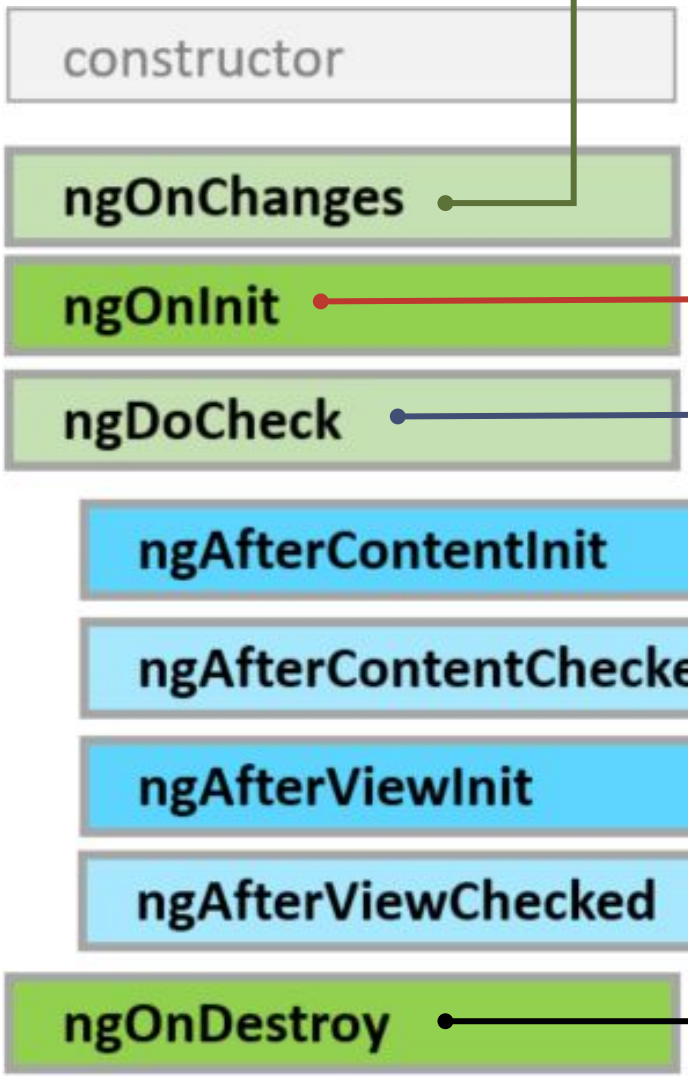
ANGULAR 2

COMPONENTS ADVANCED: LIFECYCLE

LUXOFT

LXFT LISTED NYSE

LIFECYCLE



Respond when Angular (re)sets data-bound input properties. The method receives a SimpleChanges object of current and previous property values.

Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.
Called once, after the first ngOnChanges.

Detect and act upon changes that Angular can't or won't detect on its own.

Called during every change detection run, immediately after ngOnChanges and ngOnInit.

Cleanup just before Angular destroys the directive/component. Unsubscribe observables and detach event handlers to avoid memory leaks.
Called just before Angular destroys the directive/component.

ONINIT

Use ngOnInit for two main reasons:

1) to perform complex initializations shortly after construction
2) to set up the component after Angular sets the input properties

Don't fetch data in a component constructor. Constructors should do no more than set the initial local variables to simple values.

ngOnInit is a good place for a component to fetch its initial data.

Also directive's data-bound input properties are not set until after construction. That's a problem if you need to initialize the directive based on those properties. They'll have been set when ngOninit runs.

ONDESTROY

Put cleanup logic in ngOnDestroy, the logic that must run before Angular destroys the directive.

This is the time to notify another part of the application that the component is going away.

This is the place to free resources that won't be garbage collected automatically.

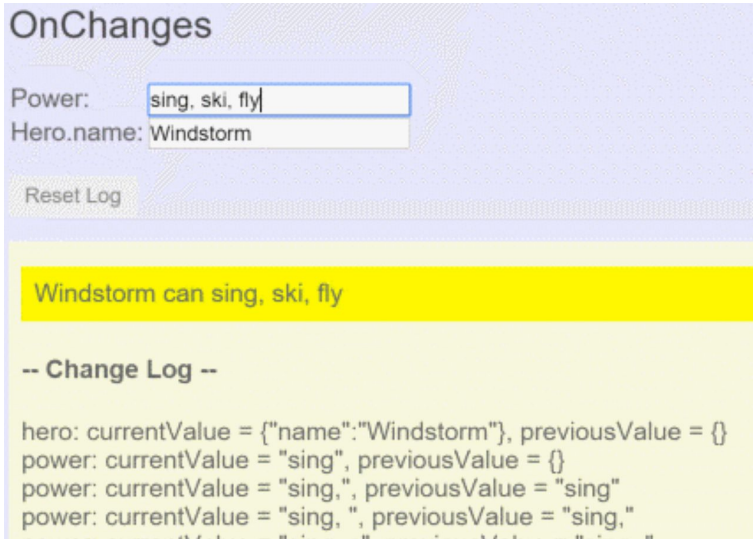
- Unsubscribe from observables and DOM events.
- Stop interval timers.
- Unregister all callbacks that this directive registered with global or application services.

You risk memory leaks if you neglect to do so.

ONCHANGES

Angular calls its ngOnChanges method whenever it detects changes to input properties of the component (or directive). This example monitors the OnChanges hook.

```
ngOnChanges(changes: SimpleChanges) {
  for (let propName in changes) {
    let chng = changes[propName];
    let cur  = JSON.stringify(chng.currentValue);
    let prev = JSON.stringify(chng.previousValue);
    this.changeLog.push(`${propName}:
      currentValue = ${cur},
      previousValue = ${prev}`);
  }
}
```



In parent component:

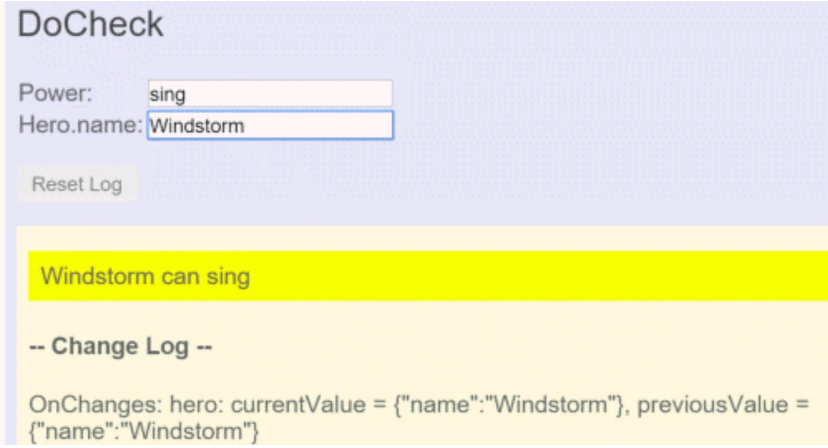
```
<on-changes [hero]="hero" [power]="power"></on-changes>
```

DOCHECK

Use the DoCheck hook to detect and act upon changes that Angular doesn't catch on its own.

```
ngDoCheck() {
  if (this.hero.name !== this.oldHeroName) {
    this.changeDetected = true;
    this.changeLog.push(`DoCheck: Hero name
     changed to "${this.hero.name}"
     from "${this.oldHeroName}"`);
    this.oldHeroName = this.hero.name;
  }

  if (this.power !== this.oldPower) {
    this.changeDetected = true;
    this.changeLog.push(`DoCheck: Power changed
     to "${this.power}" from "${this.oldPower}"`);
    this.oldPower = this.power;
  }
}
```



While the ngDoCheck hook can detect when the hero's name has changed, it has a frightful cost. This hook is called with enormous frequency — after every change detection cycle no matter where the change occurred – anywhere on the page(!).

AFTERVIEW

AfterViewInit and AfterViewChecked hooks that Angular calls after it creates a component's child views.

```
template: `
  <div>-- child view begins --</div>
    <my-child-view></my-child-view>
  <div>-- child view ends --</div>`
```

```
@Component({
    selector: 'my-child-view',
    template: '<input [(ngModel)]="hero">'})
export class ChildViewComponent {
    hero = 'Magneta';
}
```

```
export class AfterViewComponent implements  AfterViewChecked, AfterViewInit {
    private prevHero = '';
    @ViewChild(ChildViewComponent) viewChild: ChildViewComponent;
    ngAfterViewInit() {  this.logIt('AfterViewInit');  } // viewChild is set after view was initialized
    ngAfterViewChecked() {   // viewChild is updated after the view has been checked
        if (this.prevHero === this.viewChild.hero) {
            this.logIt('AfterViewChecked (no change)');
        } else {
            this.prevHero = this.viewChild.hero;
            this.logIt('AfterViewChecked');
        }
  } }
```

AFTERCONTENT

AfterContentInit and AfterContentChecked hooks that Angular calls after Angular projects external content into the component.
Content projection is a way to import HTML content from outside the component and insert that content into the component's template in a designated spot (aka transclusion in Angular 1).

ParentComponent:

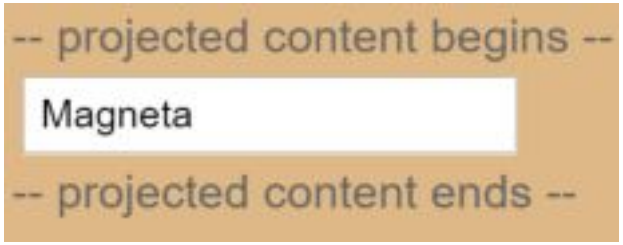
```
`<my-child>
      <my-input [value]="Magneta" >
      </my-input>
</my-child>`
```

InputComponent (selector ‘my-input’):

```
`<input [value]="value”>`
```

ChildComponent (selector ‘my-child’):

```
<div>-- projected content begins --</div>
   <ng-content></ng-content>
<div>-- projected content ends --</div>
```



@ContentChild and @ContentChildren queries will return directives existing inside the <ng-content></ng-content> element of your view, whereas @ViewChild and @ViewChildren only look at elements that are on your view template directly.

AFTERCONTENT

```
export class ParentComponent implements AfterContentChecked, AfterContentInit {
  @ContentChild(InputComponent) inputComponent: InputComponent;

  prevValue: string;
  // contentChild is set after the content has been initialized
  ngAfterContentInit() {
    this.logIt('AfterContentInit');
  }
  // contentChild is updated after the content has been checked
  ngAfterContentChecked() {
    if (this.prevValue === this.inputComponent.value) {
      this.logIt('AfterContentChecked (no change)');
    } else {
      this.prevHero = this.inputComponent.value;
      this.logIt('AfterContentChecked');
    }
  }
}
```

ParentComponent:

```
`<after-content>
  <my-child>
    <my-input
      [value]="Magneta">
    </my-input>
  </my-child>
</after-content>`
```

InputComponent (selector ‘my-input’):

```
`<input [value]="value”>`
```