* Exploring Angular Forms
  + Forms in Angular
    - Note that the **mat-form-field > input** element that we use to allow users to filter data is a way for a user to interact
    - But, if you look at the generated HTML, you will only see an **input** element (no **form** element)
    - This is because you don’t need form data to submit this request; it is done via an async JS XMLHttpRequest (XHR) (i.e. AJAX)
    - This doesn’t require a form container to handle data encoding and transmission
  + Reasons to Use Forms
    - Even if we don’t need a form to handle encoding/transmission, there are other advantages
    - Specifically, there are disadvantages to using a single input element and a text string
      * Can’t keep track of global form state
      * Can’t easily display an error message to users if a form is invalid
      * We are not validating the data in any way
    - Note that we could add these features manually in our Components and Angular directives like \*ngIf and \*ngFor, but there are easier ways
    - Specifically, Template-Driven Forms and Client-Driven Forms (aka Reactive Forms)
  + Template-Driven Forms
    - These forms are defined mainly in the Template for your component (big surprise)
    - They involve defining a form element and using the ngForm module
    - You define inputs in the form and define attributes (per normal) and use the [()] two-way property binding syntax to ensure the forms are responsive
    - Note that [(ngModel)]=”city.Name” is really Angular shorthand for [ngModel]=”city.Name” and (ngModelChanged)=”city.Name = $event”
    - Pros
      * These forms are easy to write (especially from an HTML knowledge standpoint)
      * These forms are easy to read and understand if you have a decent background in HTML
    - Cons
      * These require a lot of HTML which is difficult to maintain and more error-prone than pure TS
      * These forms cannot be unit tested; they require E2E tests with a browser to e.g. verify the validation logic
      * Readability will quickly drop as you add more and more validation and logic
    - Generally, Template-Driven is a good option when the forms are small and the validation logic is light
  + Model-Driven/Reactive Forms
    - These types of forms were introduced in Angular 2 to combat the cons of Template-Driven forms
    - The main difference is the amount of HTML required in the template is significantly less
    - In the Component you define a FormGroup element that represents the whole form and encapsulates related controls
    - Each individual data input is represented by a FormControl element within the FormGroup (either the parent or a child)
    - The FormControl tracks data related to the control’s current state and the actual control value
    - The FormGroup tracks the state of each child control; so the group is only valid if all the children are valid
    - The powerful difference here is in where the control lies
      * For template driven, we don’t really have control over the validation or submission logic once it’s out there; it’s written in the HTML and there it will stay
      * For model-driven/reactive, the HTML updates parts of a strongly-typed FormGroup and then we operate on that data directly in the component; giving us much more control
* Building Our First Reactive Form
  + CityEditComponent
    - So this is going to be a form that allows us to edit a city instance
    - To do this we are going to create a new component (new TS, CSS, and HTML files) for a **city-edit** component
    - For this component we will define a parent FormGroup that will be what the use interacts with, and a City property that will hold the actual data
    - Note the FormModel will only represent the data that we want the user to be able to edit (e.g. not the primary ID)
    - When the page is loaded and onNgInit() fires we create the new FormGroup with the 3 FormControl’s for the editable fields
    - We then get the city ID from the activated router and use this to get the city instance we need
    - We set the local City variable based on this result and then use FormGroup.patchValue() to set the form values from the result
    - On form submission we get the values of the form (I assume from the formControlName attributes in the Template), set our local City instance with them, and use those values in a PUT request to the backend
    - After submission we use the router to navigate back to the main cities page
    - Adding the Navigation Link
      * So per usual we import the Component in our app.module.ts file and we need to add a router link to navigate to the page
      * This path for this router link though uses different syntax to ensure the City ID is set in the route parameters
      * Specifically it uses path: ‘city/:id’
    - Then we update the main cities list view to have it so if you click on the ID you navigate to the CityEditComponent
    - This is does via an anchor tag binding routerLink to [‘/city’, city.id]
* Adding a New City
  + Note that a common requirement of a Detail View with editing capabilities is to be able to create a new city from the same view