Renato Athaydes Personal Website

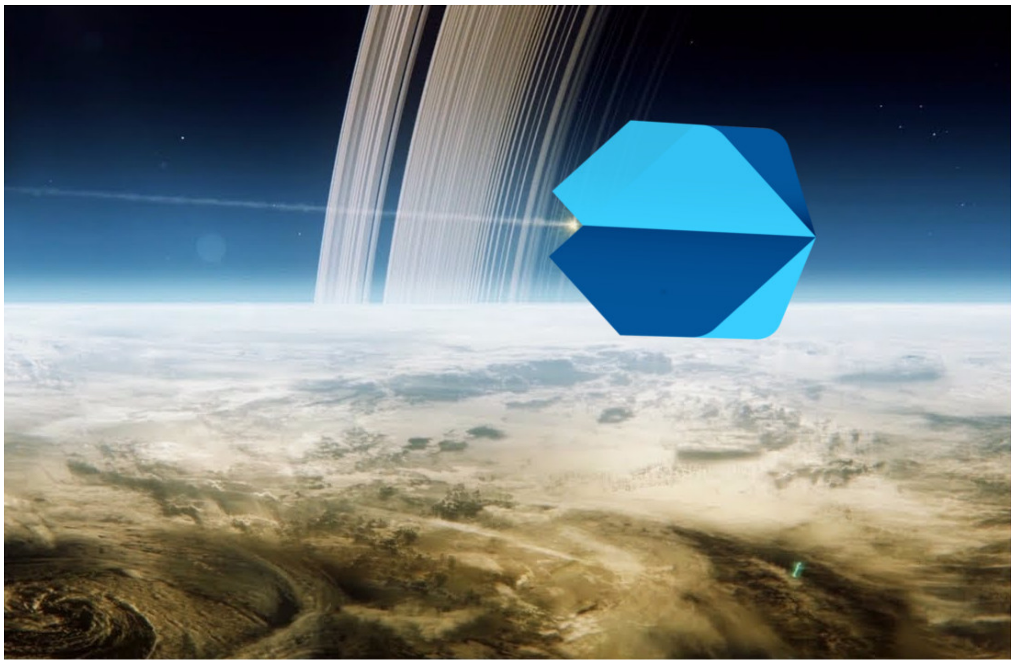
I share, therefore I am

Home Posts About

Interesting Features of the Dart Programming Language

And why it might be the first truly all-platform language

Written on 07 Jul 2019, 01:32 PM (Last updated on 10 Jul 2019, 07:22 PM)



Dart falling into Saturn (*original image from National Geographic Cassini video*)

Table of Contents:

Introduction

Dart is often compared to Java for being a “boring” language. While it’s true that if you know Java (or JavaScript for that matter) Dart will look pretty familiar, I hope you’ll agree after reading the rest of this post that Dart has quite a lot more to offer, without going to extremes and becoming inaccessible. And to be honest, I would even go as far as saying that Dart deserves to be regarded as belonging to the same league as other modern languages like Kotlin and Swift (but with a killer app the other 2 are lacking: Flutter).

Flutter developers tend to love the language from what I’ve been reading lately, and I can definitely see why.

> *Flutter is already the 3rd most loved Framework in the StackOverflow Developer Survey 2019.*

In this article, I want to show how the Dart of 2019 (version 2.4 to be exact) is actually quite an exciting language!

I’ll discuss some nice feature of the language itself, as well as some neat platform characteristics (like built-in platform-specific - like browser - testing support and hot reloading) which make developer’s lives so much easier.

By the end, I hope you’ll be as convinced as I am that Dart deserves the success it is starting to finally enjoy, and that it might be a good choice for multi-platform development at present, and even more in the future.

Where Dart is coming from

Before we start, a little bit of history is appropriate so that, in case you're not familiar with Dart and its origins, you may better understand the rest of this post.

Feel free to skip it if you're just interested in the code.

The Dart programming language was born in 2011 with the modest goal of fixing the web. In other words, to try and replace JavaScript in the browser.

> *See this discussion on HackerNews for how Brendan Eich felt about Dart at the time. TL;DR he thought it was a hostile, replace-not-embrace effort by Google to kill JS.*

Needless to say, it failed miserably.

That became clear when Google announced in March 2015 that it would not include the Dart VM in Chrome (as everyone had expected), and would instead focus on the Dart-to-JS compiler.

That seemed like a death blow to Dart. After all, it was now relegated to a mere tranpile-to-JS language, just one out of many… and not a very exciting one at that.

To most people, the language was en route to quickly fade away and become a tiny footnote in the history of programming languages.

But the language survived and, at least at Google, continued to prosper.

And then, something happened!

In May 2017, Google released the first public alpha version of Flutter, a cross-platform user interface development framework that uses Dart to create mobile apps for both iOS and Android. With time, Flutter started to get quite popular (🌟 65,792 stars on GitHub as of writing).

That sparked new life into the Dart project, and with the releases of Dart 2 in August 2018, and Flutter 1.0 soon after, it became clear that Dart is absolutely back in town, and I dare say it's going to be one of the fastest growing technologies in 2019!

> *In May 2019, Google announced its intention to expand Flutter beyond mobile, to web, embedded and Desktop.*

Quick Dart Overview

As I expect many readers to not be familiar with Dart yet, specially with Dart 2.3+, I think it's necessary to quickly introduce the language before we discuss the more interesting features.

Let's start with the basics. Dart is a statically-typed language with a sound type system.

Here's the simplest possible Hello World implementation in Dart:

```
main() {
  print("Hello World");
}
```

As you can see, Dart supports free functions. It also has classes:

```
class Person {
  final String name;
  final int age;

  Person(this.name, this.age);
  Person(this.name, this.age);

  @override String toString() =>
    "Name: $name, Age: $age";
}
```

The above example shows how Dart also supports String interpolation and arrow functions/methods (for single expression implementations).

You can use the `dart` command to run any Dart file with a `main` function using the Dart VM in interpreted mode:

```
$ dart main.dart
```

You can also compile to machine code for Linux, Windows and MacOS using `dart2aot`, then run it with `dartaotruntime`:

```
$ dart2aot main.dart main.dart.aot
$ dartaotruntime main.dart.aot
```

If you want to generate JavaScript to run Dart in the browser, you can use `dart2js`:

```
$ dart2js -o main.js main.dart
```

Usually, though, you would automate this step using `webdev`, a tool written in Dart itself that's Dart web developer's best friend!

See the official Get Started guides for more details:

- Get Started (CLI and Server).
- Get Started (web).
- Get Started with Flutter (mobile).

Dart Language Features

The rest of this article focuses on the things I believe are interesting and noteworthy about Dart. For a high level overview of most Dart features, see the Tour of Dart.

Flexible imports and exports

Let’s start with a very simple feature: Dart imports/exports.

Dart imports can refer to Dart standard library packages, third-party packages or to other local files:

```
// Local file import
import 'some_dir/other_file.dart';

// Dart standard library import
import 'dart:collection';

// External package import
import 'package:http/http.dart';
```

The above imports will expose everything that is public in the file or package to the global namespace. So, to use a type from `dart:collection` , for example, no namespace is necessary:

```
import 'dart:collection';

final queue = DoubleLinkedQueue<int>();
```

But it is also possible to namespace the import instead, to avoid conflicts:

```
import 'dart:collection' as collection;

final queue = collection.DoubleLinkedQueue<int>();
```

Namespaced imports can also make the code more readable as some packages are designed to be namespaced, such as the `http` package:

```
import 'package:http/http.dart' as http;

main() async {
  final response = await http.get('/something',
      headers: {'Accept': 'text/html'});
  assert(response.statusCode == 200);
}
```

It is also possible to only expose certain types from an import (whitelisting), or to import everything except some specific types (blacklisting):

```
import 'dart:collection' show DoubleLinkedQueue;
import 'dart:math' hide Rectangle, MutableRectangle;
```

It is also interesting to note how Dart handles package exports.

As shown above, package imports are usually of the form `import 'package:<name>/<name>.dart'`. That is, you normally import one file from the package which contains all of its exports. The file will look something like this:

```
import 'src/some_definitions.dart';
import 'src/more_definitions.dart';

export function1;
export function2;
export TypeA;
```

This results in a very simple, but efficient and concise module system.

Constructors

Dart has 3 different types of constructors:

```
class UsesNormalConstructor {
  String value;
  UsesNormalConstructor() {
    value = "hello";
  }
}

class UsesNamedConstructors {
  String value;
  UsesNamedConstructors.firstConstructor() {
    value = "first";
  }
  UsesNamedConstructors.secondConstructor() {
    value = "second";
  }
}

class UsesFactoryConstructor {
  final String name;

  // private named constructor
  UsesFactoryConstructor._create(String name) :
     this.name = name;


  factory UsesFactoryConstructor(String name) {
    // can return a new instance, some cached instance,
    // or even a subtype!
    return UsesFactoryConstructor._create(name);
  }
}

main() {
  print(UsesNormalConstructor());
  print(UsesNamedConstructors.firstConstructor());
  print(UsesFactoryConstructor("joe"));
}
```

Since Dart 2.0, it is not necessary to use the `new` *keyword when invoking a constructor.*

More interestingly, constructors that simply initialize the class' variables can be written in shorthand notation:

```
class Example {
  String text;
  int number;

  Example(this.text, this.number);
}
```

Looks quite nice, and saves the developer from writing repetitive, tedious code (e.g. `this.text = text;`).

Named and optional parameters

Dart allows the programmer to decide whether a function should take positional or named arguments.

The most basic syntax is for positional arguments (i.e. the position of an argument determines which parameter is bound to it, as in most languages):

```
add(int first, int second) => first + second;

add(1, 2);
```

To use named arguments instead, just wrap the parametes within `{` and `}` :

```
add({int first, int second}) => first + second;

add(first: 1, second: 2);
```

A better example would be a function that requires several parameters… or even better, a constructor, as in the following example:

```
class Person {
  String firstName;
  String lastName;
  int age;
  String occupation;

  Person({this.firstName, this.lastName, this.age, this.occu
}
```

A person can then be created with a very nice syntax:

```
final joe = Person(
  firstName: 'Joe',
  lastName: 'Doe',
  occupation: 'plumber',
  age: 32,
);
```

This is used extensively in Flutter code, making UI code look very easy to read:

Example from FlutterByExample.com

```
@override
Widget build(BuildContext context) {
  return Padding(
    padding: const EdgeInsets.symmetric(horizontal: 16.0, ve
    child: Container(
      height: 115.0,
      child: Stack(
        children: <Widget>[
          Positioned(
            left: 50.0,
            child: dogCard,
          ),
          Positioned(top: 7.5, child: dogImage),
        ],
      ),
    ),
  );
}
```

The `@required` annotation can be used to declare that certain named arguments must be provided:

```
    class Person {
      String firstName;
      String lastName;
      int age;
      String occupation;

      Person(
          {@required this.firstName,
          @required this.lastName,
          this.age,
          this.occupation = 'unknown'});
    }
```

Failing to provide a `@required` argument normally results in a warning, but that can be easily converted to an error if desired.

Optional parameters, similarly, are wrapped with `[` and `]`:

```
String wrap(String text, [String wrapper = ' ']) =>
    "$wrapper$text$wrapper";

print(wrap("hello")); // prints ' hello '
print(wrap("Dart", "!!!!!")); // prints '!!!!!Dart!!!!!'
```

As you can see in the examples above, both named and optional parameters may be given default values.

Constants

Constants are frozen, immutable values determined at compile-time.

As explained in Const, Static, Final, Oh My!, const objects:

- must be created from data that can be calculated at compile time.
- are deeply, transitively immutable.
- are *canonicalized*. For any given const value, a single const object will be created and re-used no matter how many times the const expression(s) are evaluated.

> *Dart's `static` and `final` keywords behave like in Java.*

All you need to make instances of your own types possibly `const` is to declare its constructor `const`:

```
class Example {
  final String text;

  const Example(this.text);
}
```

> *All of the fields of a `const` type must be final and have `const` constructor themselves.*

It is possible now to create constants of that type:

```
const constExample = Example("Something");
```

The `const` keyword is inferred when calling the constructor in this case… however, when you want to pass a `const` to a method, for example, you need to explicitly call the constructor as below:

```
takesExample(Example ex) => /* TODO */ null;

takesExample(const Example("Something"));
```

This is very common in Flutter code, where constants are used to provide settings, as in this example from flutterexamples.com:

```
Container(
  padding: const EdgeInsets.all(0.0),
  color: Colors.cyanAccent,
  width: 80.0,
  height: 80.0,
),
```

Cascade notation

The cascade notation is not so common in other languages, but it is widely used in Dart as it has proven itself to be really useful!

It allows repeated method calls on the same variable to be expressed more concisely.

For example, you might need to write something like this in Dart Web (using type-safe CSS setters):

```
final element = Element.div();
final style = element.style;
style.width = '50%';
style.height = '4em';
style.padding = '1em';
append(element);
```

With the cascade notation, this can be written as a single expression:

```
append(Element.div()
  ..style.width = '50%'
```

```
    ..style.height = '4em'
    ..style.padding = '1em');
```

Very neat.

Null-safe operators

Even though Dart currently has nullable values (variables of any type can be `null`), Dart has several null-safe operators to make it easier to avoid null pointer exceptions.

> *The Dart team is currently working on non-nullable types that will make the compiler check all access to nullable variables! This feature should be in the next Dart release, and will allow gradual migration from existing code bases! I will udpate this post once it is released.*

For example, it is possible to access nested properties of possibly-null variables using the `?.` operator:

```
print(Person().lastName?.toLowerCase()?.indexOf('a')); // nu
```

This is commonly used together with the `??` operator to use a default value in case something is null:

```
print(Person().lastName?.toLowerCase()?.indexOf('a') ?? -1);
```

As mentioned in the Named Parameters section, it is possible to get the compiler to emit a warning or an error if the value for a parameter is not provided by annotating it with the `@required` annotation.

There's also an operator `??=` for only assigning to a variable if the variables is currently `null`:

```
String s;

void something(String newValue) {
  // if s is null, it gets a new value!
  s ??= newValue;
}
```

Optional dynamic typing

This one may be controversial, but I find it very handy in some situations!

Even if most of your Dart code is type-checked at compile-time, you're still able to write some parts of it, specially those at the boundary of your system (where type checking is impossible, such as where it crosses the network), using dynamic typing.

A prime example of such circumstance is when receiving JSON from a third-party REST API.

Because of the support for dynamic typing, Dart does not force you to create types for every piece of data you might receive (though you can easily do that if you wish, using the powerful built_value package).

For example:

```
import 'dart:convert' show jsonDecode;

main() {
  // simulate a JSON String received from a HTTP request
  final json = '''
  {
    "id": 12345,
    "age": 35,
    "first_name": "Jimmy",
    "hobbies": ["jumping", "basketball"]
  }
  ''';

  dynamic object = jsonDecode(json);

  try {
    if (object["hobbies"]?.contains('jumping') ?? false) {
      print("${object["first_name"]} likes to jump!"); // Ji
    }
  } catch (e) {
    print("Unexpected Json!");
  }
}
```

Declaring a variable as `dynamic` essentially turns off type checking. This can make code much more concise and avoid a lot of boilerplate if used wisely! Having to declare a new class with several fields and nested types just to be able to parse some simple JSON that may change at any time seems wasteful in comparison, and it's not safer in any way.

Of course, passing `dynamic` Objects around in your code would be a terrible idea, but if isolated to small functions as in this example, they can be the best choice from both the readability and maintainability point of view.

Callable Objects

This is a small, but neat feature: objects of a class that exposes a `call` method can be used as functions!

This makes it possible to create “customizable functions” (similar to currying) on the go:

```
class Adder {
  final int initial;

  Adder(this.initial);

  int call(List<int> values) => values.fold(initial, (a, b)
}

main() {
  final add10 = Adder(10);

  print(add10([20])); // 30
  print(add10([20, 30, 40])); // 100
}
```

Like a normal function or method, callable objects may be passed to functions that take other functions as long as the types match:

```
void takesFun(int Function(List<int>) fun) {
  print("Fun: ${fun([50])}");
}

// add10 from the previous example can be used!
takesFun(add10);
```

Collection if/for and spreads

This is the newest, and arguably, coolest and most unique feature of Dart!

Collection- `if` and `for` make it much easier to conditionally add zero-to-many item(s) into a slot of a collection literal.

It should be easy to understand with some examples:

```
// on Windows, prints "[run, .exe]"
// on other platforms, "[run]"
print(['run', if (Platform.isWindows) '.exe']);

final numbers = Iterable.generate(10, (index) => index + 1);

final evenNumbers = [0, for (var n in numbers) if (n % 2 ==

print(evenNumbers); // [0, 2, 4, 6, 8, 10]
```

Notice that these examples may look like for-comprehension from other languages, but they are not quite the same thing!

The `for` and `if` results are spread into the slot of the collection where they are located, producing no value, one value, or many values in that slot.

Similarly, Dart provides a spread operator (`...`), which is used like this:

```
List<int> withBothEnds(int endElement, List<int> list) =>
  [endElement, ...list, endElement];

// prints "Both ends: [42, 1, 2, 3, 42]"
print("Both ends: ${withBothEnds(42, [1, 2, 3])}");
```

This makes some very common code a lot easier to read and write! Specially UI code, which is the reason these features were introduced in the first place.

Here’s a Flutter example showing how the skillful application of spreads can make code a lot nicer:

Example copied from Making Dart a better language for UI.

```
Widget build(BuildContext context) {
  return Row(
    children: [
      IconButton(icon: Icon(Icons.menu)),
      if (!isAndroid) ...[
        Expanded(child: title),
        IconButton(icon: Icon(Icons.search)),
      ]
    ],
  );
}
```

Asynchronous Programming

Asynchronous programming in Dart is absolutely first class.

Using `async/await`, as well as `Stream`s to write asynchronous code is incredibly easy.

Take this HTTP server example, based on the HTTP clients & servers article in the Dart docs:

```
import 'dart:io';

Future main() async {
  var server = await HttpServer.bind(
    InternetAddress.loopbackIPv4,
    4040,
  );
  print('Listening on localhost:${server.port}');

  await for (HttpRequest request in server) {
    request.response.write('Hello, world!');
    await request.response.close();
  }
}
```

> *This is one of my favourite code snippets in any language!*

Notice how `main` is marked as an `async` function… `async` functions are allowed to `await` on `Future`s and `Stream`s… this makes asynchronous code read as easy as synchronous code, but without ever blocking the main thread (which is incredibly important in servers and GUIs, for example), freeing the Dart Event Loop to execute other pending functions that might have been waiting.

Because `HttpServer.bind(...)` returns a `Future<HttpServer>`, we can `await` on it, which suspends the execution of the current function until a value is available from the `Future`… at which time `server` is assigned with the value and execution proceeds.

The line `await for (HttpRequest request in server)` works because `HttpServer` implements `Stream<HttpRequest>`, and in Dart, `Stream` is *a source of asynchronous data events*.

Support for `Stream`s is very rich in Dart, so their use is widespread.

For example, `dart:html` uses `Stream`s to represent user events… `onClick` returns a `Stream` that can be listened to:

```
querySelector('#my-button')
  .onClick.listen((event) => print("User clicked"));
```

A simple way to create a `Stream` is to use the `async*` generator syntax:

```
Stream<int> countStream(int to) async* {
  for (int i = 1; i <= to; i++) {
    yield i;
  }
}
```

For more information about Dart Streams, do read the Streams Documentation as well as the Creating Streams article, as Streams are one of the most important features of Dart!

For even more advanced Streams use cases, see the awesome RxDart package, which implements the ReactiveX API and

> *integrates seamlessly with Dart's native Streams.*

Isolates

Dart code is normally run in a single thread. Even asynchronous code as explained in the previous sections is bound to a single Thread.

But Dart makes it easy to use other Threads via Isolates when necessary.

Isolates are conceptually more similar to different processes than Threads, given that they do not share memory (hence their name) and can only communicate with each other via message passing.

> *Unfortunately, using native `Isolate` s directly is quite cumbersome, so you should probably go with the isolate package instead (which is from the Dart Team), as I did in the example below.*

This example shows how one can run a heavy computation on a different Isolate (likely in a different CPU core):

```
import 'package:isolate/isolate.dart';

heavyComputation(String question) {
  // do some heavy work!
  return 42;
}

void main() async {
  final runner = await IsolateRunner.spawn();
```

```
    final answer = await runner.run(heavyComputation, "What is
    print("The answer is $answer");
    await runner.kill();
  }
```

The `heavyComputation` function is passed to the [`IsolateRunner.run`] method, which executes it in another `Isolate`, passing the given argument, `"What is the answer?"`, to it *on the other side*.

The answer (`42`, evidently) is then returned asynchronously and printed from the main Isolate.

Notice how the same `async/await` pattern is used for running code in another Isolate (but this time, achieving real parallelization!).

Because Isolates do not share memory, any state that is maintained outside of the function's arguments will be separate in the main Isolate and the other Isolate. That is, only the information passed explicitly into the function that is run by the main Isolate is available inside the other Isolate (i.e. a global variable that is set in the main Isolate will not have the same value in the other one!). It's important to understand that, when sending a "message" to an Isolate, the **message is passed by value**. This guarantees memory isolation between Isolates… so, don't be surprised when you make a change to an object in your current Isolate and that change is not reflected on another Isolate you passed the object to - the other Isolate received an independent copy of the object, not the object itself!

> *Aqueduct, a web framework for Dart, handles HTTP requests in parallel by making use of Isolates.*

Dart Platform Features

Now that we've seen lots of cool feature of Dart, the language, it's time to take a look at Dart, the platform.

Multiple compilation modes

Since its inception, Dart has always been a multi-platform language. It can run natively or interpreted on its own DartVM (on Linux, Windows, MacOS, and of course, on iOS and Android with Flutter), or it can run on any browser when compiled to JS, as we saw in the beginning of this post.

We have already encountered the `dart`, `dart2aot`, `dart2js` and `dartaotruntime` commands in the introduction… but when actually working on a larger Dart project, you're likely to interact more directly with `pub`, `webdev` (if working on web), and `flutter` (for mobile). You might also like to use the `dartanalyzer`, as we'll see.

> *The IntelliJ IDE (and its sibling, Android Studio) as well as VSCode have great support for running the Dart tools automatically, but it is good to know exactly how these tools work even if you don't need to run them manually often.*

pub, the Dart Package Manager and build system

Pub can be seen as the Dart package manager and build system. It underlies the tools for all platforms.

Most of the time, you'll just use it to interpret your build file, pubspec.yaml and download dependencies from a Dart Packages repository (normally, pub.dev but anyone can run a pub_server).

> *If you have a Google Account, you can use that to publish packages to pub.dev!*

Here's an example pubspec.yaml file:

```yaml
name: my_package
description: A sample command-line application.
version: 1.0.0
homepage: https://www.example.com
author: Me <me@example.com>

environment:
  sdk: '>=2.1.0 <3.0.0'

dependencies:
  path: ^1.4.1

dev_dependencies:
  pedantic: ^1.0.0
  test: ^1.0.0
```

> *Check this blog post to learn about the `pedantic` library, an opinionated set of lints for Dart.*

Pretty self-explanatory… this is basically what the Dart scaffolding tool, `stagehand`, will generate for you (run `pub global activate stagehand`, then `stagehand console-full`, for example, to generate a new Dart application with a pubspec file and some simple Dart sources already in place).

Run `pub get` to download the dependencies.

> *Flutter developers usually interact with the `flutter` tool directly, not with `pub` … for example, Flutter developers should run `flutter pub get` rather than just `pub get`. See the Flutter Tutorial for more information.*

Dart source code is expected to be placed in different directories depending on its purpose:

- `bin/` - for CLIs and other runnable applications.
- `lib/` - for shared packages.
- `web/` - for web applications.
- `test/` - for all tests.
- `example/` - example source code (shown in the package documentation in pub.dev).

Running code on the Dart VM (interpreted or native)

To execute your application entry point (a `.dart` file with a `main` function) using the Dart VM:

```
$ dart bin/main.dart
```

> *You can make your package easily executable by other Dart users by specifying the executable files on your pubspec.yaml file, see the pubspec docs for more information.*

To pre-compile it to machine code:

```
$ dart2aot bin/main.dart main.dart.aot
```

Then, you can run it with the minimal Dart native runtime, `dartaotruntime` (only around 5MB):

```
$ dartaotruntime main.dart.aot
```

> *To create a minimalistic Docker image for your AOT-compiled Dart application, check out this Gist. This Dockerfile builds a Docker image based on the `bitnami/minideb` (mini Debian) image, which together with the `dartaotruntime` tool, takes up only around 55MB.*

If you don't mind using the full `dart` tool to run your app, you may achieve a higher peak-performance if you create an app snapshot instead of using `dart2aot`:

```
$ dart --snapshot=main.dart.snapshot
      --snapshot-kind=kernel
```

```
            bin/main.dart
```

The snapshot is run with the simple `dart` tool:

```
$ dart main.dart.snapshot
```

> *Dart may soon gain an extremely strong framework for building desktop apps with rich user interfaces, as Flutter is coming to desktop and to the web!*

Running code in the browser

Even though you can easily use `dart2js` to compile Dart directly to JS, if your focus is to develop web applications, you should definitely use the `webdev` tool.

It takes care of re-building your app and bundling assets as needed, and has a live-reloading web server.

> *Check my blog post on Using Sass and Bulma in Dart Web Projects to see how you can easily setup a comfortable web development environment.*

To build your app, in release mode, into the `build/` directory, run:

```
$ webdev build
```

To serve it locally with hot reloading (currently does not keep state, as in Flutter hot reload - but this should be fixed soon, we hope):

```
$ webdev serve --auto=restart
```

Running code on mobile

Finally, running Dart on mobile, be it Android or iOS, is as easy as it gets with Flutter!

Once you're setup, all you need to do is:

```
$ flutter run
```

Check the Flutter docs for details.

Hot reloading

In all platforms, Dart supports hot code reloading. We've seen above how to use that on the web… on Flutter, it works out-of-the-box and is one of the strong points of the framework!

Here, we'll just see how to hot reload Dart code directly on a running Dart VM.

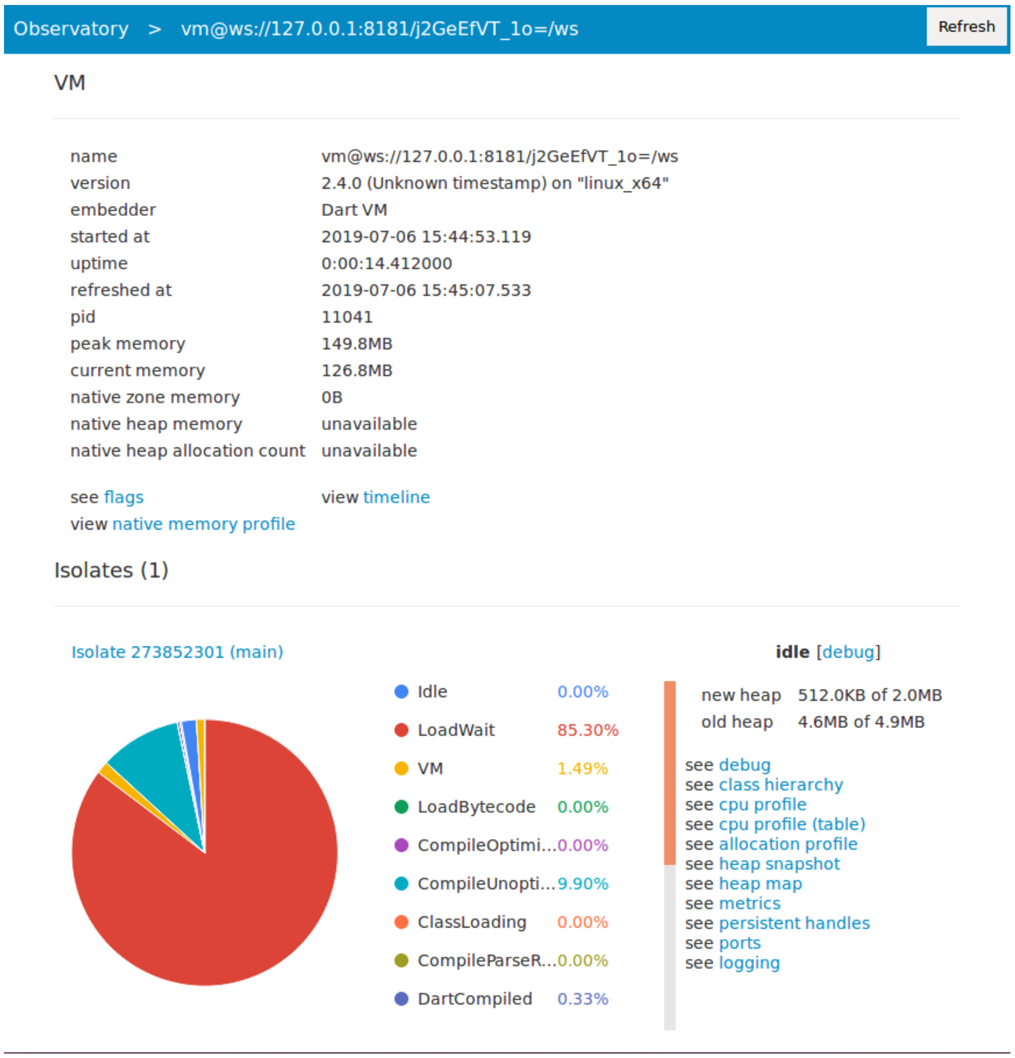
I've written a little sample CLI app to show how that works:

```
import 'dart:async';
import 'dart:convert';
import 'dart:io';

String prompt = '>> ';

void main() async {
  stdout.write("Echo program! Enter 'quit' to exit.\n$prompt
  await for (final line in readLine()) {
    final quit = processLine(line);
    if (quit) break;
  }
}

Stream<String> readLine() =>
    stdin.transform(utf8.decoder).transform(const LineSplitt

bool processLine(String answer) {
  if (answer == 'quit') {
    print("Goodbye!");
    return true;
  } else {
    stdout.write("echo: $answer\n$prompt");
    return false;
  }
}
```

To enable hot reloading (and the debugger), run the program with the `--observe` flag:

```
$ dart --observe bin/main.dart
Observatory listening on http://127.0.0.1:8181/j2GeEfVT_1o=/
```

When you open the given URL on your browser, you should see the Dart Observatory:



VM

name	vm@ws://127.0.0.1:8181/j2GeEfVT_1o=/ws
version	2.4.0 (Unknown timestamp) on "linux_x64"
embedder	Dart VM
started at	2019-07-06 15:44:53.119
uptime	0:00:14.412000
refreshed at	2019-07-06 15:45:07.533
pid	11041
peak memory	149.8MB
current memory	126.8MB
native zone memory	0B
native heap memory	unavailable
native heap allocation count	unavailable

see flags   view timeline
view native memory profile

Isolates (1)

Isolate 273852301 (main)   **idle** [debug]

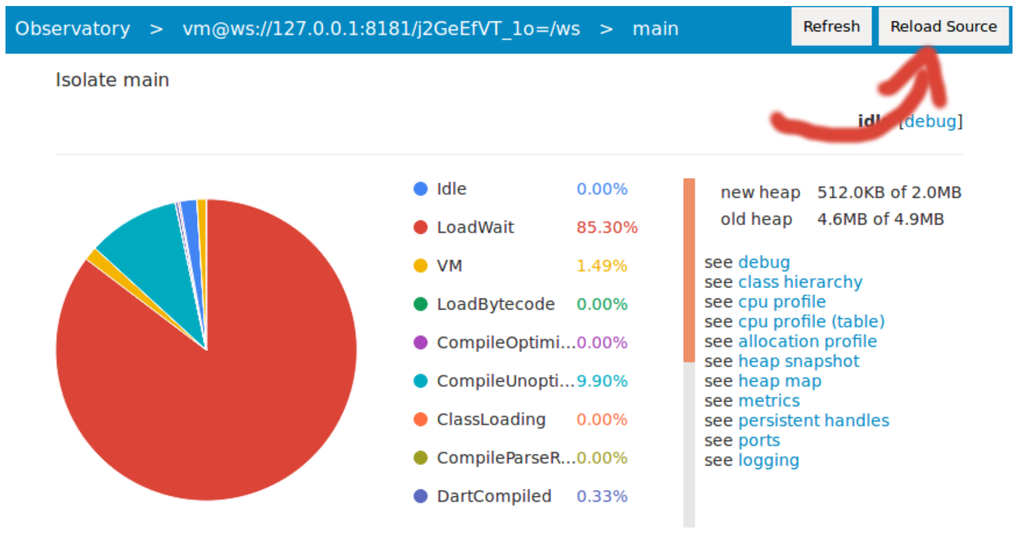
● Idle	0.00%
● LoadWait	85.30%
● VM	1.49%
● LoadBytecode	0.00%
● CompileOptimi...	0.00%
● CompileUnopti...	9.90%
● ClassLoading	0.00%
● CompileParseR...	0.00%
● DartCompiled	0.33%

new heap	512.0KB of 2.0MB
old heap	4.6MB of 4.9MB

see debug
see class hierarchy
see cpu profile
see cpu profile (table)
see allocation profile
see heap snapshot
see heap map
see metrics
see persistent handles
see ports
see logging

Work is currently in progress to replace the Observatory with something even better: DevTools, a suite of performance and debugging tools for Dart and Flutter!

Click on the `main` Isolate link, above the chart (or use the breadcrumb at the top), and you'll see the Isolate's screen:



● Idle	0.00%
● LoadWait	85.30%
● VM	1.49%
● LoadBytecode	0.00%
● CompileOptimi...	0.00%
● CompileUnopti...	9.90%
● ClassLoading	0.00%
● CompileParseR...	0.00%
● DartCompiled	0.33%

new heap	512.0KB of 2.0MB
old heap	4.6MB of 4.9MB

I added a big, red arrow to show where you need to click to reload the code.

Make changes to the Dart script, click on the "Reload source" button, and see how they reflect immediately on the CLI.

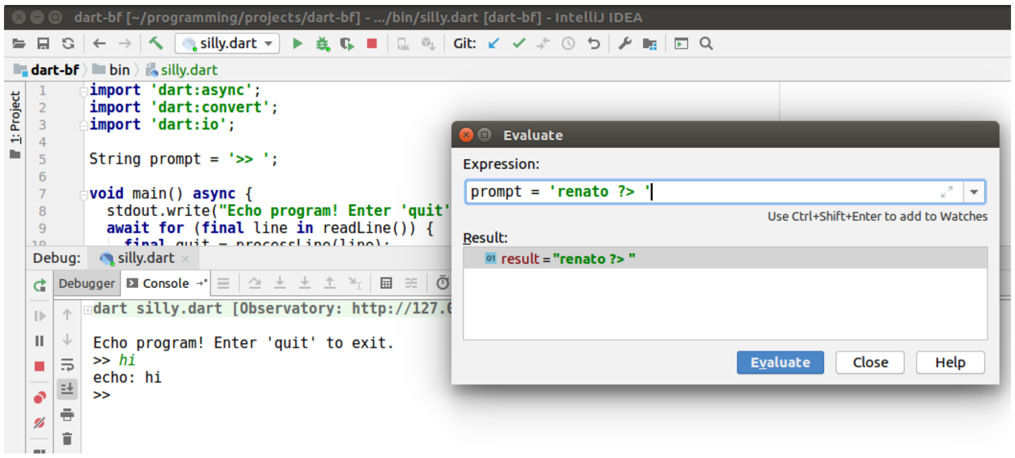
Another way of reloading the code is by entering the `reload` command in the Debugger Screen.

Pretty much anything can be modified in the code… but one thing that does not change is the state of existing objects… for example, try changing the `prompt` variable in the script:

```
String prompt = 'Me >? ';
```

After reloading, the prompt will not be changed in the running CLI… to change that, you need to actually run code that sets the variable to something else…

The Dart Observatory is able to evaluate expressions, but it is a bit buggy on FireFox from what I tried, working better on Chrome (this may cause some raging reactions - but calm down, it's very likely that the Observatory just doesn't receive much attention from the development team, as they're pretty focused on bringing in some cool new features to the language for now)… but I found out that the IntelliJ debugger can evaluate expressions without problems (on the other hand, it does not seem to be able to hot reload! Can someone please fix that!?).


dart-bf [~/programming/projects/dart-bf] - .../bin/silly.dart [dart-bf] - IntelliJ IDEA
silly.dart
Git:
dart-bf
bin
silly.dart
1: Project
Debug:
silly.dart
Debugger
Console
Evaluate
Expression:
Use Ctrl+Shift+Enter to add to Watches
Result:
Evaluate
Close
Help

```
import 'dart:async';
import 'dart:convert';
import 'dart:io';

String prompt = '>> ';

void main() async {
  stdout.write("Echo program! Enter 'quit'
  await for (final line in readLine()) {
```

```
dart silly.dart [Observatory: http://127.

Echo program! Enter 'quit' to exit.
>> hi
echo: hi
>>
```

```
prompt = 'renato ?> '
```

```
result = "renato ?> "
```

I can't comment on [DartCode, the VSCode plugin for Dart/Flutter](https://renato.athaydes.com/posts/interesting-dart-features.html) as I've only tried it briefly (and prefer JetBrains products in general), but I've seen some talks by the Dart team where they use that, so it's probably good!

It's also possible to evaluate code in the Observatory's debugger, as we'll see in the next section.

It's not too hard to hook into the Observatory API and add framework-specific hot reloading capabilities… the Angel Web Framework authors did that, and it's pretty cool.

The `jaguar_hotreload` package makes auto-hot-reloading as easy as two lines of code before your `main` function:

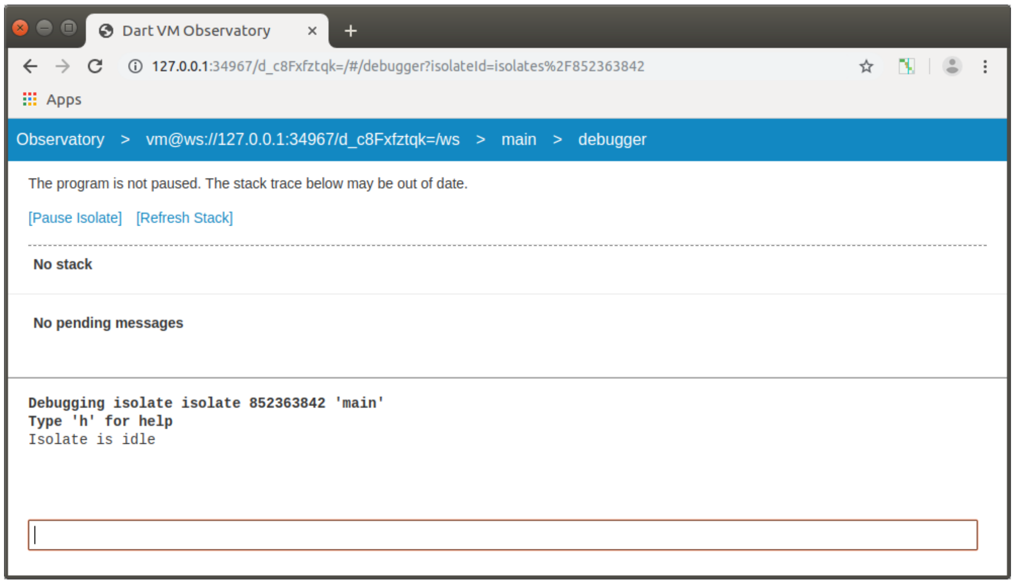
```
import 'package:jaguar_hotreload/jaguar_hotreload.dart';

main() async {
  final reloader = HotReloader()..addPath('.');
  await reloader.go();

  // TODO your code here!
}
```

Rewinding debugger

This is a really powerful feature of the Dart debugger: the capacity of rewinding code execution, going back to a previous point in the program.

> *Unfortunately, it seems that, as of writing (July 2019), the only way to do this is via the Observatory's debugger… neither IntelliJ's nor VSCode's debuggers seem to have this feature yet.*

To illustrate how this works, let's consider the example of the previous section. Run the CLI with the Observatory enabled, then open the main Isolate and go to the debugger.


Dart VM Observatory
127.0.0.1:34967/d_c8Fxfztqk=/#/debugger?isolateId=isolates%2F852363842
Apps
Observatory > vm@ws://127.0.0.1:34967/d_c8Fxfztqk=/ws > main > debugger
[Pause Isolate] [Refresh Stack]

The program is not paused. The stack trace below may be out of date.

No stack

No pending messages

```
Debugging isolate isolate 852363842 'main'
Type 'h' for help
Isolate is idle
```

Add a breakpoint at the `processLine` function:

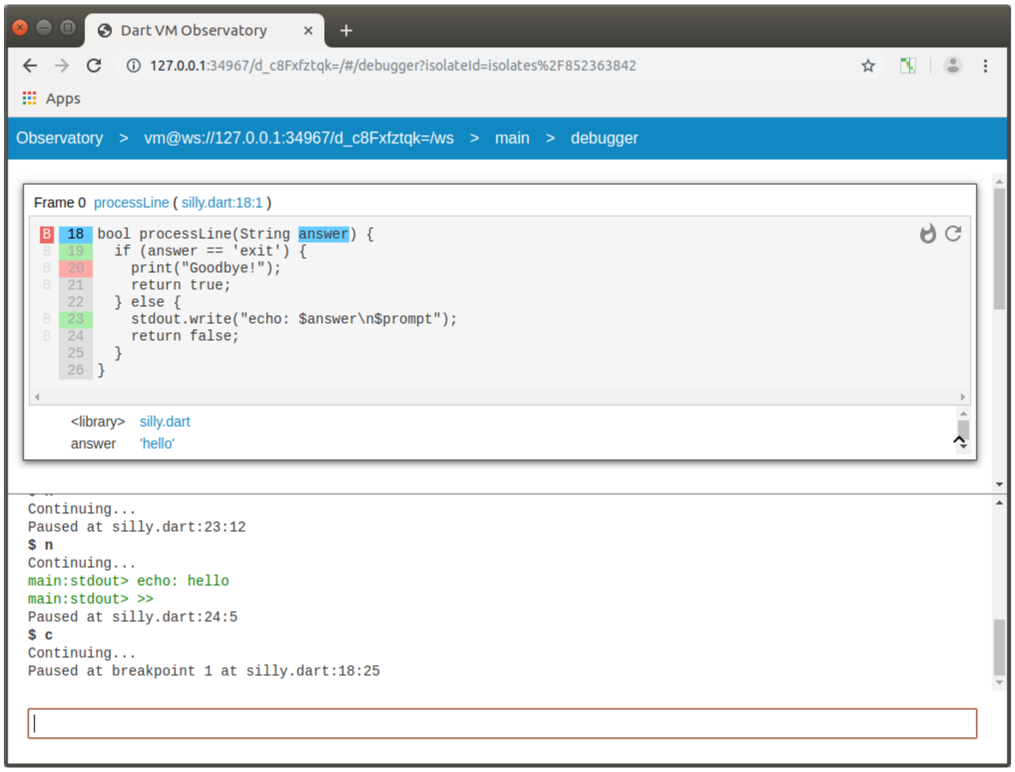
```
> break processLine
```

Tab completion, as well as up/down history navigation, work in the debugger terminal!

Then, from the app's shell, enter something…

```
Echo program! Enter 'quit' to exit.
$ >> hello
```

The debugger should stop at the `processLine` function:


Dart VM Observatory
127.0.0.1:34967/d_c8Fxfztqk=/#/debugger?isolateId=isolates%2F852363842
Apps
Observatory > vm@ws://127.0.0.1:34967/d_c8Fxfztqk=/ws > main > debugger

Frame 0 processLine (silly.dart:18:1)

```
bool processLine(String answer) {
  if (answer == 'exit') {
    print("Goodbye!");
    return true;
  } else {
    stdout.write("echo: $answer\n$prompt");
    return false;
  }
}
```

```
<library>   silly.dart
answer      'hello'
```

```
Continuing...
Paused at silly.dart:23:12
$ n
Continuing...
main:stdout> echo: hello
main:stdout> >>
Paused at silly.dart:24:5
$ c
Continuing...
Paused at breakpoint 1 at silly.dart:18:25
```

Notice the **blue** highlighting on `answer`. That indicates the next statement or expression to be executed.

You can enter commands like `s` (step), `c` (continue, resume execution), `p` (print and evaluate expression) and others… enter `h` (help) to see the full list of commands.

But now, suppose you want to redefine the `processLine` function at this point of execution. It will not work because you're already inside the execution stack of the existing function definition!

To do that, you need to `rewind` to the previous frame. This will bring you back to the point where `processLine` is called, so that you can modify its definition, `reload` the code as we saw in the previous section, then `s` (step) again into `processLine`. This time, the new definition of `processLine` will be executed!

Notice that you can `rewind` as many times as you want, running the same function again and again, until you get it exactly right!!

Code generation

Code generation is a feature that offers a great alternative to complex things like macros and runtime reflection.

I like it because it lets us write powerful things and cut off on boilerplate virtually anywhere, without losing the ability to actually see the actual code that's going to run, and follow it without magic black boxes in between.

Dart builders were designed to solve this problem. They are so flexible that they are used for all sorts of things, from transpiling and minifying Dart-to-JS code to converting and bundling `scss` files to `css`, compile-time meta-programming (e.g. MobX observables, value types, on-demand reflection), serialization, and even auto-generating React components' boilerplate.

It's quite simple to use builders! For example, `json_serializable` shows this example in their documentation, to make a Dart class JSON-serializable:

```
import 'package:json_annotation/json_annotation.dart';

part 'example.g.dart';

@JsonSerializable(nullable: false)
class Person {
  final String firstName;
  final String lastName;
  final DateTime dateOfBirth;
  Person({this.firstName, this.lastName, this.dateOfBirth});
  factory Person.fromJson(Map<String, dynamic> json) => _$Pe
  Map<String, dynamic> toJson() => _$PersonToJson(this);
}
```

Generated Dart files, by convention, are named with the `<file>.g.dart` format, where `<file>` is the name of the file from which it was generated… Dart files include the generated files via the `part` instruction (and the generated file contains a `part of` instruction, so it's easy to go both ways, from/to source/generated files).

Generated classes and functions are normally named with a `_$` prefix, hence the `_$PersonFromJson` function call above.

The generated code for the above class looks like this:

```
part of 'example.dart';

Person _$PersonFromJson(Map<String, dynamic> json) {
  return Person(
      firstName: json['firstName'] as String,
      lastName: json['lastName'] as String,
      dateOfBirth: DateTime.parse(json['dateOfBirth'] as Str
}

Map<String, dynamic> _$PersonToJson(Person instance) => <Str
      'firstName': instance.firstName,
      'lastName': instance.lastName,
      'dateOfBirth': instance.dateOfBirth.toIso8601String()
    };
```

Pretty basic, but saves developers from having to write quite a lot of error-prone, tedious code… while still being debugger-friendly.

Platform-specific testing

Last but not least, Dart has very good support for running tests in all of the platforms that it supports.

I highly recommend watching the video Dart for the Web: State of the Union (Dart Developer Summit 2015), where they demonstrate tests running simultaneously on the Dart VM, Dartium, Chrome, Firefox and even Safari!

And it's really easy, you just tell `pub` in which platform(s) to run the tests:

```
$ pub run test
    -p chrome
    -p firefox
    -p safari
    -p vm
    -p content-shell
    -p dartium test/frontend/matcher/
```

The list above is taken from the video, and seems to be a little out-of-date… check the test package docs for the (huge!) list of targets supported.

If you only want to run certain tests on some platforms, you can declare that in the test file. For example, here's a part of one of the tests I wrote for a little web widgets library I was writing:

```
@TestOn('browser')
import 'dart:html';

import 'package:flattery/flattery_widgets.dart';
import 'package:test/test.dart';

main() async {
  group('Simple Widget', () {
    Text textBox;
```

```
      setUp(() {
        textBox = Text('')
          ..root.id = 'hello'
          ..text = 'Hey there';
        querySelector('#output').append(textBox.root);
      });

      test('can be added to a HTML document, then removed', ()
        var helloEl = document.getElementById(textBox.root.id)
        expect(helloEl, isA<DivElement>());
        expect(helloEl.text, equals('Hey there'));

        textBox.removeFromDom();

        helloEl = document.getElementById(textBox.root.id);
        expect(helloEl, isNull);
      });
    });
  }
```

Check the [Flutter Docs] for details on testing Flutter apps.

`@TestOn('browser')` means that this test will only execute when running in one of the supported browsers.

It's even possible to write simple expressions to define where the test should run:

```
@TestOn("browser && !chrome")
```

Very powerful!

Conclusion

I hope that this blog post convinces at least some people that Dart is an incredibly versatile language and platform to develop on, and clears up a few misconceptions that seem to be common due to Dart's past (before Flutter?!).

It seems to be the first truly multi-platform language in the sense that it can be used on all big desktop OS's, on all modern mobile and web platforms, and can actually deliver beautiful user interfaces in all of them, not only backend/CLIs (I've heard that Flutter can even be used on embedded devices - like parking ticket terminals!).

Java got close in the early 2000's, but not quite there… and the only other languages that can claim this kind of feat are probably Haxe (but I haven't heard much about it and didn't get the chance to try it yet) and JavaScript, via myriads of third-party projects (so not really first class, like Dart).

Misconceptions

But Dart is dead!!!

No, it really isn't… Google is pouring resources into it and the language is rapidly evolving.

It is being used by more and more companies (even if Google still is the heaviest user), and the community is certainly growing at a rapid pace.

Dart is verbose like Java!

The list of features above should quickly disprove that Java and Dart are basically the same… even though Java is also evolving (a lot faster in the last couple of years than in the previous decade, in fact), Dart has, right now, a lot of things that help make developer's lifes easier… and with non-nullable types coming in the next few months (fingers crossed), it really is a nicer language to work with, and should be compared to Kotlin and Swift, if anything.

Dart is as slow as Python.

Regarding speed: Dart 2.4 (current version) is really fast! It runs my little bf interpreter in just under 2.5 seconds when compiled with `--snapshot-kind=kernel` , in the neighbourhood of languages like Nim, D and Go! Java was a fair bit slower with the default JIT (though with the new GraalVM JIT, it managed a blazing fast time of 1.4 seconds).

From a quick and dirty wrk benchmark I ran using Java Jetty VS Dart with the `http` package to handle lots of HTTP requests to serve a local file, Dart actually, to my surprise, came ahead!

So, while it's not possible to say that Dart beats Java in speed, it's absolutely fair to say that it's in the same league as Java, and by extension, even languages like Go and Nim.

Dart tooling is poor in comparison to JS or Java

I am a big fan of great tooling and find it hard to leave the comfort of Java IDEs like IntelliJ. I really care about great tooling, so let me tell you that Dart's tooling is actually amazing! It's one of the very few languages that actually gets close to Java in this regard.

As I've shown above, it has a great debugger/profiler, the Observatory… excellent support in IntelliJ (and WebStorm for web developers) and VS Code.

And due to the AOT compiler, it can be easily deployed with minimal fuss, using a tiny runtime.

The Dart tooling is actually one of its greatest advantages!

Google will drop it when it gets bored

The fact that a behemoth like Google is behind Dart is both a blessing and a curse.

Languages that do not have a huge corporation behind them tend to lack even basic tooling. Things like debuggers and IDE support require a lot of hard work and are not very interesting. It’s just not the kind of things compiler engineers like to work on in their free time.

Dart has an army of engineers to throw at the problem. And they have experience doing so as Dart is not their only language, and they have thousands of users in-house.

I know that Google takes a lot of beating for dropping projects like hot potatoes, but there seems to be zero evidence that they are going in that direction with Dart, quite the opposite. A lot of people think they should be using Kotlin in Flutter, and if they did that, maybe Dart would be in danger… but look at the new features Dart has been getting, then compare them against Kotlin. I can only think of a couple of features that really make Kotlin look more attractive than Dart (non-nullable types and extension functions - the former is almost certainly coming in the next release, the latter might come already in the release after - as the JS team needs it badly for supporting certain JS idioms). But they are not really enough, I think, to justify switching languages, or even adding an alternative language, to Flutter.

I actually work with Kotlin professionally and like it a lot, and am only a Dart enthusiast. I have nothing to win in this race… I am just trying to express why I think Google is right in sticking with Dart on Flutter. It’s a great language for the job, and they can mold it as they evolve both projects to fit their needs. For those of us on the user side, all that we should care about is that we get a good language and platform to work on: and that, we definitely get with Dart at the moment. The marginal improvement that might be obtained with another language would be just a drop in the ocean.

Finally, Dart is completely open source, and all of its tooling is also, as far as I know. And the Dart Team seems to actively listen to, and seek feeback from, the community. So I am confident that even if Google did one day decide to move away, Dart won't be dead. Specially with Flutter relying on it - there's money to be made in this space and if Google, after doing all the hard work, decides to drop that, I am sure someone else will be eagerly waiting to take it over.

Thanks for reading.

If you want to let me know what you think, ping me on Twitter.

I'm on keybase.io
And on GitHub
Twitter: @renatoathaydes