Dart & Flutter: Natively-Run App-Programming Miracles or Monsters

Faith Twardzik
University of California, Los Angeles
CS 131

Dart: The Programming Language

Dart is a relatively recently-designed programming language released by Google in 2011, for the purpose of building mobile, desktop, web, and server applications. First, we discuss some of the inherent characteristics of Dart as a programming language, within the context of several of the other programming languages we have discussed, and whether or not they constitute advantages or weaknesses. Dart has a syntax similar to that of C, enabling ease of use for programmers versed in the widely-known language. In addition, Dart is a fully object-oriented language as well as class-based, such as are Java and C++, where everything, including numbers, functions, etc. constitute objects. This is a definite strength of the language, seeing as object-oriented languages allow a variety of flexibilities, including code reuse in the form of extension/ inheritance (and in this case mixins as well, which allow the reuse of code in other classes without incurring multiple instances of inheritance), classes that can organize class-specific methods and private data members safe from corruption outside of the class utilizing the concept of encapsulation, and instances of objects that are easily mutable within the framework of the class to which they belong. Structurally, the object-oriented nature of Dart is hugely advantageous. This presumes that Dart is a form of an imperative language (like Java and C), that also includes functional counterparts, such as lambda functions, for instance,

```
int a_sum(int first_num, int second_num) => first_num + second_num;
```

and higher-order functions with currying (this is similar to functional languages we saw, like OCaml, which made heavy use of lambda and higher-order functions). The exceedingly interesting strength to such a modern language as Dart is that although it is rooted in an imperative style, its functional programming elements do not detract from its imperative functionality, but rather add to it. You can use the language in either capacity, where one is not more natural than the other, allowing grand flexibility for the programmer. For example, the functional aspect of Dart allows the following, which are all unavailable in imperative languages: passing functions as parameters to functions, assigning functions as the "values" of variables, currying, and constructing unnamed lambda functions.

Optional Typing Functionality

In addition, Dart is a statically-typed language, but has dynamic type functionality. Dart takes advantage of sound typing, which is something inherent to programming languages such as Java and C++. Dart, specifically, utilizes a static analyzer at compile-time in order to ensure variables have values that correspond with their static types. This is advantageous in the sense that it will not allow the programmer to compile code with potential type mismatches and runtime/logic type errors, the code will be more easily readable because types of values will be explicitly known, corruption in updates to code will be caught at compile-time, and AOT compilation will be more efficient. However, Dart is similar to programming languages such as Python and OCaml in the sense that it implements type inferencing, with optional type annotations but no required explicit type declarations for a variable, as is the case in C++ and Java. This allows the soundness of static type checking, as well as the programmer-friendly type inference of newer languages like Python. However, yet again with a crossover language like Dart, we have intersectional functionality, because the system of type inference is not total. The static analyzer will infer the types of objects that are not explicitly typed. We see this using the "var" keyword below:

```
var different_types = {'word': 'hello', 'number': 69}; // Dart infers this to be of type Map<String, Object>
```

However, on the other hand, you are allowed to explicitly define the type of an object. Yet, unlike variables in functional languages such as OCaml, in Dart you *are* allowed to modify the type of an object later on in the program, using the "dynamic" keyword. This allows us the benefit of strict static type checking, as well as the flexibility of type modification, a substantial strength to the language. For example:

Map<String, dynamic> different_types = {'word': 'hello', 'number': 69}; // here we explicitly define the key to be of type String, and the value to be dynamic, meaning that we are allowed to modify the values' types in the future

In essence, Dart is optionally typed, which is incredibly rare among a gamut of languages that are either typed, or not typed (such as JavaScript).

Asynchronous Concurrency in Dart's Isolates Implementation

Next, we discuss the way concurrency is implemented within Dart, which is through asynchronous operations. Multithreading and multiprocessing, core components of languages such as C++ (and even Python, to a degree), are reduced to *only* the usage of asynchrony. This is in some ways a weakness to the language, allowing less programmer flexibility in concurrency implementation. However, its concurrency system is simple, reliable, efficient, and easy to implement. It is structured through the usage of isolates, which have a single thread and a single, running event loop, but their own separate memory spaces. Apps are allowed to have virtually as many isolates as they would like, where these isolates communicate to one another via messages. Isolates are not merely asynchronous operations however, because each of them receives its own thread, as opposed to asynchronous coroutines, tasks, and futures in the Python asyncio library that all operated on a single thread. Isolates allow the branching-off of wait-intensive procedures, to allow frequent context switches within a running routine when it is blocked and/or waiting, so that other routines may execute in the meantime. Isolates utilize async syntax similar to that of the Python asyncio library – *async* is used to define an asynchronous function, and *await* is used to await a call and generate a context switch to another isolate until that call has returned, yet the procedure through which isolates are created and maintained is more similar to that of process creation in Python, utilizing the *spawn* method. This is somewhat unnecessarily confusing syntax-wise, mixing multi-process and asynchronous syntaxes.

The Strengths and Weakness of Dart

The above conversation discussed the strengths and weakness of Dart in reference to other common programming languages, yet we now discuss these in terms of the development of a natively-running mobile app running on high-end modern cellphones, utilizing image processing by machine learning. One of Dart's strengths within this framework is its AOT (ahead-of-time) and JIT (just-in-time) compilation capabilities. AOT compilation is more commonly used in programming languages used for writing mobile apps, such as Java and C#, where the app is recompiled when modifications are made. This is highly cumbersome and inefficient, especially for intensive apps in which complete re-compilation will take substantial time, as is likely the case in our garage-sale image-processing application with a work-intensive machine-learning algorithm. This type of compilation will work well for building and testing natively-run mobile aps. However, Dart is unique in that it also offers JIT functionality, where the app will not need to completely reload or recompile when modifications are made. This is exceedingly efficient during the app-development stage, offering quick updates.

Dart is also ideal for a native mobile app such as the one we are considering because of its extreme portability. Dart mobile apps can run on any OS (operating system), including MacOS, iOS, Android, and Windows, and web apps may run on all browsers. In addition, Dart is perfect for reactive programming (i.e. built upon user interactivity) based on data streams, intensive UIs, and substantial widget usage, which is the type of programming we would be using for this application. Dart's effectiveness for reactive programming stems from its usage of asynchrony, as well as an efficiently implemented garbage collector. For most languages that utilize a garbage collector, such as Python and Scheme, garbage collection greatly hinders the efficiency of the program, sometimes halting execution for seconds at a time – especially unideal for an interactive application where studies show that the patience of a user loading a page before retrying or exiting is less than three seconds. However, in the context of Flutter, Dart's garbage collector is actually remarkably efficient. This is crucial, given that Flutter apps will be implementing up to thousands of widgets for user interaction, that may be created and then destroyed relatively quickly within a short time period, requiring the heavy usage of garbage collection to avoid memory overload. Dart's garbage collector, however, utilizes scheduling, a young space scavenger, and parallel mark sweep collectors to be one of the most efficient garbage collectors currently out there. In scheduling, the garbage collector tracks the app's current activity, and if the app is running idle, a message will be sent back to the garbage collector, to begin its collection while such collection will not

impact user interaction. The first phase of garbage collection, designed for quickly created and destroyed objects, is the faster of the two, as it is used more extensively. Generated objects are allocated to locations in a memory section known as a nursery that are contiguous, almost like an array in concept. Only half of the nursery is active at once, however, and when there is no longer any available space in the active half for the next allocation, the collector searches for all objects still alive, transfers them over to the other, previously inactive half, marks it as active, does nothing to dead objects (they will have no references anyways), and terminates collection. The second collection phase is for objects with a longer lifespan, as this is the less efficient phase. The collector first searches for all objects still live and marks them, and then traverses memory once more to recycle the objects that have not been marked, restoring marks along the way. Clearly, this is a time-intensive operation, especially during marking, so only used on objects that are less likely to be marked (long lifespan objects). These phases combine to form an extremely efficient garbage collector and allocation/deallocation system, allowing Dart the programmer-friendly ease of garbage collection, safety from memory leaks, and efficiency all wrapped in one.

Flutter: An SDK Using Dart

Next, we discuss Flutter in more depth. The previous discussions focused on Dart as a programming language, though in utilizing Flutter, which uses Dart, we have been indirectly discussing Flutter's strengths and weaknesses as well. In more depth, Flutter is an SDK (software development kit) designed for building native mobile apps that are cross-sectional among operating systems. One of its strengths is standardized and reliable performance, ideal for user interactivity. For example, Flutter maintains a 60-fps frame rate, which is effective for user viewing. Flutter is also extremely portable for international optimization, as it includes package widgets that allow conversion of languages, measurement units, times, currency, and other global factors. This allows an app to be quickly internationalized, whereas traditional app-building would necessitate a baseline app in one language, one measurement unit standard, one time zone, etc., and the piecemeal addition of others. Because of Dart's JIT compilation, Flutter offers "hot reloading", where modifications update immediately, without re-compiling/reloading over several minutes, allowing efficient updating and testing during development. Flutter also offers a wide range of already-coded widgets and objects that allow construction of a user interface to be simple, quick, and non-intensive for the programmer. Objects include fonts and buttons, and all are already available and customizable for usage.

On the other hand, as with any SDK or programming language, Flutter has its fair share of weaknesses. Among them, Flutter has had issues in the past with keeping up with iOS updates, hindering its portability. While it is completely functional with Android, its functionality lags with iOS app construction. Additionally, because of built-in widgets and available libraries, Flutter apps are massive in size, which is debatably one of its worst features, especially in a framework such as the native mobile app we are building, which we would expect to take up a small amount of memory on the user's phone (they won't waste precious memory on a garage-sale-scanning app, and so we must reduce size as much as possible). Flutter is also weak in the sense that it, and Dart, which it is based upon, are both relatively new. This precludes not finding many programmers with experience, nor finding many resources for development within these frameworks, as well as a distinct lack of open-source, third-party libraries for usage. It has a substantial list of built-ins, but its libraries, packages, and tools still lag behind older SDKs with longer-standing programming languages at their core. Therefore, Flutter may be best used in the future, when more updates have been made and more functionality is available.

tflite for App Machine Learning Functionality

Lastly, we give a brief overview of tflite, which will be used for machine-learning-based functionality and image processing/detection of garage sale items. tflite is a Flutter plugin allowing access to the TensorFlow Lite API. As a plugin for Flutter, it enjoys the weaknesses and strengths of the SDK. As a plugin itself, it utilizes the encapsulated functionality of TensorFlow (machine-learning algorithms build upon convolutional neural networks naturally trained for image recognition and classification). It can perform image classification (labelling of an image), as well as object detection (finding an object and classifying), for both static images and image streams. It also supports real-time detection through flutter_realtime_detection, which accesses the mobile device's camera and utilizes the tflite plugin for immediate image recognition. This is what our app would be using. In conclusion, Dart, Flutter, and tflite offer a killer combination that will be ideal for the construction of this sort of app, despite some definite drawbacks.

References

Chetu Marketing. “Advantages and Disadvantages of Using Flutter for Mobile Development.” */ Habr*, Habr, 14 Nov. 2019, habr.com/en/post/455020/.

“Dart (Programming Language).” *Wikipedia*, Wikimedia Foundation, 29 May 2020, en.wikipedia.org/wiki/Dart_%28programming_language%29.

“The Dart Type System.” *Dart*, dart.dev/guides/language/sound-dart.

“Intro to Dart for Java Developers.” *Google*, Google, codelabs.developers.google.com/codelabs/from-java-to-dart/#5.

Joe. “Primary Menu.” *Coding With Joe*, 16 Apr. 2019, codingwithjoe.com/dart-fundamentals-isolates/.

Karkkainen, Kimmo. *Dart*. 2020, ccle.ucla.edu/pluginfile.php/3509471/mod_resource/content/0/CS131%20-%20Week%209.pdf.

Rao, Vicky Singh. “Why You Should Learn Dart Programming Language?” *Technotification*, 18 Jan. 2019, www.technotification.com/2019/01/why-dart-programming-language.html.

Sullivan, Matt. “Flutter: Don't Fear the Garbage Collector.” *Medium*, Flutter, 4 Jan. 2019, medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30.

“Tflite.” *Tflite - Dart API Docs*, pub.dev/documentation/tflite/latest/#Image-Classification.