Exercises 4 - Dart and Flapjax

Søren Krogh - 20105661
Emil Madsen - 20105376
K. Rohde Fischer - 20052356

December 1, 2014

Exercise 1

Trivially solved, no issues.

Exercise 2

Exercise 3

Exercise 4

The async keyword is added to the function declaration, and marks the function as asynchronous, making it return a `Future<T>` immidately, rather than computing the function and returning `T` itself. The function is scheduled to be executed, and runs asynchronously to the caller.

The await keyword awaits the completion of computation for a `Future<T>`, in layman terms, it suspoens execution while waiting for the future to complete. The result of the await expression on a `Future<T>` is `T`.

In short, async is used to run functions asynchronously, wrapping their return types in futures. - This allows one to easily write asynchronous code. Await is used to wait for asynchronous functions to compute, as if they were non-async. - This allows one to easily write code which is asynchrous as if it was synchronous.

In essence, we avoid the convoluted callback handling of Javascript, or the verbose use of '`.then(T)`' and heavy use of lambdas from Dart futures or the Javascript Q library. While retaining the advantages of asynchronous evaluation:

```
1  file.exists().then((bool exists) {
2      if (!exists) {
3        file.create(recursive: true).then((File file) {
4          file.writeAsString("version=1");
5        })
6        .catchError(handleError);
7      } else {
```

```
8         file.readAsString().then((String text) {
9           int version = int.parse(text.split("=").last);
10          file.writeAsString("version=" + (version+1).toString());
11        })
12        .catchError(handleError);
13    }
14  });
```

Can be rewritten to the below using async and await;

```
if(!(await file.exists())) {
      try {
        (await file.create(recursive: true)).writeAsString("version=1");
      } catch(exception, stackTrace) {
        handleError(exception);
      }
  } else {
      try {
        int version = int.parse((await file.readAsString()).split("=").last);
        file.writeAsString("version=" + (version+1).toString());
      } catch(exception, stackTrace) {
        handleError(exception);
      }
  }
```

As can be seen errors are now handled using the exception handling mechanism, rather than using the catchError scheme from futures.

We could achieve the same scheme as used in the second code snippet using the synchronous file IO system. We would however not be gaining the advantages of asynchronicity, if we were to do that.