Criterion C

Word: 1063

Technique Used
A. IF & ELSE Function
B. TRY & CATCH
C. NESTED FOR Loop
D. String manipulation
E. Streams
F. Providers
G. Models
H. Form & TextFormField
I. GUI
J. 2D arrays

I have utilised Flutter to develop this program which uses Dart as a programming language. Databases are created through the Google cloud platform, Firebase, which was connected to the program through pubspec.yaml, gradle and podfile extensions and drivers.

Classes Used

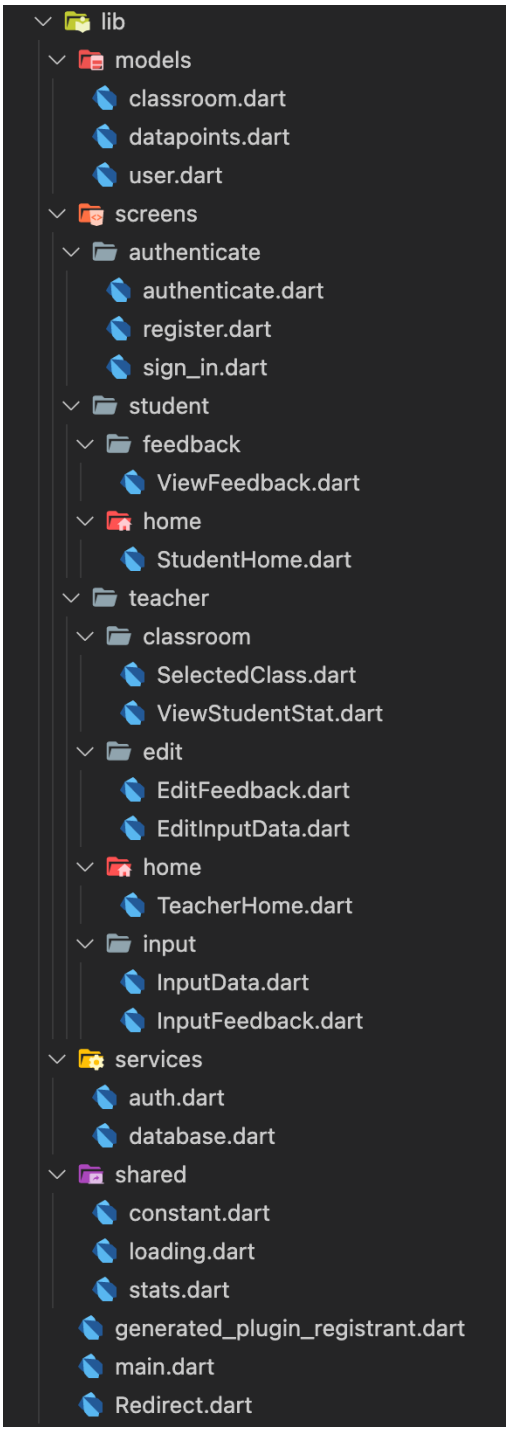

Figure 1. *Screenshot of all classes organised in separate folders but everything is stored in lib*

There are more classes than planned in criteria B as these additional classes made the ***add, delete, display,*** function easier to develop as organising similar methods together under one class such as methods to fetch data, makes it less difficult for the programmer to navigate and find the necessary methods throughout the program.

Use of Additional Dependencies

```
dependencies:
  flutter:
    sdk: flutter

  firebase_auth: ^3.0.1
  cloud_firestore: ^2.4.0
  provider: ^5.0.0
  google_fonts: ^2.1.0
  flutter_spinkit: ^5.0.0
  intl: ^0.17.0
  syncfusion_flutter_charts: ^19.2.62
```

Figure 2. *Screenshot of all the external dependencies used in this program*

Name	**What is it**	**What is it used for**	**Success Criteria**
Firebase Auth (firebase_auth: ^3.0.1)	Database software which provides register and sign-in validation and storing user's account	• Validate user's emails when signing in to see whether they exist or not in the current database • Register new users	1 & 2
Cloud Firestore (cloud_firestore: ^2.4.0)	Also a database software but this time, instead of storing user's account, this stores relevant data of user's input within the software such as student stats	• Creating necessary databases without knowing the complete code to Firebase • Storing user's data input to allow the program to function properly - analysing data • Viewing some records in the database allows easier testing to see whether data was added or deleted properly	3
Provider (provider: ^5.0.0)	A dependency that allows children of parents class to obtain streams of data from Cloud Firestore where the parent is the initializer of the stream	• For children of parents class to listen to any data changes within the database and update their variables accordingly to update the display • For all inherited classes to access the stream	All
Google Fonts (google_fonts: ^2.1.0)	A dependency that allows the use of a variety of fonts within the program	• To change fonts within the program to "Roboto" which makes it more appealing to the user	N/A
Flutter Spinkit (flutter_spinkit: ^5.0.0)	A dependency that is able to create an animated loading	• To inform the user to wait as data is being processed or retrieved for their needs	N/A

	screen		
Intl: ^0.17.0	A dependency that allows formatting of the date class types - DateTime	• For efficient storage of date data by omitting unnecessary values such as hours and minutes that are included within the date class • Makes viewing date class data easier for programmers	7
Syncfusion Flutter Charts (syncfusion_flutter_charts: ^19.2.62)	A dependency that allows graphs to be displayed from the user's input	• To allow users to comprehend the data they inputted relative to time to see improvements in student's stats	7

Models

'cUser' model:

```
class cUser {
  final String? uid;
  cUser({this.uid});
}
```

Figure 3.
This model establishes the user's unique id which is used to identify which user is currently using the program

'TeacherData' model:

```
class TeacherData {
  final String? name;
  var classid;
  final bool? teacher;
  final int? numofclass;
  TeacherData({
    this.name,
    this.classid,
    this.teacher,
    this.numofclass,
  });
}
```

Figure 4.
This model establishes the teacher's information used to display classroom list and teacher's name

'StudentData' model:

```
class StudentData {
  final String? name;
  var feedback;
  var accList;
  var effectList;
  var speedList;
  var distanceList;
  final double? accuracy;
  final double? speed;
  final double? effect;
  final double? distance;
  var datainput;
  var classid;
  final bool? teacher;
  StudentData({
    this.name,
    this.feedback,
    this.accList,
    this.effectList,
    this.speedList,
    this.distanceList,
    this.accuracy,
    this.speed,
    this.effect,
    this.distance,
    this.datainput,
    this.classid,
    this.teacher,
  });
}
```

Figure 5.
This model establishes the student's information which includes their name, feedback given, accuracy list, effectiveness list, speed list, distance list, average accuracy, average speed, average effectiveness, average distance, list of data inputs and list of class ids they are in

'UserRole' model:

```
class UserRole {
  final bool? teacher;
  UserRole({this.teacher});
}
```

Figure 6.
This model helps identify whether user is a teacher or a student so that user can be redirected to the correct screen according to their roles

Streams in DatabaseService() Class

```
Stream<UserRole?> get userrole {
  return users.snapshots().map(_userRole);
}
```

Figure 7.
Streams data of user's role - whether they are a teacher or a student

```
Stream<QuerySnapshot> get classroomdata {
  return classCollection.snapshots();
}
```

Figure 8.
Streams data of all classrooms that is available in the program

```
Stream<TeacherData> get teacherdata {
  return users.snapshots().map(_teacherDataFromSnapshot);
}
```

Figure 9.
Streams data of the teacher currently using the program

```
Stream<StudentData> get studentdata {
  return users.snapshots().map(_studentDataFromSnapshot);
}
```

Figure 10.
Streams data of the student currently using the program

Firebase Connection

Class: DatabaseService()

- Class that contains methods that converts, add, delete, and read data from the database "CS T-ball Program"

```
class DatabaseService {
  final String? uid;
  final String? ID;
  DatabaseService({this.uid, this.ID});

  late final DocumentReference users =
      FirebaseFirestore.instance.collection('users').doc(uid);

  final CollectionReference collectionUsers =
      FirebaseFirestore.instance.collection('users');

  final CollectionReference classCollection =
      FirebaseFirestore.instance.collection('classrooms');
```

Figure 11. *Top section of the DatabaseService*

How:

- The Firebase Firstore dependency was imported
- "FirebaseFirestore" class can then be utilized to get data type DocumentReference and CollectionReference
- ".collection" is a method that gets data type CollectionReference and ".doc" afterwards is to get data type DocumentReference
- DocumentReference data is a subset of CollectionReference data - CollectionReference data contains many DocumentReference data
- Data from "CS T-ball Program" is then stored into the 3 variables

Why:

- To allow methods within the DatabaseService class to add, delete, and read data

main.dart - Firebase initializer, Routes, and Stream Providers

Function: main()

- Initializes everything important to the program such as the starting class, MyApp(), the Firebase database, and the glue that binds the VSCode platform to the Flutter engine

```dart
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

Figure 12. *main() method from VsCode*

How:

- The Firebase class was called from the imports and the initializeApp method starts up all other methods in the Firebase class. runApp function is called to redirect program to starting class MyApp

Class: MyApp()

- Streams user's login status to its children class, Wrapper(), and provides routes to its children classes to later allow users to navigate in the sign-in or register home page

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return StreamProvider<cUser?>.value(
      initialData: null,
      value: AuthService().loginstatus, // stream you want to listen to
      child: MaterialApp(
        debugShowCheckedModeBanner: false,
        home: Wrapper(),
        routes: {
          '/sign_in': (context) => SignIn(),
          '/register': (context) => Register(),
        },
      ), // MaterialApp
    ); // StreamProvider.value
  }
}
```

Figure 13. *MyApp class inherits StatelessWidget class.*

How/Why:

- "***StreamProvider***<*cUser?*>*.value(...)*" provides streams of modeled user data, cUser, to its children class, Wrapper(), where it will check if a user is logged in or not.
- Parameter "routes" in MaterialApp class contains a map with a key and value pair in order to be used later through *Navigator* class to allow users to access different pages in the children class. "=>" means **return** in dart

Wrapper Class

- This class decides whether to redirect users to the home or teacher/student home screen based on whether they are logged in or not. It will also redirect users to the correct screen based on their roles.

```
import 'package:cstballprogram/screens/teacher/home/teacherhome.dart';
import 'package:cstballprogram/services/database.dart';
import 'package:cstballprogram/shared/loading.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
```

Figure 14. *Screenshot of the imported libraries for wrapper.dart*

```
final userloginuid = Provider.of<cUser?>(context);
```

Figure 15. *Screenshot of the variable that stores streams of data pertaining to the user's information unique id*

```
if (userloginuid == null) {
  return Authenticate();
} else {
  return StreamBuilder<UserRole?>(
      stream: DatabaseService(uid: userloginuid.uid.toString()).userrole,
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
          return Loading();
        }
        if (snapshot.data?.teacher == true) {
          return Home();
        }
        if (snapshot.data?.teacher == false) {
          return SHome();
        }
        return Loading();
      }); // StreamBuilder
}
```

Figure 16. *Screenshot of if function in Wrapper class*

How/Why:

- "*Wrapper()*" class **inherits** the stream of data provided in its parents class "*main()*"
- The **IF function** checks whether the user is logged in or not by checking whether the variable "*userloginuid*" is null or not. If it is null, then no data is collected from the stream which meant that no user is currently logged in.
- **Nested IF function** is then used to redirect the user to the correct home screen based on their roles

Authentication/Sign In/Register Screen

Class: Authenticate()

- Home page which helps users to navigate to sign in and register page

```
import 'package:flutter/material.dart';
import 'package:google_fonts/google_fonts.dart';
```

Figure 17. *Screenshot of imported libraries for authenticate.dart*

```
child: TextButton(
  style: ButtonStyle(
    backgroundColor:
        MaterialStateProperty.all<Color>(Colors.amber),
    shape: MaterialStateProperty.all<RoundedRectangleBorder>(
        RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(18.0),
    )), // RoundedRectangleBorder
  ), // ButtonStyle
  onPressed: () {
    setState(() {
      Navigator.pushNamed(context, '/sign_in');
    });
  },
  child: Text(
    "Sign In",
    style: GoogleFonts.lato(
```

Figure 18. *Widget for sign-in button*

How:

- Using the "*TextButton()*" widget with parameters such as "*backgroundColor, shape"* to format the button and parameters such as "*onPressed*" to initiate function to navigate to sign-in page or registration page

Class: SignIn()

- Checks if user's input of username and password matches with the ones stored on the database

```
import 'package:cstballprogram/services/auth.dart';
import 'package:cstballprogram/shared/constant.dart';
import 'package:cstballprogram/shared/loading.dart';
import 'package:flutter/material.dart';
import 'package:google_fonts/google_fonts.dart';
```

Figure 19. *Screenshot of imported libraries for sign_in.dart*

```
final AuthService _cAuth = AuthService();
```

Figure 20. *Variable to store class "AuthService()" to call signInEmailandPassword later on*

```
final _formkey = GlobalKey<FormState>();
```

Figure 21. *Variable to store multiple inputs from user (email and password)*

```
child: Form(
  key: _formkey,
  child: Column(
    children: [
      SizedBox(height: 20),
      TextFormField(
        style: TextStyle(color: Colors.amber),
        decoration:
            textInputDecoration.copyWith(hintText: 'Email'),
        validator: (val) =>
            val!.isEmpty ? 'Enter an email' : null,
        onChanged: (val) {
          email = val;
        },
      ), // TextFormField
```

Figure 22. *Widget for textbox to input email addresses*

```
ElevatedButton(
  style: TextButton.styleFrom(
    backgroundColor: Colors.amber,
  ),
  onPressed: () async {
    if (_formkey.currentState!.validate()) {
      setState(() {
        loading = true;
      });
      dynamic result = await _cAuth.signInEmailAndPassword(
          email, password);
      if (result == null) {
        setState(() {
          error = 'Could not Sign In';
          loading = false;
        });
      } else {
        setState(() {
          Navigator.pop(context);
        });
      }
    }
  },
  child: Text(
    'Sign in',
    style: TextStyle(color: Colors.grey[900]),
  ), // Text
), // ElevatedButton
```

Figure 23. *Widget for button to sign-in*

```dart
//sign in with email and pass
Future signInEmailAndPassword(String email, String password) async {
  try {
    UserCredential result = await _fAuth.signInWithEmailAndPassword(
        email: email.trim(), password: password.trim());
    User user = result.user as User;
    return _userFromFirebaseUser(user);
  } catch (e) {
    print(e.toString());
    return null;
  }
}
```

Figure 24. *Method to fetch user's information from database if the email address and password matches*

How/Why:

- Using the "***Form()***" widget to group together inputs from multiple "***TextFormField()***" widgets which is stored in private variable "*_formkey*"
- **IF function** to check whether the text boxes were left empty using
- **IF function** is also then used to check if variable "*result"* is null or not and if it is then that meant an invalid email or password
- To assign values to dynamic variable "*result*", "*signInEmailAndPassword*" method is called where it uses a **try and catch function** to prevent the program from crashing if "*_fAuth.signInWithEmailAndPassword(...)*" were unable to fetch any results from the database
- **Return** "*null*" if email and password is invalid or "*_userFromFirebaseUser(user)*" if it is valid

Teacher's Home Screen

Class: ClassroomList()

- To display a list of classrooms owned by the teacher

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:cstballprogram/models/user.dart';
import 'package:cstballprogram/screens/teacher/home/classroom_list.dart';
import 'package:cstballprogram/services/auth.dart';
import 'package:flutter/material.dart';
import 'package:google_fonts/google_fonts.dart';
import 'package:cstballprogram/services/database.dart';
import 'package:provider/provider.dart';
```

Figure 25. *Screenshot of imported libraries for teacherhome.dart*

```
if (classroomdatadoc != null) {
  for (var x in classroomdatadoc) {
    for (var y = 0; y < classidlist.length; y++) {
      if (x.get('id') == classidlist[y]) {
        /* if teacher's classroom id ("classidlist[y]") matches with
        a classroom id stored in the database ("x.get('id')") then add to array "class2dlist"*/
        class2dlist.add([
          x.get('classname'),
          classidlist[y]
        ]); //class2dlist: [[class name, class id], [..., ...]]
        var studentidlist = x.get(
            'studentid'); // "studentidlist" contains the id of students that are within the classroom
        for (var z in studentidlist) {
          //converting from an array of maps to an array of arrays for ease of access
          student[index].add([
            z.keys.toString().substring(1, z.keys.toString().length - 1),
            z.values.toString().substring(1, z.values.toString().length - 1)
          ]);
        }
        index++; //to move onto the next "slot" everytime student data is added to dynamic multi-dimensional array "student",
      }
    }
  }
}
```

Figure 26. *Function to filter out classrooms that don't belong to the teacher*

How/Why:

- **Nested for loop** to compare the teacher's class's unique id to all of the classroom id stored on the database. This is to then display a list of classrooms that the teacher owns.
- "*.substring()*" method is then used to perform **string manipulation** where the character at the start of the string and the end string are removed. This is to remove the brackets when the keys and values are obtained from the map "*studentidlist*"

```
child: FloatingActionButton(
  foregroundColor: Colors.grey[800],
  backgroundColor: Colors.amber,
  onPressed: () async {
    await DatabaseService(uid: userloginuid?.uid)
        .addTNewClass(ID, numofclass!);
    await DatabaseService().createClassroomData(
        ID, [], "New Classroom $numofclass", name!);
  },
  child: Icon(
    Icons.add,
  ), // Icon
), // FloatingActionButton
```

Figure 27. *Button to add a new classroom for the teacher*

```
Future addTNewClass(String id, int numofclass) async {
  return await users.update({
    'classid': FieldValue.arrayUnion([id]),
    'numofclass': FieldValue.increment(1),
  });
}
```

Figure 28. *Method that adds the new classroom's id to the teacher's classroom id list*

```
Future createClassroomData(
    String id, var studentid, String classname, String teachername) async {
  return await classCollection.doc(id).set({
    'teachername': teachername,
    'id': id,
    'classname': classname,
    'studentid': studentid,
  });
}
```

Figure 29. *Method to create new classroom document in classroom collection*

```
ListView.builder(
    itemCount: classidlist.length,
    itemBuilder: (BuildContext context, int index) {
      return Padding(
        padding: EdgeInsets.only(top: 8.0),
        child: GestureDetector(
          child: Card(
            margin: EdgeInsets.fromLTRB(20.0, 6.0, 20.0, 0.0),
            child: ListTile(
              leading: CircleAvatar(
                radius: 25.0,
                child: Icon(Icons.sports_baseball),
              ), // CircleAvatar
              title: Text('${class2dlist[index][0]}'),
              subtitle: Text("ID: ${class2dlist[index][1]}"),
            ), // ListTile
          ), // Card
          onTap: () async {
            Navigator.of(context).push(MaterialPageRoute(
                builder: (context) => TeacherClassroom(
                      name: class2dlist[index][0],
                      id: class2dlist[index][1],
                      student: student[index],
                    ))); // TeacherClassroom // MaterialPageRoute
          },
        ), // GestureDetector
      ); // Padding
    }), // ListView.builder
```

Figure 30. *"Listview.builder()" widget is used to form a list of classrooms and "GestureDetector()" widget is used so that the teacher can be redirected to the classroom's page with a list of students a part of that class.*

Teacher's Selected Classroom Screen

Class: TeacherClassroom()

- Displays list of students within a classroom

```
import 'package:cstballprogram/screens/teacher/classroom/teacherstudentstats.dart';
import 'package:flutter/material.dart';
```

Figure 31. *Screenshot of imported libraries for teacherclassroom.dart*

```
body: ListView.builder(
    itemCount: widget.student.length,
    itemBuilder: (context, index) {
      return Padding(
        padding: EdgeInsets.only(top: 8.0),
        child: GestureDetector(
          child: Card(
            margin: EdgeInsets.fromLTRB(20.0, 6.0, 20.0, 0.0),
            child: ListTile(
              leading: CircleAvatar(
                radius: 25.0,
                child: Icon(Icons.sports_baseball),
              ), // CircleAvatar
              title: Text('${widget.student[index][1]}'),
            ), // ListTile
          ), // Card
          onTap: () {
            Navigator.of(context).push(MaterialPageRoute(
                builder: (context) => TeacherStudentStat(
                    studentname: '${widget.student[index][1]}',
                    studentid: '${widget.student[index][0]}'))); /
          },
        ), // GestureDetector
      ); // Padding
    }), // ListView.builder
```

Figure 32. *Code to display a list of classrooms is similar to code for displaying a list of students*

Teacher View Student's Statistics Screen

Class: TeacherStudentStat()

- Displays a dropdown list for the teacher to either input statistics for students or give student feedback

```dart
import 'package:cstballprogram/models/user.dart';
import 'package:cstballprogram/screens/teacher/input/teacherfeedback.dart';
import 'package:cstballprogram/screens/teacher/input/teacherinputstudentdata.dart';
import 'package:cstballprogram/services/database.dart';
import 'package:cstballprogram/shared/stats.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
```

Figure 33. *Screenshot of imported libraries for teacherstudentstats.dart*

```dart
child: DropdownButton(
  dropdownColor: Colors.grey[800],
  underline: Container(
    color: Colors.transparent,
  ), // Container
  onChanged: (String? pageinput) {
    if (pageinput == 'Input Data') {
      Navigator.push(
          context,
          MaterialPageRoute(
              builder: (context) =>
                  StreamProvider<StudentData?>.value(
                      value:
                          DatabaseService(uid: widget.studentid)
                              .studentdata,
                      initialData: null,
                      catchError: (context, error) => null,
                      child: TeacherInputStudentData(
                        studentid: widget.studentid,
                      )))); // TeacherInputStudentData // Stream
    }
    if (pageinput == 'Feedback') {
      Navigator.push(
          context,
          MaterialPageRoute(
              builder: (context) => TeacherFeedback(
                    studentid: widget.studentid,
                  ))); // TeacherFeedback // MaterialPageRoute
    }
  },
  items: ['Input Data', 'Feedback'].map((String value) {
    return DropdownMenuItem(
      value: value,
      child: Text(value),
    ); // DropdownMenuItem
  }).toList(),
  icon: Icon(
    Icons.add_circle,
    color: Colors.amber,
    size: 35.0,
  ), // Icon
  style: TextStyle(color: Colors.amber),
)) // DropdownButton // Padding
```

Figure 34. *"Dropdownbutton()" widget to create a drop down list GUI for the teacher*

Class: Stats()

- Displays the statistics of students such as their average accuracy, average effectiveness, average distance, and average speed for the past 30, 60, 90, 180, 365, or all days

```
//Filter Function
List filter(int selectedperiod, data) {
  var filter = [];
  try {
    DateTime latestdate = DateTime.parse(data[0]
        .keys
        .toString()
        .substring(1, data.last.keys.toString().length - 1));
    //If Selected Period Is Not "All-Time" Then The FOR LOOP Will Proceed To Filter Out The Data
    if (selectedperiod != 1000) {
      for (var z = 0; z < data.length; z++) {
        if (latestdate
                .difference(DateTime.parse(data[z]
                    .keys
                    .toString()
                    .substring(1, data[z].keys.toString().length - 1)))
                .inDays
                .abs() <=
            selectedperiod) {
          filter.add(data[z]);
        }
      }
      return filter;
    } else {
      return data;
    }
  } catch (e) {
    return data;
  }
}
```

Figure 35. *"filter" method allows student's statistics data that is not in the past 30, 60, 90, 180, or 365 days to be excluded but if the option is all days then the entire data would just be returned*

How/Why:

- **IF** the selected period from the teacher is not "all days" then a **FOR loop** will be executed **ELSE** the original data will be returned. This is to prevent unnecessary operations which makes the code more efficient.
- **FOR loop** will loop for "data length" amount of times and an **IF function** within will check whether the difference between the "*latestdate*" and the current date will be smaller than the selected period in days using the "*.difference()*" and "*.inDays*" method. This is to exclude any dates that is not within the selected period.
- **ELSE** if the difference between the "*latestdate*" and the current date is greater than the selected period then the loop will **break**. This is to prevent unecessary operations because the dates are sorted in order, making the code more efficient.

```
//Averages The Data
double average(data) {
  double total = 0.0;
  int n = 0;
  for (var z = 0; z < data.length; z++) {
    double value = double.parse(data[z]
        .values
        .toString()
        .substring(1, data[z].values.toString().length - 1));
    //If The Speed Value Is Not "N/A" Which Is Equivalent To Saying "-314", Then "total" Will Be Added As Well As "n"
    if (value != -314) {
      total += value;
      n++;
    }
  }
  //Returns The Average
  return (total / n);
}
```

Figure 36. *"average" method allows the program to find the new average of accuracy, effectiveness, speed, and distance based on the selected period*

How/Why:

- **IF** function is used within the **FOR loop** to see whether value is equal to -314. This is the value I assigned for N/A speeds. This is so that N/A speeds are not included in the average speed.

```dart
//Converting Data Into A Data Point Model To Be Displayed On The Graph
List<StatsData>? convertToStatData(data) {
  //Contains The Converted Version Of The Data
  List<StatsData> statdatalist = [];

  //Contains A List Of Non-Overlapping Dates From The Data
  List<DateTime> recordeddates = [];

  //FOR LOOP To Loop Through The Dates Of The Data
  for (var x in data) {
    DateTime date = DateTime.parse(
        x.keys.toString().substring(1, x.keys.toString().length - 1));

    //IF Function To Prevent Data Point Inputted On The Same Day To Overlap
    if (!recordeddates.contains(date)) {
      recordeddates.add(date);
      double total = 0.0;
      int n = 0;

      //FOR LOOP To Find Overlapping Dates And Average Them
      for (var z in data) {
        DateTime compare = DateTime.parse(
            z.keys.toString().substring(1, z.keys.toString().length - 1));
        if (date == compare &&
            double.parse(z.values
                    .toString()
                    .substring(1, z.values.toString().length - 1)) !=
                -314) {
          total += double.parse(z.values
              .toString()
              .substring(1, z.values.toString().length - 1));
          n++;
        }
      }
      statdatalist
          .add(StatsData(date, double.parse((total / n).toStringAsFixed(1))));
    }
  }
  return statdatalist;
}
```

Figure 37. *"convertToStatData" is a method to convert the data into the "StatsData" model which is later used as a data point to plot on the graph*

How/Why:

- Any data that has the same input date will have to be merged together as one data point. To do this the average must be obtained between the similar points. **Nested FOR loop** and **IF functions** are utilised to do so. This is to prevent faulty looking graphs where there may be more than 1 point on the same x-axis label.

```
void getData(int days) {
  for (var x in filter(days, datainput)) {
    balllocation.add(x.values
        .toString()
        .substring(2, x.values.toString().length - 2)
        .split(','));
  }
  avgaccuracy =
      double.parse((average(filter(days, provacc))).toStringAsFixed(2));
  avgspeed =
      double.parse((average(filter(days, provspeed))).toStringAsFixed(1));
  avgdistance = double.parse(
      (average(filter(days, provdistance))).toStringAsFixed(1));
  avgeffect =
      double.parse((average(filter(days, proveffect))).toStringAsFixed(1));
  graphacc = convertToStatData(filter(days, provacc));
  graphspeed = convertToStatData(filter(days, provspeed));
  grapheffect = convertToStatData(filter(days, proveffect));
}
```

Figure 38. *"getData" method utilises all of the methods above to assign data to variables*

```
if (studentdata == null) {
  return Loading();
} else {
  if (selectedperiod == '30-days Average') {
    getData(30);
  }
  if (selectedperiod == '60-days Average') {
    getData(60);
  }
  if (selectedperiod == '90-days Average') {
    getData(90);
  }
  if (selectedperiod == '180-days Average') {
    getData(180);
  }
  if (selectedperiod == '365-days Average') {
    getData(365);
  }
  if (selectedperiod == 'All-time Average') {
    getData(1000);
  }
```

Figure 39. *IF function to determine what values should be assigned to variables based on the teacher's selected period*

Teacher Input Student's Statistics Screen

Class: TeacherInputStudentData()

- To allow teacher to input data for a specific student

```
import 'package:cstballprogram/models/user.dart';
import 'package:cstballprogram/services/database.dart';
import 'package:cstballprogram/shared/constant.dart';
import 'package:cstballprogram/shared/loading.dart';
import 'package:flutter/material.dart';
import 'dart:math';
import 'package:intl/intl.dart';
import 'package:provider/provider.dart';
```

Figure 40. *Screenshot of imported libraries in teacherinputstudentdata.dart*

```
child: GestureDetector(
  onTapDown: (TapDownDetails details) =>
      onTapDown(context, details),
  onDoubleTap: () => onDoubleTap(),
```

Figure 41. *Screenshot of "GestureDetector()" widget to utilize its parameters "onTapDown" and "onDoubleTap" in order to use "onTapDown" method and "onDoubleTap" method*

```
void onTapDown(BuildContext context, TapDownDetails details) {
  setState(() {
    posx = details.localPosition.dx;
    posy = details.localPosition.dy;
    distance = double.parse(sqrt(
            pow(((379 - posx) * (1 / 4)), 2) + pow((512 - posy) * (1 / 4), 2))
        .toStringAsFixed(1));
    changecolor = true;
  });
}
```

Figure 42. *Screenshot of "onTapDown" method which is used to assign user's tap position in the form of x and y coordinates to the variable "posx" and "posy" which is then used to find the distance of the ball travelling*

```
void onDoubleTap() {
  setState(() {
    posx = 100.0;
    posy = 100.0;
    distance = 0.0;
    changecolor = false;
  });
}
```

Figure 43. *Screenshot of "onDoubleTap" which a method used to remove the previous tap of the teacher*

```
Positioned(
  child: Container(
    height: 50,
    width: 100,
    child: Column(
      children: [
        Text(
          "$distance ft",
          style: TextStyle(
              color: changecolor == false
                  ? Colors.green
                  : Colors.black,
              fontSize: 13.0), // TextStyle
        ), // Text
        SizedBox(height: 5.0),
        ClipOval(
          child: Container(
            height: 20,
            width: 20,
            color: changecolor == false
                ? Colors.green
                : Colors.red,
          ), // Container
        ), // ClipOval
      ],
    ), // Column
  ), // Container
  left: posx - 50,
  top: posy - 28,
), // Positioned
```

Figure 44. *Screenshot of "Positioned()" widget which is used to display the red dot of where the teacher tapped*

No. of strikes text field

```
child: TextFormField(
    style: TextStyle(color: Colors.amber),
    decoration: textInputDecoration.copyWith(
        hintText: 'No.of Strikes'),
    onChanged: (val) {
      try {
        int.parse(val);
        setState(() {
          noofstrike = int.parse(val);
          if (noofstrike == 3) {
            seconds = 0.0;
            noofbaseran = 0;
            caughtout = 'N';
          }
        });
      } catch (e) {
        setState(() {
          noofstrike = 0;
        });
      }
    },
    validator: (val) {
      if (val!.isNotEmpty) {
        try {
          noofstrike = int.parse(val);
          if (noofstrike < 0 ||
              noofstrike > 3) {
            return 'Please Enter a No. from 0-3';
          }
        } catch (e) {
          return 'Please Input an Integer!';
        }
      } else {
        return "Please Enter a Value!";
      }
    }), // TextFormField
```

Figure 45.

Time text field

```
child: TextFormField(
    readOnly: noofstrike == 3 ? true : false,
    style: TextStyle(color: Colors.amber),
    decoration: textInputDecoration.copyWith(
        hintText: noofstrike == 3
            ? 'N/A'
            : 'Time in seconds',
        hintStyle: noofstrike == 3
            ? TextStyle(color: Colors.amber)
            : TextStyle(
                color: Color(0xFFBDBDBD))), // TextStyle
    onChanged: (val) {
      try {
        double.parse(val);
        seconds = double.parse(val);
      } catch (e) {}
    },
    validator: (val) {
      if (noofstrike != 3) {
        if (val!.isNotEmpty) {
          try {
            int.parse(val);
          } catch (e) {
            return 'Please Input an Integer!';
          }
        } else {
          return "Please Enter a Value!";
        }
      }
    }), // TextFormField
```

Figure 46. *If number of strike is 3, time in seconds is N/A*

No. of base ran text field

```
child: TextFormField(
    readOnly: noofstrike == 3 ? true : false,
    style: TextStyle(color: Colors.amber),
    decoration: textInputDecoration.copyWith(
        hintText: noofstrike == 3
            ? '0'
            : 'No.of Base Ran',
        hintStyle: noofstrike == 3
            ? TextStyle(color: Colors.amber)
            : TextStyle(
                color: Color(0xFFBDBDBD))), // TextStyle
    onChanged: (val) {
      try {
        int.parse(val);
        noofbaseran = int.parse(val);
      } catch (e) {}
    },
    validator: (val) {
      if (noofstrike != 3) {
        if (val!.isNotEmpty) {
          try {
            noofbaseran = int.parse(val);
            if (noofbaseran < 0 ||
                noofbaseran > 4) {
              return 'Please Enter a No. from 0-4';
            }
          } catch (e) {
            return 'Please Input an Integer!';
          }
        } else {
          return "Please Enter a Value!";
        }
      }
    }), // TextFormField
```

Figure 47. *If number of strike equals 3, number of base ran is 0*

Caught out text field

```
child: TextFormField(
    readOnly: noofstrike == 3 ? true : false,
    style: TextStyle(color: Colors.amber),
    decoration: textInputDecoration.copyWith(
        hintText: noofstrike == 3
            ? 'N'
            : 'Caught Out?',
        hintStyle: noofstrike == 3
            ? TextStyle(color: Colors.amber)
            : TextStyle(
                color: Color(0xFFBDBDBD))), // TextStyle
    onChanged: (val) {
      caughtout = val;
    },
    validator: (val) {
      if (noofstrike != 3) {
        if (val!.isNotEmpty) {
          if (val != 'Y' && val != 'N') {
            return "Please enter either (Y/N)";
          }
        } else {
          return "Please enter either (Y/N)";
        }
      }
    }), // TextFormField
```

Figure 48. *If number of strikes is 3, "caught out?" is no*

```
if (noofstrike < 3 &&
    (distance < 14.8 ||
        512 - posy < 0 ||
        379 - posx < 0)) {
  setState(() {
    loading = false;
    error = changecolor == false
        ? "Please enter the position of the ball below"
        : "A proper hit cannot land there at $noofstrike ${noofstrike == 2 ? 'strikes' : 'strike'}!";
  });
} else {
  newaccuracy = double.parse(
      (1 - noofstrike / 3).toStringAsFixed(2));
  newspeed = seconds == 0
      ? -314
      : double.parse(((noofbaseran * 60) / seconds)
          .toStringAsFixed(1));
  if (noofbaseran != 4 && noofbaseran != 0) {
    if (caughtout == 'Y') {
```

Figure 49. *IF function to assess whether a valid input for distance for no. of strike is smaller than 3*

```
if (noofbaseran != 4 && noofbaseran != 0) {
  if (caughtout == 'Y') {
    if (noofbaseran == 1) {
      neweffect = 0.15;
    }
    if (noofbaseran == 2) {
      neweffect = 0.45;
    }
    if (noofbaseran == 3) {
      neweffect = 0.75;
    }
  } else {
    if (noofbaseran == 1) {
      neweffect = 0.30;
    }
    if (noofbaseran == 2) {
      neweffect = 0.60;
    }
    if (noofbaseran == 3) {
      neweffect = 0.90;
    }
  }
} else {
  if (noofbaseran == 4) {
    neweffect = 1;
  } else {
    neweffect = 0;
  }
}
```

Figure 50. *To assign the effectiveness of the student based on the no. of base they ran*

```
await DatabaseService(uid: widget.studentid)
    .updateStudentStatsFromInput(
        newaccuracy,
        newspeed,
        neweffect,
        newdistance,
        avgaccuracy,
        avgspeed!,
        avgeffect,
        avgdistance,
        dateinput,
        datainput);
Navigator.pop(context);
```

Figure 51. *Screenshot of utilising "updateStudentStatsFromInput" to update the database once all variables have been assigned with values. Afterwards "Navigator.pop(context)" pops the current screen TeacherInputStudentData()*

```
Future updateStudentStatsFromInput(
  double newaccuracy,
  double newspeed,
  double neweffect,
  double newdistance,
  double avgaccuracy,
  double avgspeed,
  double avgeffect,
  double avgdistance,
  String? dateinput,
  var datainput,
) async {
  return await users.update({
    'accList': FieldValue.arrayUnion([
      {dateinput: newaccuracy}
    ]),
    'speedList': FieldValue.arrayUnion([
      {dateinput: newspeed}
    ]),
    'effectList': FieldValue.arrayUnion([
      {dateinput: neweffect}
    ]),
    'distanceList': FieldValue.arrayUnion([
      {dateinput: newdistance}
    ]),
    'acc': avgaccuracy,
    'speed': avgspeed,
    'effectiveness': avgeffect,
    'distance': avgdistance,
    'datainput': FieldValue.arrayUnion([
      {dateinput: datainput}
    ]),
  });
}
```

Figure 52. *Screenshot of "updateStudentStatsFromInput" method*

Teacher Input Feedback Screen

Class: TeacherFeedback()

- This is to allow teachers to input feedback to student based on their performance

```dart
import 'package:cstballprogram/services/database.dart';
import 'package:cstballprogram/shared/constant.dart';
import 'package:cstballprogram/shared/loading.dart';
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
```

Figure 53. *Screenshot of libraries imported to teacher feedback.dart*

```dart
child: TextFormField(
  onChanged: (val) {
    feedback = val;
  },
  validator: (val) {
    return val!.isEmpty ? "Please Write a Feedback" : null;
  },
  decoration: textInputDecoration,
  minLines: 35,
  maxLines: 35,
  keyboardType: TextInputType
      .multiline, // user can press enter to move to the next line
), // TextFormField
```

Figure 54. *"minLines" is the height the text field starts with and "maxLines" is the maximum number of lines the text field can display before the text box scrolls vertically*

```dart
Future addFeedback(String? dateinput, String feedback) async {
  return await users.update({
    'feedback': FieldValue.arrayUnion([
      {dateinput: feedback}
    ])
  });
}
```

Figure 55. *"addFeedback" method is then used to update student's feedback array*

Student Home Screen

Class: SHome()

- Provides data streams for its children, Stats() and SideBar(), and setup the app bar used to view the sidebar, feedback and logout option

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:cstballprogram/models/user.dart';
import 'package:cstballprogram/screens/student/feedback/studentfeedbackpage.dart';
import 'package:cstballprogram/screens/student/home/studentsidebar.dart';
import 'package:cstballprogram/services/auth.dart';
import 'package:cstballprogram/services/database.dart';
import 'package:cstballprogram/shared/stats.dart';
import 'package:flutter/material.dart';
import 'package:google_fonts/google_fonts.dart';
import 'package:provider/provider.dart';
```

Figure 56. *Screenshot of libraries imported for studenthome.dart*

```
return MultiProvider(
  providers: [
    StreamProvider<StudentData?>.value(
        value: DatabaseService(uid: loginuseruid?.uid).studentdata,
        initialData: null,
        catchError: (context, error) => null), // StreamProvider.value
    StreamProvider<QuerySnapshot?>.value(
      value: DatabaseService().classroomdata,
      initialData: null,
      catchError: (context, error) => null,
    ) // StreamProvider.value
  ],
```

Figure 57. *Multiple stream providers for its children widget to utilise later on to display data*

```
child: FloatingActionButton(
  onPressed: () {
    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) =>
                StreamProvider<StudentData?>.value(
                    value: DatabaseService(uid: loginuseruid?.uid)
                        .studentdata,
                    initialData: null,
                    catchError: (context, error) => null,
                    child: StudentFeedbackPage()))); // StreamProvi
  },
  child: Icon(Icons.remove_red_eye_outlined),
), // FloatingActionButton
```

Figure 58. *Button to view student feedback page*

```
child: TextButton.icon(
  style: ButtonStyle(
    backgroundColor:
        MaterialStateProperty.all<Color>(Colors.amber),
    shape: MaterialStateProperty.all<RoundedRectangleBorder>(
        RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(6.0),
    )), // RoundedRectangleBorder
  ), // ButtonStyle
  onPressed: () async {
    await _auth.logOut();
  },
  icon: Icon(
    Icons.person,
    color: Colors.grey[800],
  ), // Icon
  label: Text('Logout',
      style: GoogleFonts.lato(
        textStyle:
            TextStyle(color: Colors.grey[800], fontSize: 15.0),
      )), // Text
), // TextButton.icon
```

Figure 59. *Button to logout*

```
  //sign out
  Future logOut() async {
    try {
      return await _fAuth.signOut();
    } catch (e) {
      print(e.toString());
      return null;
    }
  }
}
```

Figure 60. *"logOut" method to logout*

Class: SideBar()

- To show what classrooms students are currently in and allows them to add more classroom

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:cstballprogram/models/user.dart';
import 'package:cstballprogram/services/database.dart';
import 'package:cstballprogram/shared/constant.dart';
import 'package:cstballprogram/shared/loading.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
```

Figure 61. *Screenshot of libraries imported into studentsidebar.dart*

```
child: FloatingActionButton(
  foregroundColor: Colors.grey[800],
  backgroundColor: Colors.amber,
  onPressed: () async {
    await joinClassroom(context, studentclassidlist,
        classroomdatadoc, loginuseruid.uid, studentname);
  },
  child: Icon(
    Icons.add,
  ), // Icon
), // FloatingActionButton
```

Figure 62. *Button to add a new classroom for student*

```
Future<void> joinClassroom(BuildContext context, var studentclassidlist,
    classroomdatadoc, uid, studentname) async {
  return await showDialog(
      context: context,
      builder: (context) {
        return StatefulBuilder(builder: (context, setState) {
          return Form(
            key: _formkey,
            child: AlertDialog(
              title: Text("Enter Class ID Below"),
```

Figure 63. *"joinClassroom" method to create a pop-up box for input of classroom's ID using "showDialog" method*

```
if (_formkey.currentState!.validate()) {
  if (studentclassidlist.length != 0) {
    for (var x in studentclassidlist) {
      if (inputid == x) {
        setState(() {
          error = 'Classroom ID already exist';
          samedata = true;
        });
      }
    }
  }
```

Figure 64. *IF function to check if the class id student inputted to join new classroom already exist in their current classroom list*

```
if (samedata == false) {
  if (classroomdatadoc != null) {
    for (var x in classroomdatadoc) {
      if (inputid == x.get('id')) {
        await DatabaseService(uid: uid)
            .addSNewClass(inputid);
        await DatabaseService(uid: uid, ID: inputid)
            .addNewStudent(studentname);
        match = true;
        Navigator.of(context).pop();
      }
      if (match == false) {
        setState(() {
          error = 'Classroom ID does not exist';
        });
      }
    }
  }
}
```

Figure 65. *IF function to check if class id student entered matches with any class id in the database*

```
Future addSNewClass(String id) async {
  return await users.update({
    'classid': FieldValue.arrayUnion([id]),
  });
}
```

Figure 66. *Adds new class id to student's class id list*

```
Future addNewStudent(String? studentname) async {
  return await classCollection.doc(ID).update({
    'studentid': FieldValue.arrayUnion([
      {uid: studentname}
    ])
  });
}
```

Figure 67. *Classroom adds new student name and id to their student id list*

Student Feedback Page

Class: StudentFeedbackPage()

- Displays app bar of feedback page

```
import 'package:cstballprogram/screens/student/feedback/student_feedback_card.dart';
import 'package:flutter/material.dart';
```

Figure 68. *Screenshot of imported libraries in studentfeedbackpage.dart*

```
appBar: AppBar(
  title: Text('Feedback'),
  backgroundColor: Colors.grey[800],
  elevation: 0,
), // AppBar
```

Figure 69. *Screenshot of "AppBar" widget used to display an app bar*

Class: StudentFeedbackCard()

- Displays teacher's feedback

```
return ListView.builder(
  itemCount: studentdata.feedback.length,
  itemBuilder: (context, int index) {
    return Card(
      child: Padding(
        padding: const EdgeInsets.all(8.0),
        child: Column(
          children: [
            Text(
                '${DateFormat('dd-MM-yyyy').format(DateTime.parse(
                 studentdata.feedback[index].keys.toString().substring(1, studentdata.feedback[index].keys.toString().length - 1)
                 ))}'), // Text
            SizedBox(
              height: 5.0,
            ), // SizedBox
            Text(
                '${studentdata.feedback[index].values.toString().substring(1, studentdata.feedback[index].values.toString().length - 1)}') // Text
          ],
        ), // Column
      ), // Padding
    ); // Card
  },
); // ListView.builder
```

Figure 70. *"ListView.builder" widget is also utilised in this case, similar to how teacher's classroom list is displayed*

References

Asynchronous programming: Streams. (n.d.). Retrieved from https://dart.dev/tutorials/language/streams

Cloud Firestore: FlutterFire. (n.d.). Retrieved from https://firebase.flutter.dev/docs/firestore/usage/#querysnapshot

GestureDetector class Null safety. (n.d.). Retrieved from https://api.flutter.dev/flutter/widgets/GestureDetector-class.html

Getting started with Flutter Cartesian Charts (SfCartesianChart). (n.d.). Retrieved from https://help.syncfusion.com/flutter/cartesian-charts/getting-started

Listview.builder in Flutter. (2021, June 30). Retrieved from https://www.geeksforgeeks.org/listview-builder-in-flutter/

Navigator class Null safety. (n.d.). Retrieved from https://api.flutter.dev/flutter/widgets/Navigator-class.html

StreamBuilder class Null safety. (n.d.). Retrieved from https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html

StreamProvider class Null safety. (n.d.). Retrieved from https://pub.dev/documentation/provider/latest/provider/StreamProvider-class.html

Using Firebase Authentication: FlutterFire. (n.d.). Retrieved from https://firebase.flutter.dev/docs/auth/usage/