Native Idl Language Specification
1th edition draft

Version 0.1

Adriano Souza

January 2, 2021

Contents

1 Scope

This Ecma standard specifies the syntax and semantics of the Dart programming language. It does not specify the APIs of the Dart libraries except where those library elements are essential to the correct functioning of the language itself (e.g., the existence of class `Object` with methods such as `noSuchMethod`, `runtimeType`).

2 Conformance

A conforming implementation of the Dart programming language must provide and support all the APIs (libraries, types, functions, getters, setters, whether top-level, static, instance or local) mandated in this specification.

A conforming implementation is permitted to provide additional APIs, but not additional syntax, except for experimental features.

3 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

1. The Unicode Standard, Version 5.0, as amended by Unicode 5.1.0, or successor.
2. Dart API Reference, https://api.dartlang.org/

4 Terms and Definitions

Terms and definitions used in this specification are given in the body of the specification proper. Such terms are highlighted in italics when they are introduced, e.g., ‘we use the term *verbosity* to refer to the property of excess ◇ verbiage’, and add a marker in the margin.

5 Layers

Layers are the means of communication between client and server. At the client side, the layer must macth the layer at the server side. There are a few buit-in layers that are the foundation for creating new libraries and other layers.

5.1 FFI

5.2 Wasm

5.3 Async

Rationale Discussion of the motivation for language design decisions appears in italics. *Distinguishing normative from non-normative helps clarify what part of the text is binding and what part is merely expository.*

Commentary Comments such as "The careful reader will have noticed that the name Dart has four characters" serve to illustrate or clarify the specification, but are redundant with the normative text. The difference between commentary and rationale can be subtle. *Commentary is more general than rationale, and may include illustrative examples or clarifications.*

Open questions (**Upcoming: in this font**). Open questions are points that are unsettled in the mind of the author(s) of the specification; expect them (the questions, not the authors; precision is important in a specification) to be eliminated in the final specification. **Upcoming: Should the text at the end of the previous bullet be rationale or commentary?**

Reserved words and built-in identifiers (**??**) appear in **bold**.

Examples would be **switch** or **class**.

Grammar productions are given in a common variant of EBNF. The left hand side of a production ends with a colon. On the right hand side, alternation is represented by vertical bars, and sequencing by spacing. As in PEGs, alternation gives priority to the left. Optional elements of a production are suffixed by a question mark like so: `anElephant?`. Appending a star to an element of a production means it may be repeated zero or more times. Appending a plus sign to a production means it occurs one or more times. Parentheses are used for grouping. Negation is represented by prefixing an element of a production with a tilde. Negation is similar to the not combinator of PEGs, but it consumes input if it matches. In the context of a lexical production it consumes a single character if there is one; otherwise, a single token if there is one.

An example would be:

$\langle aProduction \rangle$ ::= $\langle anAlternative \rangle$
| $\langle anotherAlternative \rangle$
| $\langle oneThing \rangle$ $\langle after \rangle$ $\langle another \rangle$
| $\langle zeroOrMoreThings \rangle$*
| $\langle oneOrMoreThings \rangle$+
| $\langle anOptionalThing \rangle$?
| ($\langle some \rangle$ $\langle grouped \rangle$ $\langle things \rangle$)
| ~$\langle notAThing \rangle$
| 'aTerminal'
| $\langle A\_LEXICAL\_THING \rangle$

Both syntactic and lexical productions are represented this way. Lexical productions are distinguished by their names. The names of lexical productions consist exclusively of upper case characters and underscores. As always, within grammatical productions, whitespace and comments between elements of the production are implicitly ignored unless stated otherwise. Punctuation tokens appear in quotes.

Productions are embedded, as much as possible, in the discussion of the constructs they represent.

A *term* is a syntactic construct. It may be considered to be a piece of text which is derivable in the grammar, and it may be considered to be a tree created by such a derivation. An *immediate subterm* of a given term t is a syntactic construct which corresponds to an immediate subtree of t considered as a derivation tree. A *subterm* of a given term t is t, or an immediate subterm of t, or a subterm of an immediate subterm of t.

A list $x_1, \ldots, x_n$ denotes any list of n elements of the form $x_i, 1 \leq i \leq n$. Note that n may be zero, in which case the list is empty. We use such lists extensively throughout this specification.

For $j \in 1..n$, let y_j be an atomic syntactic entity (like an identifier), x_j a composite syntactic entity (like an expression or a type), and E again a composite syntactic entity. The notation $[x_1/y_1, \ldots, x_n/y_n]E$ then denotes a copy of E in which each occurrence of $y_i, 1 \leq i \leq n$ has been replaced by x_i.

This operation is also known as *substitution*, and it is the variant that avoids capture. That is, when E contains a construct that introduces y_i into a nested scope for some $i \in 1..n$, the substitution will not replace y_i in that scope. Conversely, if such a replacement would put an identifier *id* (a subterm of x_i) into a scope where *id* is declared, the relevant declarations in E are systematically renamed to fresh names.

In short, capture freedom ensures that the "meaning" of each identifier is preserved during substitution.

We sometimes abuse list or map literal syntax, writing $[o_1, \ldots, o_n]$ (respectively $\{k_1 : o_1, \ldots, k_n : o_n\}$) where the o_i and k_i may be objects rather than expressions. The intent is to denote a list (respectively map) object whose elements are the o_i (respectively, whose keys are the k_i and values are the o_i).

The specifications of operators often involve statements such as $x \; op \; y$ is equivalent to the method invocation $x.op(y)$. Such specifications should be understood as a shorthand for:

- $x \; op \; y$ is equivalent to the method invocation $x.op'(y)$, assuming the class of x actually declared a non-operator method named op' defining the same function as the operator op.

This circumlocution is required because $x.op(y)$, where op is an operator, is not legal syntax. However, it is painfully verbose, and we prefer to state this rule once here, and use a concise and clear notation across the specification.

When the specification refers to the order given in the program, it means the order of the program source code text, scanning left-to-right and top-to-bottom.

When the specification refers to a *fresh variable*, it means a local variable with a name that doesn't occur anywhere in the current program. When the specification introduces a fresh variable bound to an object, the fresh variable is implicitly bound in a surrounding scope.

References to otherwise unspecified names of program entities (such as classes or functions) are interpreted as the names of members of the Dart core library.

Examples would be the classes `Object` and `Type` representing, respectively, the root of the class hierarchy and the reification of run-time types. It would be possible to declare, e.g., a local variable named `Object`, so it is generally incorrect to assume that the name `Object` will actually resolve to said core class. However, we will generally omit mentioning this, for brevity.

When the specification says that one piece of syntax *is equivalent to* another piece of syntax, it means that it is equivalent in all ways, and the former syntax should generate the same compile-time errors and have the same run-time behavior as the latter, if any. Error messages, if any, should always refer to the original syntax. If execution or evaluation of a construct is said to be equivalent to execution or evaluation of another construct, then only the run-time behavior is equivalent, and compile-time errors apply only for the original syntax. ◇

When the specification says that one piece of syntax s is *treated as* another piece of syntax s', it means that the static analysis of s is the static analysis of s' (in particular, exactly the same compile-time errors occur). Moreover, if s has no compile-time errors then the behavior of s at run time is exactly the behavior of s'. ◇

Error messages*, if any, should always refer to the original syntax s.*

In short, whenever s is treated as s', the reader should immediately switch to the section about s' in order to get any further information about the static analysis and dynamic semantics of s.

The notion of being 'treated as' is similar to the notion of syntactic sugar: "s is treated as s′" could as well have been worded "s is desugared into s′". Of course, it should then actually be called "semantic sugar", because the applicability of the transformation and the construction of s′ may rely on information from static analysis.

The point is that we only specify the static analysis and dynamic semantics of a core language which is a subset of Dart (just slightly smaller than Dart), and desugaring transforms any given Dart program to a program in that core language. This helps keeping the language specification consistent and comprehensible, because it shows directly that some language features are introducing essential semantics, and others are better described as mere abbreviations of existing constructs.

6 Overview

Dart is a class-based, single-inheritance, pure object-oriented programming language. Dart is optionally typed (**??**) and supports reified generics. The run-time type of every object is represented as an instance of class `Type` which can be obtained by calling the getter `runtimeType` declared in class `Object`, the root of the Dart class hierarchy.

Dart programs may be statically checked. Programs with compile-time errors do not have a specified dynamic semantics. This specification makes no attempt to answer additional questions about a library or program at the point where it is known to have a compile-time error.

However, tools may choose to support execution of some programs with errors. For instance, a compiler may compile certain constructs with errors such that a dynamic error will be raised if an attempt is made to execute such a construct, or an IDE integrated runtime may support opening an editor window when such a construct is executed, allowing developers to correct the error. It is expected that such features would amount to a natural extension of the dynamic semantics of Dart as specified here, but, as mentioned, this specification makes no attempt to specify exactly what that means.

As specified in this document, dynamic checks are guaranteed to be performed in certain situations, and certain violations of the type system throw exceptions at run time.

An implementation is free to omit such checks whenever they are guaranteed to succeed, e.g., based on results from the static analysis.

The coexistence between optional typing and reification is based on the following:

1. Reified type information reflects the types of objects at run time and may always be queried by dynamic typechecking constructs (the analogs of instanceOf, casts, typecase etc. in other languages). Reified type information includes access to instances of class `Type` representing types, the run-time type (aka class) of an object, and the actual values of type parameters to constructors and generic function invocations.

2. Type annotations declare the types of variables and functions (including methods and constructors).

3. Type annotations may be omitted, in which case they are generally filled in with the type **dynamic** (**??**).

Dart as implemented includes extensive support for inference of omitted types. This specification makes the assumption that inference has taken place, and hence inferred types are considered to be present in the program already. However, 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.

Dart programs are organized in a modular fashion into units called *libraries* (**??**). Libraries are units of encapsulation and may be mutually recursive. ◇

However they are not first class. To get multiple copies of a library running simultaneously, one needs to spawn an isolate.

A dart program execution may occur with assertions enabled or disabled. The method used to enable or disable assertions is implementation specific.

6.1 Scoping

A *compile-time namespace* is a partial function that maps names to namespace values. Compile-time namespaces are used much more frequently than run-time namespaces (defined later in this section), so when the word *namespace* is used alone, it means compile-time namespace. A *name* is a lexical token which is an $\langle IDENTIFIER \rangle$, an $\langle IDENTIFIER \rangle$ followed by '`=`', or an $\langle operator \rangle$, or `unary-`; and a *namespace value* is a declaration, a namespace, or the special value NAME_CONFLICT (**??**).

If $NS(n) = V$ then we say that NS *maps* the *key* n to the *value* V, and that NS *has the binding* $n \mapsto V$. The fact that NS is a partial function just means that each name is mapped to at most one namespace value. That is, if NS has the bindings $n \mapsto V_1$ and $n \mapsto V_2$ then $V_1 = V_2$.

Let NS be a namespace. We say that a name n *is in* NS if n is a key of NS. We say a declaration d *is in* NS if a key of NS is mapped to d.

A scope S_0 has an associated namespace NS_0. The bindings of NS_0 is specified in this document by saying that a given declaration D named n *introduces* a specific entity V into S_0, which means that the binding $n \mapsto V$ is added to NS_0.

In some cases, the name of the declaration differs from the identifier that occurs in the declaration syntax used to declare it. Setters have names that are distinct from the corresponding getters because they always have an '=' automatically added at the end, and the unary minus operator has the special name `unary-`.

It is typically the case that V is the declaration D itself, but there are exceptions. For example, a variable declaration introduces an implicitly induced getter declaration, and in some cases also an implicitly induced setter declaration into the given scope.

Note that labels (**??**) are not included in the namespace of a scope. They are resolved lexically rather then being looked up in a namespace.

It is a compile-time error if there is more than one entity with the same name declared in the same scope.

It is therefore impossible, e.g., to define a class that declares a method and a getter with the same name in Dart. Similarly one cannot declare a top-level function with the same name as a library variable or a class which is declared in the same library.

We introduce the notion of a *run-time namespace*. This is a partial function from names to run-time entities, in particular storage locations and functions. Each run-time namespace corresponds to a namespace with the same keys, but with values that correspond to the semantics of the namespace values.

A namespace typically maps a name to a declaration, and it can be used statically to figure out what that name refers to. For example, a variable is associated with an actual storage location at run time. We introduce the notion of a run-time namespace based on a namespace, such that the dynamic semantics can access run-time entities like that storage location. The same code may be executed multiple times with the same run-time namespace, or with different run-time namespaces for each execution. E.g., local variables declared inside a *function are specific to each invocation of the function, and instance variables are specific to an object.*

Dart is lexically scoped. Scopes may nest. 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. ◇ ◇

If a declaration d named n is in the namespace induced by a scope S, then d *hides* any declaration named n that is available in the lexically enclosing scope of S. ◇

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() => "?";
}
```

Names may be introduced into a scope by declarations within the scope or by other mechanisms such as imports or inheritance.

The interaction of lexical scoping and inheritance is a subtle one. Ultimately, the question is whether lexical scoping takes precedence over inheritance or vice versa. Dart chooses the former.

Allowing inherited names to take precedence over locally declared names could create unexpected situations as code evolves. Specifically, the behavior of code in a subclass could silently change if a new name is introduced in a superclass. Consider:

```
library L1;
class S {}
library L2;
import 'L1.dart';
foo() => 42;
class C extends S{ bar() => foo();}
```

Now assume a method `foo()` is added to `S`.

```
library L1;
class S {foo() => 91;}
```

If inheritance took precedence over the lexical scope, the behavior of `C` would change in an unexpected way. Neither the author of `S` nor the author of `C` are necessarily aware of this. In Dart, if there is a lexically visible method `foo()`, it will always be called.

Now consider the opposite scenario. We start with a version of `S` that contains `foo()`, but do not declare `foo()` in library `L2`. Again, there is a change in behavior - but the author of `L2` is the one who introduced the discrepancy that *effects their code, and the new code is lexically visible. Both these factors make it more likely that the problem will be detected.*

These considerations become even more important if one introduces constructs such as nested classes, which might be considered in future versions of the language.

Good tooling should of course endeavor to inform programmers of such situations (discreetly). For example, an identifier that is both inherited and lexically visible could be highlighted (via underlining or colorization). Better yet, tight integration of source control with language aware tools would detect such changes when they occur.

6.2 Privacy

Dart supports two levels of *privacy*: public and private. A declaration is *private* iff its name is private, otherwise it is *public*. A name q is *private* iff any one of the identifiers that comprise q is private, otherwise it is *public*. An identifier is *private* iff it begins with an underscore (the _ character) otherwise it is *public*.

A declaration m is *accessible to a library L* if m is declared in L or if m is public.

This means private declarations may only be accessed within the library in which they are declared.

*Privacy applies only to declarations within a library, not to the library declaration as a whole. This is because libraries do not reference each other by name, and so the idea of a private library is meaningless (**??**). Thus, if the name of a library begins with an underscore, it has no effect on the accessibility of the library or its members.*

Privacy is, at this point, a static notion tied to a particular piece of code (a library). It is designed to support software engineering concerns rather than security concerns. Untrusted code should always run in an another isolate.

Privacy is indicated by the name of a declaration—hence privacy and naming are not orthogonal. This has the advantage that both humans and machines can recognize access to private declarations at the point of use without knowledge of the context from which the declaration is derived.

6.3 Concurrency

Dart code is always single threaded. There is no shared-state concurrency in Dart. Concurrency is supported via actor-like entities called *isolates*.

An isolate is a unit of concurrency. It has its own memory and its own thread of control. Isolates communicate by message passing (**??**). No state is ever shared between isolates. Isolates are created by spawning (**??**).

7 Errors and Warnings

This specification distinguishes between several kinds of errors.

Compile-time errors are errors that preclude execution. A compile-time error must be reported by a Dart compiler before the erroneous code is executed.

A Dart implementation has considerable freedom as to when compilation takes place. Modern programming language implementations often interleave compilation and execution, so that compilation of a method may be delayed, e.g., until it is first invoked. Consequently, compile-time errors in a method m may be reported as late as the time of m's first invocation.

Dart is often loaded directly from source, with no intermediate binary representation. In the interests of rapid loading, Dart implementations may choose to avoid full parsing of method bodies, for example. This can be done by tokenizing the input and checking for balanced curly braces on method body entry. In such an implementation, even syntax errors will be detected only when the method needs to be executed, at which time it will be compiled (JITed).

In a development environment a compiler should of course report compilation errors eagerly so as to best serve the programmer.

A Dart development environment might choose to support error eliminating program transformations, e.g., replacing an erroneous expression by the invocation of a debugger. It is outside the scope of this document to specify how such transformations work, and where they may be applied.

If an uncaught compile-time error occurs within the code of a running isolate A, A is immediately suspended. The only circumstance where a compile-time error could be caught would be via code run reflectively, where the mirror system can catch it.

Typically, once a compile-time error is thrown and A is suspended, A will then be terminated. However, this depends on the overall environment. A Dart engine runs in the context of an embedder*, a program that interfaces between the engine and the surrounding computing environment. The embedder will often be a web browser, but need not be; it may be a C++ program on the server for example. When an isolate fails with a compile-time error as described above, control returns to the embedder, along with an exception describing the problem. This is necessary so that the embedder can clean up resources etc. It is then the embedder's decision whether to terminate the isolate or not.*

Static warnings are situations that do not preclude execution, but which are unlikely to be intended, and likely to cause bugs or inconveniences. A static warning must be reported by a Dart compiler before the associated code is executed.

When this specification says that a *dynamic error* occurs, it means that a corresponding error object is thrown. When it says that a *dynamic type error* occurs, it represents a failed type check at run time, and the object which is thrown implements `TypeError`.

Whenever we say that an exception *ex* is *thrown*, it acts like an expression had thrown (**??**) with *ex* as exception object and with a stack trace corresponding to the current system state. When we say that a C *is thrown*, where C is a class, we mean that an instance of class C is thrown.

If an uncaught exception is thrown by a running isolate A, A is immediately suspended.

8 Reference

8.1 Lexical Rules

Dart source text is represented as a sequence of Unicode code points. This sequence is first converted into a sequence of tokens according to the lexical rules given in this specification. At any point in the tokenization process, the longest possible token is recognized.

8.1.1 Reserved Words

A *reserved word* may not be used as an identifier; it is a compile-time error if a reserved word is used where an identifier is expected. ◇

assert, **break**, **case**, **catch**, **class**, **const**, **continue**, **default**, **do**, **else**, **enum**, **extends**, **false**, **final**, **finally**, **for**, **if**, **in**, **is**, **new**, **null**, **rethrow**, **return**, **super**, **switch**, **this**, **throw**, **true**, **try**, **var**, **void**, **while**, **with**.

⟨*LETTER*⟩ ::= ‘`a`’ .. ‘`z`’
  | ‘`A`’ .. ‘`Z`’

⟨*DIGIT*⟩ ::= ‘`0`’ .. ‘`9`’

⟨*WHITESPACE*⟩ ::= (‘`\t`’ | ‘ ’ | ⟨*NEWLINE*⟩)+

8.1.2 Comments

Comments are sections of program text that are used for documentation. ◇

⟨*SINGLE_LINE_COMMENT*⟩ ::= ‘`//`’ ~(⟨*NEWLINE*⟩)* (⟨*NEWLINE*⟩)?

⟨*MULTI_LINE_COMMENT*⟩ ::=
  ‘`/*`’ (⟨*MULTI_LINE_COMMENT*⟩ | ~ ‘`*/`’)* ‘`*/`’

Dart supports both single-line and multi-line comments. A *single line comment* begins with the token `//`. Everything between `//` and the end of line must be ignored by the Dart compiler unless the comment is a documentation comment. ◇

A *multi-line comment* begins with the token `/*` and ends with the token `*/`. Everything between `/*` and `*/` must be ignored by the Dart compiler unless the comment is a documentation comment. Comments may nest. ◇

Documentation comments are comments that begin with the tokens `///` or `/**`. Documentation comments are intended to be processed by a tool that produces human readable documentation. ◇

The scope of a documentation comment immediately preceding the declaration of a class C is the instance scope of C.

The scope of a documentation comment immediately preceding the declaration of a function f is the scope in force at the very beginning of the body of f.

8.2 Operator Precedence

Operator precedence is given implicitly by the grammar.

The following non-normative table may be helpful

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	ˆ	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	=, *=, /=, +=, -=, &=, ˆ=, etc.	Right	1