

Maddox Scott

Mr. Pierson

IB Computer Science, Period 1

17 March 2022

Criterion C: Development

Key Techniques

1. Saving user data locally and retrieving data using JSON files
2. Converting GPX files to Visible Map Routes
3. Handling input errors and exceptions

Minor Techniques Implemented

1. User-defined objects
2. Objects as data records
3. Simple Selection
4. Complex Selection
5. Loops
6. Nested Loops
7. User-defined Methods
8. User-defined Methods with Parameters
9. User-defined Methods with Return Values
10. Use of Additional Libraries

1. Saving user data locally and retrieving data using JSON files

- **Explanation of Technique**

In order to save a user's data locally, the entirety of the User instance is first converted into a Map, which is a collection of identifier / value pairs. Both classes "User" and "Activity" have toJson methods which return a Map of Strings from a User or Activity Instance. Each variable is converted to a String, and given a corresponding identifier:

User toJson()

```
//Method to convert a user object into a Map, which essentially is a long string that represents the objects variables.
Map<String, dynamic> toJson() {
    Map accumulatedStatistics = this.accumulatedStatistics != null
        ? this.accumulatedStatistics.toJson()
        : null;
    List<Map> activityList = this.activityList != null
        ? this.activityList.map((i) => i.toJson()).toList()
        : null;
    return {
        'startDateTime': startDateTime.toString(),
        'accumulatedStatistics': accumulatedStatistics,
        'activityList': activityList,
        'name': name,
        'userName': _userName,
        'password': _password,
    };
}
```

Activity toJson()

```
//Method to convert an Activity object into a Map, which essentially is a long string that represents the object and its
Map<String, dynamic> toJson() => {
    'activityDateTime': activityDateTime.toString(),
    'routeFile': routeFile.path,
    'title': title,
    'duration': duration.toString(),
    'distance': distance,
    'averageSpeed': averageSpeed,
    'elevationChange': elevationChange,
    'averageHeartRate': averageHeartRate,
};
```

Activity instances are nested inside of User instances, so the method runs a quick if-else statement to decide whether to save the activityList as the List's contents using Activity's toJson method, or simply save as null (Amorelli 2020).

After the User and its Activities are converted into a savable Map of Strings, the User Map is then saved to SharedPreferences, which is an imported library used to save text locally.

```
//Method used to save/update a user's information to SharedPreferences, and is saved locally.
void saveUser(User user, SharedPreferences prefs) async {
    String json = jsonEncode(user.toJson());
    prefs.setString(user.getUserName(), json);
}
```

The application can retrieve the User Map from Shared preferences and convert it back into a User instance with the following code:

User fromJson()

```
//Method to restore a user instance from a Map String.
User.fromJson(Map<String, dynamic> json)
: startDateTime = DateTime.parse(json['startDateTime']),
  //activityList = List<Activity>.from(json['activityList']),
  activityList = decodeActivityList(json['activityList']),
  name = json['name'],
  _userName = json['userName'],
  _password = json['password'];
```

Activity fromJson()

```
//Method used to restore an Activity file from a Map String.

Activity.fromJson(Map<String, dynamic> json)
: activityDateTime = DateTime.parse(json['activityDateTime']),
  gpxContents = json['gpxContents'],
  title = json['title'],
  duration = durationFromString(json['duration']),
  distance = json['distance'],
  avgerageSpeed = json['averageSpeed'],
  elevationChange = json['elevationChange'],
  averageHeartRate = json['averageHeartRate'];
```

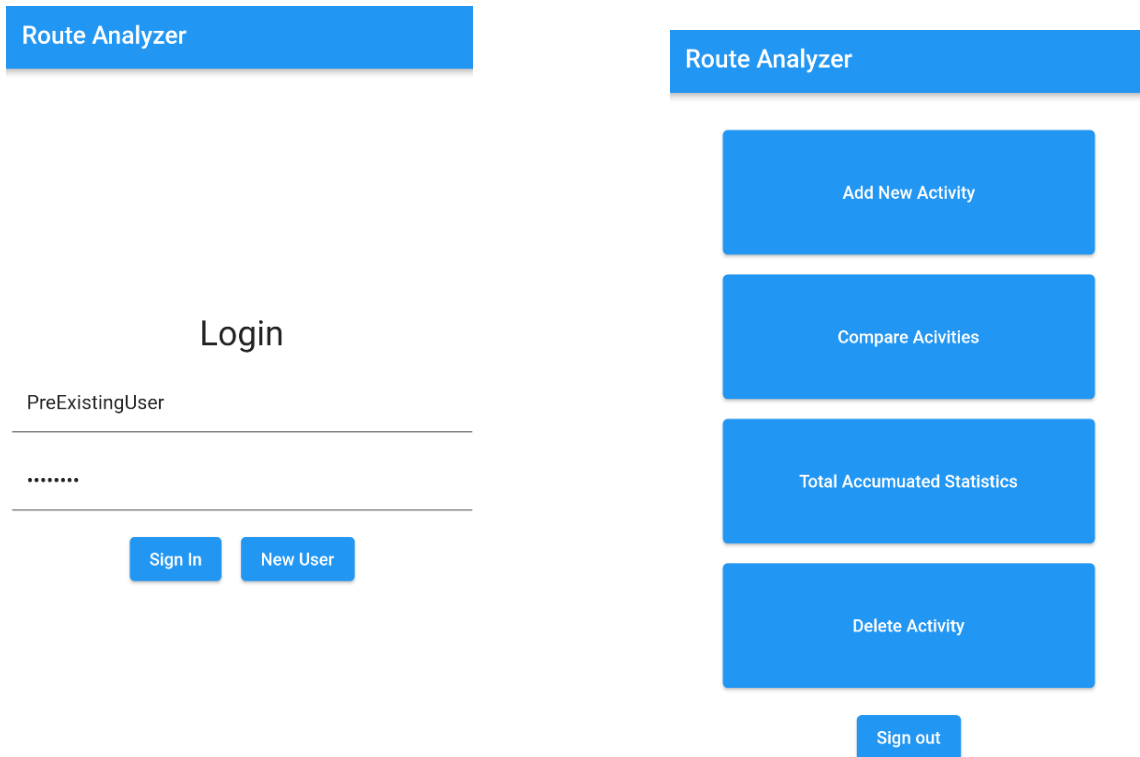
- **Ingenuity**

When saving a user's data locally, if the user's information were to be converted into a single string, it would be difficult to locate and extract a single piece of information from the user. Therefore, a Map is used rather than String. Maps can store several different strings, which each have a corresponding identifier used to locate the string, which allow for easy access to specific users or variables.

Shared Preferences automatically creates new files or directories as necessary, saving the user a large amount of work. Furthermore, SharedPreferences allows for the User's data to be saved in a single location accessible from anywhere, making organizing and accessing data effortless (Bezkodeer 2022).

- **Appropriateness**

It is essential for the application to save data between app loads so the user does not need to re-add activities during each use of the application. The user can now bypass the new user screen if they have a user made.



Furthermore, my client was concerned with how the application would function without a connection. No other functions require a connection to the internet, so by saving data locally rather than online, the user can use the application from anywhere, including remote biking routes.

- **Source**

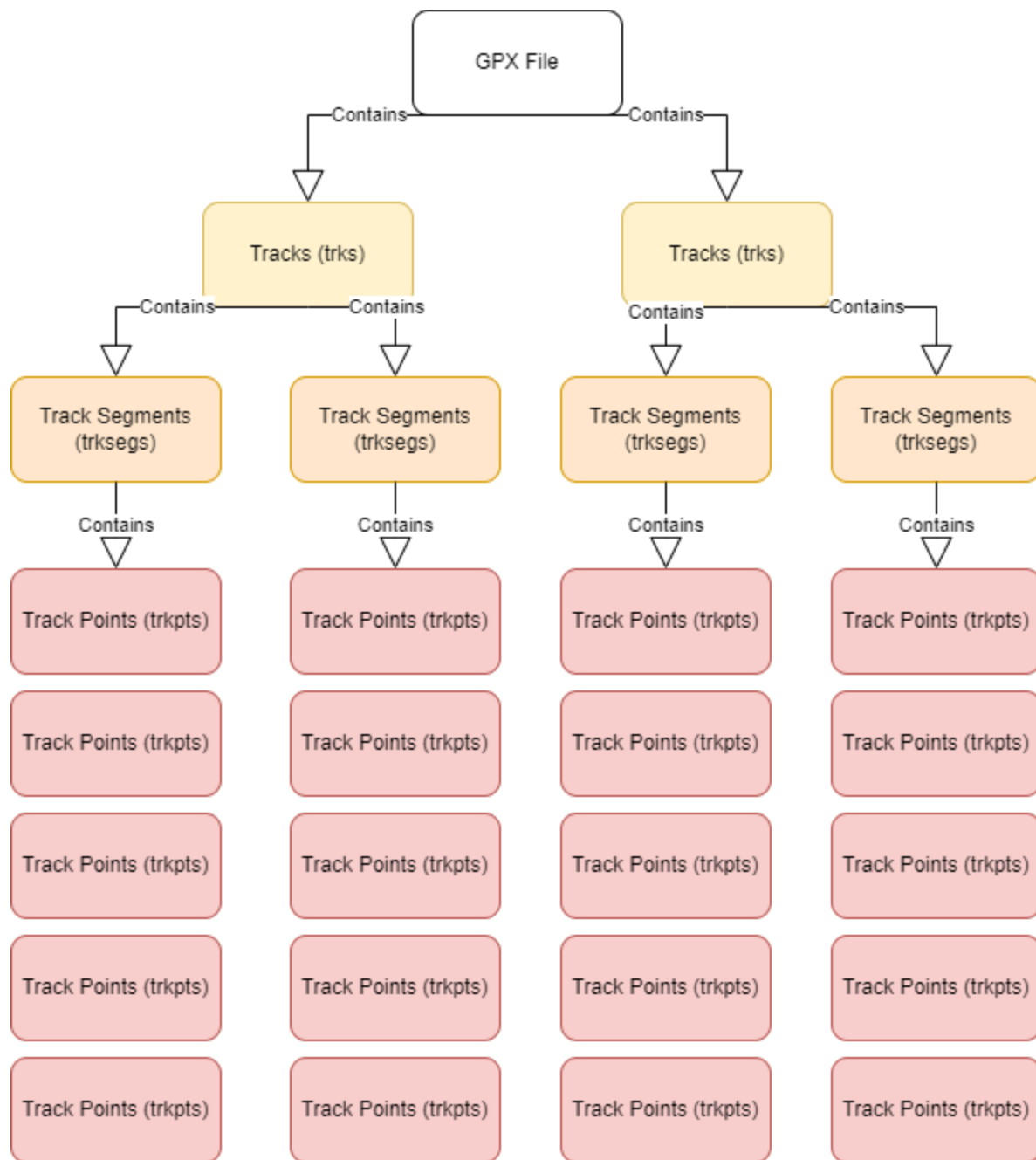
Amorelli (2020). *How do I save a list (of objects) to local memory?* Available at : stackoverflow.com/questions/65359242/flutter-shared-preferences-how-do-i-save-a-list-of-object-to-local-memory. (Accessed 5 Mar. 2022).

Bezkodeer (2022). *Dart/Flutter – Convert Object, List to JSON string* Available at: www.bezkoder.com/dart-flutter-convert-object-to-json-string/ (Accessed 7 Mar. 2022).

2. Converting GPX files to Visible Map Routes

- **Explanation of Technique**

Each Activity registered under the user has a corresponding GPX file, which contains all of the latitude and longitude coordinates, elevations, and time stamps required to display the bike route on a map. GPX files are split into segments, and the structure is as such:



Tracks are the entirety of a bike ride's route. Track Segments are a portion of a route. A new segment is made if the user stops and restarts their tracking application during a ride. Track Points are each latitude and longitude pair taken by the device's gps during a ride. There can be thousands of these for even a short ride.

First, a list of latitude longitude pairs is created to hold every track point of the file. Afterwards, a Gpx variable is initialized from the Activity's GPX file.

```
void constructPolyline(Activity activity) {
    print('CONSTRUCT POLYLINE CALLED');
    List<LatLng> polylineLatLongs = [];
    Gpx gpx = GpxReader().fromString(activity.routeFile.readAsStringSync());
```

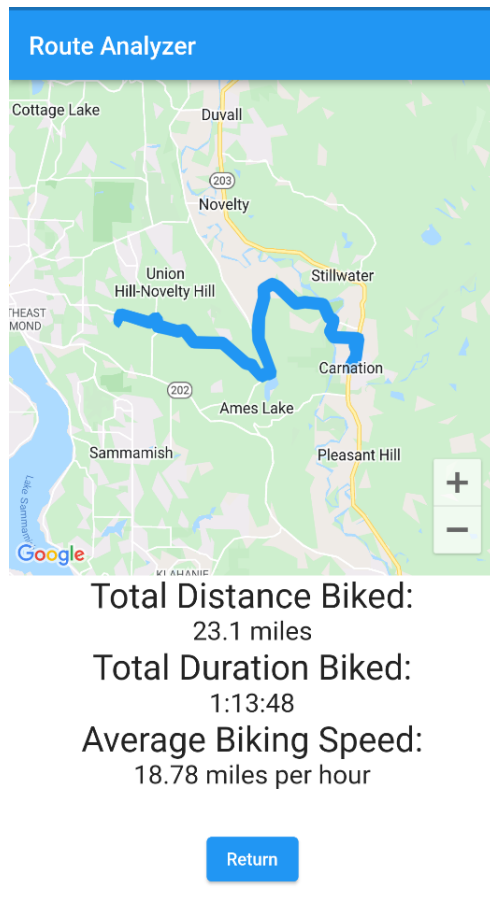
Next, a nested for-loop loops through the entirety of the GPX file structure, adding every Track Point from every Track Segment to the list of LatLngs.

```
for (int i = 0; i < gpx.trks.length; i++) {
    for (int j = 0; j < gpx.trks[i].trksegs.length; j++) {
        for (int k = 0; k < gpx.trks[i].trksegs[j].trkpts.length; k++) {
            polylineLatLongs.add(new LatLng(gpx.trks[i].trksegs[j].trkpts[k].lat,
                gpx.trks[i].trksegs[j].trkpts[k].lon));
            print('ITERATION DONE');
        }
    }
}
```

After every Track Point Has been added, a Polyline is initialized using the list of LatLngs. Polylines are a class under the GoogleMaps Flutter Library, which are used to display lines or shapes in GoogleMaps (Pinkesh 2021).

```
_polylines.add(Polyline(
    polylineId: PolylineId('0'),
    points: polylineLatLongs,
    color: Colors.blue,
    visible: true,
    width: 10)); // Polyline
print(_polylines.toString());
}
```

Application Output:



- **Ingenuity**

The GoogleMaps widget, as well as all other map widgets, lacks the ability to display a raw GPX file, due to the recent creation and implementation of GPX files in the Industry. Furthermore, there is no preexisting public library capable of converting a GPX file directly to a Polyline (Marcus 2021).

The nested for-loop also allows the polyline constructor to work for all kinds of GPX bike routes that could be called “outliers”, such as routes where the user paused the tracking app, moved to a different location, and clicked resume, or routes that are collections of several other GPX routes.

- **Appropriateness**

A success criteria of the application is the ability to display Activities and their bike routes within the application. This requirement allows the user to view where and where not they

have biked, as well as compare the routes biked between two activities. This technique fulfills that requirement by displaying GPX files in a map widget.

By having the application convert the GPX file to a polyline, the app maintains its ease of use by only requiring the user to import a GPX file. If the user had to manually convert the GPX to a displayable format themselves, the app would be monotonous and fail in its purpose of being a quick-to-use route comparison app.

- **Source**

Pinkesh (2021) *Adding Google Maps to a Flutter app* Available At:

blog.logrocket.com/adding-google-maps-to-a-flutter-app/. (Accessed 17 Mar. 2022).

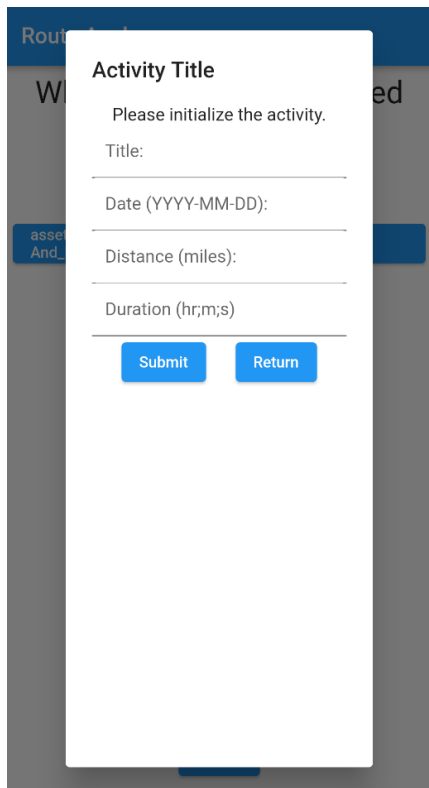
Marcus (2021) *Flutter Google Maps API Tutorial | Markers, Polylines, & Directions API*

Available at: www.youtube.com/watch?v=Zz5hMvgiWmY. (Accessed 15 Mar. 2022).

3. Handling Input Exceptions and Errors

- **Explanation of Technique**

Within the “create new activity screen”, the user types in information related to the activity, such as:



The screenshot shows a mobile application interface with a dark blue header and a light gray background. A white modal dialog box is centered on the screen, titled "Activity Title". Inside the dialog, there is a message "Please initialize the activity." followed by four input fields: "Title:", "Date (YYYY-MM-DD):", "Distance (miles):", and "Duration (hr;m;s)". At the bottom of the dialog are two blue buttons labeled "Submit" and "Return". The background of the app is partially visible, showing a map and some text like "Route", "W", "asse", and "And_".

The user can type in anything, including strings which are not acceptable for Activity initialization. Once all fields are filled and the user taps “submit”, several conditionals are fired. The first is below, which checks if the inputted date is correctly formatted:

```
if (DateTime.TryParse(_currentDate) != null) { ...  
} else { ...
```

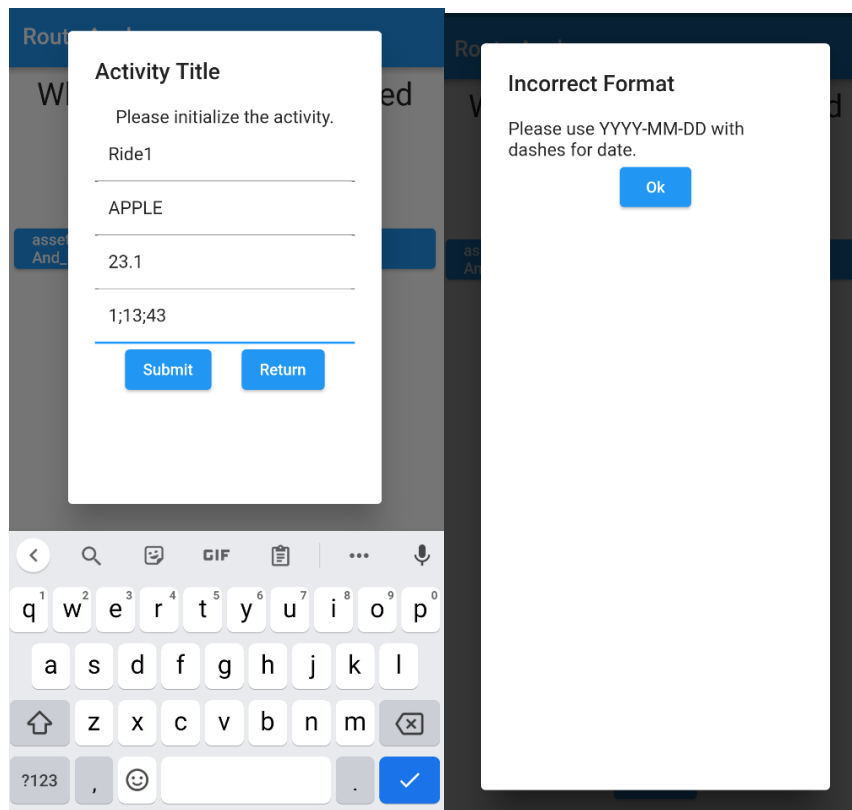
If the format is correct, the next conditional is ran, which checks if the duration string is correctly formatted.

```
if (double.TryParse(_currentDistance) != null) {
```

Lastly, the code tries to parse a Duration from the inputted string. If successful, the application creates a new Activity.

```
try {  
    Duration _test =  
        durationFromString(_currentDuration);  
    print(_test);  
} catch (e) {
```

If at any point one of the conditionals returned false or threw an exception, an alert notification as such is displayed (Filip 2021).



- **Ingenuity**

For inputted date and distance, the conditional uses premade methods which return null if the input is incorrect. This allows the conditional to catch exceptions before they are even thrown. For the Duration however, incorrect input can throw a variety of errors. Therefore, the catch block catches all errors indiscriminately, regardless of what the input string is.

- **Appropriateness**

The conditionals determine whether the input is valid before creating an Activity. This prevents the creation of corrupted or incomplete Activities, which could cause issues down the line. Also, the error messages tell the user what exactly they are doing wrong, and allows them to return and fix it (Filip 2021).

- **Source**

Filip (2021) *Write your first Flutter app, part 1* Available At: docs.flutter.dev/get-started/codelab. (Accessed 2 Mar. 2022).

Word Count: 996 Words