4C Forms

Getting ready for a robust form

1. In preparation for submitting a purchase, let's create an object to represent this purchase of tickets. Add this as property of the class:

```
Map<String, dynamic> _purchase = {
  "seats": [1, 2, 3],
  "showing_id": 1,
};
```

Later, we'll allow the ability for the user to add seats to this purchase object but for now we're hardcoding something to submit. Of course, when we're submitting a purchase from the server, we'll need to provide it with a name, email/phone, and credit card info which we can get from the form.

2. Edit the _checkout() method. Print(_purchase) so that when the user presses the FAB, we'll see the contents of the purchase in the debug console.

3. Give it a test to make sure you can see the _purchase object and we'll get on with Forms() ...

We're reading data from the user which is the most important part of the activity. But these fields are unaware of each other and can't be handled as a group. We have some opportunity for improvements:

- Control initial field values
- Make sure the data is valid
- Submit it to a server as a group
- Update/setState on a field basis rather than the whole widget

To handle all of these, we need a Form() widget.

4. Surround all of your fields with a Form() widget. It can be anywhere around the fields, so it may be convenient to put it around the Column. You can decide how you want to do it.

Saving the data

Remember, to save a Form you need a key.

5. Create a GlobalKey<FormState> called *_key* and assign it to the Form. It's of type FormState so that it'll have the needed .validate() and .save() methods.

The fields will need validator, and onSaved properties. TextFields and DropdownButtons don't have them. We need to wrap them with FormField() widgets. Or better yet, \ convert them to TextFormField()s and DropdownButtonFormField()s respectively.

6. Convert the DropdownButtons to DropdownButtonFormFields.

7. Convert the TextFields to TextFormFields.

8. In _checkout, you'll call the key.currentState's save() method.

Remember that this will fire the onSaved event handlers in every field inside the Form.

9. Since onSaved is being fired, we should write an onSaved() event for each field. Each onSaved will receive a value. Go ahead and add that to the _purchase Map. Here's an example for the firstName field:

```
onSaved: (val) => _purchase["firstName"] = val,
```

10. Write an onSaved for each field like the above.

11. Run and test. Run through the debugger and/or add some print statements to make sure you're populating the _purchase object properly.

Validating the data

We're now getting data from the user that we can send to the API server. That's very cool. But if you put in nonsense data, it still submits. Shouldn't we make sure we send good data to the server? Let's validate the data as soon as the user enters it.

12. Put in validators. Pick one or two of these business requirements and implement them:
    - First name, last name, and credit card number are required.
    - Email OR cell are required
    - Credit card number looks like a credit card number.
    - Month and Year must be in the future.
    - CVV is three digits

    Hints:
    - Each Form Field can have its own validate().
    - A validate property must be a function that receives a val and returns ...
        - A null if everything is okay
        - A String with the error message if it isn't valid.
    - Regular expressions come in really handy when you want to make sure strings match a pattern. Dart handles them with the RegExp class.

13. Look in the _checkout function. Add this line to it:

```
if (!_key.currentState!.validate()) return;
```

14. Bonus!! If you want your validation to happen as the user is entering data, add this to your form:

```
autovalidateMode: AutovalidateMode.onUserInteraction,
```