USER INPUT

Developer Preview Only - some details may change

User input triggers DOM events. We listen to those events with event bindings that funnel updated values back into our components and models.

When the user clicks a link, pushes a button, or enters text we want to know about it. These user actions all raise DOM events. In this chapter we learn to bind to those events using the Angular event binding syntax.

Run the live example

Binding to user input events

We can use Angular event bindings to respond to any DOM event.

The syntax is simple. We surround the DOM event name in parentheses and assign a quoted template statement to it. As an example, here's an event binding that implements a click handler:

```
<button (click)="onClickMe()">Click me!</button>
```

The `(click)` to the left of the equal sign identifies the button's click event as the **target of the binding**. The text within quotes on the right is the **template statement** in which we respond to the click event by calling the component's `onClickMe` method. A template statement is a subset of JavaScript with restrictions and a few added tricks.

When writing a binding we must be aware of a template statement's **execution context**. The identifiers appearing within a statement belong to a specific context object. That object is usually the Angular component that controls the template ... which it definitely is in this case because that snippet of HTML belongs to the following component:

app/click-me.component.ts

```
@Component({
  selector: 'click-me',
  template: `
    <button (click)="onClickMe()">Click me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';

  onClickMe(){
    this.clickMessage ='You are my hero!';
  }
}
```

When the user clicks the button, Angular calls the component's `onClickMe` method.

Get user input from the $event object

We can bind to all kinds of events. Let's bind to the keyup event of an input box and replay what the user types back onto the screen.

This time we'll (1) listen to an event and (2) grab the user's input.

app/keyup.components.ts (template v.1)

```
template: `
  <input (keyup)="onKey($event)">
  <p>{{values}}</p>
`
```

Angular makes an event object available in the `$event` variable, which we pass to the component's `onKey()` method. The user data we want is in that variable somewhere.

app/keyup.components.ts (class v.1)

```
export class KeyUpComponent_v1 {
  values='';

  // without strong typing
```

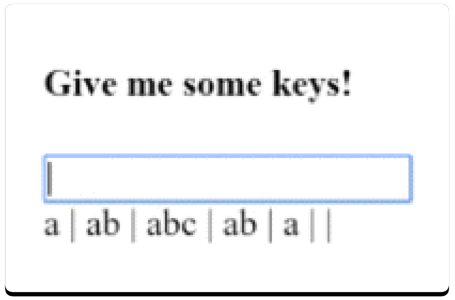
```
    onKey(event:any) {
      this.values += event.target.value + ' | ';
    }
  }
```

The shape of the `$event` object is determined by whatever raises the event. The `keyup` event comes from the DOM, so `$event` must be a standard DOM event object. The `$event.target` gives us an `HTMLInputElement`, which has a `value` property that contains our user input data.

The `onKey()` component method is where we extract the user's input from the event object, adding that input to the list of user data that we're accumulating in the component's `values` property. We then use interpolation to display the accumulating `values` property back on screen.

Enter the letters "abc", and then backspace to remove them. Here's what the UI displays:

```
a | ab | abc | ab | a | |
```


Give me some keys!

a | ab | abc | ab | a | |

We cast the `$event` as an `any` type, which means we've abandoned strong typing to simplify our code. We generally prefer the strong typing that TypeScript affords. We can rewrite the method, casting to HTML DOM objects like this.

app/keyup.components.ts (class v.1 - strongly typed)

```
export class KeyUpComponent_v1 {
  values='';

  // with strong typing
  onKey(event:KeyboardEvent) {
    this.values += (<HTMLInputElement>event.target).value + ' | ';
  }
}
```

Strong typing reveals a serious problem with passing a DOM event into the method: too much awareness of template details, too little separation of concerns.

We'll address this problem in our next try at processing user keystrokes.

Get user input from a local template variable

There's another way to get the user data without the `$event` variable.

Angular has a syntax feature called **local template variables**. These variables grant us direct access to an element. We declare a local template variable by preceding an identifier with a hash/pound character (#).

Here's an example of using a local template variable to implement a clever keystroke loopback in an ultra-simple template.

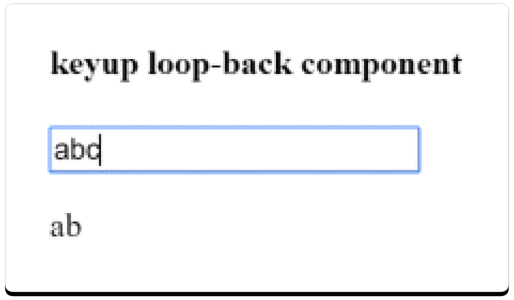
app/loop-back.component.ts

```
@Component({
  selector: 'loop-back',
  template:`
    <input #box (keyup)="0">
    <p>{{box.value}}</p>
  `
})
export class LoopbackComponent { }
```

We've declared a template local variable named `box` on the `<input>` element. The `box` variable is a reference to the `<input>` element itself, which means we can grab the input element's `value` and display it with interpolation between `<p>` tags.

The template is completely self contained. It doesn't bind to the component, and the component does nothing.

Type in the input box, and watch the display update with each keystroke. *Voila!*



> **This won't work at all unless we bind to an event**.
>
> Angular only updates the bindings (and therefore the screen) if we do something in response to asynchronous events such as keystrokes.
>
> That's why we bind the `keyup` event to a statement that does ... well, nothing. We're binding to the number 0, the shortest statement we can think of. That is all it takes to keep Angular happy. We said it would be clever!

That local template variable is intriguing. It's clearly easier to get to the textbox with that variable than to go through the `$event` object. Maybe we can rewrite our previous keyup example so that it uses the variable to get the user's input. Let's give it a try.

app/keyup.components.ts (v2)

```
@Component({
  selector: 'key-up2',
  template: `
    <input #box (keyup)="onKey(box.value)">
    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v2 {
  values='';
  onKey(value:string) {
    this.values += value + ' | ';
  }
}
```

That sure seems easier. An especially nice aspect of this approach is that our component code gets clean data values from the view. It no longer requires knowledge of the `$event` and its structure.

Key event filtering (with `key.enter`)

Perhaps we don't care about every keystroke. Maybe we're only interested in the input box value when the user presses Enter, and we'd like to ignore all other keys. When we bind to the `(keyup)` event, our event handling statement hears *every keystroke*. We could filter the keys first, examining every `$event.keyCode`, and update the `values` property only if the key is Enter.

Angular can filter the key events for us. Angular has a special syntax for keyboard events. We can listen for just the Enter key by binding to Angular's `keyup.enter` pseudo-event.

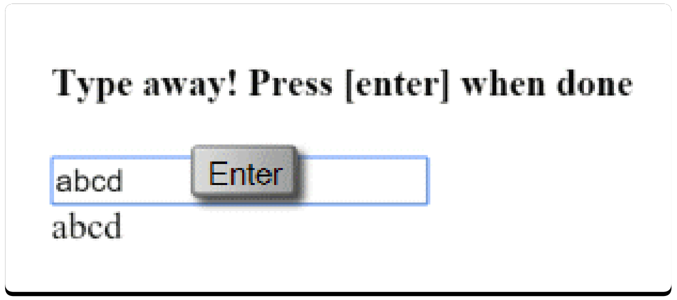
Only then do we update the component's `values` property. (In this example, the update happens inside the event binding statement. A better practice would be to put the update code in the component.)

app/keyup.components.ts (v3)

```
@Component({
  selector: 'key-up3',
  template: `
    <input #box (keyup.enter)="values=box.value">
```

```
      <p>{{values}}</p>
    `
  })
  export class KeyUpComponent_v3 {
    values='';
  }
```

Here's how it works.


Type away! Press [enter] when done

abcd

abcd

On blur

Our previous example won't transfer the current state of the input box if the user mouses away and clicks elsewhere on the page. We update the component's `values` property only when the user presses Enter while the focus is inside the input box.

Let's fix that by listening to the input box's blur event as well.

app/keyup.components.ts (v4)

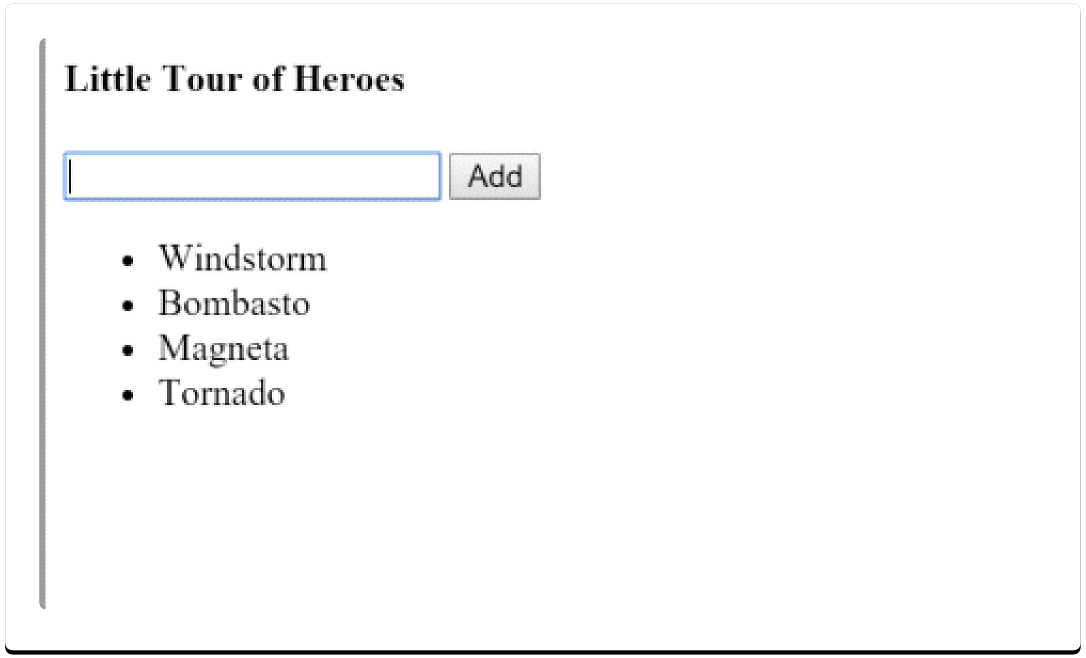
```
@Component({
  selector: 'key-up4',
  template: `
    <input #box
      (keyup.enter)="values=box.value"
      (blur)="values=box.value">

    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v4 {
  values='';
}
```

Put it all together

We learned how to display data in the previous chapter. We've acquired a small arsenal of event binding techniques in this chapter.

Let's put it all together in a micro-app that can display a list of heroes and add new heroes to that list. The user can add a hero by first typing in the input box and then pressing Enter, clicking the Add button, or clicking elsewhere on the page.



Below is the "Little Tour of Heroes" component. We'll call out the highlights after we bask briefly in its minimalist glory.

app/little-tour.component.ts

```
@Component({
  selector: 'little-tour',
  template: `
    <input #newHero
      (keyup.enter)="addHero(newHero.value)"
      (blur)="addHero(newHero.value); newHero.value='' ">

    <button (click)=addHero(newHero.value)>Add</button>

    <ul><li *ngFor="#hero of heroes">{{hero}}</li></ul>
  `
})
export class LittleTourComponent {
  heroes=['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
  addHero(newHero:string) {
    if (newHero) {
      this.heroes.push(newHero);
    }
  }
}
```

We've seen almost everything here before. A few things are new or bear repeating.

Use template variables to refer to elements

The `newHero` template variable refers to the `<input>` element. We can use `newHero` from any sibling or child of the `<input>` element.

Getting the element from a template variable makes the button click handler simpler. Without the variable, we'd have to use a fancy CSS selector to find the input element.

Pass values, not elements

We could have passed the `newHero` into the component's `addHero` method.

But that would require `addHero` to pick its way through the `<input>` DOM element, something we learned to dislike in our first try at a keyup component.

Instead, we grab the input box *value* and pass *that* to `addHero`. The component knows nothing about HTML or the DOM, which is the way we like it.

Keep template statements simple

We bound `(blur)` to *two* JavaScript statements.

We like the first one, which calls `addHero`. We do not like the second one, which assigns an empty string to the input box value.

The second statement exists for a good reason. We have to clear the input box after adding the new hero to the list. The component has no way to do that itself because it has no access to the input box (our design choice).

Although the example *works*, we are rightly wary of JavaScript in HTML. Template statements are powerful. We're supposed to use them responsibly. Complex JavaScript in HTML is irresponsible.

Should we reconsider our reluctance to pass the input box into the component?

There should be a better third way. And there is, as we'll see when we learn about `NgModel` in the Forms chapter.

Source code

Here is all the code we talked about in this chapter.

```
import {Component} from 'angular2/core';
```

```

@Component({
  selector: 'click-me',
  template: `
    <button (click)="onClickMe()">Click me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';

  onClickMe(){
    this.clickMessage ='You are my hero!';
  }
}
```

Summary

We've mastered the basic primitives for responding to user input and gestures. As powerful as these primitives are, they are a bit clumsy for handling large amounts of user input. We're operating down at the low level of events when we should be writing two-way bindings between data entry fields and model properties.

Angular has a two-way binding called `NgModel`, which we'll learn about in the `Forms` chapter.

Next Step

Forms