Lists and Trees

In October-November 2018, we addressed some performance issues of our tree widget. In doing so, we took the chance to rewrite it and ended up creating a couple of different widgets available to use within VS Code. This document aims to introduce the core list and tree widgets available, as well as summarize each widget's particularities, which should help in choosing what widget to use. Finally, we present some performance numbers after adopting the new `ObjectTree` in the Problems panel.

Widgets

All the following widgets are compositions of one another. Let's start with the bottom of the composition chain: the list.

List

At its core, the List is a ***virtual* rendering engine**. It can render a collection of elements in a scrollable view, while making sure only the visible elements actually end up in the DOM at any given point in time. Scrolling up and down will trigger a series of computations to determine whether DOM elements should be inserted in or removed from the document. Minimizing DOM access is what lets the List scale to large quantities and keep up with performance. You can easily add 100k elements to it without breaking a sweat.

A requirement of virtual rendering is the need to know each element's height in pixels before it gets rendered. At runtime, the list will ask the height to a provided virtual rendering controller, keep an in-memory map of each element's height and track the viewport's position in order to know which elements should be rendered in and out of the DOM. Note that each item can have its own height.

> *List splice animation*

Given that the collection model for a list is a simple array, the API to modify a list's element collection is very simple. The `splice` call allows for deleting and inserting continuous items in the list. Here's a simplified version of it:

```
class List<T> {
  splice(start: number, deleteCount: number, toInsert: T[]): void
}
```

Apart from being a virtual rendering engine, the `List` provides quite a lot of functionality that actually make it a usable widget: keyboard and mouse navigation, focus and selection traits, accessibility roles, etc. These features are what defines `List` as a usable widget across our workbench.

Index Tree

A tree UI widget should be able to represent tree-like data, eg. a file explorer. A rendered tree can always be rendered as a list; it is each row's indentation level and twistie indicators which give the user the perception of tree structure. By leveraging the virtual rendering functionalites of the list we can use composition to create a tree widget.

There is the question of API: how can we keep a tree-like structure API relatively simple, yet allow for complex operations, such as removing and inserting whole subtrees? If we take the same `splice` analogy as the list's (pick a location, remove some elements and add other elements), it's possible to come up with a tree `splice` call, in which the location is multi-dimensional (the first element of the `start` array represents the index in the root's children array; the `n` -th element of the `start` array represents the index in the `n - 1` -th children array) and the elements to insert are entire subtrees.

```
interface TreeElement<T> {
  element: T;
  children: TreeElement<T>[];
}

class IndexTree<T> {
  splice(start: number[], deleteCount: number, toInsert: TreeElement<T>[]): void;
}
```

The `IndexTree` 's main responsibility is to map tree `splice` calls into list `splice` calls. It is a composition of the `List` widget in order to provide enough functionality to render tree-like models with the correct identation levels and twistie states per element. It also provides additional keyboard and mouse handling in order to support tree operations such as expand, collapse, select parent node, etc.

Additionally, the `IndexTree` supports filtering. An instance of `IndexTree` can be created with a filter which can have fine grained control of which elements should be filtered out of the view. The `filter` call can optionally return additional data computed during its operation, which will be passed along to the tree renderer; this is useful in cases where substring highlights are computed during the filtering phase and need to be reused during the rendering phase.

Object Tree

While the `IndexTree` is both powerful and performant, it also has a sub-optimal API. Often, it is not trivial to compute a `start: number[]` location to call `splice` with. It is a much more common scenario to simply replace complete subtrees by addressing an actual element. Given ES6 maps, we can provide just that:

```
class ObjectTree<T> {
  setChildren(element: T | null, children: TreeElement<T>[]): void;
}
```

The `ObjectTree` is a mapper between `setChildren` calls to tree `splice` calls. Again, it is a composition of the `IndexTree` widget and provides an API which easier to use since the user can always reference elements already in the tree. The `null` value is specially reserved to represent the root element.

Data Tree

There are certain use cases in which data models are not fully known: they are lazily discovered. A file explorer is a good example. Once in its initial state, a file explorer renders only the top-level files and folders. Once expanded, a folder should resolve its children by doing some IO and render its children once done. There is an obvious issue: what if the IO is slow?

```
interface IDataSource<T> {
  hasChildren(element: T | null): boolean;
  getChildren(element: T | null): Thenable<T[]>;
}

class DataTree<T> {
  constructor(dataSource: IDataSource<T>);
  refresh(element: T | null): Thenable<void>;
}
```

The `DataTree` abstracts away the user's model using a data source interface. Once again, it is nothing but a composition on top of the `ObjectTree`. Tree elements can be refreshed via API or by expanding for the very first time. Refreshing a tree element eventually calls the data source which returns a thenable of its children elements. It makes sure to correctly handle conflicting refresh calls (concurrent calls to refresh an ancestor and descendant are problematic) as well as render an appropriate loading indicator for long-running `refresh` calls, replacing the tree twistie. Once again, the `null` value represents the root.

Performance

In order to measure a meaningful performance improvement, we adopted the `ObjectTree` in the Problems panel. This was a great choice given its simple model (files have problems which have suggested fixes), potential model size (problems can easily grow to the thousands), model update patterns (problems can change very often as the user types, lots of frequent and large updates) as well as desired features (we want the tree to be filtered based on user input).

In order to get a consistent test case, we created a dummy extension which creates fake problems, enough to populate the tree and run a few operations on it. The following example is based on creating a total of 20.000 problems on 10.000 files, 2 problems each:

1. We first set that model on the tree: **Initial Population**;
2. We then use the filter box to filter down results yet end up typing an expression which matches **all** problems, thus causing no visible changes in the tree, apart from rendering the textual highlights for the matches: **Filter to Same**;
3. We then use the filter box to filter down results to a single result, by searching for a specific problem: **Filter to One**;
4. Finally, we run the **Collapse All** action to collapse all tree nodes, which was fairly problematic in the old tree performance wise.

Here are the results from profiling all these cases on the same machine along with some speedup calculations:

Test	Before	After	Speedup
Initial population	2.990 ms	524 ms	5.7x
Filter to Same	1400 ms	204 ms	6.7x
Filter to One	970 ms	78 ms	12.4x
Collapse All	30.000 ms 😱	625 ms	**48x** 😎

While the speedup results are impressive, some of the absolute values are still unnaceptable: 625ms for collapsing all tree nodes will eat up a large chunk of JavaScript time, drastically reducing our frame rate, even if collapsing all nodes is a punctual operation. The good news is that there is still lots of room for improvement, given that we now have a solid algorithm and data structure foundation on which can iterate much easier. Stay tuned!