JAVAFX INTERFACE ELEMENTS

Search bars: Dynamically Filter a TableView in JavaFX

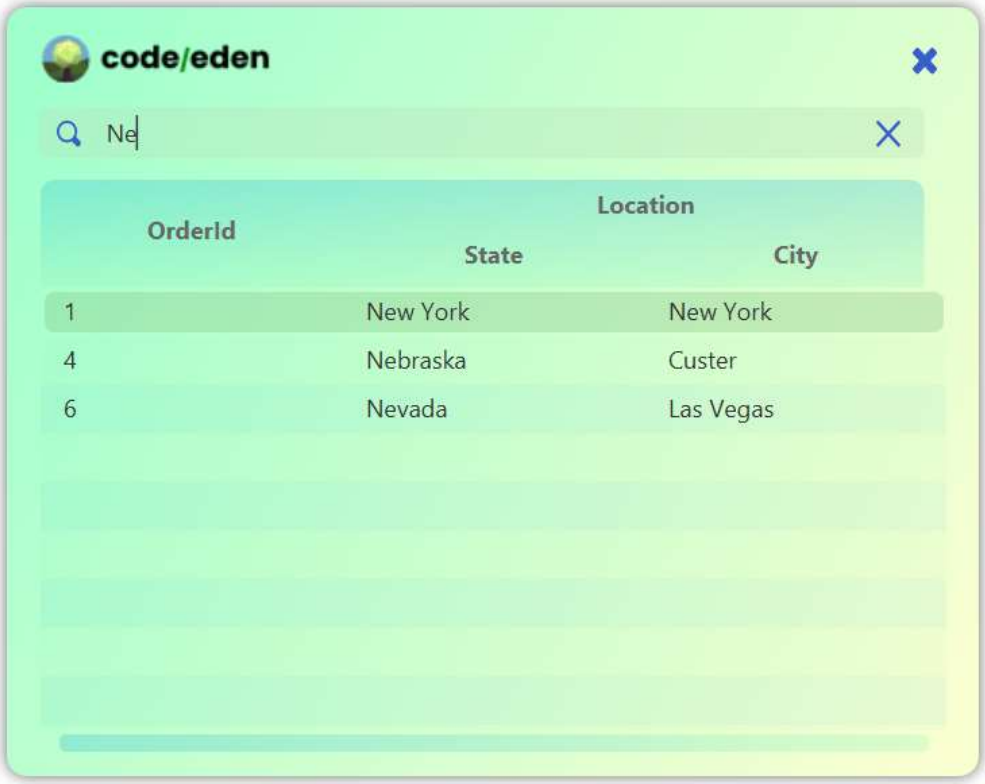
by
Ed Eden-Rump

posted on
July 19, 2020 (July 19, 2020)

If you have a lot of data and no way to query it, you're in for a bad time. If you're preparing data for users to sift through, one of the most common expectations is to be able to search for, or filter data. As more rows get added to a table, the usability goes down quickly if users can only scroll for data.

Fortunately, JavaFX provides direct support for content filtering. **Dynamically filtering content can be achieved by setting a Predicate on a FilteredList. This predicate should be updated as user input changes the search criteria. Placing a listener on the search box TextField is a common way to achieve this.**

The starting code for this tutorial is here, which we've taken from the tutorial on styling a TableView. In this project, we'll add a search bar, using SVG images to give the interface a clean look, and we'll walk through ways to filter the results based on the search. We'll then dynamically add suggested text to the `TextField` to provide search suggestions.



Finally, we'll cover some ways to combine your filters with multiple predicates for an even better user experience.

How does JavaFX store it's data?

Before we filter our results, we'll look a little into how JavaFX stores its data. If you already know about ObservableLists, you can skip down to filtering and get started.

In this section, we'll talk through what purpose the `ObservableList` serves in JavaFX and how a `TableView` generates its content.

ObservableList

Almost every list of objects maintained by JavaFX nodes is an `ObservableList`, just as almost every attribute is a `Property`. JavaFX does this so that users can establish change listeners easily. Although this comes with a performance overhead, JavaFX guesses that user interfaces will almost never take over your computer's performance. For the most part, that's absolutely on the money.

So, everything that's displayed onto a Scene by JavaFX is stored in an ObservableList. That, of course, includes the `TableView` and the objects that populate its rows.

TableView

If you were printing a list of objects to the screen, the easiest thing to do would be to loop through the contents of the list, and for each object, print the `toString()` method for each object. That's not too far from what the `TableView` actually does...

When creating or updating a `TableView`, every `TableColumn` has value factory. You can customise these, but by default, they'll get the value of the `Property` with `get()` and access the contents for display with `toString()`.

Obviously, it's important to realise that we're not necessarily showing all attributes of an Object when we show it in a table. Every column for which a Property is assigned gets a display, but only those columns. We therefore need to make a conscious decision. Are we going to let the users search for stuff that they might not necessarily be able to see at search time.

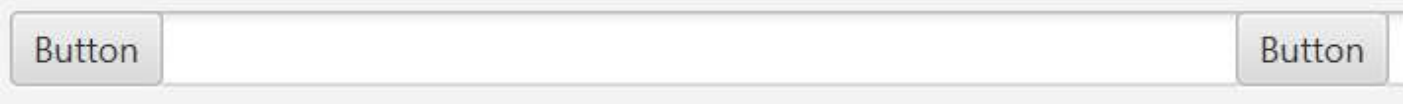
For this example, we'll filter the results based only on what the user can see in the table.

Search box

Creating the search box is pretty simple. We'll create a background `Node`, which is going to store our `TextField` and `Buttons`.

```
<StackPane>
    <TextField fx:id="searchBox" styleClass="transparent"/>
    <Button text="Button" StackPane.alignment="CENTER_RIGHT"/>
    <Button text="Button" StackPane.alignment="CENTER_LEFT"/>
</StackPane>
```

Of course, the results are a little un-inspiring...



In a similar way to how we used SVG files to create the GitHub button, we'll use vector graphics to create some components here. The left button will be used to highlight the fact that this is a search bar. For that, we'll conform to the user expectation of a magnifying glass. And on the right, we'll use a cross to highlight the delete text button.

We'll also use a little padding so the TextField stretches to the edges of the buttons rather than over them. Finally, a little CSS and we're done.

If you'd like to see how that's achieved, check out the code on our GitHub here.

How to search through data

There are multiple ways of searching through content. We won't explore indexing and hashing, because we don't have that much data to search through! But it's good to know the option's there.

Instead, we'll simply loop through the properties of the object we want to query, testing the fields we want to check against the contents of the search bar.

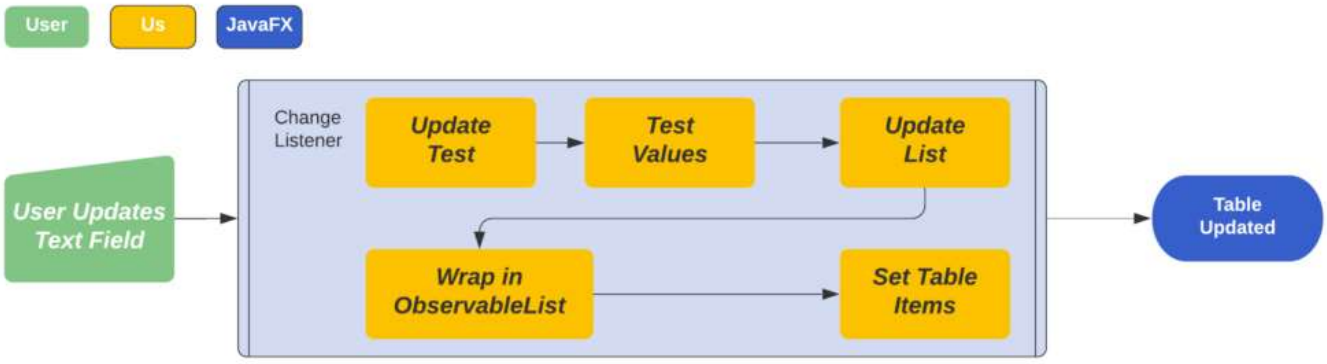
We'll look at two main ways to achieve this:

1. Manually hooking into the search box's `ChangeListener` and updating the list ourselves.
2. Using JavaFX's `FilteredList` and applying a `Predicate`.

Manually updating the ObservableList

The first thing we need to define – regardless of how we perform the searching – is a method inside the ChangeListener of the search box. That's going to update our search every time the user enters or edits their search.

Then, to manually search through each of the fields, we need to define and update our test criteria (the search box text). We'll test that against the properties of Objects in our list, and generate a new list of objects that pass the test. Finally, we'll wrap our new list in an `ObservableList` and use it to set the items on the Table.



We'll implement each of these steps in turn, and finally create the ChangeListener, passing it our test method.

Firstly, we'll define a method that tests whether an Order matches the search text. In this case, we'll simply search through the City, State and Order ID properties. It's not the most graceful, but it works..

```
private boolean searchFindsOrder(Order order, String searchText){
    return (order.getCity().toLowerCase().contains(searchText.toLowerCase())) ||
            (order.getState().toLowerCase().contains(searchText.toLowerCase())) |
            Integer.valueOf(order.getId()).toString().equals(searchText.toLowerCa
}
```

Next, we'll create a method that loops through a list of Orders and performs this test on each `Order` in turn. It should create a new `List<Order>`, containing only the orders that have passed the test and pass it back to the caller as an `ObservableList`.

```
private ObservableList<Order> filterList(List<Order> list, String searchText){
    List<Order> filteredList = new ArrayList<>();
    for (Order order : list){
        if(searchFindsOrder(order, searchText)) filteredList.add(order);
    }
    return FXCollections.observableList(filteredList);
}
```

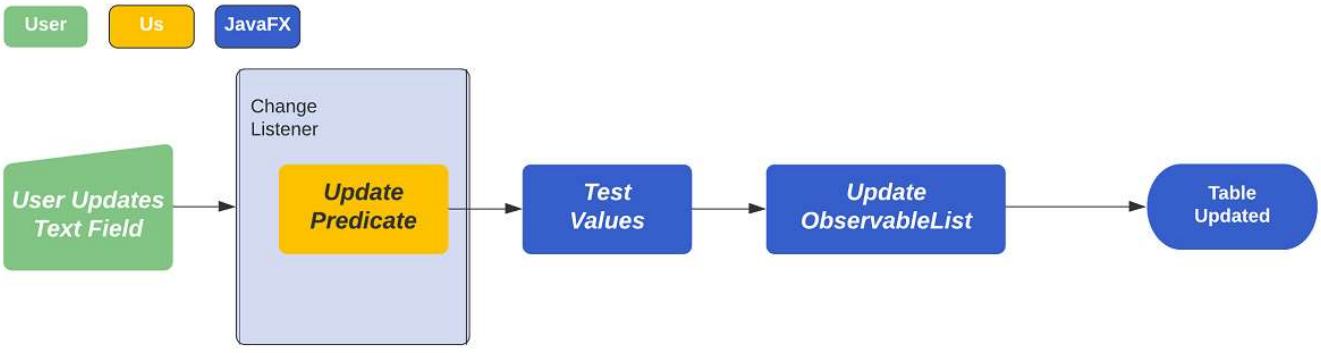
Finally, we hook into that `ChangeListener`, using the `newValue`, which represents the text currently in the search box. Then, using the `ObservableList` we get from our `filterList()` method, we set the items on the table.

```
searchBox.textProperty().addListener((observable, oldValue, newValue) ->
    exampleTable.setItems(filterList(orders, newValue))
);
```

Search bar with predicates

If the manual looping through our Orders seemed like a bit of a trek, that's probably because it was. In fact, JavaFX provides support for dynamically filtering content with predicates on the fly, through its `FilteredList` class.

Handily, the `FilteredList` extends `ObservableList`, meaning that if we pass it to the `TableView` before searching, the filtering and updating happens *automatically*. The `TableView` will even update it's contents automatically rather than us having to set the list each time. So, instead of multiple functions, we simply have to update the predicate.



When we use the FilteredList, we're responsible for setting the `Predicate` or conditions on which the decision is made. The `FilteredList` maintains two lists – the complete list, and those items that pass the test. It updates these whenever it detects a change in the list, or in the predicate.

The `Predicate` can be set when we create the FilteredList. We could set it, at creation, based on the `searchBox.getText()`, but that would set it only on the *value* of the text as it is at creation. It wouldn't update later. Instead, we create the `FilteredList` with the default `Predicate`.

```
FilteredList<Order> filteredData = new FilteredList<>(FXCollections.observableLis
exampleTable.setItems(filteredData);
```

With the default `Predicate`, all values are returned, so the list isn't filtered yet.

Next, we can create a method that will test for the current value of the search box and return a predicate for Orders that match that text. Then, we'll add it to the `ChangeListener` as before. That way it'll be fired when the text box is edited.

```
private Predicate<Order> createPredicate(String searchText){
    return order -> {
        if (searchText == null || searchText.isEmpty()) return true;
        return searchFindsOrder(order, searchText);
    };
}
```

This time, the only thing we'll do inside this listener is update the predicate by calling the method we just created.

```
searchBox.textProperty().addListener((observable, oldValue, newValue) ->
        filteredData.setPredicate(createPredicate(newValue))
);
```

It's usually good practice to include a statement to return a default value if the text is empty, or in case the text box is removed. We'll return true in both cases, so if anything happens, the table will show all its data.

That's it for filtering! Using either of the two methods above, the `TableView` should update as you type.

Conclusions

Dynamically filtering content is relatively easy to achieve in JavaFX.

We can filter `TableView` content in two main ways – manually, or by using the FilteredList class JavaFX provides. In either case, we can update our search criteria by placing a ChangeListener on the search box TextField. This way, each time the user changes their search, the `TableView` is updated automatically.

As always, all of the code we've used for this project can be found on our Github, here.

---

data