ESSENTIALS OF DART

Henry Nguyen
Department of Computer Science
University of Arizona
Tucson, AZ 85721
henryn098@email.arizona.edu

Adam Cunningham
Department of Computer Science
University of Arizona
Tucson, AZ 85721
laser@email.arizona.edu

April 30, 2020

ABSTRACT

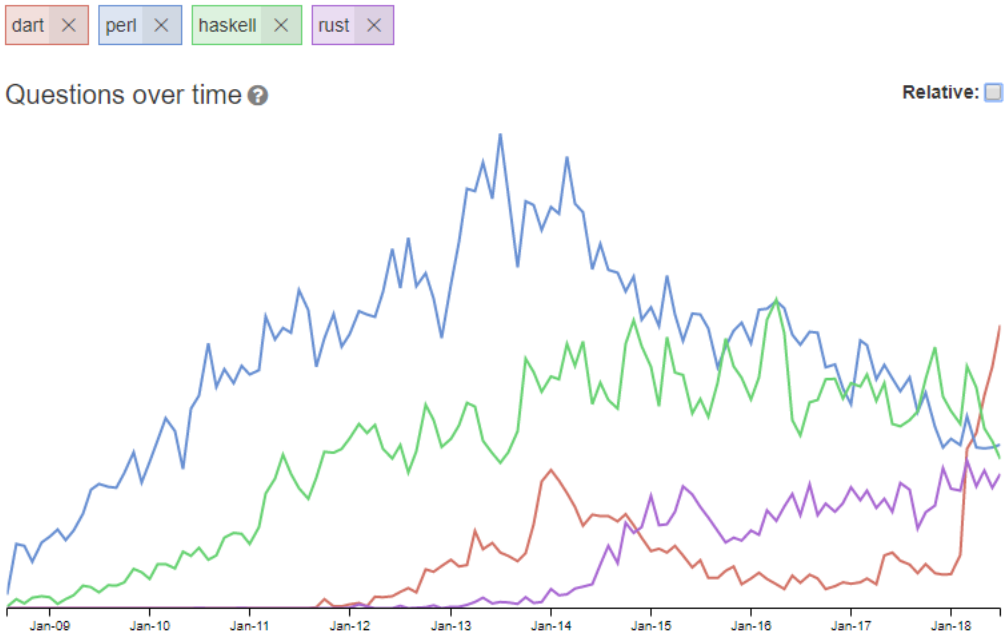
A programming language released by Google in 2011 provides features such as type analysis and object orientation. It is used to create cross-platform apps that saves companies development time and lowers their technical debt. Coupled with Google's UI framework, Flutter, it is apparent that Dart is becoming competitive with languages such as JavaScript when it comes to building web apps. In addition to having the ability to replace JavaScript, Dart has interesting features within the domain of OOP such as callable classes, typedefs and mixins that make it much more than a simple language whose sole purpose is to run in an internet browser.

1 Introduction

Before 2011, JavaScript was the dominant programming language for building web apps. Many users became frustrated with JavaScript and wanted support for features such as type analysis. Then other programming languages such as Elixir and Kotlin came out that same year. But Dart had features such as cross-compatibility and the ability to compile down to JavaScript. Dart did not gain traction until the release of Flutter, a UI framework, in 2017.

2 History

Dart first appeared in 2011 with Elixir and Kotlin. A year later, TypeScript was released by Microsoft. The language was essentially designed to solve the frustrations of JavaScript at the time. Created at Google by Lars Bak and Kasper Lund, Dart is an optionally typed object-oriented language. Dart can act as a superset of JavaScript with the dart2js compiler which included optional static type analysis [1]. Dart is known in the community as the JavaScript killer [2], partly due to the large increase in popularity of Flutter, Google's UI framework for building native interfaces in iOS and Android.



The recent popularity of Dart, coupled with its ongoing support and use in upcoming Google projects makes Dart a compelling language

Flutter is a major project that allows AOT (Ahead of time) and JIT (Just in Time) compilation. This leads to innovative development with hot-reload and a smoother UX [3]. In addition, Flutter is able to render 120 fps on supported devices. Flutter is ran via the Dart platform, and Google is continuing to build and support Dart and Flutter. In 2016, Google began using Dart to create a new operating system, Fuchsia, which is rumored to possibly replace the Android operating system [4].

3 Control Structures

A control structure is a building block in computer programming that allows the direction of a program to change upon given parameters. Languages such as C and Java have identical syntax for the basic control structures in Dart.

3.1 Loops

3.1.1 For loop

A for loop is used to iterate for a fixed number of times, typically for the purposes of counting or traversing a data structure.

```
1 //do something 10 times
2 for (int i; i < 10; i = i+1){
3     //do something
4 }
```

3.1.2 While loop

A while loop is used to do something a possibly unknown number of times. The conditional must be a statement which evaluates to true or false. All for loops can be written as while loops. The following calls a function which returns randomly "heads" or "tails". The loop will continue to run until the result is "heads";

```
string heads = "heads";
string result = "";

while (result != "heads") { //executes until result is heads
    result = CoinFlip();
}
```

3.1.3 Do-while loop

Sometimes a user may want the loop to always be entered at least once, then evaluated. This can be accomplished with the do-while syntax

```
do {
    //do something at least once
} while (cond...)
```

3.1.4 For-in loop

A for-in loop is used to iterate over some structure until the end of that structure has been reached. For example, the following code iterates over a list of numbers and sums them together.

```
sum = 0;
for (int i in list){
    sum += i;
}
print(sum);
```

3.1.5 Await-for loop

The await-for loop is used to iterate over a stream of events asynchronously. More can be read about streams in Data Types. "Asynchronous for loop (commonly just called await for) iterates over the events of a stream like the for loop iterates over an Iterable" (dart.dev).

```
Future<int> sumStream(Stream<int> stream) async {
    var sum = 0;
    await for (var value in stream) {
        sum += value;
    }
    return sum;
}
```

3.2 Conditionals

A switch statement is identical to a series of if statements, which takes some value and checks its equality with any number of cases. An error is thrown if a case does not break, but this can be caught, allowing the code to fall through to the next case.

```
switch (val)
case 1:{
    // if val is equal to the integer 1, this portion of the code will execute
    break;
    }
case 2:{
    // if val is equal to 2, this executes.
    break;
}
default:{
    // this always executes if the switch has not been broken out of
}
```

3.2.1 If-else

If-else statements evaluate blocks of code on a specified condition. The else statements are not necessary, and the default implied else is to do nothing.

```
if (cond) {
    print("do something");
} else if (cond){
    print("do something else");
} else{
    print("default");
}
```

3.2.2 Break and Continue

Break and continue statements are used inside loops to alter the normal flow of a program. Break is used to bring the flow of the program outside of the current loop. On the other hand, continue is used to skip the remaining portion of the while loop, and continue back to the beginning, or to a specified label within the loop. Break and continue are similar to goto in other languages. Similar to goto, misuse of break and continue can result in spaghetti code. It is recommended that these be used sparingly if at all. The following would print the numbers 1,2,3...9 (without commas, each on their own line).

```
int i = 0;
while (true){
    i++ //this is equivalent to i = i+1
    if (i < 10){
        print(i);
        continue; //skips the break and restarts the loop
    }
    break; //exits the loop
}
```

3.2.3 Try/Catch

```
try {
    7 ~/ 0
}
on IntegerDivisionByZeroException {
    print('Cannot divide by zero');
}
catch (e) {
    //this is a catch-all block
}
finally {
    print("Dart is cool!");
}
```

3.3 Misc.

A return statement is similar to continue, but it causes the function to be evaluated to the expression after the call to return. The default return value in Dart is null. The following function will evaluate to true.

```
returnTrueFunc(){
    return true;
}
```

4 Data Types

Dart is a object-oriented and statically-typed language and hence, every variable is an object and the type of a variable must be declared by the user. During static analysis, the data type of a variable is inferred from its initial value (section 4.2) [5]. Type argument inference provides the user with an alternative syntax when declaring a variable; helpful towards convoluted variable declarations. Contrary to other languages such as JavaScript, a variable that is declared with the *var* keyword must keep the same data type during reassignment; failing to do so results in a compilation error. On the other hand, the *dynamic* keyword allows a variable to be reassigned to a different data type i.e. type checking is "disabled" during compilation. This mechanism can lead to type safety issues but it is helpful if your data type becomes *unpredictable* during its lifetime. This results in slower performance because any operation involving that variable must be checked at runtime to ensure that it is legal. Below are the basic data types in Dart.

4.1 Integer

Unlike other languages such as Java where integers are a fixed-size of 32 bits, it varies in Dart. When compiled on the VM, Dart 1 uses "infinite-precision integers (aka bigints)". A BigInt is a built-in JavaScript object that is used for whole numbers larger than $2^{53} - 1$. According to the official Github documentation, "almost every number-operation must check if the result overflowed, and if yes, allocate a next-bigger number type. In practice this means that most numbers are represented as SMIs (Small Integers), a tagged number type, that overflow into "mint"s (medium integers), and finally overflow into arbitrary-size big-ints". This design decision led to users often bit-anding their numbers to "ensure that the compiler can see that a number will never need more than a SMI" e.g. Jenkin's hash function used in Flutter's engine [6]. Upon the release of Dart 2, the size of int was switched to represent a 64 bit two's complement integer [7]. Using dart2js gives a double precision floating point number in the range of $[-2^{53}\ to\ 2^{53}]$ since that is the only number type that JavaScript supports [8].

```
int n = 8;
```

4.2 Doubles

```
double myNum = 8; // inferred as 8.0
```

4.3 Numbers

A number is either an integer or a double.

```
num x = 3;
x = 6.9;
```

4.4 Strings

In programming languages such as C and Java, Strings are an array of characters but in Dart, the character type does not exist and a String is an immutable "sequence of UTF-16 code units". This design allows users to create Strings containing foreign letters, mathematical symbols and emojis. Strings can be initialized with matching single or double quotes and can be multiline. [9].

```
String s1 = "goodbye";
String s2 = '''
This is my multiline string!
'''

// print out list of code units
String s = "hi there;

print(s.codeUnits); // prints [104, 105, 32, 116, 104, 101, 114, 101]
```

4.5 Boolean

A boolean is a true or false value. When a data type such as a String is invoked in a boolean expression, boolean conversion occurs. [10]

```
bool val1 = true;
bool val2 = false;

String s1 = "hi";
String s2 = "there";
print(s1 && s2); // false, because boolean conversion turns s1 into false
```

5 Subprograms

Modularity and clarity are provided in other object-oriented languages through classes and objects and Dart is no different in this aspect.

5.0.1 Classes

Akin to Java, all classes extend Object and class members/methods can be invoked with the object name followed by a period.

```
class Person {
    String first;
    String last;

    Person(String first, String last) {
      this.first = first;
      this.last = last;
    }

    // Alternative constructor syntax
    Person(this.first, this.last);
}

Person p = new Person();
```

5.1 Subroutines

Subroutines can be defined as a set of instructions that perform a task, e.g. a function. In OOP, subroutines are attached to classes and Dart follows this exact same format. Subroutines can be included from libraries as well (see section 5.3.1) and this allows users to "plug n' play" with frameworks right out of the box so no development time is wasted. Subroutines are not bound to the built-in features of a language i.e. if a feature is not provided out of the box, users can download libraries from Github repositories. Google Map View is an open source Flutter library provided by App Tree Software that users can easily download and use in Flutter apps. Dart is a compelling language that is backed by a large and supported community which allows the scope of shared subroutines to be limitless.

5.1.1 Basic Subroutine

A function is a type of subroutine that is an object of type *Function*. Functions in dart can take in optional named, positional and default parameters as arguments. Named parameters allow the user to change the order of the passed in arguments. While positional parameters allow users to optionally provide a parameter to use; this is similar to method overloading in Java. Default values give a specified value to a parameter if one is not provided. [11]

```
void sayHello(String name) {
    print("Hello, ${name}!");
}

// Type annotations can be omitted although it is not recommended
void sayHello(name) {
    print("Hello, ${name}!");
}

// Named parameters
void printName({String first, String last}) {
  print("Your name is ${first} ${last}"); // prints "Your name is Bob Builder"
}

printName(last: "Builder", first: "Bob");

// Positional parameters
void printName(String first, String last, [String middle]) {
  print("Your name is ${first} ${last}");
}

printName("Bob", "Builder", "The");

// Default parameter
void printName(String first, String last, {String middle="???"}) {
  print("Your name is ${first} ${middle} ${last}"); // prints "Your name is Bob ??? Builder"
}

// If a 3rd parameter is provided, ??? is overridden
printName("Bob", "Builder");
```

5.1.2 Mixins

Before Java 8, a class can inherit properties and methods of a base class. However, this leads to the diamond problem if that class wants to extend another class. To remedy that interfaces were created but interfaces forced classes to implement methods. But, interfaces were not allowed to have default implementations and thus, Mixins were born. Mixins are "a way of reusing a class's code in multiple class hierarchies". [12]. In other words, Mixins can be thought of as more flexible interfaces; similar to Java 8+ interfaces.

5.1.3 Typedefs

Type definitions in Dart allow the user to define an alias for a function. Not to be confused with typedefs in C that are used to define an alias for a data type. A declared typedef's function signature must match the function signature of the given function which is defined by its parameters and their respective types. [13]

```
typedef MathOp(int num1, int num2);

int sum(int a, int b) {
    return a + b;
}

int product(int a, int b) {
    return a * b;
}

MathOp myOp;

myOp = sum; // myOp now points to the sum function
myOp(1, 1); // returns 2

myOp = product; // myOp now points to the product function
myOp(5, 2); // returns 10
```

5.1.4 Callable Classes

A callable class is a class than can be treated as if it were a function. This is helpful in debugging purposes but the call method must implemented. [11]

```
class Point {
    String call(int x, int y) {
        print("(" + x.toString() + ", " + y.toString() + ")");
    }
}

Point p = new Point();
p(5, 3); // prints (5, 3)
```

5.1.5 Generator Functions

Generator functions are used to generate a sequence of values. As a result of behaving like an iterator, generator functions can be stopped and only used to provide the next value in the sequence. This allows the user to only generate what's needed at the moment. This is useful if a function needs to be suspended and a "placeholder" is needed to keep track of a position in a sequence. The two types of generators in Dart are synchronous and asynchronous. Synchronous generators return an Iterable object while on the other hand, asynchronous generators return a Stream object. [14]

Synchronous Generator

```
Iterable<int> syncGen(int n) sync* {
    while (n > 0) {
      yield n--;
    }
}
```

Asynchronous Generator

```
Stream<int> asynchGen(int n) async* {
  while (n > 0) {
    yield n--;
  }
}
```

5.1.6 First-Class Citizens

In Dart, functions are first-class citizens, which means that they can be stored in a variable, passed as a parameter to another function or returned as data from a function.

5.1.7 Higher Order Functions

A higher order function is a function that takes another function as an argument or returns one such as forEach and map. The forEach higher order function takes another function as an argument and applies that function to each item in the provided array. Map does the same, but *returns* a new array which can be stored in a variable since functions are first-class citizens as discussed in the previous section.

```
var data = [1, 2, 3];

data.forEach(function(num) {
    print(num);
});

data.map(function(num) {
    return 1 + num; // returns array of [2, 3, 4]
});

// storing new array in a variable
var newData = data.map(function(num) {
                return 1 + num; // returns array of [2, 3, 4]
              });
```

5.1.8 Fat Arrow Expressions (Lambdas)

A lambda expression is shorthand syntactic sugar for defining a function, it's parameters and it's body. Below are 3 examples using the lambda syntax. Note that since functions are first-class citizens in Dart, we can store them in a variable and treat them as data. Since 1 line functions don't require the *return* statement nor the *curly braces*, these are omitted in example 2. In example 3, the return value of the lambda expression is returned and stored in a function. Example 4 demonstrates the usage of the higher-order function, map, alongside the anonymous lambda syntax.

```
// Example 1
int sum(int a, int b) => {
    return a + b;
}

// Example 2
int sum(int a, int b) => a + b;

// Example 3
int mySum = (a, b) => a + b;

// Example 4
var newData = data.map((num) =>
                return 1 + num; // returns array of [2, 3, 4]
              );
```

5.2 Extension Methods

Prior to Dart 2.7, users wanted to add functionality to a pre-defined class but couldn't. This desire came from dissatisfaction from built-in or 3rd party classes. Users could make changes such as modifying the class definition or have a stray method not attached to a class and thus, would reduce the modularity of their codebase. The solution to this were extension methods, which allowed the injection of additional methods into a class and thus extending its functionality without recompiling. In the code snippet below, parseInt() is not a method inside the String class; it is an extension method from "string_apis.dart".

```
import 'string_apis.dart';
...
print('25'.parseInt());
```

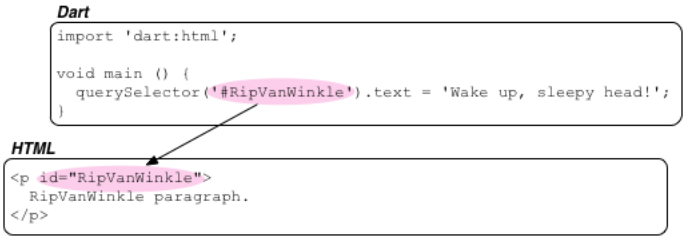
5.3 Closures

"A closure is a bundle of a function and variables from the outer scope that the function depends on." This is a bit more than lexical scoping which allow JavaScript functions to use variables defined in a parent scope. Closures provide users access to an outer scope from an inner scope. [15]

```
void main() {
  int x = a()(); // a() returns a function that is then called upon
  print(x);      // prints 5
}

Function a() { // return type is a function
  int x = 5;

  int b() {
    return x;
  }

  return b;
}
```

5.3.1 Shared Subroutines

Uncommon subroutines can be imported directly from the Dart library. For example, to connect Dart and HTML, the HTML library will have to be imported using the statement below. Building web apps with Dart requires the dart2js compiler and the imported JavaScript file to be defined in the HTML. Since the Dart file is being compiled using dart2js, the file is accessing the HTML DOM and injecting elements into it. In this case, Dart is being used as a "wrapper" for JavaScript so any Dart code that is written is compiled down to JavaScript and executed.




```
1 import 'dart:html';
```

6 Summary

Dart is closely tied to Java through its philosophy of OOP and JavaScript through its ability to build web apps. With cross platform compatibility and features such as implicit interfaces and mixins, Dart displays a bright future for the field of Computer Science. As Flutter flourishes in the coming years, it is guaranteed to succeed. Similar to various web frameworks, it is still prone to be heavily reliable upon the community to succeed. If Google keeps up the fantastic documentation and continues to demonstrate the power of Flutter, there is no doubt that Dart and Flutter together will succeed in the future.

References

[1] et al. Dart (programming language). (accessed: 03.26.2020).

[2] David Bolton. The fall and rise of dart, google's 'javascript killer'. (accessed: 03.26.2020).

[3] Wm Leler. Why flutter uses dart. (accessed: 03.26.2020).

[4] et al. Google fuchsia. (accessed: 03.26.2020).

[5] Uday Hiwarale. Dart (dartlang) introduction: Variables and data types. (accessed: 04.13.2020).

[6] Google. engine/hashcodes.dart at master - flutter/engine. (accessed: 04.13.2020).

[7] Google. Dart - fixed-size integers. (accessed: 04.13.2020).

[8] Google. The dart type system | dart. (accessed: 04.13.2020).

[9] Google. String class- dart:core library- dart api. (accessed: 04.13.2020).

[10] Seth Ladd. Booleans in dart. (accessed: 04.13.2020).

[11] Google. Language tour | dart. (accessed: 04.19.2020).

[12] Google. Language tour | dart. (accessed: 04.17.2020).

[13] TutorialsPrint. Dart programming - typedef. (accessed: 04.19.2020).

[14] Google. Language tour | dart. (accessed: 04.17.2020).

[15] Uday Hiwarale. Dart (dartlang) introduction: Functions and fat arrow expression. (accessed: 04.17.2020).