Dry Exercise HW1, part 1

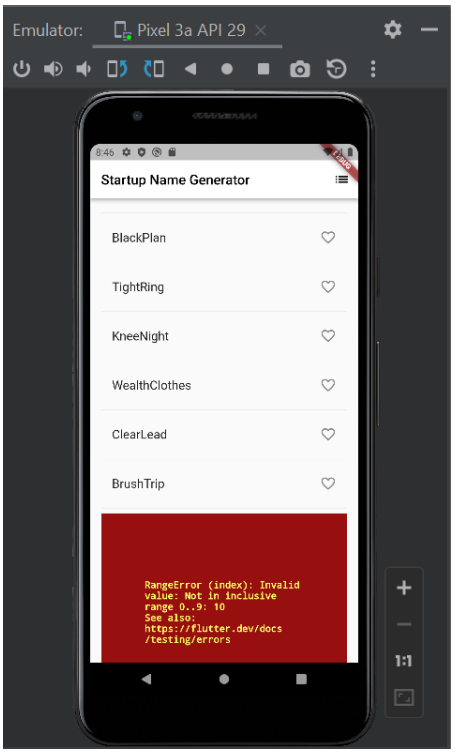
1. The 2 lines of code that make the list infinitely scrolling are:

```
// If you've reached the end of the available word pairings...
if (index >= _suggestions.length) {
  // ...then generate 10 more and add them to the suggestions list.
  _suggestions.addAll(generateWordPairs().take(10));
}
```

   If we remove these 2 lines, we'll get an error when scrolling to the end of the list (as can be seen in the picture. After initiating _suggestion with:

```
generateWordPairs().take(10).toList()
```

).



2. The ListView.seperated(..., `separatorBuilder: (BuildContext context, int index) => const Divider(),`) can be used to construct a list with dividers. Given that the list is finite, it's possible to use this method.
   I think that using ListView.seperated() is better, since, both methods give the same UI, both are building the list view items (and separators) on demand, but, the separated() is suitable for finite lists (as given in the question). This way, the separators are also built on demand, separately from the other children (items), so each item building on demand won't trigger a divider building on demand – more efficient. In addition, the code is simpler – no need to check the index to decide whether to trigger a divider building, passing the index is enough.
3. The setState() is needed in the onTap() handler because that's is how the app can render and the changed/updated UI can reflects. It calls the build() of the app (which will be optimized) so the widgets will appear (rendered and built on the platform)

with the updated changes according to the code logic. This is our way to notify the framework that the internal state of this object (list item = pair) has changed.

Dry Exercise HW1, part 2

1. The MaterialApp purpose is to wrap a number of widgets that are commonly required for material design applications. It provides the basic framework for a Material (Google UI-style) app. It includes the following properties:
    a. Theme - Default visual properties, like colors fonts and shapes, for this app's material widgets.
    b. Home - The widget for the default route of the app.
    c. Locale - The initial locale for this app's Localizations widget is based on this value. (aka the initial language for the app). If the 'locale' is null then the system's locale value is used.
2. The key property in general is used by the flutter element-tree when replacing/deleting/adding widgets of the same type in a list of widgets. When performing one of these actions, the skeleton tree (the element tree that is created (for each widget, an element is created)), need to be updated and the rendered. It is needed in stateful widgets. Specifically in the Dismissible widget, the key controls how one widget replaces another widget in the tree. If the runtime type and key properties of the two widgets are the same, then the new widget replaces the old widget by updating the underlying element. Otherwise, the old element is removed from the tree, the new widget is inflated into an element, and the new element is inserted into the tree. Without the key, it relays only on the type, which can be sometimes be the same, although the objects value is different (and we want to avoid this). It is required in Dismissible because usually the Dismissible is used on lists (like in our case, a ListView). Without keys, the default behavior is to sync widgets based on their index in the list, which means the item after the dismissed item would be synced with the state of the dismissed item. Using keys causes the widgets to sync according to their keys and avoids this pitfall.