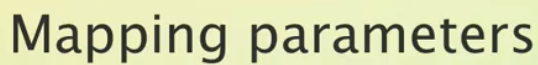
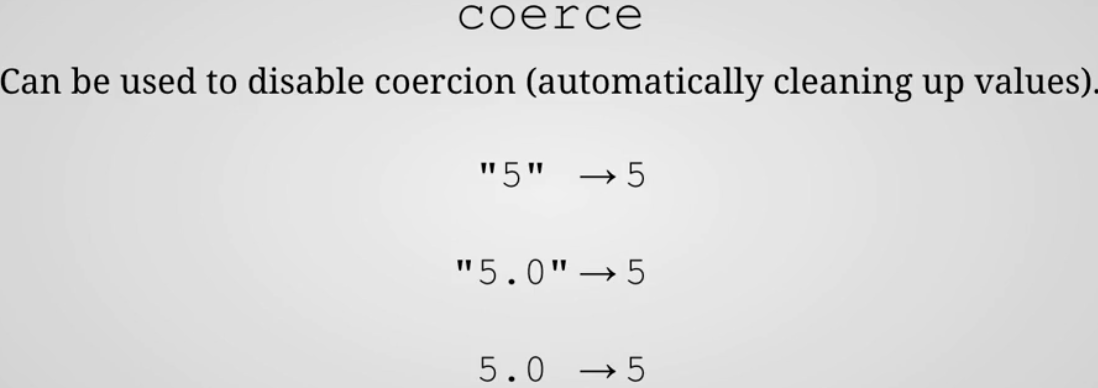
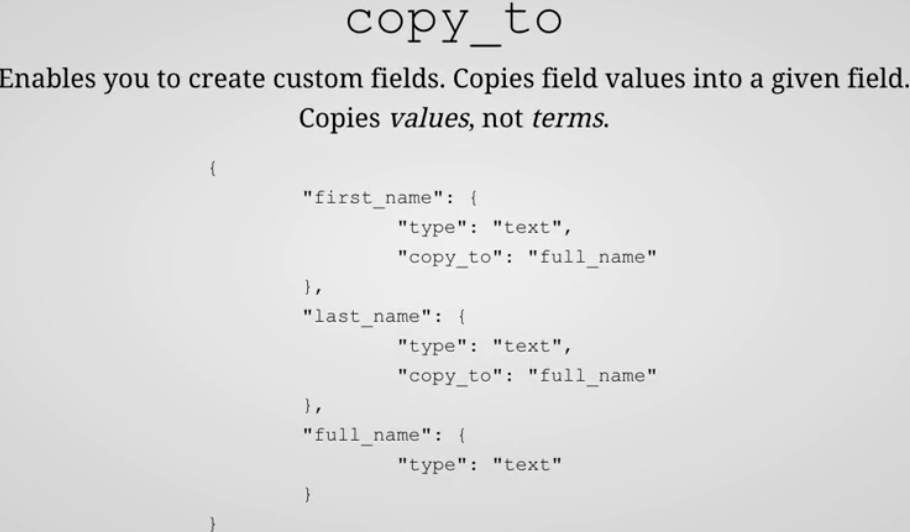
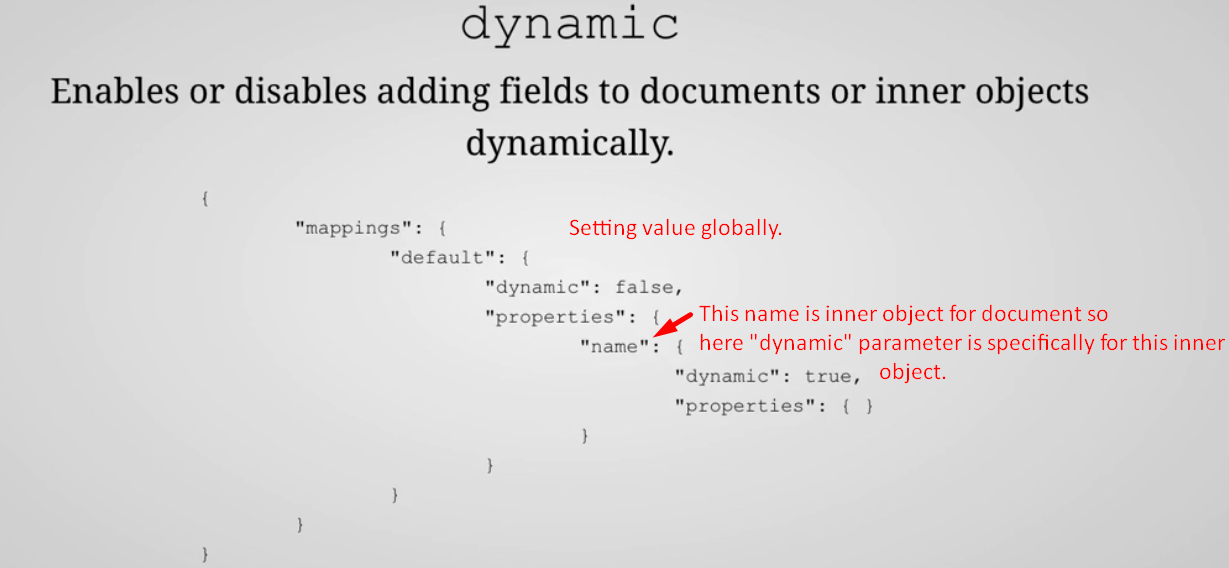
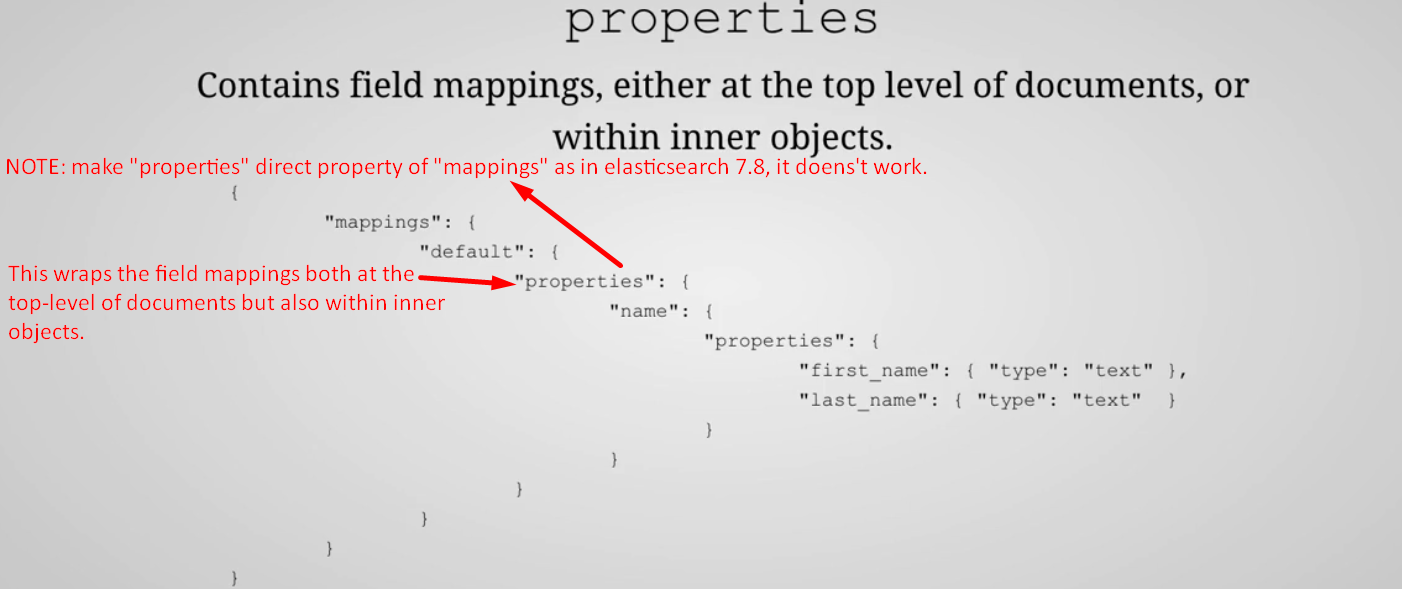
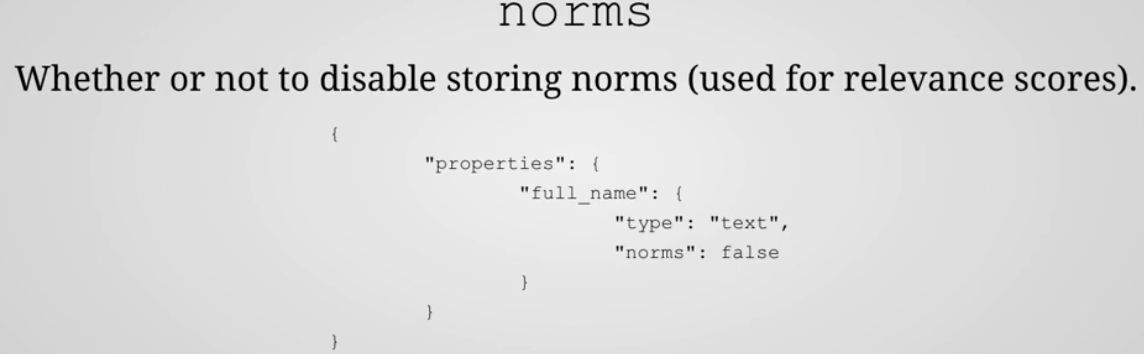
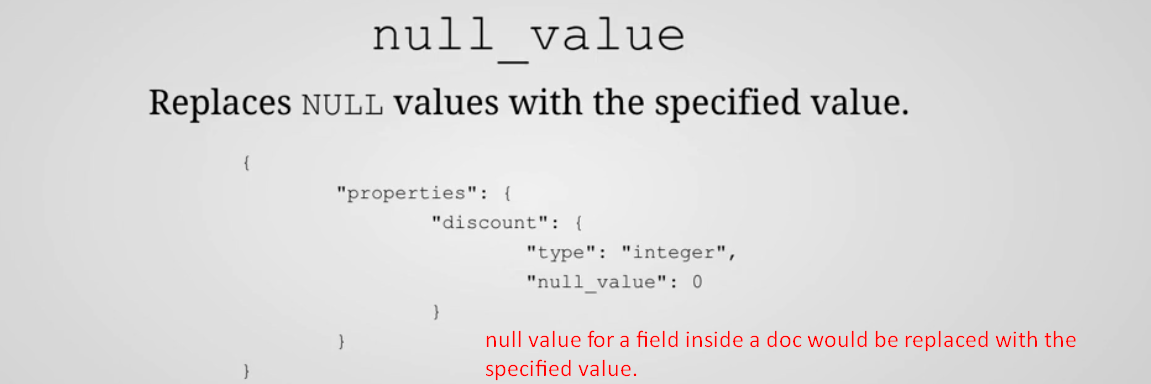
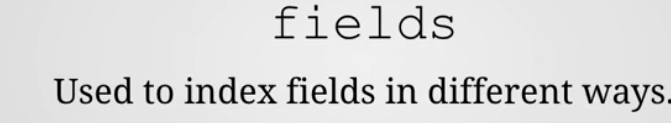
1. 
2. 
3. **Parameters**:
   1. **coerce**:  
      
      1. When indexing data, this data may come from various sources, and it may not always be in the exact format that you expect.
      2. Elasticsearch handles this by using coercion meaning that it converts values to the proper data type behind the scenes.
      3. For instance, a document might contain a value of “5” as a string for an integer field.   
         The two data types don’t match. But Elasticsearch can reason that the intension was probably to send the number 5.
      4. Elasticsearch will try to clean up this data such as coercing strings to numbers as in this example.
      5. It could also be the case that a floating point is specified for an “integer” field in which case Elasticsearch will truncate the floating point to an integer value.
      6. This all happens by default but can be disabled by setting the “coerce” parameter to “false”.  
         After that, Elasticsearch would reject the document that doesn’t contain the correct data type for the field and not try to coerce the value into the correct data type.
   2. copy\_to:  
      
      1. This parameter enables us to build a custom field with fields that we choose.
      2. For instance, if we have the “first\_name” and “last\_name” fields, we can specify that the values of these fields should be copied to a “full\_name” field.  
         This field then consists of both values and can be queried like any other field.
      3. It is field’s value that is copied to the custom field not the terms that are output by the analyzer used for the field.  
         NOTE: One more note 🡺 Copied values will not show up within the “\_source” meta field.
   3. **dynamic**:  
      
      1. The “dynamic” parameter can be used to disable dynamic mapping for new fields as you have seen.
      2. This parameter involves both new top-level document fields and new fields for inner objects.  
         For instance: We might have a “name” object consisting of a “first\_name” and a “last\_name” field.  
         Should we be allowed to add new fields to the object, such as a “middle\_name” field?  
         By default, we can but that can be disabled with the “dynamic” parameter.  
         This parameter can both be used globally for the type (“default” in this case) or for a specific field.  
         The two approaches can even be combined to override the global configuration for the type.  
         For example, the example that you can see here disables dynamic mapping for the “default” type but enables it specifically for the “name” inner object, meaning that we can add new fields to this object without having to first add a mapping explicitly.
      3. The “dynamic” parameter may contain one of three values:
         1. true: To enable dynamic mapping which is the default.
         2. false: new fields are simply ignored and the mapping should be added explicitly for new fields to be used.
         3. strict: Elasticsearch will throw an exception when detecting an unknown field, and the document will therefore be rejected and not indexed.
   4. properties:  
      
      1. You actually saw this parameter in action in the previous lecture when adding a mapping at index creation time.   
         In that context, it’s used at the root level of the mapping object as a wrapper around field mappings.  
         That’s one use of the parameter, but it is also used in two other scenarios; when defining fields within an “object” (nested field) and when Elasticsearch automatically adds field mappings through dynamic mapping.
   5. norms:  
      
      1. When running a search query, Elasticsearch doesn’t just determine whether or not a document matches; it also works out how well a document matches.  
         That is of course to give the user the most relevant search results first, which you would expect from a search engine.
      2. We will see how Elasticsearch calculates the relevance of a matching document later.  
         For now, know that Elasticsearch stores some info that enables calculating relevance scores.  
         More formally, Elasticsearch stores so-called normalization factors for fields that have scoring enabled.  
         These factors are referred to as norms.
      3. The “norms” parameter can be used to disable the storage of this information.   
         The main benefit of doing so would be to save disk space.
      4. When disabling the storage of the norms, Elasticsearch loses the ability to sort documents matching a given query by relevance.
      5. For fields those are used for aggregations or just filtering out documents and thereby not scoring documents that would not be an issue.  
         In situations like that, storing the norms would be redundant in the first place.  
         So, if you know that you will only use a field for those purposes, then you can disable norms and save quite a lot of disk space.  
         **NOTE**: We can’t re-enable norms at a later point without re-creating your index.
   6. format:   
      
      1. This can either be done by specifying a custom format using the JODA format, or by using one of the formats that are built into Elasticsearch.
   7. null\_value:  
        
      **Example**: for “discount” field, if we set null for some reason, it can be replaced with 0.
   8. fields:  
      
      1. This parameter is used for indexing a field in different ways for different purposes.
      2. You actually saw that earlier when I showed you the mapping that Elasticsearch automatically added for us based on the test data that we have imported. More specifically, Elasticsearch added an additional mapping with the “keyword” type for every “text” field.  
         The purpose of that is that you can then use text fields for sorting and aggregations.  
         Let’s see example later.