Kelly Hancox, CIS343

Dart Programming Language

Introduction

Dart was first unveiled by Google at the GOTO conference in Aarhus, Germany in 2011.[1] Dart was then released in 2013 and Dart 2.0 was released more recently in 2018. Dart is an object oriented language that uses C-style syntax. It can be fully converted to Javascript. When it's run as Javascript, then it's fully precompiled into Javascript and works as if it's Javascript in browsers, so there do not need to be any add ons to run Dart applications when this happens. It's also used in mobile app development and web. It's most popular framework is flutter, which people use to create native mobile applications in both IOS and Android with the same dart code. As of April 18th, Dart was 26th on the TIOBE index.

Data Abstractions

Dart's data types are nums, integers, doubles, strings, booleans, lists, and maps. These data types seem common from other object oriented languages like Java and C. Dart is considered an optionally typed language.[2] As stated in Heinz ,Møller, and Strocco, "Optional typing is traditionally viewed as a compromise between static and dynamic type checking, where code without type annotations is not checked until runtime." Therefore, programmers can decide whether they want their variables statically checked or not. And when they are dynamically checked, the Dart Programming Language Specification V2 also states, "As specified in this

---

1 http://gotocon.com/aarhus-2011/presentation/Opening%20Keynote:%20Dart,%20a%20new%20programming%20language%20for%20structured%20web%20programming

2 http://delivery.acm.org/10.1145/2990000/2989226/p1-heinze.pdf?ip=35.40.65.120&id=2989226&acc=ACTIVE%20SERVICE&key=B5D9E165A72B697C%2E6C5D91AC24E185AA%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1556042358_2e02e24f001ef4277c86722d5f5e6274

document, dynamic checks are guaranteed to be performed in certain situations, and certain violations of the type system throw exceptions at run time."[3] The manual also qualifies that it's assumed that inference takes place for dynamically checked types, but that "in some cases no information is available to infer an omitted type annotation, and hence this specification still needs to specify how to deal with that. A future version of this specification will also specify type inference." Heinz, Møller, and Strocco believe that this is a successful method for type checking for Dart, as they write, "Experimental results show that the technique is remarkably effective, even without context sensitivity: 99.3% of all property lookup operations are reported type safe in a collection of benchmark programs."

Dart is considered strongly typed. Generally, strong typing means that errors are always detected in a language. It helps with readability and writability as compared to weakly typed languages. Dartlang.org, however, qualifies that, "Although Dart is strongly typed, type annotations are optional because Dart can infer types … When you want to explicitly say that no type is expected, use the special type dynamic." This also has to do with being dynamically checked during runtime.

Dart uses implicit coercion. According to Bob Nystrom, engineer on the Dart team, while most statically typed languages like Java would require an explicit type cast to downcast, Dart has "something called 'assignment compatibility' to determine which assignments are valid. Most languages just use the normal subtyping rules for this: an assignment is safe if you assign from a sub- to a supertype. Dart's assignment compatibility rules also allow assigning from a super- to a subtype."[4] This means that if a programmer wanted to declare an int like so: double x

[3] https://www.dartlang.org/guides/language/specifications/DartLangSpec-v2.2.pdf
[4] https://news.dartlang.org/2012/05/types-and-casting-in-dart.html

= 3.99 or int y = x. Then this code would compile and determine if there is an error only after compiling. Likewise, a runtime error would also occur in Java, but they would need an explicit type cast just to compile. It also allows for upcasting, going from a suptype to a supertype like an integer to a double implicitly, as well, however it's recognized that this is more popular in other languages, as well. This may be more convenient and may increase writability, but decreases readability in the end because it increases difficulty in reading and determining which variables are which types.

Control Abstractions

An expression is a fragment of Dart code that can be evaluated at run time. Below is a picture of how expressions are created in Dart.

$$\begin{aligned}\langle expression \rangle ::= & \langle assignableExpression \rangle \ \langle assignmentOperator \rangle \ \langle expression \rangle \\ | \quad & \langle conditionalExpression \rangle \ \langle cascadeSection \rangle * \\ | \quad & \langle throwExpression \rangle\end{aligned}$$

This means that these expressions evaluate on the right, just like Java or C. The DartLangSpec guide also mentions that an expression produces an object or throws an exception object with a stack trace. And if an evaluation, e, of is defined in terms of an evaluation of another object, e1, then the evaluation of the e1 throws the exception and the evaluation of e stops at that point to throw the exception.

As for operators and precedence rules, they use similar rules of other languages and they are described in the table below, taken from the DartLangSpecs.

Description	Operator	Associativity	Precedence
Unary postfix	$e.$, $e?.$, e++, e--, $e1[e2]$, $e()$	None	16
Unary prefix	-e, !e, ~e, ++e, --e, **await** e	None	15
Multiplicative	*, /, ~/, %	Left	14
Additive	+, -	Left	13
Shift	<<, >>, >>>	Left	12
Bitwise AND	&	Left	11
Bitwise XOR	^	Left	10
Bitwise Or	|	Left	9
Relational	<, >, <=, >=, **as**, **is**, **is!**	None	8
Equality	==, !=	None	7
Logical AND	&&	Left	6
Logical Or	||	Left	5
If-null	??	Left	4
Conditional	$e1 ? e2 : e3$	Right	3
Cascade	..	Left	2
Assignment	=, *=, /=, +=, -=, &=, ^=, etc.	Right	1

This describes that the bottom operators would be evaluated first. Associativity means that that expression is evaluated from left to right or right to left. For instance, the additive operator is evaluated from left to right and 3+4 would evaluate 3 first, meanwhile the = operator would evaluate the expression on the right side first.

In terms of selection constructs, Dart uses conditional if statements. Below is a picture that describes how conditional statements that can be created within Dart.

$$\langle conditionalExpression \rangle ::= \langle ifNullExpression \rangle \\ \quad (\text{`?'} \ \langle expressionWithoutCascade \rangle \ \text{`:'} \ \langle expressionWithoutCascade \rangle)?$$

Dart supports if statements, else if statements, and else statements. These work much like Java's if else statements.

For iteration, Dart's for loops and while loops support iteration. Dart has three forms of for loops: the traditional for loop, the for-in statement in a synchronous and asynchronous form. It also has the traditional while loop and the do while loop. The traditional for loop uses syntax much like Java's where the following would be acceptable:

Var num = 5;

Var factorial = 1;

for(var i = num; i >= 1; i--){[5] factorial *= i }

The for-in loop is much like using a for each loop in Java. For instance, if I were to loop through an array, the following code would be acceptable:

Var obj = [12,13,14];

for(var prop in obj){ print(prop); }

This would print each variable in the object one by one. The for-in loop that is asynchronous uses an await statement inside of it. For instance: for(var o in objects) { await doSomething(o); } is viable. Overall an await statement allows for more statements to execute while one is backed up, then completes the unfinished business. We would expect this to behave similarly and return when the for loop is finished through. As for the while loops, the traditional while loop looks very much like a Java's while loop. For instance: while(expression){ //statements }.[6] For the do while loop, the syntax has the do at the top with while after the second bracket as follows:

Var n = 5;

do{ print(n); n--; } while(n >= 0);

These should provide insight into iteration with Dart.

Function syntax is much like Javascript's function syntax. It does not need a return type specified for it, likely also because it can be dynamically compiled. Here is the sample code of a function: function(int a){ print(a); }.[7] Calling this function with 'function(4)' inside the main method would work to print the number 4 onto the console.

---

[5] https://www.tutorialspoint.com/dart_programming/dart_programming_for_loop.htm
[6] https://www.tutorialspoint.com/dart_programming/dart_programming_while_loop.htm
[7] https://www.dartlang.org/guides/language/language-tour

As for parameter passing into these, it seems that Dart allows for objects to be passe dby as reference and that primitives like int, bool, and num are passed by value.[8] This means that when an object is created and passed through a method as a parameter, it's actual location and everything about it is not duplicated to pass through it, but this is the case for primitive values.

The scope rules of Dart are that Dart is lexically scoped and that scopes can nest. The DartLangSpecs guide states, "A name or declaration d is available in scope S if d is in the namespace induced by S or if d is available in the lexically enclosing scope of S. We say that a name or declaration d is in scope if d is available in the current scope."[9] This means that when you declare a variable with in the enclosing of a scope, you can see the variable you declared within that namespace. It also mentions, "A consequence of these rules is that it is possible to hide a type with a method or variable. Naming conventions usually prevent such abuses. Nevertheless, the following program is legal: class HighlyStrung { String( ) => "?"; }" Therefore, it can hide some of its declared variables.

In terms of imports, Dart calls its import 'Libraries.' These libraries are imported like: "import 'dart:math';" at the beginning of a program. The DartLangSpecs guide says, "The current library is the library currently being compiled. The import modifies the namespace of the current library in a manner that is determined by the imported library and by the optional elements of the import." Therefore, the current class can be considered the current namespace. There does not seem to be a limit to the amount of imports that can exist in a program. It also seems that Dart has its own libraries and you can import other libraries, presumably ones that

[8] https://groups.google.com/a/dartlang.org/forum/#!topic/misc/jyn35RSuT9Q
[9] https://www.dartlang.org/guides/language/specifications/DartLangSpec-v2.2.pdf

another programmer has written in dart. There can also be multiple imports within an import, as the DartLangSpecs guide also notes that 'dart:core' is imported into each library.

The included exceptions with the Dart library are as follows: DeferredLoadException, FormatExcepion, IntegerDivisionByZeroException, IOException, IsolateSpawnException, and Timeout. Dart uses try and catch blocks to catch these exceptions. Users can also create their own custom exceptions by implementing the Exception class. Below is an example of a try catch block in Dart:

main( ){ int x = 12; int y = 0; int res;

try{ res = x~/ y; }

on IntegerDivisionByZeroException[10]{ print('cannot divide by zero'); } }

These are used for error handling and Dart also supports Finally block after the catch blocks.

Object Oriented Support

Dart is solely an object oriented program. It gives a built in process to create a class with the following syntax: class class_name { ///fields, getters/setters, constructors, functions }. These classes are used as blueprints to create objects. Programmers can create instances of classes using the following syntax: var object_name = new class_name([args]).[11]

Dart's interfaces are implemented after class names like so: class ConsolePrinter implements Printer { //code }. And the class Printer would look like: class Printer{ void print_data( ) { print("_____ printing _____" _; } }. This seems to be much like Java interfaces. Dart has public and private data protection. It does not use something like protected in its classes.

[10] https://www.tutorialspoint.com/dart_programming/dart_programming_exceptions.htm
[11] https://www.tutorialspoint.com/dart_programming/dart_programming_classes.htm

A file in this github repository is calculator.dart that simulates a calculator that accepts continual input.