array reference has changed. It executes the pipe and updates the display with the new array, which includes the new flying hero.

If you *mutate* the array, no pipe is invoked and the display isn't updated; if you *replace* the array, the pipe executes and the display is updated. The Flying Heroes application extends the code with checkbox switches and additional displays to help you experience these effects.



Replacing the array is an efficient way to signal Angular to update the display. When do you replace the array? When the data changes. That's an easy rule to follow in *this* example where the only way to change the data is by adding a hero.

More often, you don't know when the data has changed, especially in applications that mutate data in many ways, perhaps in application locations far away. A component in such an application usually can't know about those changes. Moreover, it's unwise to distort the component design to accommodate a pipe. Strive to keep the component class independent of the HTML. The component should be unaware of pipes.

For filtering flying heroes, consider an *impure pipe*.

Pure and impure pipes

There are two categories of pipes: *pure* and *impure*. Pipes are pure by default. Every pipe you've seen so far has been pure. You make a pipe impure by setting its pure flag to false. You could make the `FlyingHeroesPipe` impure like this:

src/app/flying-heroes.pipe.ts

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

Before doing that, understand the difference between pure and impure, starting with a pure pipe.

Pure pipes

Angular executes a *pure pipe* only when it detects a *pure change* to the input value. A pure change is either a change to a primitive input value (`String`, `Number`, `Boolean`, `Symbol`) or a changed object reference (`Date`, `Array`, `Function`, `Object`).

Angular ignores changes within (composite) objects. It won't call a pure pipe if you change an input month, add to an input array, or update an input object property.

This may seem restrictive but it's also fast. An object reference check is fast—much faster than a deep check for differences—so Angular can quickly determine if it can skip both the pipe execution and a view update.

For this reason, a pure pipe is preferable when you can live with the change detection strategy. When you can't, you *can* use the impure pipe.

> Or you might not use a pipe at all. It may be better to pursue the pipe's purpose with a property of the component, a point that's discussed later in this page.

Impure pipes

Angular executes an *impure pipe* during every component change detection cycle. An impure pipe is called often, as often as every keystroke or mouse-move.

With that concern in mind, implement an impure pipe with great care. An expensive, long-running pipe could destroy the user experience.

An impure *FlyingHeroesPipe*

A flip of the switch turns the `FlyingHeroesPipe` into a `FlyingHeroesImpurePipe`. The complete implementation is as follows:

FlyingHeroesImpurePipe *FlyingHeroesPipe*

```
@Pipe({
  name: 'flyingHeroesImpure',
  pure: false
})
export class FlyingHeroesImpurePipe extends FlyingHeroesPipe {}
```

You inherit from `FlyingHeroesPipe` to prove the point that nothing changed internally. The only difference is the `pure` flag in the pipe metadata.

This is a good candidate for an impure pipe because the `transform` function is trivial and fast.

src/app/flying-heroes.pipe.ts (filter)

```
return allHeroes.filter(hero => hero.canFly);
```

You can derive a `FlyingHeroesImpureComponent` from `FlyingHeroesComponent`.

src/app/flying-heroes-impure.component.html (excerpt)

```
<div *ngFor="let hero of (heroes | flyingHeroesImpure)">
  {{hero.name}}
</div>
```

The only substantive change is the pipe in the template. You can confirm in the live example / download example that the *flying heroes* display updates as you add heroes, even when you mutate the `heroes` array.

The impure *AsyncPipe*

The Angular `AsyncPipe` is an interesting example of an impure pipe. The `AsyncPipe` accepts a `Promise` or `Observable` as input and subscribes to the input automatically, eventually returning the emitted values.

The `AsyncPipe` is also stateful. The pipe maintains a subscription to the input `Observable` and keeps delivering values from that `Observable` as they arrive.

This next example binds an `Observable` of message strings (`message$`) to a view with the `async` pipe.

src/app/hero-async-message.component.ts

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

import { Observable, interval } from 'rxjs';
import { map, take } from 'rxjs/operators';

@Component({
  selector: 'app-hero-message',
  template: `
    <h2>Async Hero Message and AsyncPipe</h2>
    <p>Message: {{ message$ | async }}</p>
    <button (click)="resend()">Resend</button>`,
})
export class HeroAsyncMessageComponent {
  message$: Observable<string>;

  private messages = [
    'You are my hero!',
    'You are the best hero!',
    'Will you be my hero?'
  ];

  constructor() { this.resend(); }

  resend() {
    this.message$ = interval(500).pipe(
      map(i => this.messages[i]),
      take(this.messages.length)
    );
  }
}
```

The Async pipe saves boilerplate in the component code. The component doesn't have to subscribe to the async data source, extract the resolved values and expose them for binding, and have to unsubscribe when it's destroyed (a potent source of memory leaks).

An impure caching pipe

Write one more impure pipe, a pipe that makes an HTTP request.

Remember that impure pipes are called every few milliseconds. If you're not careful, this pipe will punish the server with requests.

In the following code, the pipe only calls the server when the requested URL changes and it caches the server response. The code uses the Angular http client to retrieve data:

src/app/fetch-json.pipe.ts

```
import { HttpClient }          from '@angular/common/http';
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'fetch',
  pure: false
})
export class FetchJsonPipe implements PipeTransform {
  private cachedData: any = null;
  private cachedUrl = '';

  constructor(private http: HttpClient) { }

  transform(url: string): any {
    if (url !== this.cachedUrl) {
      this.cachedData = null;
      this.cachedUrl = url;
      this.http.get(url).subscribe(result => this.cachedData = result);
    }

    return this.cachedData;
  }
}
```

Now demonstrate it in a harness component whose template defines two bindings to this pipe, both requesting the heroes from the `heroes.json` file.

src/app/hero-list.component.ts

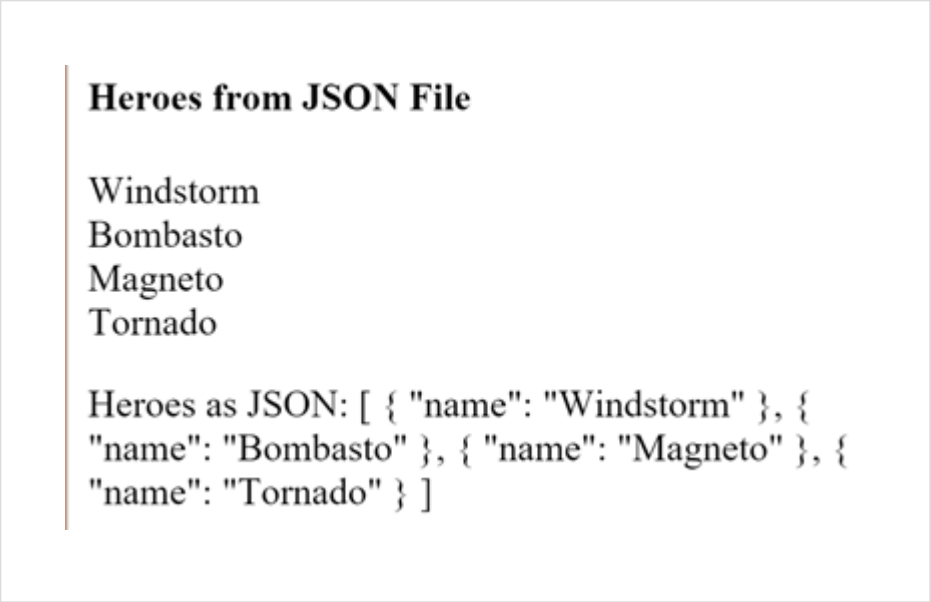
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-hero-list',
  template: `
    <h2>Heroes from JSON File</h2>

    <div *ngFor="let hero of ('assets/heroes.json' | fetch) ">
      {{hero.name}}
    </div>

    <p>Heroes as JSON:
      {{'assets/heroes.json' | fetch | json}}
    </p>`
})
export class HeroListComponent { }
```

The component renders as the following:



A breakpoint on the pipe's request for data shows the following:

- Each binding gets its own pipe instance.
- Each pipe instance caches its own URL and data.
- Each pipe instance only calls the server once.

JsonPipe

In the previous code sample, the second `fetch` pipe binding demonstrates more pipe chaining. It displays the same hero data in JSON format by chaining through to the built-in `JsonPipe`.

DEBUGGING WITH THE JSON PIPE

The JsonPipe provides an easy way to diagnose a mysteriously failing data binding or inspect an object for future binding.

Pure pipes and pure functions

A pure pipe uses pure functions. Pure functions process inputs and return values without detectable side effects. Given the same input, they should always return the same output.

The pipes discussed earlier in this page are implemented with pure functions. The built-in `DatePipe` is a pure pipe with a pure function implementation. So are the `ExponentialStrengthPipe` and `FlyingHeroesPipe`. A few steps back, you reviewed the `FlyingHeroesImpurePipe`—an impure pipe with a pure function.

But always implement a *pure pipe* with a *pure function*. Otherwise, you'll see many console errors regarding expressions that changed after they were checked.

Next steps

Pipes are a great way to encapsulate and share common display-value transformations. Use them like styles, dropping them into your template's expressions to enrich the appeal and usability of your views.

Explore Angular's inventory of built-in pipes in the API Reference. Try writing a custom pipe and perhaps contributing it to the community.

Appendix: No *FilterPipe* or *OrderByPipe*

Angular doesn't provide pipes for filtering or sorting lists. Developers familiar with AngularJS know these as `filter` and `orderBy`. There are no equivalents in Angular.

This isn't an oversight. Angular doesn't offer such pipes because they perform poorly and prevent aggressive minification. Both `filter` and `orderBy` require parameters that reference object properties. Earlier in this page, you learned that such pipes must be impure and that Angular calls impure pipes in almost every change-detection cycle.

Filtering and especially sorting are expensive operations. The user experience can degrade severely for even moderate-sized lists when Angular calls these pipe methods many times per second. `filter` and `orderBy` have often been abused in AngularJS apps, leading to complaints that Angular itself is slow. That charge is fair in the indirect sense that AngularJS prepared this performance trap by offering `filter` and `orderBy` in the first place.

The minification hazard is also compelling, if less obvious. Imagine a sorting pipe applied to a list of heroes. The list might be sorted by hero `name` and `planet` of origin properties in the following way:

```
<!-- NOT REAL CODE! -->
<div *ngFor="let hero of heroes | orderBy:'name,planet'"></div>
```

You identify the sort fields by text strings, expecting the pipe to reference a property value by indexing (such as `hero['name']`). Unfortunately, aggressive minification manipulates the `Hero` property names so that `Hero.name` and `Hero.planet` become something like `Hero.a` and `Hero.b`. Clearly `hero['name']` doesn't work.

While some may not care to minify this aggressively, the Angular product shouldn't prevent anyone from minifying aggressively. Therefore, the Angular team decided that everything Angular provides will minify safely.

The Angular team and many experienced Angular developers strongly recommend moving filtering and sorting logic into the component itself. The component can expose a `filteredHeroes` or `sortedHeroes` property and take control over when and how often to execute the supporting logic. Any capabilities that you would have put in a pipe and shared across the app can be written in a filtering/sorting service and injected into the component.

If these performance and minification considerations don't apply to you, you can always create your own such pipes (similar to the FlyingHeroesPipe) or find them in the community.