This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Pokédex

Distribution Code

Download this project's distribution code (https://cdn.cs50.net/2019/fall/tracks/android/pokedex/pokedex.zip).

To open the distribution code, extract the ZIP, open Android Studio, select "Import project", and select the folder you extracted from the ZIP.

What To Do

- Searching
- Catching
- Saving State
- Sprites
- Description

Searching

Let's add some new functionality to our Pokédex app! First, let's give users the ability to search the Pokédex for their favorite Pokémon.

To start, we're going to use a built-in feature of `Adapter` called `Filterable`. This interface allows us to apply a filter to the data stored in our `Adapter`, which is exactly what we need! We'll filter out any Pokémon whose names don't match the search text.

First, make sure that the `adapter` variable in `MainActivity` has the type `PokedexAdapter`, like this:

```
private PokedexAdapter adapter;
```

We'll be calling methods that are specific to our `PokedexAdapter` that don't exist on the base `Adapter` class, so we need to use the `PokedexAdapter` type.

Next, open up the `PokedexAdapter` class. We can specify that our `PokedexAdapter` implements `Filterable` by changing the class declaration to:

```
public class PokedexAdapter extends RecyclerView.Adapter<PokedexAdapter.PokedexViewHolder> implements Filterable {
```

Recall that an interface is just a list of methods that any class can implement. Now that we've implemented `Filterable`, we can add a new method called `getFilter` to the `PokedexAdapter`.

```
@Override
public Filter getFilter() {
  return new PokemonFilter();
}
```

Of course, we don't have a class called `PokemonFilter` yet, so let's create one! We can create this class inside of `PokedexAdapter`, just as we did with `PokedexViewHolder`, like this:

```
private class PokemonFilter extends Filter {
  @Override
  protected FilterResults performFiltering(CharSequence constraint) {
    // implement your search here!
  }

  @Override
  protected void publishResults(CharSequence constraint, FilterResults results) {
  }
}
```

You can implement your search inside `performFiltering`. The argument to this method, `constraint`, will be whatever text the user has typed into the search bar, which you can use for your filter. The `performFiltering` method should return an instance of `FilterResults`. Here's an example:

```
@Override
protected FilterResults performFiltering(CharSequence constraint) {
  // implement your search here!
  FilterResults results = new FilterResults();
  results.values = filteredPokemon; // you need to create this variable!
  results.count = filteredPokemon.size();
  return results
}
```

The instance of `FilterResults` that you return from `performFiltering` will then be passed to `publishResults`. Inside of `publishResults`, you probably want to store the results of the search in another class variable, so you don't lose your copy of the list containing all Pokémon (i.e., the `pokemon` variable). Assuming you call this variable `List<Pokemon> filtered`, then your implementation of `publishResults` might look like this:

```
@Override
protected void publishResults(CharSequence constraint, FilterResults results) {
  filtered = (List<Pokemon>) results.values;
  notifyDataSetChanged();
}
```

Then, rather than using the `pokemon` variable inside of methods like `onBindViewHolder` and `getItemCount`, use your new `filtered` variable.

Now that the filtering logic is done, let's add a search bar above our `RecyclerView`. On the left-hand side of Android Studio, expand the `app` folder, and you should see a folder called `res`. Recall that this is where the XML files for our layouts are stored. Right click on `res`, then select New > Android Resource Directory. Enter `menu` for both `Directory name` and `Resource type`, then press `OK`. You should now see a new directory called `menu` underneath `res`.

Next, right click on that `menu` directory and select New > Menu resource file. Call this file `main_menu.xml` and then click `OK`. This new XML file will contain the layout for our menu. Paste the below into that file:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item android:id="@+id/action_search"
        android:title="Search"
        app:actionViewClass="androidx.appcompat.widget.SearchView"
        app:showAsAction="always" />
</menu>
```

As you can see, we're creating a new `menu` element with one `item` child. The `item` represents a search icon, that when pressed, will open up a `SearchView`.

Now, we can wire up that `SearchView` to our `MainActivity`. First, we need to make `MainActivity` implement an interface called `SearchView.OnQueryTextListener`. To tell Android that our main activity class implements `SearchView.OnQueryTextListener`, change the declaration of the class to the below:

```
public class MainActivity extends AppCompatActivity implements SearchView.OnQueryTextListener {
```

Next, to use the layout file we just created, we need to implement a method on our `MainActivity` called `onCreateOptionsMenu`.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main_menu, menu);
    MenuItem searchItem = menu.findItem(R.id.action_search);
    SearchView searchView = (SearchView) searchItem.getActionView();
    searchView.setOnQueryTextListener(this);

    return true;
}
```

As you'd guess, this method is called when an activity is creating a menu. Let's walk through this code line-by-line. First, we're specifying that this activity should use `R.menu.main_menu` , which is the name of the XML file we created. Then, we're grabbing a reference to the `item` inside our menu using its ID, `action_search` . Finally, we're calling `setOnQueryTextListener` on the `SearchView` in order to specify that our search code will be specified in our `MainActivity` class (which is what `this` references).

Now, our `SearchView` will automatically call methods on `MainActivity` when the user types text into the `SearchView` . Specifically, a method called `onQueryTextChange` will be called, and the argument passed to that method will be a `String` representing the current text of the `SearchView` . We then want to pass that along to the `PokemonFilter` we created earlier, like this, so our UI will update:

```
@Override
public boolean onQueryTextChange(String newText) {
    adapter.getFilter().filter(newText);
    return false;
}
```

Along the same line, a method called `onQueryTextSubmit` will be called when the user presses the "submit" button on the keyboard, which you can handle in the same way:

```
@Override
public boolean onQueryTextSubmit(String newText) {
    adapter.getFilter().filter(newText);
    return false;
}
```

At this point, everything should be wired up, so you can test out your new search functionality!

Catching

Any good Pokédex keeps track of which Pokémon have been caught and which haven't. Let's add that functionality to our Pokédex as well.

First, let's add a new `Button` to the `PokemonActivity` . Open up the layout XML file, and then add a new `<Button>` element. You can set the text of this button to whatever you'd like, but we'll go with `Catch` for simplicity.

To handle taps on the `Button` , we can use the attribute `android:onClick="toggleCatch"` . Add that to your `Button` , and then a method called `public void toggleCatch(View view)` will automatically be called whenever the user presses on the button.

Naturally, you'll want to add that method to your `PokemonActivity` , like this:

```
public void toggleCatch(View view) {
  // gotta catch 'em all!
}
```

Now, we can implement catching. To start, add a new boolean class variable that keeps track of whether or not the Pokémon is caught. If a Pokémon is caught, change the text of the button to something like `Release` , and vice-versa when it's released. The `Button` method `setText(String text)` method will come in handy.

Saving State

You'll notice that if you stop running your app and then run it again, your Pokédex will forget which Pokémon are caught and which aren't! Let's fix that by saving that state to disk.

As your last task, use the `SharedPreferences` class to save which Pokémon are caught. With this class, you can store state that will be remembered each time your app launches, which is just what you need. How you store this state is up to you—you might consider storing a list of all Pokémon that are caught, or you might consider using a map from Pokémon to boolean values.

Here's an example:

```
getPreferences(Context.MODE_PRIVATE).edit().putString("course", "cs50").commit();
String course = getPreferences(Context.MODE_PRIVATE).getString("course", "cs50");
// course is equal to "cs50"
```

To test saving state, you should be able to catch a Pokémon, stop the simulator, start the simulator again, and still see that Pokémon as caught.

Sprites

Every Pokémon aficionado has noticed by now that our Pokédex doesn't yet have arguably its most important feature: the ability to display what each Pokémon looks like! Luckily for us, the API we chose contains links to images for each Pokémon.

Let's add that functionality to our app. First, add a new `ImageView` to the layout for `PokemonActivity`. Give it a unique ID, and then create an `ImageView` class variable inside of `PokemonActivity`, and use `findViewById` to map that variable to your layout.

Next, when parsing the response from the API call, take a look at the key called `sprites`. You'll notice that it's a dictionary, and the key `front_default` contains a URL pointing to an image of a Pokémon. Use the value of that key to load in an image to your `ImageView`. You'll want to follow a similar pattern as before—use methods like `getJSONObject` and `getString` to parse the JSON strings into Java objects.

Once you have the URL of the image, you'll need to download it from the Internet. To do so, we'll use an Android built-in called `AsyncTask`. An `AsyncTask` executes some code in the background, so your app doesn't lock up as the image is downloading. To use an `AsyncTask`, create a new class that looks like this:

```
private class DownloadSpriteTask extends AsyncTask<String, Void, Bitmap> {
  @Override
  protected Bitmap doInBackground(String... strings) {
    try {
      URL url = new URL(strings[0]);
      return BitmapFactory.decodeStream(url.openStream());
    }
    catch (IOException e) {
      Log.e("cs50", "Download sprite error", e);
      return null;
    }
  }

  @Override
  protected void onPostExecute(Bitmap bitmap) {
    // load the bitmap into the ImageView!
  }
}
```

Let's walk through this. On the first line, we're specifying that our `AsyncTask` takes a `String` as input, and will return a `Bitmap`. That makes sense, since we'll be passing in a URL as a `String`, and we expect a `Bitmap` object, which represents an image, in exchange. The `doInBackground` method is where we'll put the logic to actually download an image. You'll notice that this method actually takes an array of strings, but we only need to download one, so we're just taking the first element in that array with `strings[0]`.

After `doInBackground` completes, the method called `onPostExecute` will be called. The `Bitmap` argument that's passed in represents a loaded image, so load that into your `ImageView` using the method `setImageBitmap`.

Finally, you can use this new class to trigger a download of a string URL with:

```
new DownloadSpriteTask().execute(url); // you need to get the url!
```

You can test your code by selecting Pokémon from the list, and you should see images in the `ImageView`!

Description

Let's add one last feature to our Pokédex: a description of each Pokémon. From the API documentation, we can see that we can use /api/v2/pokemon-species/{id} to retrieve a description for a given Pokémon: https://pokeapi.co/docs/v2.html#pokemon-species (https://pokeapi.co/docs/v2.html#pokemon-species). For instance, the URL `https://pokeapi.co/api/v2/pokemon-species/133/` will give you the description text for everyone's favorite Pokémon.

Specifically, what we're looking for can be found in the key called `flavor_text_entries`. This key happens to contain entries for several different languages, but we're just concerned with English for now. You might need a few additional structs to model the data for these new keys.

After a user selects a Pokémon from the list, make a separate API call to this second endpoint to retrieve the description of the selected Pokémon. Filter for just the first English description, and then display it somewhere on the screen. (Some Pokémon have more than one English description, and it suffices to just display the first one.) You'll probably want to wire up a new `TextView` to display this final piece of data.

You should see a few sentences about each Pokémon after selecting it from the list!

How to Submit

To submit your code with `submit50`, you may either: (1) upload your code to CS50 IDE and run `submit50` from inside of your IDE, or (2) install `submit50` on your own computer by running `pip3 install submit50` (assuming you have Python 3 (https://www.python.org/downloads/) installed).

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/tracks/android/pokedex
```