* Optimizations and Tweaks
  + Template Improvements
    - We’ve already done some of this, specifically consolidating validation logic into methods for each detail view
    - But we can do better by removing the duplicate methods in each of these edit views
  + Class Inheritance
    - TS allows for class inheritance and polymorphism just like any good OOP language
    - A child class created through inheritance acquires all properties of the parent, except for ctors, dtors, overloaded operators, and private members
    - To extend a class we use the extends keyword and the name of the base class that we want to extend off of
    - Note that we can override any element in the base class just by creating a function/property with the same name (no keyword required)
* Bug Fixes and Improvements
  + There were a couple bugs we fixed related to adding patterns and range limits to the lat and lon fields
  + We also added a total city count for countries that is a count of all the cities in a country (i.e. the cities that have that country as its parent)
  + To do this and avoid loading all the city data for each country on the GET countries call, we created a Country DTO class
  + This class does not have a list of all the child cities, only the count; this field is calculated on each GET call (so that it is always up to date)
  + DTO Classes – Should we Really Use Them?
    - So the DTO class we created is not very DRY and duplicates most of the elements in our Country entity
    - We could instead add the total city count to the Country entity, or inherit off of Country to create our Country DTO
    - But, this can cause issues related to a separation of concerns and security considerations
    - Separation of Concerns
      * Note that we typically want to separate the entity modeling the domain object from client-side needs
      * If something is needed solely for the client-side framework, it should be added to a separate DTO class
      * This is because these front-end elements often have no real bearing on the domain object
    - Security Considerations
      * Additionally, transferring all the data in your entities over the wire to your front-end can be dangerous
      * This is obviously more clear if your entities have sensitive info that should be kept secret
    - DTO Classes vs Anonymous Types
      * Note that we could also use anonymous types to project the Country DbSet to a list containing the DTO elements
      * This works and is a quick way to achieve the desired result
      * But, this will cause some headaches later, especially when adding UTs where we want strongly-typed objects
* Data Services
  + In all our work here we’ve used Angular’s built-in HTTP API client in the @angular/common/http package
  + This client rests on the XMLHttpRequest interface and has many advantages (discussed previously)
  + This makes using the HttpClient class a logic choice for folks using ANgular to build out their front end
  + But, there are other alternatives to how we implement its usage
  + XMLHttpRequest vs Fetch vs HttpClient
    - The HttpClient class is build on XMLHttpRequest (XHR), which is an API provided through the browsers’ JS engine
    - This API can transfer data between the browser and a web server in an async way, without needing to reload the entire page
    - This was pretty much the only way to do this until ~2017 when the Fetch API came out
    - This is another interface for fetching resources that is a modern alternative to the XHR API
    - XMLHttpRequest
      * This concept first made an appearance in 1999 which MS released the first version of Outlook Wed Access (OWA) for MS Exchange Server 2000
      * A second version was implemented in the MS XML Core Services (MSXML) library and shipped with Internet Explorer
      * Shortly after that Mozilla created an nsIMXLHttpRequest interface that was similar to the MS interface
      * The difference here was that it also included a wrapper that allowed the object to be used through JS due to an object that was returned by the browser
      * This object was called XMLHttpRequest, and quickly became a de facto standard in major browsers
      * The term Asynchronous JavaScript and XML (AJAX) was coined in 2005 for a set of web dev techniques used to create async web apps on the client side (where XMLHttpReqeust existed)
      * The World Wide Web Consortium (W3C) release of draft of a spec for XMLHttpRequest in 2006
      * This spec paved the way for wide adoption of AJAX development, although initial usage was difficult due to browser differences
      * Cross-browser JS libraries (e.g. jQuery, Axios, MooTools) provided a standard implementation without needing to work direcectly with the underlying XMLHttpRequest library
      * The XHR data format switched from XML to JSON, HTML, and plain text
      * When the Reactive Extensions for JavaScript (RxJS) library released, the XHR object could easily be put behind an Observable
      * This allowed you to mix and match with other observables, subscribe/unsubscribe, pipe/map, etc.
      * This is what is behind Angular’s HttpClient class; it is Angular’s way of dealing with the XMLHttpRequest object that wraps the object in an Observable pattern
    - Fetch
      * The Fetch API was released as a way to do similar things to the XHR object, but in a cleaner way
      * The XHR code (especially before cross-browser tooling) often required significant code overhead
      * The code for Fetch is more readable, has built-in JSON capabilities, and Promises allowing for chaining and async/await tasks w/o needing callbacks
      * Generally, the Fetch API is preferable to raw XHR code
    - HttpClient
      * But Angular’s HttpClient class provides a clean Observable-based approach to using XHR
      * This has many of the benefits of the Fetch API while still using XHR behind the scenes
      * The class is also maintained by the Angular framework and as such is injectable (which makes it easier to UT)
      * But, keep in mind that if you also need to work outside of the Angular framework in your app, that Fetch API provides similar benefits and can be used in both places effectively
* Building a Data Service
  + We are going to stick with using HttpClient in Angular
  + But, note that we currently have all our HTTP calls in the Components; this is not ideal for a number of reasons
  + If we want to process errors differently (e.g. sending them to a separate server), post-process or cache data requested from the server, implement retry logic, etc.
  + Adding all this logic to each component is obviously bad, and putting all this in a base class is not ideal as the child class logic can quickly become complex
  + Instead, we are going to define a data service that will encapsulate the DAL from the Data Presentation Logic and inject that service into our components
  + Creating the BaseService
    - We have separate contexts here that have define separate concerns (Country and City)
    - So we will define a separate service for each context, and a base service that will define a common interface
      * Note that we need to do this via an **abstract class** here, instead of a direct interface like we did for the city and country interfaces
      * This is because Angular does not allow us to have interfaces as providers
      * So, to create an interface for a service, we need to create an abstract class