Dart Programming Language Specification
5th edition draft

Version 2.0.0-dev

June 18, 2018

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 in support of null-aware cascades that are likely to be introduced in the next revision of this specification.

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'.

5 Notation

We distinguish between normative and non-normative text. Normative text defines the rules of Dart. It is given in this font. At this time, non-normative text includes:

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 (**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. **Should the text at the end of the previous bullet be rationale or commentary?**

Reserved words and built-in identifiers (16.34) 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:

```
AProduction:
  AnAlternative |
  AnotherAlternative |
  OneThing After Another |
  ZeroOrMoreThings* |
  OneOrMoreThings+ |
  AnOptionalThing? |
  (Some Grouped Things) |
  ~NotAThing |
  A_LEXICAL_THING
  ;
```

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 variable with a name that doesn't occur anywhere in the current program. When the specification introduces a fresh variable bound to a value, 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 the root of the class hierarchy and the reification of run-time types respectively.

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 static warnings and have the same run-time behavior as the latter. 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 only the static warnings or errors mentioned for the original syntax applies.

6 Overview

Dart is a class-based, single-inheritance, pure object-oriented programming language. Dart is optionally typed (19) 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. The static checker will report some violations of the type rules, but such violations do not abort compilation or preclude execution.

Dart programs may be executed in one of two modes: production mode or checked mode. In production mode, static type annotations (19.1) have absolutely no effect on execution with the exception of reflection and structural type tests.

Reflection, by definition, examines the program structure. If we provide reflective access to the type of a declaration, or to source code, it will inevitably produce results that depend on the types used in the underlying code.

Type tests also examine the types in a program explicitly. Nevertheless, in most cases, these will not depend on type annotations. The exceptions to this rule are type tests involving function types. Function types are structural, and so depend on the types declared for their parameters and on their return types.

In checked mode, assignments are dynamically checked, and certain violations of the type system throw exceptions at run time.

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 class declarations, the run-time type (aka class) of an object, and type arguments to constructors.

2. Static type annotations determine the types of variables and function declarations (including methods and constructors).

3. Production mode respects optional typing. Static type annotations do not affect run-time behavior.

4. Checked mode utilizes static type annotations and dynamic type information aggressively yet selectively to provide early error detection during development.

Dart programs are organized in a modular fashion into units called *libraries* (18). 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.

6.1 Scoping

A *namespace* is a mapping of names denoting declarations to actual declarations. 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 maps to d.

A scope S_0 induces a namespace NS_0 that maps the simple name of each variable, type or function declaration d declared in S_0 to d. Labels are not included in the induced namespace of a scope; instead they have their own dedicated namespace.

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.

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

In some cases, the name of the declaration differs from the identifier 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 unary minus has the special name unary-.

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 can create unexpected situations as code evolves. Specifically, the behavior of code

in a subclass can change without warning 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 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 library declarations themselves.

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. It is possible that libraries will become first class objects and privacy will be a dynamic notion tied to a library instance.

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 (16.18.4). No state is ever shared between isolates. Isolates are created by spawning (16.13).

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.

As a web language, 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.

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 those errors reported by the static checker. They have no effect on execution. Many, but not all, static warnings relate to types, in which case they are known as *static type warnings*. Static warnings must be provided by Dart compilers used during development such as those incorporated in IDEs or otherwise intended to be used by developers for developing code. Compilers that are part of run-time execution environments such as virtual machines should not issue static warnings.

Dynamic type errors are type errors reported in checked mode.

Run-time errors are exceptions thrown during execution. Whenever we say that an exception *ex* is *thrown*, it acts like an expression had *thrown* (17) 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 Variables

Variables are storage locations in memory.

variableDeclaration:
  declaredIdentifier (‘, ’ identifier)*
;

declaredIdentifier:
  metadata finalConstVarOrType identifier
;

finalConstVarOrType:
  final type? |
  const type? |
  varOrType
;

varOrType:
  var |
  type
;

initializedVariableDeclaration:
declaredIdentifier ('=' expression)? (', ' initializedIdentifier)*
;

initializedIdentifier:
identifier ('=' expression)?
;

initializedIdentifierList:
initializedIdentifier (', ' initializedIdentifier)*
;

A variable that has not been initialized has the null object (16.2) as its initial value.

A variable declared at the top-level of a library is referred to as either a *library variable* or simply a top-level variable.

A *static variable* is a variable that is not associated with a particular instance, but rather with an entire library or class. Static variables include library variables and class variables. Class variables are variables whose declaration is immediately nested inside a class declaration and includes the modifier **static**. A library variable is implicitly static. It is a compile-time error to preface a top-level variable declaration with the built-in identifier (16.34) **static**.

Static variable declarations are initialized lazily. When a static variable v is read, iff it has not yet been assigned, it is set to the result of evaluating its initializer. The precise rules are given in section 8.1.

The lazy semantics are given because we do not want a language where one tends to define expensive initialization computations, causing long application startup times. This is especially crucial for Dart, which must support the coding of client applications.

A *final variable* is a variable whose binding is fixed upon initialization; a final variable v will always refer to the same object after v has been initialized. The declaration of a final variable must include the modifier **final**.

It is a static warning if a final instance variable that has been initialized at its point of declaration is also initialized in a constructor. It is a compile-time error if a local variable v is final and v is not initialized at its point of declaration. It is a static warning and a dynamic error to assign to a final local variable.

A library or static variable is guaranteed to have an initializer at its declaration by the grammar.

Attempting to assign to a final variable anywhere except in its declaration or in a constructor header will cause a run-time error to be thrown as discussed below. The assignment will also give rise to a static warning. Any repeated assignment to a final variable will also lead to a run-time error.

Taken as a whole, the rules ensure that any attempt to execute multiple assignments to a final variable will yield static warnings and repeated assignments will fail dynamically.

A *constant variable* is a variable whose declaration includes the modifier **const**. A constant variable is always implicitly final. A constant variable must be initialized to a compile-time constant (16.1) or a compile-time error occurs.

We say that a variable v is *potentially mutated* in some scope s if v is not final or constant and an assignment to v occurs in s.

If a variable declaration does not explicitly specify a type, the type of the declared variable(s) is **dynamic**, the unknown type (19.6).

A variable is *mutable* if it is not final. Static and instance variable declarations always induce implicit getters. If the variable is mutable it also introduces an implicit setter. The scope into which the implicit getters and setters are introduced depends on the kind of variable declaration involved.

A library variable introduces a getter into the top level scope of the enclosing library. A static class variable introduces a static getter into the immediately enclosing class. An instance variable introduces an instance getter into the immediately enclosing class.

A mutable library variable introduces a setter into the top level scope of the enclosing library. A mutable static class variable introduces a static setter into the immediately enclosing class. A mutable instance variable introduces an instance setter into the immediately enclosing class.

Local variables are added to the innermost enclosing scope. They do not induce getters and setters. A local variable may only be referenced at a source code location that is after its initializer, if any, is complete, or a compile-time error occurs. The error may be reported either at the point where the premature reference occurs, or at the variable declaration.

We allow the error to be reported at the declaration to allow implementations to avoid an extra processing phase.

The example below illustrates the expected behavior. A variable x is declared at the library level, and another x is declared inside the function f.

```
var x = 0;
f(y) {
  var z = x; // compile-time error
  if (y) {
    x = x + 1; // two compile-time errors
    print(x); // compile-time error
  }
  var x = x++; // compile-time error
  print(x);
}
```

The declaration inside f hides the enclosing one. So all references to x inside f refer to the inner declaration of x. However, many of these references are illegal, because they appear before the declaration. The assignment to z is one such case. The assignment to x in the **if** statement suffers from multiple problems. The right hand side reads x before its declaration, and the left hand side assigns to x before

its declaration. Each of these are, independently, compile-time errors. The print statement inside the **if** is also illegal.

The inner declaration of x is itself erroneous because its right hand side attempts to read x before the declaration has terminated. The left hand side is not, technically, a reference or an assignment but a declaration and so is legal. The last print statement is perfectly legal as well.

As another example **var** x = 3, y = x; is legal, because x is referenced after its initializer.

A particularly perverse example involves a local variable name shadowing a type. This is possible because Dart has a single namespace for types, functions and variables.

```
class C {}
perverse() {
  var v = new C(); // compile-time error
  C aC; // compile-time error
  var C = 10;
}
```

Inside perverse(), C denotes a local variable. The type C is hidden by the variable of the same name. The attempt to instantiate C causes a compile-time error because it references a local variable prior to its declaration. Similarly, for the declaration of aC (even though it is only a type annotation).

As a rule, type annotations are ignored in production mode. However, we do not want to allow programs to compile legally in one mode and not another, and in this extremely odd situation, that consideration takes precedence.

The following rules apply to all static and instance variables.

A variable declaration of one of the forms $T\ v$;, $T\ v = e$; **const** $T\ v = e$;, **final** $T\ v$; or **final** $T\ v = e$; always induces an implicit getter function (10.2) with signature

T **get** v

whose invocation evaluates as described below (8.1).

A variable declaration of one of the forms **var** v;, **var** $v = e$;, **const** $v = e$;, **final** v; or **final** $v = e$; always induces an implicit getter function with signature

get v

whose invocation evaluates as described below (8.1).

A non-final variable declaration of the form $T\ v$; or the form $T\ v = e$; always induces an implicit setter function (10.3) with signature

void set $v = (T\ x)$

whose execution sets the value of v to the incoming argument x.

A non-final variable declaration of the form **var** v; or the form **var** $v = e$; always induces an implicit setter function with signature

set $v = (x)$

whose execution sets the value of v to the incoming argument x.

8.1 Evaluation of Implicit Variable Getters

Let d be the declaration of a static or instance variable v. If d is an instance

variable, then the invocation of the implicit getter of v evaluates to the value stored in v. If d is a static or library variable then the implicit getter method of v executes as follows:

- **Non-constant variable declaration with initializer**. If d is of one of the forms **var** $v = e$;, T $v = e$;, **final** $v = e$;, **final** T $v = e$;, **static** $v = e$;, **static** T $v = e$; , **static final** $v = e$; or **static final** T $v = e$; and no value has yet been stored into v then the initializer expression e is evaluated. If, during the evaluation of e, the getter for v is invoked, a CyclicInitializationError is thrown. If the evaluation succeeded yielding an object o, let r be o, otherwise let r be the null object (16.2). In any case, r is stored into v. The result of executing the getter is r.
- **Constant variable declaration**. If d is of one of the forms **const** $v = e$;, **const** T $v = e$;, **static const** $v = e$; or **static const** T $v = e$; the result of the getter is the value of the compile-time constant e. Note that a compile-time constant cannot depend on itself, so no cyclic references can occur. Otherwise
- **Variable declaration without initializer**. The result of executing the getter method is the value stored in v.

9 Functions

Functions abstract over executable actions.

functionSignature:
  metadata returnType? identifier formalParameterPart
;

formalParameterPart:
  typeParameters? formalParameterList
;

returnType:
  void |
  type
;

functionBody:
  async? '=>' expression ';' |
  (**async** | **async*** | **sync***)? block
;

block:
  '{' statements '}'

;

Functions can be introduced by function declarations (9.1), method declarations (10.1, 10.7), getter declarations (10.2), setter declarations (10.3), and constructor declarations (10.6); and they can be introduced by function literals (16.10).

Each declaration that introduces a function has a signature that specifies its return type, name, and formal parameter part, except that the return type may be omitted, and getters never have a formal parameter part. Function literals have a formal parameter part, but no return type and no name. The formal parameter part optionally specifies the formal type parameter list of the function, and it always specifies its formal parameter list. A function body is either:

- A block statement (17.1) containing the statements (17) executed by the function, optionally marked with one of the modifiers: **async**, **async*** or **sync***.

  Because Dart is optionally typed, we cannot guarantee that a function that does not return a value will not be used in the context of an expression. Therefore, every function must return a value. A function body that ends without doing a throw or return will cause the function to return the null object (16.2), as will a **return** without an expression. For generator functions, the situation is more subtle. See further discussion in section 17.12.

  OR

- of the form $=> e$ or the form **async** $=> e$, which both return the value of the expression e as if by a return e. The other modifiers do not apply here, because they apply only to generators, discussed below, and generators do not allow to return a value, values are added to the generated stream or iterable using **yield** instead. Let R be the static type of e and let T be the actual return type (19.8.1) of the function that has this body. It is a static warning if T is not **void** and either the function is synchronous and the static type of R is not assignable to T, or the function is asynchronous and Future$<flatten(R)>$ is not assignable to T.

A function is *asynchronous* if its body is marked with the **async** or **async*** modifier. Otherwise the function is *synchronous*. A function is a *generator* if its body is marked with the **sync*** or **async*** modifier.

Whether a function is synchronous or asynchronous is orthogonal to whether it is a generator or not. Generator functions are a sugar for functions that produce collections in a systematic way, by lazily applying a function that *generates* individual elements of a collection. Dart provides such a sugar in both the synchronous case, where one returns an iterable, and in the asynchronous case, where one returns a stream. Dart also allows both synchronous and asynchronous functions that produce a single value.

It is a compile-time error if an **async**, **async*** or **sync*** modifier is attached to the body of a setter or constructor.

An asynchronous setter would be of little use, since setters can only be used in the context of an assignment (16.20), and an assignment expression always evaluates to the value of the assignment's right hand side. If the setter actually did its work asynchronously, one might imagine that one would return a future that resolved to the assignment's right hand side after the setter did its work. However, this would require dynamic tests at every assignment, and so would be prohibitively expensive.

An asynchronous constructor would, by definition, never return an instance of the class it purports to construct, but instead return a future. Calling such a beast via ***new*** *would be very confusing. If you need to produce an object asynchronously, use a method.*

One could allow modifiers for factories. A factory for Future *could be modified by* ***async****, a factory for* Stream *could be modified by* ***async***** and a factory for* Iterable *could be modified by* ***sync*****. No other scenario makes sense because the object returned by the factory would be of the wrong type. This situation is very unusual so it is not worth making an exception to the general rule for constructors in order to allow it.* It is a static warning if the declared return type of a function marked **async** is not a supertype of Future<T> for some type T. It is a static warning if the declared return type of a function marked **sync*** is not a supertype of Iterable<T> for some type T. It is a static warning if the declared return type of a function marked **async*** is not a supertype of Stream<T> for some type T.

9.1 Function Declarations

A *function declaration* is a function that is neither a member of a class nor a function literal. Function declarations include *library functions*, which are function declarations at the top level of a library, and *local functions*, which are function declarations declared inside other functions. Library functions are often referred to simply as top-level functions.

A function declaration consists of an identifier indicating the function's name, possibly prefaced by a return type. The function name is followed by a signature and body. For getters, the signature is empty. The body is empty for functions that are external.

The scope of a library function is the scope of the enclosing library. The scope of a local function is described in section 17.4. In both cases, the name of the function is in scope in its formal parameter scope (9.2).

It is a compile-time error to preface a function declaration with the built-in identifier **static**.

When we say that a function f_1 *forwards* to another function f_2, we mean that invoking f_1 causes f_2 to be executed with the same arguments and/or receiver as f_1, and returns the result of executing f_2 to the caller of f_1, unless f_2 throws an exception, in which case f_1 throws the same exception. Furthermore, we only use the term for synthetic functions introduced by the specification.

9.2 Formal Parameters

Every non-getter function declaration includes a *formal parameter list*, which consists of a list of required positional parameters (9.2.1), followed by any optional parameters (9.2.2). The optional parameters may be specified either as a set of named parameters or as a list of positional parameters, but not both.

Some function declarations include a *formal type parameter list* (9), in which case we say that it is a *generic function*. A *non-generic function* is a function which is not generic.

The *formal parameter part* of a function declaration consists of the formal type parameter list, if any, and the formal parameter list.

The following kinds of functions cannot be generic: Getters, setters, operators, and constructors.

The formal type parameter list of a function declaration introduces a new scope known as the function's *type parameter scope*. The type parameter scope of a generic function f is enclosed in the scope where f is declared. Every formal type parameter introduces a type into the type parameter scope.

If it exists, the type parameter scope of a function f is the current scope for the signature of f, and for the formal type parameter list itself; otherwise the scope where f is declared is the current scope for the signature of f.

This means that formal type parameters are in scope in the bounds of parameter declarations, allowing for so-called F-bounded type parameters like

class C<X **extends** Comparable<X>> { ... },

and the formal type parameters are in scope for each other, allowing dependencies like class D<X **extends** Y, Y> { ... }.

The formal parameter list of a function declaration introduces a new scope known as the function's *formal parameter scope*. The formal parameter scope of a non-generic function f is enclosed in the scope where f is declared. The formal parameter scope of a generic function f is enclosed in the type parameter scope of f. Every formal parameter introduces a local variable into the formal parameter scope. The current scope for the function's signature is the scope that encloses the formal parameter scope.

This means that in a generic function declaration, the return type and parameter type annotations can use the formal type parameters, but the formal parameters are not in scope in the signature.

The body of a function declaration introduces a new scope known as the function's *body scope*. The body scope of a function f is enclosed in the scope introduced by the formal parameter scope of f.

It is a compile-time error if a formal parameter is declared as a constant variable (8).

formalParameterList:
'(' ')' |
'(' normalFormalParameters ','? ')' |
'(' normalFormalParameters ',' optionalFormalParameters ')' |

‘(’ optionalFormalParameters ‘)’
;

normalFormalParameters:
normalFormalParameter (‘,’ normalFormalParameter)*
;

optionalFormalParameters:
optionalPositionalFormalParameters |
namedFormalParameters
;

optionalPositionalFormalParameters:
‘[’ defaultFormalParameter (‘,’ defaultFormalParameter)* ‘,’? ‘]’
;

namedFormalParameters:
‘{’ defaultNamedParameter (‘,’ defaultNamedParameter)* ‘,’? ‘}’
;

Formal parameter lists allow an optional trailing comma after the last parameter (‘,’?). A parameter list with such a trailing comma is equivalent in all ways to the same parameter list without the trailing comma. All parameter lists in this specification are shown without a trailing comma, but the rules and semantics apply equally to the corresponding parameter list with a trailing comma.

9.2.1 Required Formals

A *required formal parameter* may be specified in one of three ways:

- By means of a function signature that names the parameter and describes its type as a function type (19.5). It is a compile-time error if any default values are specified in the signature of such a function type.
- As an initializing formal, which is only valid as a parameter to a generative constructor (10.6.1).
- Via an ordinary variable declaration (8).

normalFormalParameter:
functionFormalParameter |
fieldFormalParameter |
simpleFormalParameter
;

functionFormalParameter:
  metadata **covariant**? returnType? identifier
      formalParameterPart
;

simpleFormalParameter:
  metadata **covariant**? finalConstVarOrType? identifier |

;

fieldFormalParameter:
  metadata finalConstVarOrType? **this** '.' identifier
      formalParameterPart?
;

It is possible to include the modifier **covariant** in some forms of parameter declarations. This modifier has no effect.

*The modifier **covariant** is used in strong mode. The modifier is allowed here even though it has no effect, such that source code can be used in both contexts.*

It is a compile-time error if the modifier **covariant** occurs on a parameter of a function which is not an instance method, instance setter, or instance operator.

9.2.2 Optional Formals

Optional parameters may be specified and provided with default values.

defaultFormalParameter:
  normalFormalParameter ('=' expression)?
;

defaultNamedParameter:
  normalFormalParameter ('=' expression)? |
  normalFormalParameter (':' expression)?
;

A **defaultNamedParameter** of the form: normalFormalParameter : expression is equivalent to one of the form: normalFormalParameter = expression The colon-syntax is included only for backwards compatibility. It is deprecated and will be removed in a later version of the language specification.

It is a compile-time error if the default value of an optional parameter is not a compile-time constant (16.1). If no default is explicitly specified for an optional parameter an implicit default of **null** is provided.

It is a compile-time error if the name of a named optional parameter begins with an '_' character.

The need for this restriction is a direct consequence of the fact that naming and privacy are not orthogonal. If we allowed named parameters to begin with an underscore, they would be considered private and inaccessible to callers from outside the library where it was defined. If a method outside the library overrode a method with a private optional name, it would not be a subtype of the original method. The static checker would of course flag such situations, but the consequence would be that adding a private named formal would break clients outside the library in a way they could not easily correct.

9.3 Type of a Function

If a function declaration does not declare a return type explicitly, its return type is **dynamic** (19.6), unless it is a constructor function, in which case its return type is the immediately enclosing class, or it is a setter or operator []=, in which case its return type is **void**.

A function declaration may declare formal type parameters. The type of the function includes the names of the type parameters and their upper bounds. When consistent renaming of type parameters can make two function types identical, they are considered to be the same type.

It is convenient to include the type parameter names in function types because they are needed in order to express such things as relations among different type parameters, and F-bounds. However, we do not wish to distinguish two function types if they have the same structure and only differ in the choice of names. This treatment of names is also known as alpha-equivalence.

In the following three paragraphs, if the number m of formal type parameters is zero then the type parameter list in the function type should be omitted.

Let F be a function with formal type parameters $X_1\ B_1, \ldots,\ X_m\ B_m$, required formal parameters $T_1\ p_1, \ldots,\ T_n\ p_n$, return type T_0 and no optional parameters. Then the type of F is $<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n) \rightarrow T_0$.

Let F be a function with formal type parameters $X_1\ B_1, \ldots,\ X_m\ B_m$, required formal parameters $T_1\ p_1, \ldots,\ T_n\ p_n$, return type T_0 and positional optional parameters $T_{n+1}\ p_{n+1}, \ldots,\ T_{n+k}\ p_{n+k}$. Then the type of F is $<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n, [T_{n+1}\ p_{n+1}, \ldots,\ T_{n+k}\ p_{n+k}]) \rightarrow T_0$.

Let F be a function with formal type parameters $X_1\ B_1, \ldots,\ X_m\ B_m$, required formal parameters $T_1\ p_1, \ldots,\ T_n\ p_n$, return type T_0 and named optional parameters $T_{n+1}\ p_{n+1}, \ldots,\ T_{n+k}\ p_{n+k}$. Then the type of F is $<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n, \{T_{n+1}\ p_{n+1}, \ldots,\ T_{n+k}\ p_{n+k}\}) \rightarrow T_0$.

The run-time type of a function object always implements the class **Function**.

One cannot assume, based on the above, that given a function f, f.runtimeType will actually be **Function**, or that any two distinct function objects necessarily have the same run-time type.

It is up to the implementation to choose an appropriate representation for function objects. For example, consider that a function object produced via property extraction treats equality differently from other function objects, and is therefore likely a different class. Implementations may also use different classes

for function objects based on arity and or type. Arity may be implicitly affected by whether a function is an instance method (with an implicit receiver parameter) or not. The variations are manifold, and so this specification only guarantees that function objects are instances of some class that implements ***Function****.*

9.4 External Functions

An *external function* is a function whose body is provided separately from its declaration. An external function may be a top-level function (18), a method (10.1, 10.7), a getter (10.2), a setter (10.3) or a non-redirecting constructor (10.6.1, 10.6.2). External functions are introduced via the built-in identifier **external** (16.34) followed by the function signature.

External functions allow us to introduce type information for code that is not statically known to the Dart compiler.

Examples of external functions might be foreign functions (defined in C, or Javascript etc.), primitives of the implementation (as defined by the Dart run-time system), or code that was dynamically generated but whose interface is statically known. However, an abstract method is different from an external function, as it has *no* body.

An external function is connected to its body by an implementation specific mechanism. Attempting to invoke an external function that has not been connected to its body will throw a NoSuchMethodError or some subclass thereof.

The actual syntax is given in sections 10 and 18 below.

10 Classes

A *class* defines the form and behavior of a set of objects which are its *instances*. Classes may be defined by class declarations as described below, or via mixin applications (12.1).

classDefinition:
  metadata **abstract**? **class** identifier typeParameters?
      (superclass mixins?)? interfaces?
      '{' (metadata classMemberDefinition)* '}' |
  metadata **abstract**? **class** mixinApplicationClass
;

mixins:
  with typeList
;

classMemberDefinition:
  declaration ';' |
  methodSignature functionBody

;

methodSignature:
constructorSignature initializers? |
factoryConstructorSignature |
static? functionSignature |
static? getterSignature |
static? setterSignature |
operatorSignature
;

declaration:
constantConstructorSignature (redirection | initializers)? |
constructorSignature (redirection | initializers)? |
external constantConstructorSignature |
external constructorSignature |
((**external static**?))? getterSignature |
((**external static**?))? setterSignature |
external? operatorSignature |
((**external static**?))? functionSignature |
static (**final** | **const**) type? staticFinalDeclarationList |
final type? initializedIdentifierList |
(**static** | **covariant**)? (**var** | type) initializedIdentifierList
;

staticFinalDeclarationList:
staticFinalDeclaration (',' staticFinalDeclaration)*
;

staticFinalDeclaration:
identifier '=' expression
;

It is possible to include the modifier **covariant** in some forms of declarations. This modifier has no effect.

*The modifier **covariant** is used in strong mode. The modifier is allowed here even though it has no effect, such that source code can be used in both contexts.*

A class has constructors, instance members and static members. The instance members of a class are its instance methods, getters, setters and instance variables. The static members of a class are its static methods, getters, setters and static variables. The members of a class are its static and instance members.

A class has several scopes:

- A *type-parameter scope*, which is empty if the class is not generic (14).

The enclosing scope of the type-parameter scope of a class is the enclosing scope of the class declaration.

- A *static scope*. The enclosing scope of the static scope of a class is the type parameter scope (14) of the class.
- An *instance scope*. The enclosing scope of a class' instance scope is the class' static scope.

The enclosing scope of an instance member declaration is the instance scope of the class in which it is declared.

The enclosing scope of a static member declaration is the static scope of the class in which it is declared.

Every class has a single superclass except class Object which has no superclass. A class may implement a number of interfaces by declaring them in its implements clause (10.10).

An *abstract class* is a class that is explicitly declared with the **abstract** modifier, either by means of a class declaration or via a type alias (19.3.1) for a mixin application (12.1). A *concrete class* is a class that is not abstract.

We want different behavior for concrete classes and abstract classes. If A is intended to be abstract, we want the static checker to warn about any attempt to instantiate A, and we do not want the checker to complain about unimplemented methods in A. In contrast, if A is intended to be concrete, the checker should warn about all unimplemented methods, but allow clients to instantiate it freely.

The *interface of class C* is an implicit interface that declares instance members that correspond to the instance members declared by C, and whose direct superinterfaces are the direct superinterfaces of C (10.10). When a class name appears as a type, that name denotes the interface of the class.

It is a compile-time error if a class declares two members of the same name, either because it declares the same name twice in the same scope (6.1), or because it declares a static member and an instance member with the same name (10.11).

Here are simple examples, that illustrate the difference between "has a member" and "declares a member". For example, B *declares* one member named f, but *has* two such members. The rules of inheritance determine what members a class has.

```
class A {
  var i = 0;
  var j;
  f(x) => 3;
}
class B extends A {
  int i = 1; // getter i and setter i= override versions from A
  static j; // compile-time error: static getter & setter conflict with
  // instance getter & setter
  /* compile-time error: static method conflicts with instance method */
  static f(x) => 3;
}
```

It is a compile-time error if a class C declares a member with the same

name as C. It is a compile-time error if a generic class declares a type variable with the same name as the class or any of its members or constructors.

10.1 Instance Methods

Instance methods are functions (9) whose declarations are immediately contained within a class declaration and that are not declared **static**. The instance methods of a class C are those instance methods declared by C and the instance methods inherited by C from its superclass.

It is a static warning if an instance method m_1 overrides (10.9.1) an instance member m_2 and m_1 has a greater number of required parameters than m_2. It is a static warning if an instance method m_1 overrides an instance member m_2 and m_1 has fewer positional parameters than m_2. It is a static warning if an instance method m_1 overrides an instance member m_2 and m_1 does not declare all the named parameters declared by m_2.

It is a static warning if an instance method m_1 overrides an instance member m_2 and the type of m_1 is not a subtype of the type of m_2. It is a static warning if an instance method m_1 overrides an instance member m_2, the signature of m_2 explicitly specifies a default value for a formal parameter p, and the signature of m_1 implies a different default value for p.

A method declaration may conflict with other declarations (10.11).

10.1.1 Operators

Operators are instance methods with special names.

operatorSignature:
  returnType? **operator** operator formalParameterList
;

operator:
  '~' |
  binaryOperator |
  '[]' |
  '[]='
;

binaryOperator:
  multiplicativeOperator |
  additiveOperator |
  shiftOperator |
  relationalOperator |
  '==' |
  bitwiseOperator
;

An operator declaration is identified using the built-in identifier (16.34) **operator**.

The following names are allowed for user-defined operators: <, >, <=, >=, ==, -, +, /, ~/, *, %, |, ^, &, <<, >>, >>>, []=, [], ~.

It is a compile-time error if the arity of the user-declared operator []= is not 2. It is a compile-time error if the arity of a user-declared operator with one of the names: <, >, <=, >=, ==, -, +, ~/, /, *, %, |, ^, &, <<, >>, >>>, [] is not 1. It is a compile-time error if the arity of the user-declared operator - is not 0 or 1.

The - operator is unique in that two overloaded versions are permitted. If the operator has no arguments, it denotes unary minus. If it has an argument, it denotes binary subtraction.

The name of the unary operator - is unary-.

This device allows the two methods to be distinguished for purposes of method lookup, override and reflection.

It is a compile-time error if the arity of the user-declared operator ~ is not 0.

It is a compile-time error to declare an optional parameter in an operator.

It is a static warning if the return type of the user-declared operator []= is explicitly declared and not **void**.

If no return type is specified for a user-declared operator []=, its return type is **void** (9.3).

10.2 Getters

Getters are functions (9) that are used to retrieve the values of object properties.

getterSignature:
  returnType? **get** identifier
;

If no return type is specified, the return type of the getter is **dynamic**.

A getter definition that is prefixed with the **static** modifier defines a static getter. Otherwise, it defines an instance getter. The name of the getter is given by the identifier in the definition. The effect of a static getter declaration in class C is to add an instance getter with the same name and signature to the Type object for class C that forwards (9.1) to the static getter.

The instance getters of a class C are those instance getters declared by C, either implicitly or explicitly, and the instance getters inherited by C from its superclass. The static getters of a class C are those static getters declared by C.

A getter declaration may conflict with other declarations (10.11). In particular, a getter can never override a method, and a method can never override a getter or an instance variable.

It is a static warning if the return type of a getter is **void**. It is a static warning if a getter m_1 overrides (10.9.1) a getter m_2 and the type of m_1 is not a subtype of the type of m_2.

10.3 Setters

Setters are functions (9) that are used to set the values of object properties.

setterSignature:
returnType? **set** identifier formalParameterList
;

If no return type is specified, the return type of the setter is **void** (9.3).

A setter definition that is prefixed with the **static** modifier defines a static setter. Otherwise, it defines an instance setter. The name of a setter is obtained by appending the string '=' to the identifier given in its signature. The effect of a static setter declaration in class C is to add an instance setter with the same name and signature to the Type object for class C that forwards (9.1) to the static setter.

Hence, a setter name can never conflict with, override or be overridden by a getter or method.

The instance setters of a class C are those instance setters declared by C either implicitly or explicitly, and the instance setters inherited by C from its superclass. The static setters of a class C are those static setters declared by C.

It is a compile-time error if a setter's formal parameter list does not consist of exactly one required formal parameter p. *We could enforce this via the grammar, but we'd have to specify the evaluation rules in that case.*

It is a static warning if a setter declares a return type other than **void**. It is a static warning if a setter m_1 overrides (10.9.1) a setter m_2 and the type of m_1 is not a subtype of the type of m_2. It is a static warning if a class has a setter named $v =$ with argument type T and a getter named v with return type S, and T may not be assigned to S.

A setter declaration may conflict with other declarations (10.11).

10.4 Abstract Instance Members

An *abstract method* (respectively, *abstract getter* or *abstract setter)* is an instance method, getter or setter that is not declared **external** and does not provide an implementation. A *concrete method* (respectively, *concrete getter* or *concrete setter)* is an instance method, getter or setter that is not abstract.

*Earlier versions of Dart required that abstract members be identified by prefixing them with the modifier **abstract**. The elimination of this requirement is motivated by the desire to use abstract classes as interfaces. Every Dart class induces an implicit interface.*

Using an abstract class instead of an interface has important advantages. An abstract class can provide default implementations; it can also provide static

methods, obviating the need for service classes such as Collections or Lists, whose entire purpose is to group utilities related to a given type.

Eliminating the requirement for an explicit modifier on members makes abstract classes more concise, making abstract classes an attractive substitute for interface declarations.

Invoking an abstract method, getter or setter results in an invocation of noSuchMethod exactly as if the declaration did not exist, unless a suitable member a is available in a superclass, in which case a is invoked. The normative specification for this appears under the definitions of lookup for methods, getters and setters.

The purpose of an abstract method is to provide a declaration for purposes such as type checking and reflection. In classes used as mixins, it is often useful to introduce such declarations for methods that the mixin expects will be provided by the superclass the mixin is applied to.

It is a static warning if an abstract member m is declared or inherited in a concrete class C unless:

- m overrides a concrete member, or
- C has a concrete noSuchMethod() method distinct from the one declared in class Object.

We wish to warn if one declares a concrete class with abstract members. However, code like the following should work without warnings:

```
class Base {
  int get one => 1;
}
abstract class Mix {
  int get one;
  int get two => one + one;
}
class C extends Base with Mix { }
```

At run time, the concrete method one declared in Base will be executed, and no problem should arise. Therefore no warning should be issued and so we suppress warnings if a corresponding concrete member exists in the hierarchy.

10.5 Instance Variables

Instance variables are variables whose declarations are immediately contained within a class declaration and that are not declared **static**. The instance variables of a class C are those instance variables declared by C and the instance variables inherited by C from its superclass.

It is a compile-time error if an instance variable is declared to be constant.

The notion of a constant instance variable is subtle and confusing to programmers. An instance variable is intended to vary per instance. A constant instance variable would have the same value for all instances, and as such is already a dubious idea.

The language could interpret const instance variable declarations as instance getters that return a constant. However, a constant instance variable could not be treated as a true compile-time constant, as its getter would be subject to overriding.

Given that the value does not depend on the instance, it is better to use a static class variable. An instance getter for it can always be defined manually if desired.

10.6 Constructors

A *constructor* is a special function that is used in instance creation expressions (16.12) to obtain objects, typically by creating or initializing them. Constructors may be generative (10.6.1) or they may be factories (10.6.2).

A *constructor name* always begins with the name of its immediately enclosing class, and may optionally be followed by a dot and an identifier *id*. It is a compile-time error if the name of a constructor is not a constructor name.

A constructor declaration may conflict with static member declarations (10.11).

Iff no constructor is specified for a class C, it implicitly has a default constructor C() : **super**() {}, unless C is class Object.

10.6.1 Generative Constructors

A *generative constructor* consists of a constructor name, a constructor parameter list, and either a redirect clause or an initializer list and an optional body.

constructorSignature:
  identifier ('.' identifier)? formalParameterList
;

A *constructor parameter list* is a parenthesized, comma-separated list of formal constructor parameters. A *formal constructor parameter* is either a formal parameter (9.2) or an initializing formal. An *initializing formal* has the form **this**.*id*, where *id* is the name of an instance variable of the immediately enclosing class. It is a compile-time error if *id* is not an instance variable of the immediately enclosing class. It is a compile-time error if an initializing formal is used by a function other than a non-redirecting generative constructor.

If an explicit type is attached to the initializing formal, that is its static type. Otherwise, the type of an initializing formal named *id* is T_{id}, where T_{id} is the type of the instance variable named *id* in the immediately enclosing class. It is a static warning if the static type of *id* is not a subtype of T_{id}.

Initializing formals constitute an exception to the rule that every formal parameter introduces a local variable into the formal parameter scope (9.2). When the formal parameter list of a non-redirecting generative constructor contains any initializing formals, a new scope is introduced, the *formal parameter initializer scope*, which is the current scope of the initializer list of the constructor,

and which is enclosed in the scope where the constructor is declared. Each initializing formal in the formal parameter list introduces a final local variable into the formal parameter initializer scope, but not into the formal parameter scope; every other formal parameter introduces a local variable into both the formal parameter scope and the formal parameter initializer scope.

This means that formal parameters, including initializing formals, must have distinct names, and that initializing formals are in scope for the initializer list, but they are not in scope for the body of the constructor. When a formal parameter introduces a local variable into two scopes, it is still one variable and hence one storage location. The type of the constructor is defined in terms of its formal parameters, including the initializing formals.

Initializing formals are executed during the execution of generative constructors detailed below. Executing an initializing formal **this**.*id* causes the instance variable *id* of the immediately surrounding class to be assigned the value of the corresponding actual parameter, unless *id* is a final variable that has already been initialized, in which case a run-time error occurs.

The above rule allows initializing formals to be used as optional parameters:

```
class A {
  int x;
  A([this.x]);
}
```

is legal, and has the same effect as

```
class A {
  int x;
  A([int x]): this.x = x;
}
```

A *fresh instance* is an instance whose identity is distinct from any previously allocated instance of its class. A generative constructor always operates on a fresh instance of its immediately enclosing class.

The above holds if the constructor is actually run, as it is by **new**. If a constructor c is referenced by **const**, c may not be run; instead, a canonical object may be looked up. See the section on instance creation (16.12).

If a generative constructor c is not a redirecting constructor and no body is provided, then c implicitly has an empty body {}.

Redirecting Constructors

A generative constructor may be *redirecting*, in which case its only action is to invoke another generative constructor. A redirecting constructor has no body; instead, it has a redirect clause that specifies which constructor the invocation is redirected to, and with what arguments.

redirection:
  ':' **this** ('.' identifier)? arguments
;

Initializer Lists

An initializer list begins with a colon, and consists of a comma-separated list of individual *initializers*. There are two kinds of initializers.

- A *superinitializer* identifies a *superconstructor* - that is, a specific constructor of the superclass. Execution of the superinitializer causes the initializer list of the superconstructor to be executed.
- An *instance variable initializer* assigns a value to an individual instance variable.

initializers:
  ':' initializerListEntry (',' initializerListEntry)*
;

initializerListEntry:
  super arguments |
  super '.' identifier arguments |
  fieldInitializer |
  assertion
;

fieldInitializer:
  (**this** '.')? identifier '=' conditionalExpression cascadeSection*
;

Let k be a generative constructor. Then k may include at most one superinitializer in its initializer list or a compile-time error occurs. If no superinitializer is provided, an implicit superinitializer of the form **super**() is added at the end of k's initializer list, unless the enclosing class is class Object. It is a compile-time error if more than one initializer corresponding to a given instance variable appears in k's initializer list. It is a compile-time error if k's initializer list contains an initializer for a variable that is initialized by means of an initializing formal of k. It is a compile-time error if k's initializer list contains an initializer for a final variable f whose declaration includes an initialization expression. It is a compile-time error if k includes an initializing formal for a final variable f whose declaration includes an initialization expression.

Each final instance variable f declared in the immediately enclosing class must have an initializer in k's initializer list unless it has already been initialized by one of the following means:

- Initialization at the declaration of f.
- Initialization by means of an initializing formal of k.

or a static warning occurs. It is a compile-time error if k's initializer list contains an initializer for a variable that is not an instance variable declared in the immediately surrounding class.

The initializer list may of course contain an initializer for any instance variable declared by the immediately surrounding class, even if it is not final.

It is a compile-time error if a generative constructor of class Object includes a superinitializer.

Execution of a generative constructor k of type T to initialize a fresh instance i is always done with respect to a set of bindings for its formal parameters and the type parameters of the immediately enclosing class bound to a set of actual type arguments of T, $V_1, \ldots, V_m$.

These bindings are usually determined by the instance creation expression that invoked the constructor (directly or indirectly). However, they may also be determined by a reflective call.

If k is redirecting then its redirect clause has the form

this.$g(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$

where g identifies another generative constructor of the immediately surrounding class. Then execution of k to initialize i proceeds by evaluating the argument list $(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$, and then executing g to initialize i with respect to the bindings resulting from the evaluation of $(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ and with **this** bound to i and the type parameters of the immediately enclosing class bound to $V_1, \ldots, V_m$.

Otherwise, execution proceeds as follows:

The instance variable declarations of the immediately enclosing class are visited in the order they appear in the program text. For each such declaration d, if d has the form $finalConstVarOrType\ v = e;$ then e is evaluated to an object o and the instance variable v of i is bound to o.

Any initializing formals declared in k's parameter list are executed in the order they appear in the program text. Then, the initializers of k's initializer list are executed to initialize i in the order they appear in the program.

We could observe the order by side effecting external routines called. So we need to specify the order.

Then if any instance variable of i declared by the immediately enclosing class is not yet bound to a value, all such variables are initialized with the null object (16.2).

Then, unless the enclosing class is Object, the explicitly specified or implicitly added superinitializer (10.6.1) is executed to further initialize i.

The super constructor call can be written anywhere in the initializer list of k, but the actual call always happens after all initializers have been processed. It is not equivalent to moving the super call to the end of the initializer list because the argument expressions may have visible side effects which must happen in the order the expressions occur in the program text.

After the superinitializer has completed, the body of k is executed in a scope where **this** is bound to i.

*This process ensures that no uninitialized final instance variable is ever seen by code. Note that **this** is not in scope on the right hand side of an initializer*

(see 16.11) so no instance method can execute during initialization: an instance method cannot be directly invoked, nor can ***this*** *be passed into any other code being invoked in the initializer.*

During the execution of a generative constructor to initialize an instance i, execution of an initializer of the form **this**.$v = e$ proceeds as follows:

First, the expression e is evaluated to an object o. Then, the instance variable v of i is bound to o. In checked mode, it is a dynamic type error if o is not the null object (16.2) and the interface of the class of o is not a subtype of the actual type of the instance variable v.

An initializer of the form $v = e$ is equivalent to an initializer of the form **this**.$v = e$.

Execution of an initializer that is an assertion proceeds by executing the assertion (17.17).

Execution of a superinitializer of the form

super$(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k}: a_{n+k})$

(respectively **super**.$id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k}: a_{n+k})$)

proceeds as follows:

First, the argument list $(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ is evaluated.

Then, after the remainder of the initializer list of k has been executed, the superconstructor is executed as follows:

Let C be the class in which the superinitializer appears and let S be the superclass of C. If S is generic (14), let $U_1, \ldots, U_m$ be the actual type arguments passed to S in the superclass clause of C.

The generative constructor S (respectively $S.id$) of S is executed to initialize i with respect to the bindings that resulted from the evaluation of $(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$, and the type parameters (if any) of class S bound to $U_1, \ldots, U_m$.

It is a compile-time error if class S does not declare a generative constructor named S (respectively $S.id$).

10.6.2 Factories

A *factory* is a constructor prefaced by the built-in identifier (16.34) **factory**.

factoryConstructorSignature:
  factory identifier ('.' identifier)? formalParameterList
  ;

The *return type* of a factory whose signature is of the form **factory** M or the form **factory** $M.id$ is M if M is not a generic type; otherwise the return type is $M<T_1, \ldots, T_n>$ where $T_1, \ldots, T_n$ are the type parameters of the enclosing class.

It is a compile-time error if M is not the name of the immediately enclosing class.

In checked mode, it is a dynamic type error if a factory returns a non-null object whose type is not a subtype of its actual (19.8.1) return type.

It seems useless to allow a factory to return the null object (16.2). But it is more uniform to allow it, as the rules currently do.

Factories address classic weaknesses associated with constructors in other languages. Factories can produce instances that are not freshly allocated: they can come from a cache. Likewise, factories can return instances of different classes.

Redirecting Factory Constructors

A *redirecting factory constructor* specifies a call to a constructor of another class that is to be used whenever the redirecting constructor is called.

redirectingFactoryConstructorSignature:
  const? **factory** identifier ('.' identifier)? formalParameterList
    '=' type ('.' identifier)?
;

Calling a redirecting factory constructor k causes the constructor k' denoted by *type* (respectively, $type.identifier$) to be called with the actual arguments passed to k, and returns the result of k' as the result of k. The resulting constructor call is governed by the same rules as an instance creation expression using **new** (16.12).

It follows that if *type* or *type.id* are not defined, or do not refer to a class or constructor, a dynamic error occurs, as with any other undefined constructor call. The same holds if k is called with fewer required parameters or more positional parameters than k' expects, or if k is called with a named parameter that is not declared by k'.

It is a compile-time error if k explicitly specifies a default value for an optional parameter. Default values specified in k would be ignored, since it is the *actual* parameters that are passed to k'. Hence, default values are disallowed.

It is a run-time error if a redirecting factory constructor redirects to itself, either directly or indirectly via a sequence of redirections.

If a redirecting factory F_1 redirects to another redirecting factory F_2 and F_2 then redirects to F_1, then both F_1 and F_2 are ill-defined. Such cycles are therefore illegal.

It is a static warning if *type* does not denote a class accessible in the current scope; if *type* does denote such a class C it is a static warning if the referenced constructor (be it *type* or *type.id*) is not a constructor of C.

Note that it is not possible to modify the arguments being passed to k'. *At first glance, one might think that ordinary factory constructors could simply create instances of other classes and return them, and that redirecting factories are unnecessary. However, redirecting factories have several advantages:*

- *An abstract class may provide a constant constructor that utilizes the constant constructor of another class.*

- *A redirecting factory constructors avoids the need for forwarders to repeat the default values for formal parameters in their signatures.*

It is a compile-time error if k is prefixed with the **const** modifier but k' is not a constant constructor (10.6.3).

It is a static warning if the function type of k' is not a subtype of the type of k.

This implies that the resulting object conforms to the interface of the immediately enclosing class of k.

It is a static type warning if any of the type arguments to k' are not subtypes of the bounds of the corresponding formal type parameters of *type*.

10.6.3 Constant Constructors

A *constant constructor* may be used to create compile-time constant (16.1) objects. A constant constructor is prefixed by the reserved word **const**.

constantConstructorSignature:
const qualified formalParameterList
;

All the work of a constant constructor must be handled via its initializers.

It is a compile-time error if a constant constructor is declared by a class that has a non-final instance variable.

The above refers to both locally declared and inherited instance variables.

It is a compile-time error if a constant constructor is declared by a class C if any instance variable declared in C is initialized with an expression that is not a constant expression.

A superclass of C cannot declare such an initializer either, because it must necessarily declare constant constructor as well (unless it is Object, which declares no instance variables).

The superinitializer that appears, explicitly or implicitly, in the initializer list of a constant constructor must specify a constant constructor of the superclass of the immediately enclosing class or a compile-time error occurs.

Any expression that appears within the initializer list of a constant constructor must be a potentially constant expression, or a compile-time error occurs.

A *potentially constant expression* is an expression e that would be a valid constant expression if all formal parameters of e's immediately enclosing constant constructor were treated as compile-time constants that were guaranteed to evaluate to an integer, boolean or string value as required by their immediately enclosing superexpression, and where e is also a valid expression if all the formal parameters are treated as non-constant variables.

Note that a parameter that is not used in a superexpression that is restricted to certain types can be a constant of any type. For example

```
class A {
 final m;
```

```
  const A(this.m);
}
```

can be instantiated via **const** A(**const**[]);

The difference between a potentially constant expression and a compile-time constant expression (16.12.2) deserves some explanation.

The key issue is whether one treats the formal parameters of a constructor as compile-time constants.

If a constant constructor is invoked from a constant object expression, the actual arguments will be required to be compile-time constants. Therefore, if we were assured that constant constructors were always invoked from constant object expressions, we could assume that the formal parameters of a constructor were compile-time constants.

However, constant constructors can also be invoked from ordinary instance creation expressions (16.12.1), and so the above assumption is not generally valid.

Nevertheless, the use of the formal parameters of a constant constructor within the constructor is of considerable utility. The concept of potentially constant expressions is introduced to facilitate limited use of such formal parameters. Specifically, we allow the usage of the formal parameters of a constant constructor for expressions that involve built-in operators, but not for constant objects, lists and maps. This allows for constructors such as:

```
class C {
  final x; final y; final z;
  const C(p, q): x = q, y = p + 100, z = p + q;
}
```

The assignment to x is allowed under the assumption that q is a compile-time constant (even though q is not, in general a compile-time constant). The assignment to y is similar, but raises additional questions. In this case, the superexpression of p is p + 100, and it requires that p be a numeric compile-time constant for the entire expression to be considered constant. The wording of the specification allows us to assume that p evaluates to an integer. A similar argument holds for p and q in the assignment to z.

However, the following constructors are disallowed:

```
class D {
  final w;
  const D.makeList(p): w = const [p]; // compile-time error
  const D.makeMap(p): w = const {"help": q}; // compile-time error
  const D.makeC(p): w = const C(p, 12); // compile-time error
}
```

The problem is not that the assignments to w are not potentially constant; they are. However, all these run afoul of the rules for constant lists (16.7), maps (16.8) and objects (16.12.2), all of which independently require their subexpressions to be constant expressions.

All of the illegal constructors of D above could not be sensibly invoked via ***new****, because an expression that must be constant cannot depend on a formal parameter, which may or may not be constant. In contrast, the legal examples*

make sense regardless of whether the constructor is invoked via ***const*** *or via* ***new****.*

Careful readers will of course worry about cases where the actual arguments to C() *are constants, but are not numeric. This is precluded by the following rule, combined with the rules for evaluating constant objects (16.12.2).*

When invoked from a constant object expression, a constant constructor must throw an exception if any of its actual parameters is a value that would prevent one of the potentially constant expressions within it from being a valid compile-time constant.

10.7 Static Methods

Static methods are functions, other than getters or setters, whose declarations are immediately contained within a class declaration and that are declared **static**. The static methods of a class C are those static methods declared by C.

The effect of a static method declaration in class C is to add an instance method with the same name and signature to the Type object for class C that forwards (9.1) to the static method.

Inheritance of static methods has little utility in Dart. Static methods cannot be overridden. Any required static function can be obtained from its declaring library, and there is no need to bring it into scope via inheritance. Experience shows that developers are confused by the idea of inherited methods that are not instance methods.

Of course, the entire notion of static methods is debatable, but it is retained here because so many programmers are familiar with it. Dart static methods may be seen as functions of the enclosing library.

Static method declarations may conflict with other declarations (10.11).

10.8 Static Variables

Static variables are variables whose declarations are immediately contained within a class declaration and that are declared **static**. The static variables of a class C are those static variables declared by C.

10.9 Superclasses

The superclass S' of a class C that has a with clause **with** $M_1, \ldots, M_k$ and an extends clause **extends** S is the application of mixin composition (12) $M_k * \cdots * M_1$ to S. The name S' is a fresh identifier. If no **with** clause is specified then the **extends** clause of a class C specifies its superclass. If no **extends** clause is specified, then either:

- C is Object, which has no superclass. OR
- Class C is deemed to have an **extends** clause of the form **extends** Object, and the rules above apply.

It is a compile-time error to specify an **extends** clause for class Object.

superclass:
  extends type
;

The scope of the **extends** and **with** clauses of a class C is the type-parameter scope of C.

It is a compile-time error if the **extends** clause of a class C specifies a type variable (14), a type alias (19.3.1), an enumerated type (13), a malformed type, or a deferred type (19.1) as a superclass. It is a compile-time error if the **extends** clause of a class C specifies type **dynamic** as a superinterface.

The type parameters of a generic class are available in the lexical scope of the superclass clause, potentially shadowing classes in the surrounding scope. The following code is therefore illegal and should cause a compile-time error:

```
class T {}
/* Compilation error: Attempt to subclass a type parameter */
class G<T> extends T {}
```

A class S is *a superclass* of a class C iff either:

- S is the superclass of C, or
- S is a superclass of a class S', and S' is the superclass of C.

It is a compile-time error if a class C is a superclass of itself.

10.9.1 Inheritance and Overriding

Let C be a class, let A be a superclass of C, and let $S_1, \ldots, S_k$ be superclasses of C that are also subclasses of A. C *inherits* all accessible instance members of A that have not been overridden by a declaration in C or in at least one of $S_1, \ldots, S_k$.

It would be more attractive to give a purely local definition of inheritance, that depended only on the members of the direct superclass S. However, a class C can inherit a member m that is not a member of its superclass S. This can occur when the member m is private to the library L_1 of C, whereas S comes from a different library L_2, but the superclass chain of S includes a class declared in L_1.

A class may override instance members that would otherwise have been inherited from its superclass.

Let $C = S_0$ be a class declared in library L, and let $\{S_1, \ldots, S_k\}$ be the set of all superclasses of C, where S_i is the superclass of S_{i-1} for $i \in 1..k$. Let C declare a member m, and let m' be a member of $S_j, j \in 1..k$, that has the same name as m, such that m' is accessible to L. Then m overrides m' if m' is not already overridden by a member of at least one of $S_1, \ldots, S_{j-1}$ and neither m nor m' are instance variables.

Instance variables never override each other. The getters and setters induced by instance variables do.

Again, a local definition of overriding would be preferable, but fails to account for library privacy.

Whether an override is legal or not is described elsewhere in this specification (see 10.1, 10.2 and 10.3).

For example getters may not legally override methods and vice versa. Setters never override methods or getters, and vice versa, because their names always differ.

It is nevertheless convenient to define the override relation between members in this way, so that we can concisely describe the illegal cases.

Note that instance variables do not participate in the override relation, but the getters and setters they induce do. Also, getters don't override setters and vice versa. Finally, static members never override anything.

It is a static warning if a concrete class inherits an abstract method.

For convenience, here is a summary of the relevant rules. Remember that this is not normative. The controlling language is in the relevant sections of the specification.

1. There is only one namespace for getters, setters, methods and constructors (6.1). An instance or static variable f introduces a getter f and a non-final instance or static variable f also introduces a setter $f=$ (10.5, 10.8). When we speak of members here, we mean accessible instance or static variables, getters, setters, and methods (10).

2. You cannot have two members with the same name in the same class - be they declared or inherited (6.1, 10).

3. Static members are never inherited.

4. It is a warning if you have an static member named m in your class or any superclass (even though it is not inherited) and an instance member of the same name (10.1, 10.2, 10.3).

5. It is a warning if you have a static setter $v=$, and an instance member v (10.3).

6. It is a warning if you have a static getter v and an instance setter $v=$ (10.2).

7. If you define an instance member named m, and your superclass has an instance member of the same name, they override each other. This may or may not be legal.

8. If two members override each other, it is a static warning if their type signatures are not assignable to each other (10.1, 10.2, 10.3) (and since these are function types, this means the same as "subtypes of each other").

9. If two members override each other, it is a static warning if the overriding member has more required parameters than the overridden one (10.1).

10. If two members override each other, it is a static warning if the overriding member has fewer positional parameters than the overridden one (10.1).

11. If two members override each other, it is a static warning if the overriding member does not have all the named parameters that the overridden one has (10.1).

12. Setters, getters and operators never have optional parameters of any kind; it's a compile-time error (10.1.1, 10.2, 10.3).

13. It is a compile-time error if a member has the same name as its enclosing class (10).

14. A class has an implicit interface (10).

15. Superinterface members are not inherited by a class, but are inherited by its implicit interface. Interfaces have their own inheritance rules (11.1.1).

16. A member is abstract if it has no body and is not labeled **external** (10.4, 9.4).

17. A class is abstract iff it is explicitly labeled **abstract**.

18. It is a static warning if a concrete class has an abstract member (declared or inherited).

19. It is a static warning and a dynamic error to call a non-factory constructor of an abstract class (16.12.1).

20. If a class defines an instance member named m, and any of its superinterfaces have a member named m, the interface of the class overrides m.

21. An interface inherits all members of its superinterfaces that are not overridden and not members of multiple superinterfaces.

22. If multiple superinterfaces of an interface define a member with the same name m, then at most one member is inherited. That member (if it exists) is the one whose type is a subtype of all the others. If there is no such member, then:

    - A static warning is given.
    - If possible the interface gets a member named m that has the minimum number of required parameters among all the members in the superinterfaces, the maximal number of positionals, and the superset of named parameters. The types of these are all **dynamic**. If this is impossible then no member m appears in the interface.

    (11.1.1)

23. Rule 8 applies to interfaces as well as classes (11.1.1).

24. It is a static warning if a concrete class does not have an implementation for a method in any of its superinterfaces unless it has a concrete noSuchMethod method (10.10) distinct from the one in class Object.

25. The identifier of a named constructor cannot be the same as the name of a member declared (as opposed to inherited) in the same class (10.6).

10.10 Superinterfaces

A class has a set of direct superinterfaces. This set includes the interface of its superclass and the interfaces specified in the **implements** clause of the class.

interfaces:
  implements typeList
 ;

The scope of the **implements** clause of a class C is the type-parameter scope of C.

It is a compile-time error if the **implements** clause of a class C specifies a type variable (14), a type alias (19.3.1), an enumerated type (13), a malformed type, or a deferred type (19.1) as a superinterface. It is a compile-time error if the **implements** clause of a class C specifies type **dynamic** as a superinterface. It is a compile-time error if the **implements** clause of a class C specifies a type T as a superinterface more than once. It is a compile-time error if the superclass of a class C is specified as a superinterface of C. It is a compile-time error if a class C has two superinterfaces that are different instantiations of the same generic class. For example, a class may not have both 'List<int>' and 'List<num>' as superinterfaces.

One might argue that it is harmless to repeat a type in the superinterface list, so why make it an error? The issue is not so much that the situation described in program source is erroneous, but that it is pointless. As such, it is an indication that the programmer may very well have meant to say something else - and that is a mistake that should be called to her or his attention. Nevertheless, we could simply issue a warning; and perhaps we should and will. That said, problems like these are local and easily corrected on the spot, so we feel justified in taking a harder line.

It is a compile-time error if the interface of a class C is a superinterface of itself.

Let C be a concrete class that does not have a concrete noSuchMethod() method distinct from the one declared in class Object. It is a static warning if the implicit interface of C includes an instance member m of type F and C does not declare or inherit a corresponding concrete instance member m of type F' such that $F' <: F$.

A class does not inherit members from its superinterfaces. However, its implicit interface does.

We choose to issue these warnings only for concrete classes; an abstract class might legitimately be designed with the expectation that concrete subclasses will implement part of the interface. We also disable these warnings if a concrete noSuchMethod() *declaration is present or inherited from any class other than* Object*. In such cases, the supported interface is going to be implemented via* noSuchMethod() *and no actual declarations of the implemented interface's members are needed. This allows proxy classes for specific types to be implemented without provoking type warnings.*

It is a static warning if the implicit interface of a class C includes an instance member m of type F and C declares or inherits a corresponding instance member m of type F' if F' is not a subtype of F.

However, if a class does explicitly declare a member that conflicts with its superinterface, this always yields a static warning.

10.11 Class Member Conflicts

Some pairs of class member declarations cannot coexist, even though they do not both introduce the same name into the same scope. This section specifies these errors.

The *basename* of a getter or method named n is n; the basename of a setter named $n=$ is n.

Let C be a class. It is a compile-time error if C declares a

- constructor named $C.n$ and a static member with basename n.
- getter or a setter with basename n, and has a method named n.
- method named n, and has a getter or a setter with basename n.
- static member with basename n, and has an instance member with basename n.

These errors occur when the getters or setters are defined explicitly as well as when they are induced by variable declarations.

11 Interfaces

An *interface* defines how one may interact with an object. An interface has methods, getters and setters and a set of superinterfaces.

11.1 Superinterfaces

An interface has a set of direct superinterfaces.

An interface J is a superinterface of an interface I iff either J is a direct superinterface of I or J is a superinterface of a direct superinterface of I.

11.1.1 Inheritance and Overriding

Let J be an interface and K be a library. We define $inherited(J, K)$ to be the set of members m such that all of the following hold:

- m is accessible to K and
- A is a direct superinterface of J and either
  - A declares a member m or
  - m is a member of $inherited(A, K)$.
- m is not overridden by J.

Furthermore, we define $overrides(J, K)$ to be the set of members m' such that all of the following hold:

- J is the implicit interface of a class C.
- C declares a member m.
- m' has the same name as m.
- m' is accessible to K.
- A is a direct superinterface of J and either
  - A declares a member m' or
  - m' is a member of $inherited(A, K)$.

Let I be the implicit interface of a class C declared in library L. I *inherits* all members of $inherited(I, L)$ and I *overrides* m' if $m' \in overrides(I, L)$.

All the static warnings pertaining to the overriding of instance members given in section 10 above hold for overriding between interfaces as well.

It is a static warning if m is a method and m' is a getter, or if m is a getter and m' is a method.

However, if the above rules would cause multiple members $m_1, \ldots, m_k$ with the same name n to be inherited (because identically named members existed in several superinterfaces) then at most one member is inherited.

If some but not all of the $m_i, 1 \leq i \leq k$ are getters none of the m_i are inherited, and a static warning is issued.

Otherwise, if the static types $T_1, \ldots, T_k$ of the members $m_1, \ldots, m_k$ are not identical then there must be an $x \in 1..k$ such that $T_x <: T_i$ for all $i \in 1..k$, or a static type warning occurs. The member that is inherited is m_x, if it exists; otherwise: let $numberOfPositionals(f)$ denote the number of positional parameters of a function f, and let $numberOfRequiredParams(f)$ denote the number of required parameters of a function f. Furthermore, let s denote the set of all named parameters of the $m_1, \ldots, m_k$. Then let

$h = max(numberOfPositionals(m_i))$,

$r = min(numberOfRequiredParams(m_i)), i \in 1..k$.

Then I has a method named n, with r required parameters of type **dynamic**, h positional parameters of type **dynamic**, named parameters s of type **dynamic** and return type **dynamic**.

The only situation where the run-time system would be concerned with this would be during reflection, if a mirror attempted to obtain the signature of an interface member.

The current solution is a tad complex, but is robust in the face of type annotation changes. Alternatives: (a) No member is inherited in case of conflict. (b) The first m is selected (based on order of superinterface list). (c) Inherited member chosen at random.

(a) means that the presence of an inherited member of an interface varies depending on type signatures. (b) is sensitive to irrelevant details of the declaration, and (c) is liable to give unpredictable results between implementations or even between different compilation sessions.

12 Mixins

A mixin describes the difference between a class and its superclass. A mixin is always derived from an existing class declaration.

It is a compile-time error to derive a mixin from a class which explicitly declares a generative constructor. It is a compile-time error to derive a mixin from a class which has a superclass other than Object.

This restriction is temporary. We expect to remove it in later versions of Dart.

The restriction on constructors simplifies the construction of mixin applications because the process of creating instances is simpler.

12.1 Mixin Application

A mixin may be applied to a superclass, yielding a new class. Mixin application occurs when one or more mixins are mixed into a class declaration via its **with** clause. The mixin application may be used to extend a class per section (10); alternatively, a class may be defined as a mixin application as described in this section. It is a compile-time error if the **with** clause of a mixin application C includes a type variable (14), a type alias (19.3.1), an enumerated type (13), a malformed type, or a deferred type (19.1).

mixinApplicationClass:
  identifier typeParameters? '=' mixinApplication ';'
;

mixinApplication:
  type mixins interfaces?
;

A mixin application of the form S **with** M; for the name N defines a class C with superclass S and name N.

A mixin application of the form S **with** $M_1, \ldots, M_k$; for the name N defines a class C whose superclass is the application of the mixin composition (12.2) $M_{k-1} * \ldots * M_1$ to S of a name that is a fresh identifer, and whose name is N. *The name of the resulting class is necessary because it is part of the names of the introduced constructors.*

In both cases above, C declares the same instance members as M (respectively, M_k), and it does not declare any static members. If any of the instance variables of M (respectively, M_k) have initializers, they are executed in the instance scope of M (respectively, M_k) to initialize the corresponding instance variables of C.

Let L_C be the library containing the mixin application. That is, the library containing the clause S **with** M or the clause S_0 **with** $M_1, \ldots, M_k, M$.

Let N_C be the name of the mixin application class C, let S be the superclass of C, and let S_N be the name of S.

For each generative constructor of the form $S_q(T_1\ a_1, \ldots, T_k\ a_k)$ of S that is accessible to L_C, C has an implicitly declared constructor of the form

$C_q(T_1\ a_1, \ldots, T_k\ a_k)$:**super**$_q(a_1, \ldots, a_k)$;

where C_q is obtained from S_q by replacing occurrences of S_N, which denote the superclass, by N_C, and **super**$_q$ is obtained from S_q by replacing occurrences of S_N which denote the superclass by **super**. If S_q is a generative const constructor, and M does not declare any fields, C_q is also a const constructor.

For each generative constructor of the form $S_q(T_1\ a_1, \ldots, T_k\ a_k, [T_{k+1}\ a_{k+1} = d_1, \ldots, T_{k+p}\ a_{k+p} = d_p])$ of S that is accessible to L_C, C has an implicitly declared constructor of the form

$C_q(T_1\ a_1, \ldots, T_k\ a_k, [T_{k+1}\ a_{k+1} = d'_1, \ldots, T_{k+p}\ a_{k+p} = d'_p])$
: **super**$_q(a_1, \ldots, a_k, a_{k+1}, \ldots, a_p)$;

where C_q is obtained from S_q by replacing occurrences of S_N, which denote the superclass, by N_C, **super**$_q$ is obtained from S_q by replacing occurrences of S_N which denote the superclass by **super**, and d'_i, $i \in 1..p$, is a compile-time constant expression evaluating to the same value as d_i. If S_q is a generative const constructor, and M does not declare any fields, C_q is also a const constructor.

For each generative constructor of the form $S_q(T_1\ a_1, \ldots, T_k\ a_k, \{T_{k+1}\ a_{k+1} = d_1, \ldots, T_{k+n}\ a_{k+n} = d_n\})$ of S that is accessible to L_C, C has an implicitly declared constructor of the form

$C_q(T_1\ a_1, \ldots, T_k\ a_k, \{T_{k+1}\ a_{k+1} = d'_1, \ldots, T_{k+n}\ a_{k+n} = d'_n\})$
: **super**$_q(a_1, \ldots, a_k, a_{k+1}: a_{k+1}, \ldots, a_p: a_p)$;

where C_q is obtained from S_q by replacing occurrences of S_N which denote the superclass by N_C, **super**$_q$ is obtained from S_q by replacing occurrences of S_N which denote the superclass by **super**, and d'_i, $i \in 1..n$, is a compile-time constant expression evaluating to the same value as d_i. If S_q is a generative const constructor, and M does not declare any fields, C_q is also a const constructor.

If the mixin application class declares interfaces, the resulting class also implements those interfaces.

It is a compile-time error if S is an enumerated type (13) or a malformed

type. It is a compile-time error if M (respectively, any of $M_1, \ldots, M_k$) is an enumerated type (13) or a malformed type. It is a compile-time error if a well formed mixin cannot be derived from M (respectively, from each of $M_1, \ldots, M_k$).

Let K be a class declaration with the same constructors, superclass and interfaces as C, and the instance members declared by M (respectively $M_1, \ldots, M_k$). It is a static warning if the declaration of K would cause a static warning. It is a compile-time error if the declaration of K would cause a compile-time error.

If, for example, M declares an instance member im whose type is at odds with the type of a member of the same name in S, this will result in a static warning just as if we had defined K by means of an ordinary class declaration extending S, with a body that included im.

The effect of a class definition of the form **class** $C = M;$ or the form **class** $C < T_1, \ldots, T_n > = M;$ in library L is to introduce the name C into the scope of L, bound to the class (10) defined by the mixin application M for the name C. The name of the class is also set to C. Iff the class is prefixed by the built-in identifier **abstract**, the class being defined is an abstract class.

Let M_A be a mixin derived from a class M with direct superclass S_{static}, e.g., as defined by the class declaration class M extends S_{static} { ... }.

Let A be an application of M_A. It is a static warning if the superclass of A is not a subtype of S_{static}.

Let C be a class declaration that includes M_A in a with clause. It is a static warning if C does not implement, directly or indirectly, all the direct superinterfaces of M.

12.2 Mixin Composition

Dart does not directly support mixin composition, but the concept is useful when defining how the superclass of a class with a mixin clause is created.

The *composition of two mixins*, $M_1 < T_1, \ldots, T_{k_{M_1}} >$ and $M_2 < U_1, \ldots, U_{k_{M_2}} >$, written $M_1 < T_1, \ldots, T_{k_{M_1}} > * M_2 < U_1, \ldots, U_{k_{M_2}} >$ defines an anonymous mixin such that for any class $S < V_1, \ldots, V_{k_S} >$, the application of
$M_1 < T_1, \ldots, T_{k_{M_1}} > * M_2 < U_1, \ldots, U_{k_{M_2}} >$
to $S < V_1, \ldots, V_{k_S} >$ for the name C is equivalent to
abstract class $C < T_1, \ldots, T_{k_{M_1}}, U_1, \ldots, U_{k_{M_2}}, V_1, \ldots, V_{k_S} > =$
$\quad Id_2 < U_1, \ldots, U_{k_{M_2}}, V_1 \ldots V_{k_S} >$ **with** $M_1 < T_1, \ldots, T_{k_{M_1}} >;$
where Id_2 denotes
abstract class $Id_2 < U_1, \ldots, U_{k_{M_2}}, V_1, \ldots, V_{k_S} > =$
$\quad S < V_1, \ldots, V_{k_S} >$ **with** $M_2 < U_1, \ldots, U_{k_{M_2}} >;$
and Id_2 is a unique identifier that does not exist anywhere in the program.

The intermediate classes produced by mixin composition are regarded as abstract because they cannot be instantiated independently. They are only introduced as anonymous superclasses of ordinary class declarations and mixin applications. Consequently, no warning is given if a mixin composition includes abstract members, or incompletely implements an interface.

Mixin composition is associative.

Note that any subset of M_1, M_2 and S may or may not be generic. For any non-generic declaration, the corresponding type parameters may be elided, and if no type parameters remain in the derived declarations C and/or Id_2 then the those declarations need not be generic either.

13 Enums

An *enumerated type*, or *enum*, is used to represent a fixed number of constant values.

enumType:
  metadata **enum** identifier
      '{' enumEntry (',' enumEntry)* (',')? '}'
;

enumEntry:
  metadata identifier
;

The declaration of an enum of the form m **enum** E $\{m_0\ id_0, \ldots, m_{n-1}\ id_{n-1}\}$ has the same effect as a class declaration

m **class** E {
  final int index;
  const E(**this**.index);
  m_0 **static const** E id_0 = **const** E(0);
  ...
  m_{n-1} **static const** E id_{n-1} = const E(n - 1);
  static const List<E> values = const <E>[id_0, ..., id_{n-1}];
  String toString() => { 0: 'E.id_0', ..., n-1: 'E.id_{n-1}'}[index]
}

It is also a compile-time error to subclass, mix-in or implement an enum or to explicitly instantiate an enum. These restrictions are given in normative form in sections 10.9, 10.10, 12.1 and 16.12 as appropriate.

14 Generics

A class declaration (10), type alias (19.3.1), or function (9) G may be *generic*, that is, G may have formal type parameters declared.

When an entity in this specification is described as generic, and the special case is considered where the number of type arguments is zero, the type argument list should be omitted.

This allows non-generic cases to be included implicitly as special cases. For example, an invocation of a non-generic function arises as the special case where the function takes zero type arguments, and zero type arguments are passed. In

this situation some operations are also omitted (have no effect), e.g., operations where formal type parameters are replaced by actual type arguments.

A *generic class declaration* introduces a generic class into the enclosing library scope. A *generic class* is a mapping that accepts a list of actual type arguments and maps them to a class. Consider a generic class declaration G named C with formal type parameter declarations X_1 **extends** $B_1, \ldots, X_m$ **extends** B_m, and a parameterized type T of the form C<$T_1, \ldots, T_l$>.

It is a static warning if $m \neq l$. It is a static warning if T is not well-bounded (14.2).

That is, if the number of type arguments is wrong, or one or more of the upper bounds has been violated.

Otherwise, said parameterized type C<$T_1, \ldots, T_m$> denotes an application of the generic class declared by G to the type arguments $T_1, \ldots, T_m$. This yields a class C' whose members are equivalent to those of a class declaration which is obtained from the declaration of G by replacing each occurrence of X_j by T_j.

Other properties of C' such as the subtype relationships are specified elsewhere (19.4).

A *generic type alias* introduces a mapping from actual type argument lists to types. Consider a generic type alias declaration G named F with formal type parameter declarations X_1 **extends** $B_1, \ldots, X_m$ **extends** B_m, and right hand side T, and the parameterized type S of the form F<$T_1, \ldots, T_l$>.

It is a static warning if $m \neq l$. It is a static warning if S is not well-bounded (14.2).

That is, if the number of type arguments is wrong, or one or more of the upper bounds has been violated.

Otherwise, said parameterized type F<$T_1, \ldots, T_m$> denotes an application of the mapping denoted by G to the type arguments $T_1, \ldots, T_m$. This yields the type $[T_1/X_1, \ldots, T_m/X_m]T$.

A generic type alias does not correspond to any entities at run time, it is only an alias for an existing type. Hence, we may consider it as syntactic sugar which is eliminated before the program runs.

A *generic type* is a type which is introduced by a generic class declaration or a generic type alias, or it is the type FutureOr.

A *generic function declaration* introduces a generic function (9.2) into the enclosing scope. Consider a function invocation expression of the form f<$T_1, \ldots, T_l$>(...), where the static type of f is a generic function type with formal type parameters X_1 **extends** $B_1, \ldots, X_m$ **extends** B_m.

It is a static warning if $m \neq l$. It is a static warning if there exists a j such that T_j is not a subtype of $[T_1/X_1, \ldots, T_m/X_m]B_j$.

That is, if the number of type arguments is wrong, or if the jth actual type argument is not a subtype of the corresponding bound, where each formal type parameter has been replaced by the corresponding actual type argument.

typeParameter:
metadata identifier (**extends** type)?

;

typeParameters:
'<' typeParameter (',' typeParameter)* '>'
;

A type parameter T may be suffixed with an **extends** clause that specifies the *upper bound* for T. If no **extends** clause is present, the upper bound is Object. It is a static type warning if a type parameter is a supertype of its upper bound. The bounds of type variables are a form of type annotation and have no effect on execution in production mode.

Type parameters are declared in the type parameter scope of a class or function. The type parameters of a generic G are in scope in the bounds of all of the type parameters of G. The type parameters of a generic class declaration G are also in scope in the **extends** and **implements** clauses of G (if these exist) and in the body of G. However, a type parameter of a generic class is considered to be a malformed type when referenced by a static member.

The scopes associated with the type parameters of a generic function are described in (9.2).

*The restriction on static members is necessary since a type variable has no meaning in the context of a static member, because statics are shared among all generic instantiations of a generic class. However, a type variable may be referenced from an instance initializer, even though **this** is not available.*

Because type parameters are in scope in their bounds, we support F-bounded quantification (if you don't know what that is, don't ask). This enables typechecking code such as:

```
class Ordered<T> {
  operator > (T x);
}
class Sorter<T extends Ordered<T>> {
  sort(List<T> l) ... l[n] < l[n+1] ...
}
```

Even where type parameters are in scope there are numerous restrictions at this time:

- A type parameter cannot be used to name a constructor in an instance creation expression (16.12).
- A type parameter cannot be used as a superclass or superinterface (10.9, 10.10, 11.1).
- A type parameter cannot be used as a generic type.

The normative versions of these are given in the appropriate sections of this specification. Some of these restrictions may be lifted in the future.

14.1 Variance

We say that a type S *occurs covariantly* in a type T iff S occurs in a covariant position in T, but not in a contravariant position, and not in an invariant position.

We say that a type S *occurs contravariantly* in a type T iff S occurs in a contravariant position in T, but not in a covariant position, and not in an invariant position.

We say that a type S *occurs invariantly* in a type T iff S occurs in an invariant position in T, or S occurs in a covariant position as well as a contravariant position.

We say that a type S occurs *in a covariant position* in a type T iff one of the following conditions is true:

- T is S
- T is of the form $G<S_1, \ldots, S_n>$ where G denotes a generic class and S occurs in a covariant position in S_j for some $j \in 1..n$.
- T is of the form S_0 **Function**$<X_1$ **extends** $B_1, \ldots>(S_1\ x_1, \ldots)$ where the type parameter list may be omitted, and S occurs in a covariant position in S_0.
- T is of the form

  S_0 **Function**$<X_1$ **extends** $B_1, \ldots>$

  $(S_1\ x_1, \ldots, S_k\ x_k, [S_{k+1}\ x_{k+1} = d_{k+1}, \ldots, S_n\ x_n = d_n])$

  or of the form

  S_0 **Function**$<X_1$ **extends** $B_1, \ldots>$

  $(S_1\ x_1, \ldots, S_k\ x_k, \{S_{k+1}\ x_{k+1} = d_{k+1}, \ldots, S_n\ x_n = d_n\})$

  where the type parameter list and each default value may be omitted, and S occurs in a contravariant position in S_j for some $j \in 1..n$.
- T is of the form $G<S_1, \ldots, S_n>$ where G denotes a parameterized type alias such that $j \in 1..n$, the formal type parameter corresponding to S_j is covariant, and S occurs in a covariant position in S_j.
- T is of the form $G<S_1, \ldots, S_n>$ where G denotes a parameterized type alias such that $j \in 1..n$, the formal type parameter corresponding to S_j is contravariant, and S occurs in a contravariant position in S_j.

We say that a type S occurs *in a contravariant position* in a type T iff one of the following conditions is true:

- T is of the form $G<S_1, \ldots, S_n>$ where G denotes a generic class and S occurs in a contravariant position in S_j for some $j \in 1..n$.

- T is of the form S_0 **Function**$<X_1$ **extends** $B_1, \ldots>(S_1\ x_1, \ldots)$ where the type parameter list may be omitted, and S occurs in a contravariant position in S_0.

- T is of the form

  S_0 **Function**$<X_1$ **extends** $B_1, \ldots>$

  $(S_1\ x_1, \ldots,\ S_k\ x_k, [S_{k+1}\ x_{k+1} = d_{k+1}, \ldots,\ S_n\ x_n = d_n])$

  or of the form

  S_0 **Function**$<X_1$ **extends** $B_1, \ldots>$

  $(S_1\ x_1, \ldots,\ S_k\ x_k, \{S_{k+1}\ x_{k+1} = d_{k+1}, \ldots,\ S_n\ x_n = d_n\})$

  where the type parameter list and each default value may be omitted, and S occurs in a covariant position in S_j for some $j \in 1..n$.

- T is of the form $G<S_1, \ldots,\ S_n>$ where G denotes a parameterized type alias such that $j \in 1..n$, the formal type parameter corresponding to S_j is covariant, and S occurs in a contravariant position in S_j.

- T is of the form $G<S_1, \ldots,\ S_n>$ where G denotes a parameterized type alias such that $j \in 1..n$, the formal type parameter corresponding to S_j is contravariant, and S occurs in a covariant position in S_j.

We say that a type S occurs *in an invariant position* in a type T iff one of the following conditions is true:

- T is of the form $G<S_1,\ \ldots,\ S_n>$ where G denotes a generic class or a parameterized type alias, and S occurs in an invariant position in S_j for some $j \in 1..n$.

- T is of the form

  S_0 **Function**$<X_1$ **extends** $B_1, \ldots,\ X_m$ **extends** $B_m>$

  $(S_1\ x_1, \ldots,\ S_k\ x_k, [S_{k+1}\ x_{k+1} = d_{k+1}, \ldots,\ S_n\ x_n = d_n])$

  or of the form

  S_0 **Function**$<X_1$ **extends** $B_1, \ldots,\ X_m$ **extends** $B_m>$

  $(S_1\ x_1, \ldots,\ S_k\ x_k, \{S_{k+1}\ x_{k+1} = d_{k+1}, \ldots,\ S_n\ x_n = d_n\})$

  where the type parameter list and each default value may be omitted, and S occurs in an invariant position in S_j for some $j \in 0..n$, or S occurs in B_i for some $i \in 1..m$.

- T is of the form $G<S_1, \ldots,\ S_n>$ where G denotes a parameterized type alias, $j \in 1..n$, the formal type parameter corresponding to S_j is invariant, and S occurs in S_j.

Consider a generic type alias declaration G with formal type parameter declarations X_1 **extends** $B_1, \ldots, X_m$ **extends** B_m, and right hand side T. Let $j \in 1..m$. We say that *the formal type parameter X_j is invariant* iff X_j occurs invariantly in T, X_j *is covariant* iff X_j occurs covariantly in T, and X_j *is contravariant* iff X_j occurs contravariantly in T.

Variance gives a characterization of the way a type varies as the value of a subterm varies, e.g., a type variable: Assume that T is a type where a type variable X occurs, and L and U are types such that L is a subtype of U. If X occurs covariantly in T then $[L/X]T$ is a subtype of $[U/X]T$. Similarly, if X occurs contravariantly in T then $[U/X]T$ is a subtype of $[L/X]T$. If X occurs invariantly then $[L/X]T$ and $[U/X]T$ are not guaranteed to be subtypes of each other in any direction. In short: with covariance, the type covaries; with contravariance, the type contravaries; with invariance, all bets are off.

14.2 Super-Bounded Types

This section describes how the declared upper bounds of formal type parameters are enforced, including some cases where a limited form of violation is allowed.

A *top type* is a type T such that Object is a subtype of T. For instance, Object, **dynamic**, and **void** are top types, and so are FutureOr<**void**> and FutureOr<FutureOr<**dynamic**>>.

Every type which is not a parameterized type is *regular-bounded*. Let T be a parameterized type of the form $G<S_1, \ldots, S_n>$ where G denotes a generic class or a parameterized type alias. Let X_1 **extends** $B_1, \ldots, X_n$ **extends** B_n be the formal type parameter declarations of G. T is *regular-bounded* iff S_j is a subtype of $[S_1/X_1, \ldots, S_n/X_n]B_j$, for all $j \in 1..n$.

This means that each actual type argument satisfies the declared upper bound for the corresponding formal type parameter.

Let T be a parameterized type of the form $G<S_1, \ldots, S_n>$ where G denotes a generic class or a parameterized type alias. T is *super-bounded* iff the following conditions are both true:

- T is not regular-bounded.
- For each $j \in 1..n$, let S'_j be the result of replacing every occurrence of a top type in a covariant position in S_j by Null, and every occurrence of Null in a contravariant position in S_j by Object. It is then required that $G<S'_1, \ldots, S'_n>$ is regular-bounded.

In short, at least one type argument violates its bound, but the type is regular-bounded after replacing all occurrences of an extreme type by an opposite extreme type, depending on their variance.

A type T is *well-bounded* iff it is either regular-bounded or super-bounded.

Any use of a type T which is not well-bounded is a compile-time error.

It is a compile-time error if a parameterized type T is super-bounded when it is used in any of the following ways:

- T is an immediate subterm of a new expression (16.12.1) or a constant object expression (16.12.2).
- T is an immediate subterm of a redirecting factory constructor signature (10.6.2).
- T is an immediate subterm of an **extends** clause (10.9), or it occurs as an element in the type list of an **implements** clause (10.10), or a **with** clause (10).

Types of members from super-bounded class types are computed using the same rules as types of members from other types. Types of function applications involving super-bounded types are computed using the same rules as types of function applications involving other types. Here is an example:

```
class A<X extends num> {
  X x;
}
A<Object> a;
```

With this, a.x has static type Object, even though the upper bound on the type variable X is num.

Super-bounded types enable the expression of informative common supertypes of some sets of types whose common supertypes would otherwise be much less informative. For example, consider the following class:

```
class C<X extends C<X>> {
  X next;
}
```

Without super-bounded types, there is no type T which makes C<T> a common supertype of all types of the form C<S> (noting that all types must be regular-bounded when we do not have the notion of super-bounded types). So if we wish to allow a variable to hold any instance "of type C" then that variable must use Object or another top type as its type annotation, which means that a member like next is not known to exist (which is what we mean by saying that the type is 'less informative').

We could introduce a notion of recursive (infinite) types, and express the least upper bound of all types of the form C<S> as some syntax whose meaning could be approximated by C<C<C<C<...>>>>.

However, we expect that any such concept in Dart would incur a significant cost on developers and implementations in terms of added complexity and subtlety, so we have chosen not to do that. Super-bounded types are finite, but they offer a useful developer-controlled approximation to such infinite types.

For example, C<Object> and C<C<C<**void**>>> are types that a developer may choose to use as a type annotation. This choice serves as a commitment to a finite level of unfolding of the infinite type, and it allows for a certain amount of control at the point where the unfolding ends: If c has type C<C<**dynamic**>> then c.next.next has type **dynamic** and c.next.next.whatever has no compile-time

error, but if c has type C<C<**void**>> then Object x = c.next.next; is a compile-time error. It is thus possible for developers to get a more or less strict treatment of expressions whose type proceeds beyond the given finite unfolding.

15 Metadata

Dart supports metadata which is used to attach user defined annotations to program structures.

metadata:
('@' qualified ('.' identifier)? (arguments)?)*
;

Metadata consists of a series of annotations, each of which begin with the character @, followed by a constant expression that starts with an identifier. It is a compile-time error if the expression is not one of the following:

- A reference to a compile-time constant variable.
- A call to a constant constructor.

Metadata is associated with the abstract syntax tree of the program construct p that immediately follows the metadata, assuming p is not itself metadata or a comment. Metadata can be retrieved at run time via a reflective call, provided the annotated program construct p is accessible via reflection.

Obviously, metadata can also be retrieved statically by parsing the program and evaluating the constants via a suitable interpreter. In fact many if not most uses of metadata are entirely static.

It is important that no run-time overhead be incurred by the introduction of metadata that is not actually used. Because metadata only involves constants, the time at which it is computed is irrelevant so that implementations may skip the metadata during ordinary parsing and execution and evaluate it lazily.

It is possible to associate metadata with constructs that may not be accessible via reflection, such as local variables (though it is conceivable that in the future, richer reflective libraries might provide access to these as well). This is not as useless as it might seem. As noted above, the data can be retrieved statically if source code is available.

Metadata can appear before a library, part header, class, typedef, type parameter, constructor, factory, function, parameter, or variable declaration and before an import, export or part directive.

The constant expression given in an annotation is type checked and evaluated in the scope surrounding the declaration being annotated.

16 Expressions

An *expression* is a fragment of Dart code that can be evaluated at run time. Evaluating an expression either *produces a value* (an object), or it *throws* an exception object and an associated stack trace. In the former case, we also say that the expression *evaluates to a value*.

Every expression has an associated static type (19.1). Every value has an associated dynamic type (19.2).

If evaluation of one expression, e, is defined in terms of evaluation of another expression, typically a subexpression of e, and the evaluation of the other expression throws an exception and a stack trace, the evaluation of e stops at that point and throws the same exception object and stack trace.

expression:
  assignableExpression assignmentOperator expression |
  conditionalExpression cascadeSection* |
  throwExpression
 ;

expressionWithoutCascade:
  assignableExpression assignmentOperator
      expressionWithoutCascade |
  conditionalExpression |
  throwExpressionWithoutCascade
 ;

expressionList:
  expression (‘,’ expression)*
 ;

primary:
  thisExpression |
  super unconditionalAssignableSelector |
  functionExpression |
  literal |
  identifier |
  newExpression |
  constObjectExpression |
  ‘(’ expression ‘)’
 ;

An expression e may always be enclosed in parentheses, but this never has any semantic effect on e.

Sadly, it may have an effect on the surrounding expression. Given a class C with static method $m => 42$, $C.m()$ returns 42, but $(C).m()$ produces a NoSuchMethodError. This anomaly can be corrected by removing the restrictions on calling the

members of instances of Type. This issue may be addressed in future versions of Dart.

16.0.1 Object Identity

The predefined Dart function identical() is defined such that identical(c_1, c_2) iff:

- c_1 evaluates to either the null object (16.2) or an instance of bool and c_1 == c_2, OR
- c_1 and c_2 are instances of int and c_1 == c_2, OR
- c_1 and c_2 are constant strings and c_1 == c_2, OR
- c_1 and c_2 are instances of double and one of the following holds:
  - c_1 and c_2 are non-zero and c_1 == c_2.
  - Both c_1 and c_2 are $+0.0$.
  - Both c_1 and c_2 are -0.0.
  - Both c_1 and c_2 represent a NaN value with the same underlying bit pattern.

  OR
- c_1 and c_2 are constant lists that are defined to be identical in the specification of literal list expressions (16.7), OR
- c_1 and c_2 are constant maps that are defined to be identical in the specification of literal map expressions (16.8), OR
- c_1 and c_2 are constant objects of the same class C and the value of each instance variable of c_1 is identical to the value of the corresponding instance variable of c_2. OR
- c_1 and c_2 are the same object.

The definition of identity for doubles differs from that of equality in that a NaN is identical to itself, and that negative and positive zero are distinct.

The definition of equality for doubles is dictated by the IEEE 754 standard, which posits that NaNs do not obey the law of reflexivity. Given that hardware implements these rules, it is necessary to support them for reasons of efficiency.

The definition of identity is not constrained in the same way. Instead, it assumes that bit-identical doubles are identical.

The rules for identity make it impossible for a Dart programmer to observe whether a boolean or numerical value is boxed or unboxed.

16.1 Constants

A *constant expression* is an expression whose value can never change, and that can be evaluated entirely at compile time.

A constant expression is one of the following:

- A literal number (16.3).
- A literal boolean (16.4).
- A literal string (16.5) where any interpolated expression (16.5.1) is a compile-time constant that evaluates to a numeric, string or boolean value or to the null object (16.2). *It would be tempting to allow string interpolation where the interpolated value is any compile-time constant. However, this would require running the* toString() *method for constant objects, which could contain arbitrary code.*
- A literal symbol (16.6).
- **null** (16.2).
- A qualified reference to a static constant variable (8) that is not qualified by a deferred prefix. For example, If class C declares a constant static variable v, C.v is a constant. The same is true if C is accessed via a prefix p; p.C.v is a constant unless p is a deferred prefix.
- An identifier expression that denotes a constant variable.
- A simple or qualified identifier denoting a class or type alias that is not qualified by a deferred prefix. For example, If C is a class or typedef, C is a constant, and if C is imported with a prefix p, p.C is a constant unless p is a deferred prefix.
- A constant constructor invocation (16.12.2) that is not qualified by a deferred prefix.
- A constant list literal (16.7).
- A constant map literal (16.8).
- A simple or qualified identifier denoting a top-level function (9) or a static method (10.7) that is not qualified by a deferred prefix.
- A parenthesized expression (e) where e is a constant expression.
- An expression of the form identical(e_1, e_2) where e_1 and e_2 are constant expressions and identical() is statically bound to the predefined dart function identical() discussed above (16.0.1).
- An expression of one of the forms $e_1 == e_2$ or $e_1 != e_2$ where e_1 and e_2 are constant expressions, and either both evaluate to a numeric, string or boolean value, or at least one of e_1 or e_2 evaluates to the null object (16.2).

- An expression of one of the forms !e, e_1 && e_2 or $e_1||e_2$, where e, e_1 and e_2 are constant expressions that evaluate to a boolean value.
- An expression of one of the forms ~e, e_1 ^ e_2, e_1 & e_2, $e_1|e_2$, $e_1 << e_2$, $e_1 >> e_2$ or $e_1 >>> e_2$, where e, e_1 and e_2 are constant expressions that evaluate to an integer value or to the null object (16.2).
- An expression of the form $e_1 + e_2$ where e_1 and e_2 are constant expressions that evaluate to a numeric or string value or to the null object (16.2).
- An expression of one of the forms $-e$, e_1 - e_2, e_1 * e_2, e_1 / e_2, e_1 ~/ e_2, $e_1 > e_2$, $e_1 < e_2$, $e_1 >= e_2$, $e_1 <= e_2$ or e_1 % e_2, where e, e_1 and e_2 are constant expressions that evaluate to a numeric value or to the null object (16.2).
- An expression of the form $e_1?e_2:e3$ where e_1, e_2 and e_3 are constant expressions and e_1 evaluates to a boolean value.
- An expression of the form $e_1??e_2$ where e_1 and e_2 are constant expressions.
- An expression of the form e.length where e is a constant expression that evaluates to a string value.

It is a compile-time error if an expression is required to be a constant expression but its evaluation would throw an exception. It is a compile-time error if an assertion is part of a compile-time constant constructor invocation and the assertion would throw an exception.

Note that there is no requirement that every constant expression evaluate correctly. Only when a constant expression is required (e.g., to initialize a constant variable, or as a default value of a formal parameter, or as metadata) do we insist that a constant expression actually be evaluated successfully at compile time.

The above is not dependent on program control-flow. The mere presence of a required compile-time constant whose evaluation would fail within a program is an error. This also holds recursively: since compound constants are composed out of constants, if any subpart of a constant would throw an exception when evaluated, that is an error.

On the other hand, since implementations are free to compile code late, some compile-time errors may manifest quite late.

```
const x = 1 / 0;
final y = 1 / 0;
class K {
 m1() {
  var z = false;
  if (z) {return x; }
  else { return 2; }
 }
 m2() {
  if (true) {return y; }
```

```
    else { return 3; }
  }
}
```

An implementation is free to immediately issue a compilation error for x, but it is not required to do so. It could defer errors if it does not immediately compile the declarations that reference x. For example, it could delay giving a compilation error about the method m1 until the first invocation of m1. However, it could not choose to execute m1, see that the branch that refers to x is not taken and return 2 successfully.

The situation with respect to an invocation m2 is different. Because y is not a compile-time constant (even though its value is), one need not give a compile-time error upon compiling m2. An implementation may run the code, which will cause the getter for y to be invoked. At that point, the initialization of y must take place, which requires the initializer to be compiled, which will cause a compilation error.

The treatment of ***null*** *merits some discussion. Consider* ***null*** *+ 2. This expression always causes an error. We could have chosen not to treat it as a constant expression (and in general, not to allow* ***null*** *as a subexpression of numeric or boolean constant expressions). There are two arguments for including it:*

1. *It is constant. We can evaluate it at compile time.*
2. *It seems more useful to give the error stemming from the evaluation explicitly.*

One might reasonably ask why $e_1?e_1 : e_3$ and $e_1??e_2$ have constant forms. For example, if e_1 is known statically, why do we need to test it?. The answer is that there are contexts where e_1 is a variable. In particular, constant constructor initializers such as

const *C(foo) :* ***this****.foo = foo ?? someDefaultValue;*

It is a compile-time error if the value of a compile-time constant expression depends on itself.

As an example, consider:

```
class CircularConsts{
// Illegal program - mutually recursive compile-time constants
  static const i = j; // a compile-time constant
  static const j = i; // a compile-time constant
}
```

literal:
  nullLiteral |
  booleanLiteral |
  numericLiteral |
  stringLiteral |
  symbolLiteral |
  mapLiteral |
  listLiteral

;

16.2 Null

The reserved word **null** evaluates to the *null object*.

nullLiteral:
  null
;

The null object is the sole instance of the built-in class Null. Attempting to instantiate Null causes a run-time error. It is a compile-time error for a class to extend, mix in or implement Null. The Null class extends the Object class and declares no methods except those also declared by Object.

The static type of **null** is the Null type.

16.3 Numbers

A *numeric literal* is either a decimal or hexadecimal numeral representing an integer value, or a decimal double representation.

numericLiteral:
  NUMBER |
  HEX_NUMBER
;

NUMBER:
  DIGIT+ (‘.’ DIGIT+)? EXPONENT? |
  ‘.’ DIGIT+ EXPONENT?
;

EXPONENT:
  (‘e’ | ‘E’) (‘+’ | ‘-’)? DIGIT+
;

HEX_NUMBER:
  ‘0x’ HEX_DIGIT+ |
  ‘0X’ HEX_DIGIT+
;

HEX_DIGIT:
  ‘a’..‘f’ |
  ‘A’..‘F’ |

DIGIT
;

A numeric literal starting with '0x' or '0X' is a *hexadecimal integer literal*. It has the numeric integer value of the hexadecimal numeral following '0x' (respectively '0X').

A numeric literal that contains only decimal digits is a *decimal integer literal*. It has the numeric integer value of the decimal numeral.

An *integer literal* is either a hexadecimal integer literal or a decimal integer literal. The static type of an integer literal is int.

A numeric literal that is not an integer literal is a *double literal*. A double literal always contains either a decimal point or an exponent part. The static type of a double literal is double.

A hexadecimal integer literal with numeric value i is a compile-time error if $i \geq 2^{64}$, unless it is prefixed by a unary minus operator, in which case it is a compile-time error if $i > 2^{63}$. If the int class is implemented as signed 64-bit two's complement integers, $i \geq 2^{63}$, and the literal is not prefixed by a unary minus operator, then the literal evaluates to an instance of the int class representing the integer value $i - 2^{64}$. Otherwise the literal evaluates to an instance of the int class representing the integer value i, and it is a compile-time error if the integer i cannot be represented exactly by an instance of int.

A decimal integer literal with numeric value i is a compile-time error if $i \geq 2^{63}$, unless i is prefixed by a unary minus operator, in which case it is only a compile-time error if $i > 2^{63}$. Otherwise the literal evaluates to an instance of the int class representing the integer value i. It is a compile-time error if the value i cannot be represented exactly by an instance of int.

A double literal evaluates to a an instance of the double class representing a 64 bit double precision floating point number as specified by the IEEE 754 standard.

Integers in Dart are designed to be implemented as 64-bit two's complement integer representations. In practice, implementations may be limited by other considerations. For example, Dart compiled to JavaScript may use the JavaScript number type, equivalent to Dart double, to represent integers, and if so, integer literals with more than 53 bits of precision cannot be represented exactly.

It is a compile-time error for a class to extend, mix in or implement int. It is a compile-time error for a class to extend, mix in or implement double. It is a compile-time error for any class other than int and double to extend, mix in or implement num.

16.4 Booleans

The reserved words **true** and **false** evaluate to objects *true* and *false* that represent the boolean values true and false respectively. They are the *boolean literals*.

```
booleanLiteral:
  true |
  false
;
```

Both *true* and *false* are instances of the built-in class bool, and there are no other objects that implement bool. It is a compile-time error for a class to extend, mix in or implement bool.

Invoking the getter runtimeType on a boolean value returns the Type object that is the value of the expression bool. The static type of a boolean literal is bool.

16.4.1 Boolean Conversion

Boolean conversion maps any object *o* into a boolean. Boolean conversion is defined by the function application

```
(bool v){
      assert(v != null);
      return identical(v, true);
}(o)
```

Boolean conversion is used as part of control-flow constructs and boolean expressions. Ideally, one would simply insist that control-flow decisions be based exclusively on booleans. This is straightforward in a statically typed setting. In a dynamically typed language, it requires a dynamic check. Sophisticated virtual machines can minimize the penalty involved. Alas, Dart must be compiled into Javascript. Boolean conversion allows this to be done efficiently.

At the same time, this formulation differs radically from Javascript, where most numbers and objects are interpreted as ***true****. Dart's approach prevents usages such* ***if*** *(a-b) ... ; because it does not agree with the low level conventions whereby non-null objects or non-zero numbers are treated as* ***true****. Indeed, there is no way to derive* ***true*** *from a non-boolean object via boolean conversion, so this kind of low level hackery is nipped in the bud.*

Dart also avoids the strange behaviors that can arise due to the interaction of boolean conversion with autoboxing in Javascript. A notorious example is the situation where ***false*** *can be interpreted as* ***true****. In Javascript, booleans are not objects, and instead are autoboxed into objects where "needed". If* ***false*** *gets autoboxed into an object, that object can be coerced into* ***true*** *(as it is a non-null object).*

Because boolean conversion requires its parameter to be a boolean, any construct that makes use of boolean conversion will cause a dynamic type error in checked mode if the value to be converted is not a boolean.

16.5 Strings

A *string* is a sequence of UTF-16 code units.

This decision was made for compatibility with web browsers and Javascript. Earlier versions of the specification required a string to be a sequence of valid Unicode code points. Programmers should not depend on this distinction.

stringLiteral:
(multilineString | singleLineString)+
;

A string can be a sequence of single line strings and multiline strings.

singleLineString:
‘"’ stringContentDQ* ‘"’ |
‘’’ stringContentSQ* ‘’’ |
‘r’’ (~(‘’’ | NEWLINE))* ‘’’ |
‘r"’ (~(‘"’ | NEWLINE))* ‘"’
;

A single line string is delimited by either matching single quotes or matching double quotes.

Hence, ‘abc’ and “abc” are both legal strings, as are ‘He said “To be or not to be” did he not?’ and “He said ‘To be or not to be’ didn’t he”. However “This ‘ is not a valid string, nor is ‘this”.

The grammar ensures that a single line string cannot span more than one line of source code, unless it includes an interpolated expression that spans multiple lines.

Adjacent strings are implicitly concatenated to form a single string literal.

Here is an example

```
print("A string" "and then another"); // prints: A stringand then another
```

Dart also supports the operator + for string concatenation.

The + operator on Strings requires a String argument. It does not coerce its argument into a string. This helps avoid puzzlers such as

```
print("A simple sum: 2 + 2 = " +
          2 + 2);
```

which this prints ’A simple sum: 2 + 2 = 22’ rather than ’A simple sum: 2 + 2 = 4’. However, the use of the concatenation operation is still discouraged for efficiency reasons. Instead, the recommended Dart idiom is to use string interpolation.

```
print("A simple sum: 2 + 2 = ${2+2}");
```

String interpolation works well for most cases. The main situation where it is not fully satisfactory is for string literals that are too large to fit on a line. Multiline strings can be useful, but in some cases, we want to visually align the code. This can be expressed by writing smaller strings separated by whitespace, as shown here:

```
'Imagine this is a very long string that does not fit on a line. What shall we do? '
'Oh what shall we do? '
'We shall split it into pieces '
'like so'.
```

multilineString:
‘"""’ stringContentTDQ* ‘"""’ |
‘’’’ stringContentTSQ* ‘’’’ |
‘r"""’ (~ ‘"""’)* ‘"""’ |
‘r’’’’ (~ ‘’’’)* ‘’’’
;

ESCAPE_SEQUENCE:
‘\n’ |
‘\r’ |
‘\f’ |
‘\b’ |
‘\t’ |
‘\v’ |
‘\x’ HEX_DIGIT HEX_DIGIT |
‘\u’ HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT |
‘\u{’ HEX_DIGIT_SEQUENCE ‘}’
;

HEX_DIGIT_SEQUENCE:
HEX_DIGIT HEX_DIGIT? HEX_DIGIT?
HEX_DIGIT? HEX_DIGIT? HEX_DIGIT?
;

Multiline strings are delimited by either matching triples of single quotes or matching triples of double quotes. If the first line of a multiline string consists solely of the whitespace characters defined by the production *WHITESPACE* (20.1), possibly prefixed by \, then that line is ignored, including the line break at its end.

The idea is to ignore a whitespace-only first line of a multiline string, where whitespace is defined as tabs, spaces and the final line break. These can be represented directly, but since for most characters prefixing by backslash is an identity in a non-raw string, we allow those forms as well.

Strings support escape sequences for special characters. The escapes are:

- \n for newline, equivalent to \x0A.
- \r for carriage return, equivalent to \x0D.
- \f for form feed, equivalent to \x0C.
- \b for backspace, equivalent to \x08.
- \t for tab, equivalent to \x09.
- \v for vertical tab, equivalent to \x0B.

- \x HEX_DIGIT_1 HEX_DIGIT_2, equivalent to \u{HEX_DIGIT_1 HEX_DIGIT_2}.
- \u HEX_DIGIT_1 HEX_DIGIT_2 HEX_DIGIT_3 HEX_DIGIT_4, equivalent to \u{HEX_DIGIT_1 HEX_DIGIT_2 HEX_DIGIT_3 HEX_DIGIT_4}.
- \u{$HEX\_DIGIT\_SEQUENCE$} is the Unicode code point represented by the $HEX\_DIGIT\_SEQUENCE$. It is a compile-time error if the value of the $HEX\_DIGIT\_SEQUENCE$ is not a valid Unicode code point.
- $ indicating the beginning of an interpolated expression.
- Otherwise, \$k$ indicates the character k for any k not in $\{n, r, f, b, t, v, x, u\}$.

Any string may be prefixed with the character 'r', indicating that it is a *raw string*, in which case no escapes or interpolations are recognized.

Line breaks in a multiline string are represented by the *NEWLINE* production. A line break introduces a single newline character into the string value.

It is a compile-time error if a non-raw string literal contains a character sequence of the form \x that is not followed by a sequence of two hexadecimal digits. It is a compile-time error if a non-raw string literal contains a character sequence of the form \u that is not followed by either a sequence of four hexadecimal digits, or by curly brace delimited sequence of hexadecimal digits.

stringContentDQ:
  ~('\' | '"' | '$' | NEWLINE) |
  '\' ~(NEWLINE) |
  stringInterpolation
  ;

stringContentSQ:
  ~('\' | ''' | '$' | NEWLINE) |
  '\' ~(NEWLINE) |
  stringInterpolation
  ;

stringContentTDQ:
  ~('\' | '"""' | '$') |
  stringInterpolation
  ;

stringContentTSQ:
  ~('\' | ''''' | '$') |
  stringInterpolation
  ;

NEWLINE:
\n |
\r |
\r\n
;

All string literals evaluate to instances of the built-in class String. It is a compile-time error for a class to extend, mix in or implement String. The static type of a string literal is String.

16.5.1 String Interpolation

It is possible to embed expressions within non-raw string literals, such that these expressions are evaluated, and the resulting values are converted into strings and concatenated with the enclosing string. This process is known as *string interpolation*.

stringInterpolation:
'$' IDENTIFIER_NO_DOLLAR |
'${' expression '}'
;

The reader will note that the expression inside the interpolation could itself include strings, which could again be interpolated recursively.

An unescaped $ character in a string signifies the beginning of an interpolated expression. The $ sign may be followed by either:

- A single identifier *id* that must not contain the $ character.
- An expression e delimited by curly braces.

The form $id is equivalent to the form ${id}. An interpolated string, s, with content '$s_0\${e_1}s_1 \ldots s_{n-1}\${e_n}s_n$' (where any of $s_0, \ldots, s_n$ can be empty) is evaluated by evaluating each expression e_i ($1 \leq i \leq n$) in to a string r_i in the order they occur in the source text, as follows:

- Evaluate e_i to an object o_i.
- Invoke the toString method on o_i with no arguments, and let r_i be the returned value.
- If r_i is not an instance of the built-in type String, throw an Error.

Finally, the result of the evaluation of s is the concatenation of the strings s_0, r_1, ..., r_n, and s_n.

16.6 Symbols

A *symbol literal* denotes the name of a declaration in a Dart program.

symbolLiteral:
'#' (operator | (identifier ('.' identifier)*))
;

A symbol literal *#id* where *id* does not begin with an underscore ('_') is equivalent to the expression **const** Symbol('*id*').

A symbol literal *#_id* evaluates to the object that would be returned by the call MirrorSystem.getSymbol("*_id*", *libraryMirror*) where *libraryMirror* is an instance of the class LibraryMirror defined in the library dart:mirrors, reflecting the current library.

One may well ask what is the motivation for introducing literal symbols? In some languages, symbols are canonicalized whereas strings are not. However literal strings are already canonicalized in Dart. Symbols are slightly easier to type compared to strings and their use can become strangely addictive, but this is not nearly sufficient justification for adding a literal form to the language. The primary motivation is related to the use of reflection and a web specific practice known as minification.

Minification compresses identifiers consistently throughout a program in order to reduce download size. This practice poses difficulties for reflective programs that refer to program declarations via strings. A string will refer to an identifier in the source, but the identifier will no longer be used in the minified code, and reflective code using these would fail. Therefore, Dart reflection uses objects of type Symbol rather than strings. Instances of Symbol are guaranteed to be stable with repeat to minification. Providing a literal form for symbols makes reflective code easier to read and write. The fact that symbols are easy to type and can often act as convenient substitutes for enums are secondary benefits.

The static type of a symbol literal is Symbol.

16.7 Lists

A *list literal* denotes a list, which is an integer indexed collection of objects.

listLiteral:
const? typeArguments? '[' (expressionList ','?)? ']'
;

A list may contain zero or more objects. The number of elements in a list is its size. A list has an associated set of indices. An empty list has an empty set of indices. A non-empty list has the index set $\{0, \ldots, n-1\}$ where n is the size of the list. It is a run-time error to attempt to access a list using an index that is not a member of its set of indices.

If a list literal begins with the reserved word **const**, it is a *constant list literal* which is a compile-time constant (16.1) and therefore evaluated at compile time. Otherwise, it is a *run-time list literal* and it is evaluated at run time. Only run-time list literals can be mutated after they are created. Attempting to mutate a constant list literal will result in a dynamic error.

It is a compile-time error if an element of a constant list literal is not a compile-time constant. It is a compile-time error if the type argument of a constant list literal includes a type parameter. *The binding of a type parameter is not known at compile time, so we cannot use type parameters inside compile-time constants.*

The value of a constant list literal **const** $< E > [e_1, \ldots, e_n]$ is an object a whose class implements the built-in class $List < E >$. The ith element of a is v_{i+1}, where v_i is the value of the compile-time expression e_i. The value of a constant list literal **const** $[e_1, \ldots, e_n]$ is defined as the value of the constant list literal **const**$<$ **dynamic** $> [e_1, \ldots, e_n]$.

Let $list_1 =$ **const** $< V > [e_{11}, \ldots, e_{1n}]$ and $list_2 =$ **const** $< U > [e_{21}, \ldots, e_{2n}]$ be two constant list literals and let the elements of $list_1$ and $list_2$ evaluate to $o_{11}, \ldots, o_{1n}$ and $o_{21}, \ldots, o_{2n}$ respectively. Iff identical(o_{1i}, o_{2i}) for $i \in 1..n$ and $V = U$ then identical($list_1$, $list_2$).

In other words, constant list literals are canonicalized.

A run-time list literal $< E > [e_1, \ldots, e_n]$ is evaluated as follows:

- First, the expressions $e_1, \ldots, e_n$ are evaluated in order they appear in the program, producing objects $o_1, \ldots, o_n$.
- A fresh instance (10.6.1) a, of size n, whose class implements the built-in class $List < E >$ is allocated.
- The operator []= is invoked on a with first argument i and second argument $o_{i+1}, 0 \leq i < n$.
- The result of the evaluation is a.

Note that this document does not specify an order in which the elements are set. This allows for parallel assignments into the list if an implementation so desires. The order can only be observed in checked mode (and may not be relied upon): if element i is not a subtype of the element type of the list, a dynamic type error will occur when $a[i]$ is assigned o_{i-1}.

A run-time list literal $[e_1, \ldots, e_n]$ is evaluated as $<$ **dynamic** $> [e_1, \ldots, e_n]$.

There is no restriction precluding nesting of list literals. It follows from the rules above that $< List < int >> [[1, 2, 3], [4, 5, 6]]$ is a list with type parameter $List < int >$, containing two lists with type parameter **dynamic**.

The static type of a list literal of the form **const**$< E > [e_1, \ldots, e_n]$ or the form $< E > [e_1, \ldots, e_n]$ is $List < E >$. The static type a list literal of the form **const** $[e_1, \ldots, e_n]$ or the form $[e_1, \ldots, e_n]$ is $List <$ **dynamic** $>$.

It is tempting to assume that the type of the list literal would be computed based on the types of its elements. However, for mutable lists this may be unwarranted. Even for constant lists, we found this behavior to be problematic. Since

compile time is often actually run time, the run-time system must be able to perform a complex least upper bound computation to determine a reasonably precise type. It is better to leave this task to a tool in the IDE. It is also much more uniform (and therefore predictable and understandable) to insist that whenever types are unspecified they are assumed to be the unknown type ***dynamic****.*

16.8 Maps

A *map literal* denotes a map object.

mapLiteral:
  const? typeArguments?
      '{' (mapLiteralEntry (',' mapLiteralEntry)* ','?)? '}'
;

mapLiteralEntry:
  expression ':' expression
;

A *map literal* consists of zero or more entries. Each entry has a *key* and a *value*. Each key and each value is denoted by an expression.

If a map literal begins with the reserved word **const**, it is a *constant map literal* which is a compile-time constant (16.1) and therefore evaluated at compile time. Otherwise, it is a *run-time map literal* and it is evaluated at run time. Only run-time map literals can be mutated after they are created. Attempting to mutate a constant map literal will result in a dynamic error.

It is a compile-time error if either a key or a value of an entry in a constant map literal is not a compile-time constant. It is a compile-time error if the key of an entry in a constant map literal is an instance of a class that implements the operator == unless the key is a string, an integer, a literal symbol or the result of invoking a constant constructor of class Symbol. It is a compile-time error if the type arguments of a constant map literal include a type parameter.

The value of a constant map literal $\textbf{const} < K, V > \{k_1 : e_1, \ldots, k_n : e_n\}$ is an object m whose class implements the built-in class $Map < K, V >$. The entries of m are $u_i : v_i, i \in 1..n$, where u_i is the value of the compile-time expression k_i and v_i is the value of the compile-time expression e_i. The value of a constant map literal $\textbf{const}\ \{k_1 : e_1, \ldots, k_n : e_n\}$ is defined as the value of a constant map literal $\textbf{const} < \textbf{dynamic}, \textbf{dynamic} > \{k_1 : e_1, \ldots, k_n : e_n\}$.

Let $map_1 = \textbf{const} < K, V > \{k_{11} : e_{11}, \ldots, k_{1n} : e_{1n}\}$ and $map_2 = \textbf{const} < J, U > \{k_{21} : e_{21}, \ldots, k_{2n} : e_{2n}\}$ be two constant map literals. Let the keys of map_1 and map_2 evaluate to $s_{11}, \ldots, s_{1n}$ and $s_{21}, \ldots, s_{2n}$ respectively, and let the elements of map_1 and map_2 evaluate to $o_{11}, \ldots, o_{1n}$ and $o_{21}, \ldots, o_{2n}$ respectively. Iff identical(o_{1i}, o_{2i}) and identical(s_{1i}, s_{2i}) for $i \in 1..n$, and $K = J, V = U$ then identical(map_1, map_2).

In other words, constant map literals are canonicalized.

A run-time map literal $< K, V > \{k_1 : e_1, \ldots, k_n : e_n\}$ is evaluated as follows:

- For each $i \in 1..n$ in numeric order, first the expression k_i is evaluated producing object u_i, and then e_i is evaluated producing object o_i. This produces all the objects $u_1, o_1, \ldots, u_n, o_n$.
- A fresh instance (10.6.1) m whose class implements the built-in class $Map < K, V >$ is allocated.
- The operator []= is invoked on m with first argument u_i and second argument o_i for each $i \in 1..n$.
- The result of the evaluation is m.

A run-time map literal $\{k_1 : e_1, \ldots, k_n : e_n\}$ is evaluated as $< \textbf{dynamic}, \textbf{dynamic} > \{k_1 : e_1, \ldots, k_n : e_n\}$.

Iff all the keys in a map literal are compile-time constants, it is a static warning if the values of any two keys in a map literal are equal.

A map literal is ordered: iterating over the keys and/or values of the maps always happens in the order the keys appeared in the source code.

Of course, if a key repeats, the order is defined by first occurrence, but the value is defined by the last.

The static type of a map literal of the form **const**$< K, V > \{k_1 : e_1, \ldots, k_n : e_n\}$ or the form $< K, V > \{k_1 : e_1, \ldots, k_n : e_n\}$ is $Map < K, V >$. The static type of a map literal of the form **const**$\{k_1 : e_1, \ldots, k_n : e_n\}$ or the form $\{k_1 : e_1, \ldots, k_n : e_n\}$ is $Map <$ **dynamic**, **dynamic** $>$.

16.9 Throw

The *throw expression* is used to throw an exception.

throwExpression:
  throw expression
;

throwExpressionWithoutCascade:
  throw expressionWithoutCascade
;

Evaluation of a throw expression of the form **throw** e; proceeds as follows:

The expression e is evaluated to a value v (16).

There is no requirement that the expression e must evaluate to any special kind of object.

If v is the null object (16.2), then a NullThrownError is thrown. Otherwise let t be a stack trace corresponding to the current execution state, and the **throw** statement throws with v as exception object and t as stack trace (16).

If v is an instance of class Error or a subclass thereof, and it is the first time that Error object is thrown, the stack trace t is stored on v so that it will be returned by the v's stackTrace getter

If the same Error object is thrown more than once, its stackTrace getter will return the stack trace from the *first* time it was thrown.

The static type of a throw expression is $\bot$.

16.10 Function Expressions

A *function literal* is an object that encapsulates an executable unit of code.

functionExpression:
formalParameterPart functionBody
;

The class of a function literal implements the built-in class **Function**.

The static type of a function literal of the form
$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n, [T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k]) => e$
is
$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n, [T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}]) \rightarrow T_0$,
where T_0 is the static type of e.

The static type of a function literal of the form
$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n, [T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k])$ **async** $=> e$
is
$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1, \ldots,\ T_n, [T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}]) \rightarrow$ Future$<flatten(T_0)>$,
where T_0 is the static type of e.

In the previous two paragraphs, the type argument lists are omitted in the case where $m = 0$, and $flatten(T)$ is defined as follows:

- If $T =$ FutureOr$<S>$ then $flatten(T) = S$.
- Otherwise if $T <:$ Future then let S be a type such that $T <<$ Future$<S>$ and for all R, if $T <<$ Future$<R>$ then $S << R$.

  This ensures that Future$<S>$ *is the most specific generic instantiation of* Future *that is a supertype of* T. *Note that* S *is well-defined because of the requirements on superinterfaces.*

  Then $flatten(T) = S$.
- In any other circumstance, $flatten(T) = T$.

The static type of a function literal of the form
$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n, \{T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k\}) => e$

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n,\ \{T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}\}) \rightarrow T_0$,
where T_0 is the static type of e.

The static type of a function literal of the form

$<X_1 B_1, \ldots,\ X_m B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n,\ \{T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k\})$ **async** $=>$ e

is

$<X_1 B_1, \ldots,\ X_m B_m>$
$(T_1, \ldots,\ T_n,\ \{T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}\}) \rightarrow$ Future$<flatten(T_0)>$,
where T_0 is the static type of e.

The static type of a function literal of the form

$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n,\ [T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k])\ \{\ s\ \}$

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n,\ [T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}]) \rightarrow$ **dynamic**.

The static type of a function literal of the form

$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n,\ [T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k])$ **async** $\{\ s\ \}$

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n,\ [T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}]) \rightarrow$ Future.

The static type of a function literal of the form

$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n,\ [T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k])$ **async*** $\{\ s\ \}$

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n,\ [T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}]) \rightarrow$ Stream.

The static type of a function literal of the form

$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n,\ [T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k])$ **sync*** $\{\ s\ \}$

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n,\ [T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}]) \rightarrow$ Iterable.

The static type of a function literal of the form

$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n,\ [T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k])\ \{\ s\ \}$

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n,\ [T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}]) \rightarrow$ **dynamic**.

The static type of a function literal of the form

$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n,\ \{T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k\})$ **async** $\{\ s\ \}$

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n,\ \{T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}\}) \rightarrow$ Future.

The static type of a function literal of the form

$<X_1\ B_1, \ldots,\ X_m\ B_m>$
$(T_1\ a_1, \ldots,\ T_n\ a_n,\ \{T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k\})$ **async*** $\{\ s\ \}$

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n, \{T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}\}) \rightarrow$ Stream.

The static type of a function literal of the form

$<X_1\ B_1, \ldots,\ X_m\ B_m>$

$(T_1\ a_1, \ldots,\ T_n\ a_n, \{T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k\})$ **sync*** { s }

is

$<X_1\ B_1, \ldots,\ X_m\ B_m>(T_1, \ldots,\ T_n, \{T_{n+1}\ x_{n+1}, \ldots,\ T_{n+k}\ x_{n+k}\}) \rightarrow$ Iterable.

In all of the above cases, the type argument lists are omitted when $m = 0$, and whenever $T_i, 1 \leq i \leq n + k$, is not specified, it is considered to have been specified as **dynamic**.

16.11 This

The reserved word **this** denotes the target of the current instance member invocation.

thisExpression:
  this
;

The static type of **this** is the interface of the immediately enclosing class.

We do not support self-types at this point.

It is a compile-time error if **this** appears, implicitly or explicitly, in a top-level function or variable initializer, in a factory constructor, or in a static method or variable initializer, or in the initializer of an instance variable.

16.12 Instance Creation

Instance creation expressions invoke constructors to produce instances.

It is a static type warning if the type T in an instance creation expression of one of the forms

new $T.id(a_1, \ldots,\ a_n,\ x_{n+1}: a_{n+1}, \ldots,\ x_{n+k}: a_{n+k})$,

new $T(a_1, \ldots,\ a_n,\ x_{n+1}: a_{n+1}, \ldots,\ x_{n+k}: a_{n+k})$,

const $T.id(a_1, \ldots,\ a_n,\ x_{n+1}: a_{n+1}, \ldots,\ x_{n+k}: a_{n+k})$,

const $T(a_1, \ldots,\ a_n,\ x_{n+1}: a_{n+1}, \ldots,\ x_{n+k}: a_{n+k})$ is malformed (19.2) or malbounded (19.8).

It is a compile-time error if the type T in an instance creation expression of one of the forms

new $T.id(a_1, \ldots,\ a_n,\ x_{n+1}: a_{n+1}, \ldots,\ x_{n+k}: a_{n+k})$,

new $T(a_1, \ldots,\ a_n,\ x_{n+1}: a_{n+1}, \ldots,\ x_{n+k}: a_{n+k})$,

const $T.id(a_1, \ldots,\ a_n,\ x_{n+1}: a_{n+1}, \ldots,\ x_{n+k}: a_{n+k})$,

const $T(a_1, \ldots,\ a_n,\ x_{n+1}: a_{n+1}, \ldots,\ x_{n+k}: a_{n+k})$

is an enumerated type (13).

16.12.1 New

The *new expression* invokes a constructor (10.6).

newExpression:
new type ('.' identifier)? arguments
;

Let e be a new expression of the form
new $T.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ or the form
new $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$.
If T is a class or parameterized type accessible in the current scope then:

- If e is of the form **new** $T.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ it is a static warning if $T.id$ is not the name of a constructor declared by the type T.
- If e is of the form **new** $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ it is a static warning if the type T does not declare a constructor with the same name as the declaration of T.

If T is a parameterized type (19.8) $S<U_1, \ldots, U_m>$, let $R = S$. If T is not a parameterized type, let $R = T$. Furthermore, if e is of the form **new** $T.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ then let q be the constructor $T.id$, otherwise let q be the constructor T.

If R is a generic with $l = m$ type parameters then

- If T is not a parameterized type, then for $i \in 1..l$, let $V_i =$ **dynamic**.
- If T is a parameterized type then let $V_i = U_i$ for $i \in 1..m$.

If R is a generic with $l \neq m$ type parameters then for $i \in 1..l$, let $V_i =$ **dynamic**. In any other case, let $V_i = U_i$ for $i \in 1..m$.

Evaluation of e proceeds as follows:

First, the argument list $(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ is evaluated.

If T is a deferred type with prefix p, then if p has not been successfully loaded, a dynamic error occurs.

Then, if q is a non-factory constructor of an abstract class then an AbstractClassInstantiationError is thrown.

If T is malformed or if T is a type variable a dynamic error occurs. In checked mode, if T or any of its superclasses is malbounded a dynamic error occurs. Otherwise, if q is not defined or not accessible, a NoSuchMethodError is thrown. If q has fewer than n positional parameters or more than n required parameters, or if q lacks any of the named parameters $\{x_{n+1}, \ldots, x_{n+k}\}$ a NoSuchMethodError is thrown.

Otherwise, if q is a generative constructor (10.6.1), then:

Note that at this point we are assured that the number of actual type arguments match the number of formal type parameters.

A fresh instance (10.6.1), i, of class R is allocated. Then q is executed to initialize i with respect to the bindings that resulted from the evaluation of the argument list, and, if R is a generic class, with its type parameters bound to $V_1, \ldots, V_m$.

If execution of q completes normally (17), e evaluates to i. Otherwise execution of q throws an exception object x and stack trace t, and then evaluation of e also throws exception object x and stack trace t (16).

Otherwise, q is a factory constructor (10.6.2). Then:

If q is a redirecting factory constructor of the form $T(p_1, \ldots, p_{n+k}) = c$; or of the form $T.id(p_1, \ldots, p_{n+k}) = c$; then the result of the evaluation of e is equivalent to evaluating the expression

$[V_1/X_1, \ldots, V_m/X_m]$(**new** $c(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$)

where $X_1, \ldots, X_m$ are the formal type parameters of R. If evaluation of q causes q to be re-evaluated cyclically, with only factory constructor redirections in-between, a run-time error occurs.

Otherwise, the body of q is executed with respect to the bindings that resulted from the evaluation of the argument list, and with the type parameters (if any) of q bound to the actual type arguments $V_1, \ldots, V_l$. If this execution returns a value (17), then e evaluates to the returned value. Otherwise, if the execution completes normally or returns with no value, then e evaluates to the null object (16.2). Otherwise the execution throws an exception x and stack trace t, and then evaluation of e also throws x and t (16).

It is a static warning if q is a constructor of an abstract class and q is not a factory constructor.

The above gives precise meaning to the idea that instantiating an abstract class leads to a warning. A similar clause applies to constant object creation in the next section.

In particular, a factory constructor can be declared in an abstract class and used safely, as it will either produce a valid instance or lead to a warning inside its own declaration.

The static type of an instance creation expression of either the form

new $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$

or the form

new $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$

is T. It is a static warning if the static type of $a_i, 1 \leq i \leq n+k$ may not be assigned to the type of the corresponding formal parameter of the constructor $T.id$ (respectively T).

16.12.2 Const

A *constant object expression* invokes a constant constructor (10.6.3).

constObjectExpression:
const type ('.' identifier)? arguments

;

Let e be a constant object expression of the form
const $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$
or the form **const** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$. It is a compile-time error if T does not denote a class accessible in the current scope. It is a compile-time error if T is a deferred type (19.1).

In particular, T may not be a type variable.

If T is a parameterized type, it is a compile-time error if T includes a type variable among its type arguments.

If e is of the form **const** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ it is a compile-time error if $T.id$ is not the name of a constant constructor declared by the type T. If e is of the form **const** $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ it is a compile-time error if the type T does not declare a constant constructor with the same name as the declaration of T.

In all of the above cases, it is a compile-time error if $a_i, i \in 1..n+k$, is not a compile-time constant expression.

Evaluation of e proceeds as follows:
First, if e is of the form
const $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$
then let i be the value of the expression
new $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.
Otherwise, e must be of the form
const $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$,
in which case let i be the result of evaluating
new $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$.
Then:

- If during execution of the program, a constant object expression has already evaluated to an instance j of class R with type arguments $V_i, 1 \leq i \leq m$, then:
  - For each instance variable f of i, let v_{if} be the value of the instance variable f in i, and let v_{jf} be the value of the instance variable f in j. If identical(v_{if}, v_{jf}) for all instance variables f in i then the value of e is j, otherwise the value of e is i.
- Otherwise the value of e is i.

In other words, constant objects are canonicalized. In order to determine if an object is actually new, one has to compute it; then it can be compared to any cached instances. If an equivalent object exists in the cache, we throw away the newly created object and use the cached one. Objects are equivalent if they have identical type arguments and identical instance variables. Since the constructor cannot induce any side effects, the execution of the constructor is unobservable. The constructor need only be executed once per call site, at compile time.

The static type of a constant object expression of either the form
const $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$
or the form
const $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$
is T. It is a static warning if the static type of $a_i, 1 \leq i \leq n+k$ may not be assigned to the type of the corresponding formal parameter of the constructor $T.id$ (respectively T).

It is a compile-time error if evaluation of a constant object results in an uncaught exception being thrown.

To see how such situations might arise, consider the following examples:

```
class A {
  final x;
  const A(p): x = p * 10;
}
const A("x"); // compile-time error
const A(5); // legal
class IntPair {
  const IntPair(this.x, this.y);
  final int x;
  final int y;
  operator *(v) => new IntPair(x*v, y*v);
}
const A(const IntPair(1,2)); // compile-time error: illegal in a subtler way
```

Due to the rules governing constant constructors, evaluating the constructor A() with the argument "x" or the argument **const** IntPair(1, 2) would cause it to throw an exception, resulting in a compile-time error.

Given an instance creation expression of the form **const** $q(a_1, \ldots, a_n)$ it is a static warning if q is a constructor of an abstract class (10.4) but q is not a factory constructor.

16.13 Spawning an Isolate

Spawning an isolate is accomplished via what is syntactically an ordinary library call, invoking one of the functions spawnUri() or spawn() defined in the dart:isolate library. However, such calls have the semantic effect of creating a new isolate with its own memory and thread of control.

An isolate's memory is finite, as is the space available to its thread's call stack. It is possible for a running isolate to exhaust its memory or stack, resulting in a run-time error that cannot be effectively caught, which will force the isolate to be suspended.

As discussed in section 7, the handling of a suspended isolate is the responsibility of the embedder.

16.14 Function Invocation

Function invocation occurs in the following cases: when a function expres-

sion (16.10) is invoked (16.14.4), when a method (16.18), getter (16.17, 16.19) or setter (16.20) is invoked, or when a constructor is invoked (either via instance creation (16.12), constructor redirection (10.6.1) or super initialization). The various kinds of function invocation differ as to how the function to be invoked, f, is determined, as well as whether **this** (16.11) is bound. Once f has been determined, formal type parameters of f are bound to the corresponding actual type arguments, and the formal parameters of f are bound to corresponding actual arguments. When the body of f is executed it will be executed with the aforementioned bindings.

Executing a body of the form $=> e$ is equivalent to executing a body of the form { return e; }. Execution a body of the form **async** $=> e$ is equivalent to executing a body of the form **async** { return e; }.

If f is synchronous and is not a generator (9) then execution of the body of f begins immediately. If the execution of the body of f returns a value, v, (17), the invocation evaluates to v. If the execution completes normally or it returns without a value, the invocation evaluates to the null object (16.2). If the execution throws an exception object and stack trace, the invocation throws the same exception object and stack trace (16).

A complete function body can never break or contine (17) because a **break** or **continue** statement must always occur inside the statement that is the target of the **break** or **continue**. This means that a function body can only either complete normally, throw, or return. Completing normally or returning without a value is treated the same as returning the null object (16.2), so the result of executing a function body can always be used as the result of evaluating an expression, either by evaluating to a value or by the evaluation throwing.

If f is marked **sync*** (9), then a fresh instance (10.6.1) i implementing Iterable<U> is immediately returned, where U is determined as follows: Let T be the actual return type of f (19.8.1). If T is Iterable<S> for some type S, then U is S, otherwise U is Object.

A Dart implementation will need to provide a specific implementation of Iterable that will be returned by **sync*** methods. A typical strategy would be to produce an instance of a subclass of class IterableBase defined in dart:core. The only method that needs to be added by the Dart implementation in that case is iterator.

The iterable implementation must comply with the contract of Iterable and should not take any steps identified as exceptionally efficient in that contract.

The contract explicitly mentions a number of situations where certain iterables could be more efficient than normal. For example, by precomputing their length. Normal iterables must iterate over their elements to determine their length. This is certainly true in the case of a synchronous generator, where each element is computed by a function. It would not be acceptable to pre-compute the results of the generator and cache them, for example.

When iteration over the iterable is started, by getting an iterator j from the iterable and calling moveNext(), execution of the body of f will begin. When execution of the body of f completes (17),

- If it returns without a value or it completes normally (17), j is positioned

after its last element, so that its current value is the null object (16.2) and the current call to moveNext() on j returns false, as must all further calls.

- If it throws an exception object e and stack trace t then the current value of j is the null object (16.2) and the current call to moveNext() throws e and t as well. Further calls to moveNext() must return false.

Each iterator starts a separate computation. If the **sync*** function is impure, the sequence of values yielded by each iterator may differ.

One can derive more than one iterator from a given iterable. Note that operations on the iterable itself can create distinct iterators. An example would be length. It is conceivable that different iterators might yield sequences of different length. The same care needs to be taken when writing **sync*** functions as when writing an Iterator class. In particular, it should handle multiple simultaneous iterators gracefully. If the iterator depends on external state that might change, it should check that the state is still valid after every yield (and maybe throw a ConcurrentModificationError if it isn't).

Each iterator runs with its own shallow copies of all local variables; in particular, each iterator has the same initial arguments, even if their bindings are modified by the function. Two executions of an iterator interact only via state outside the function.

If f is marked **async** (9), then a fresh instance (10.6.1) o is associated with the invocation, where the dynamic type of o implements Future<$flatten(T)$>, and T is the actual return type of f (19.8.1). Then the body of f is executed until it either suspends or completes, at which point o is returned. The body of f may suspend during the evaluation of an **await** expression or execution of an asynchronous **for** loop. The future o is completed when execution of the body of f completes (17). If execution of the body returns a value, o is completed with that value, if it completes normally or returns without a value, o is completed with the null object (16.2), and if it throws an exception e and stack trace t, o is completed with the error e and stack trace t. If execution of the body throws before the body suspends the first time, completion of o happens at some future time after the invocation has returned. *The caller needs time to set up error handling for the returned future, so the future is not completed with an error* before *it has been returned.*

If f is marked **async*** (9), then a fresh instance (10.6.1) s implementing Stream<U> is immediately returned, where U is determined as follows: Let T be the actual return type of f (19.8.1). If T is Stream<S> for some type S, then U is S, otherwise U is Object. When s is listened to, execution of the body of f will begin. When execution of the body of f completes:

- If it completes normally or returns with no value (17), then if s has been canceled then its cancellation future is completed with the null object (16.2).

- If it throws an exception object e and stack trace t:

  - If s has been canceled then its cancellation future is completed with error e and stack trace t.
  - otherwise the error e and stack trace t are emitted by s.

- s is closed.

The body of an asynchronous generator function cannot break, continue or return a value (17). The first two are only allowed in contexts that will handle the break or continue, and return statements with an expression are not allowed in generator functions.

When an asynchronous generator's stream has been canceled, cleanup will occur in the ***finally*** *clauses (17.11) inside the generator. We choose to direct any exceptions that occur at this time to the cancellation future rather than have them be lost.*

16.14.1 Actual Argument List Evaluation

Function invocation involves evaluation of the list of actual arguments to the function and binding of the results to the function's formal parameters.

arguments:
  '(' (argumentList ','?)? ')'
;

argumentList:
  namedArgument (',' namedArgument)* |
  expressionList (',' namedArgument)*
;

namedArgument:
  label expression
;

Argument lists allow an optional trailing comma after the last argument (','?). An argument list with such a trailing comma is equivalent in all ways to the same argument list without the trailing comma. All argument lists in this specification are shown without a trailing comma, but the rules and semantics apply equally to the corresponding argument list with a trailing comma.

When parsing an argument list, an ambiguity may arise because the same source code could be one generic function invocation, and it could be two or more relational expressions and/or shift expressions. In this situation, the expression is always parsed as a generic function invocation.

An example is f(a<B, C>(d)), which may be an invocation of f passing two actual arguments of type bool, or an invocation of f passing the result returned by an invocation of the generic function a. Note that the ambiguity can be eliminated

by omitting the parentheses around the expression d, or adding parentheses around one of the relational expressions.

When the intention is to pass several relational or shift expressions as actual arguments and there is an ambiguity, the source code can easily be adjusted to a form which is unambiguous. Also, we expect that it will be more common to have generic function invocations as actual arguments than having relational or shift expressions that happen to match up and have parentheses at the end, such that the ambiguity arises.

Evaluation of an actual argument part of the form

$<A_1, \ldots, A_r>(a_1, \ldots, a_m, q_1: a_{m+1}, \ldots, q_l: a_{m+l})$

proceeds as follows:

The type arguments $A_1, \ldots, A_r$ are evaluated in the order they appear in the program, producing types $t_1, \ldots, t_r$. The arguments $a_1, \ldots, a_{m+l}$ are evaluated in the order they appear in the program, producing objects $o_1, \ldots, o_{m+l}$.

Simply stated, an argument part consisting of s type arguments, m positional arguments, and l named arguments is evaluated from left to right. Note that the type argument list is omitted when $r = 0$ (14).

16.14.2 Binding Actuals to Formals

Let f be a function with s type parameters and h required parameters; let $p_1, \ldots, p_n$ be the positional parameters of f; and let $p_{h+1}, \ldots, p_{h+k}$ be the optional parameters declared by f.

Note that the type argument lists in the following are omitted in the case where $s = 0$, and similarly for empty type parameter lists (14).

An evaluated actual argument part $<t_1, \ldots, t_r>(o_1, \ldots, o_{m+l})$ derived from an actual argument part of the form

$<A_1, \ldots, A_r>(a_1, \ldots, a_m, q_1: a_{m+1}, \ldots, q_l: a_{m+l})$

is bound to the formal type parameters and formal parameters of f as follows:

We have an actual argument list consisting of r type arguments, m positional arguments, and l named arguments. We have a function with s type parameters, h required parameters, and k optional parameters. The number of type arguments must match the number of type parameters. The number of positional arguments must be at least as large as the number of required parameters, and no larger than the number of positional parameters. All named arguments must have a corresponding named parameter. A given named argument cannot be provided more than once. If an optional parameter has no corresponding argument, it gets its default value. In checked mode, all arguments must belong to subtypes of the type of their corresponding formal.

If $r = 0$ and $s > 0$ then replace the actual type argument list: let r be s and $t_i =$ **dynamic** for $i \in 1..s$. If $r \neq s$, a NoSuchMethodError is thrown. If $l > 0$ and $n \neq h$, a NoSuchMethodError is thrown. If $m < h$, or $m > n$, a NoSuchMethodError is thrown. Furthermore, each $q_i, i \in 1..l$, must have a corresponding named parameter in the set $\{p_{h+1}, \ldots, p_{h+k}\}$, or a NoSuchMethodError is thrown. Then p_i is bound to $o_i, i \in 1..m$, and q_j is bound to $o_{m+j}, j \in 1..l$. All remaining formal parameters of f are bound to their default values.

All of these remaining parameters are necessarily optional and thus have default values.

In checked mode, it is a dynamic type error if t_i is not a subtype of the actual bound (19.8.1) of the ith type argument of f, for actual type arguments $t_1, \ldots, t_r$. In checked mode, it is a dynamic type error if o_i is not the null object (16.2) and the actual type (19.8.1) of p_i is not a supertype of the type of $o_i, i \in 1..m$. In checked mode, it is a dynamic type error if o_{m+j} is not the null object and the actual type (19.8.1) of q_j is not a supertype of the type of $o_{m+j}, j \in 1..l$.

It is a compile-time error if $q_i = q_j$ for any $i \neq j$.

Let T_i be the static type of a_i. If the static type of f is **dynamic** or the built-in class **Function**, no further static checks are performed. Otherwise, it is a static type warning if the static type of f is not a function type.

Otherwise, let $X_1, \ldots, X_s$ be the formal type parameters of the static type of f, let S_i be the type of $p_i, i \in 1..h+k$, and let S_q be the type of the named parameter q of f, where each parameter type is obtained by replacing X_j by $A_j, j \in 1..s$, in the given parameter type annotation.

Checks regarding the number of type parameters and their bounds is specified in (14).

It is a static warning if T_j may not be assigned to $S_j, j \in 1..m$. It is a static warning if $m < h$ or if $m > n$. Furthermore, each $q_i, i \in 1..l$, must have a corresponding named parameter in the set $\{p_{h+1}, \ldots, p_{h+k}\}$ or a static warning occurs. It is a static warning if T_{m+j} may not be assigned to $S_{q_j}, j \in 1..l$.

16.14.3 Unqualified Invocation

An unqualified function invocation i has the form

$id\langle A_1, \ldots, A_r\rangle(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$,

where id is an identifier.

Note that the type argument list is omitted when $r = 0$ (14).

If there exists a lexically visible declaration named id, let f_{id} be the innermost such declaration. Then:

- If id is a type literal, then i is interpreted as a function expression invocation (16.14.4) with (id) as the expression e_f. The expression (id) where id is a type literal always evaluates to an instance of class Type which is not a function. This ensures that a run-time error occurs when trying to call a type literal.
- If f_{id} is a prefix object, a compile-time error occurs.
- If f_{id} is a local function, a library function, a library or static getter or a variable then i is interpreted as a function expression invocation (16.14.4).
- Otherwise, if f_{id} is a static method of the enclosing class C, i is equivalent to $C.id\langle A_1, \ldots, A_r\rangle(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

- Otherwise, f_{id} is equivalent to the ordinary method invocation
  this.id<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k}).

Otherwise, if i occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer, evaluation of i causes a NoSuchMethodError to be thrown.

If i does not occur inside a top level or static function, i is equivalent to **this**.id<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k}).

16.14.4 Function Expression Invocation

A function expression invocation i has the form

e_f<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k}),

where e_f is an expression.

Note that the type argument list is omitted when $r = 0$ (14).

If e_f is an identifier id, then id must necessarily denote a local function, a library function, a library or static getter or a variable as described above, or i is not considered a function expression invocation. If e_f is a type literal, then it is equivalent to the expression (e_f).

The expression (e_f) where e_f is a type literal always evaluates to an instance of class Type which is not a function. This ensures that a run-time error occurs when trying to call a type literal.

If e_f is a property extraction expression (16.19), then i isn't a function expression invocation and is instead recognized as an ordinary method invocation (16.18.1).

$a.b(x)$ is parsed as a method invocation of method $b()$ on object a, not as an invocation of getter b on a followed by a function call $(a.b)(x)$. If a method or getter b exists, the two will be equivalent. However, if b is not defined on a, the resulting invocation of noSuchMethod() would differ. The Invocation passed to noSuchMethod() would describe a call to a method b with argument x in the former case, and a call to a getter b (with no arguments) in the latter.

Otherwise, evaluation of a function expression invocation

e_f<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k})

proceeds to evaluate e_f, yielding an object o. Let f be a fresh variable bound to o. If o is a function object then the following function invocation is evaluated and its result is the result of evaluating i:

f<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k}).

Otherwise, if o has a method named **call**, the following method invocation is evaluated and its result is the result of evaluating i:

f.call<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k}).

Otherwise, when o has no method named **call**, a new instance im of the predefined class Invocation is created, such that:

- im.isMethod evaluates to **true**.
- im.memberName evaluates to the symbol #call.

- im.positionalArguments evaluates to an unmodifiable list with the values resulting from evaluation of <Object>$[a_1, \ldots, a_n]$.
- im.namedArguments evaluates to an unmodifiable map with the keys and values resulting from evaluation of

  <Symbol, Object>$\{\#x_{n+1}: a_{n+1}, \ldots, x_{n+k}: \#a_{n+k}\}$.
- im.typeArguments evaluates to an unmodifiable list with the values resulting from evaluation of <Type>$[A_1, \ldots, A_r]$.

Then the method invocation f.noSuchMethod(im) is evaluated, and its result is the result of evaluating i.

Let F be the static type of e_f. It is a static warning unless one of the following conditions is satisfied:

1. F is **dynamic**.
2. F is **Function**.
3. F is a function type

   $<X_1$ **extends** $B_1, \ldots, A_s$ **extends** $B_s>$

   $(T_1, \ldots, T_n, \{T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}, \ldots, T_{n+k+p}\ x_{n+k+p}\}) \rightarrow T_0$

   where $r = s$, and the static type of a_j is assignable to $[A_1/X_1, \ldots, A_r/X_s]T_j$, for all $j \in 1..n+k$.
4. k is zero, $0 \leq m \leq n$, and F is a function type

   $<X_1$ **extends** $B_1, \ldots, A_s$ **extends** $B_s>$

   $(T_1, \ldots, T_m, [T_{m+1}, \ldots, T_n, \ldots, T_{n+p}]) \rightarrow T_0$

   where $r = s$, and the static type of a_j is assignable to $[A_1/X_1, \ldots, A_r/X_s]T_j$, for all $j \in 1..n$.
5. F is an interface type that has a method named **call**, and then, considering F to denote the function type of that method, criterion 3 or 4 in this list is satisfied.

This means that the types of the actual arguments must match the declared types of the corresponding parameters, where the formal type parameters have been replaced by the actual type arguments. Note that the type parameter lists are omitted when $s = 0$ (14), and that checks on the number of type arguments and their bounds is specified elsewhere (14). Criterion 5 ensures that a function expression invocation may amount to an invocation of the instance method **call** when such a method is statically known, and the run-time semantics ensures that a function invocation may amount to an invocation of the instance method **call** also for an invocation of a function of type **dynamic** or **Function**, but that an interface type with a method named **call** is not itself a subtype of any function type.

If F is not a function type or $r \neq s$, the static type of i is **dynamic**. Otherwise, the static type of i is the return type $[A_1/X_1, \ldots, A_r/X_s]T_0$ of F.

16.15 Function Closurization

Let f be an expression denoting a declaration of a local function, a static method, or a top-level function (16.34) or let f be a function literal (16.10). Evaluation of f yields a function object which is the outcome of a *function closurization* applied to the declaration denoted by f respectively to the function literal f considered as a function declaration. *Closurization* denotes instance method closurization (16.19.3) as well as function closurization, and it is also used as a shorthand for either of them when there is no ambiguity.

Function closurization applied to a function declaration f amounts to the creation of a function object o which is an instance of a class whose interface is a subtype of the actual type (19.8.1) corresponding to the signature in the function declaration f, using the current bindings of type variables, if any. An invocation of o with a given argument list will bind actuals to formals in the same way as an invocation of f (16.14.2), and then execute the body of f in the captured scope amended with the bound parameter scope, yielding the same completion (17) as the invocation of f would have yielded.

16.16 Lookup

16.16.1 Method Lookup

The result of a lookup of a method m in object o with respect to library L is the result of a lookup of method m in class C with respect to library L, where C is the class of o.

The result of a lookup of method m in class C with respect to library L is: If C declares a concrete instance method named m that is accessible to L, then that method is the result of the lookup, and we say that the method was *looked up in* C. Otherwise, if C has a superclass S, then the result of the lookup is the result of looking up m in S with respect to L. Otherwise, we say that the method lookup has failed.

The motivation for skipping abstract members during lookup is largely to allow smoother mixin composition.

16.16.2 Getter and Setter Lookup

The result of a lookup of a getter (respectively setter) m in object o with respect to library L is the result of looking up getter (respectively setter) m in class C with respect to L, where C is the class of o.

The result of a lookup of a getter (respectively setter) m in class C with respect to library L is: If C declares a concrete instance getter (respectively setter) named m that is accessible to L, then that getter (respectively setter) is the result of the lookup, and we say that the getter (respectively setter) was *looked up in* C. Otherwise, if C has a superclass S, then the result of the lookup is the result of looking up getter (respectively setter) m in S with respect to L. Otherwise, we say that the lookup has failed.

The motivation for skipping abstract members during lookup is largely to allow smoother mixin composition.

16.17 Top level Getter Invocation

Evaluation of a top-level getter invocation i of the form m, where m is an identifier, proceeds as follows:

The getter function m is invoked. The value of i is the result returned by the call to the getter function. Note that the invocation is always defined. Per the rules for identifier references, an identifier will not be treated as a top-level getter invocation unless the getter i is defined.

The static type of i is the declared return type of m.

16.18 Method Invocation

Method invocation can take several forms as specified below.

16.18.1 Ordinary Invocation

Note that non-generic invocations arise as the special case where the number of type arguments is zero, in which case the type argument list is omitted, and similarly for formal type parameter lists (14).

An ordinary method invocation can be *conditional* or *unconditional*.

Evaluation of a *conditional ordinary method invocation* i of the form

$e?.m<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$

proceeds as follows:

If e is a type literal, i is equivalent to

$e.m<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$.

Otherwise, evaluate e to an object o. If o is the null object, i evaluates to the null object (16.2). Otherwise let v be a fresh variable bound to o and evaluate $v.m<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ to a value r. Then e evaluates to r.

The static type of i is the same as the static type of

$e.m<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$.

Exactly the same static warnings that would be caused by

$e.m<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$

are also generated in the case of

$e?.m<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$.

An *unconditional ordinary method invocation* i has the form

$e.m<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$.

Evaluation of an unconditional ordinary method invocation i of the form $e.m<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ proceeds as follows:

First, the expression e is evaluated to a value o. Next, the argument part $<A_1, \ldots, A_r>(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ is evaluated yielding actual type arguments $t_1, \ldots, t_r$ and actual argument objects $o_1, \ldots, o_{n+k}$. Let

f be the result of looking up (16.16.1) method m in o with respect to the current library L.

Let $X_1, \ldots, X_s$ be the formal type parameters of f, let $p_1, \ldots, p_h$ be the required parameters of f, let $p_1, \ldots, p_m$ be the positional parameters of f, and let $p_{h+1}, \ldots, p_{h+l}$ be the optional parameters declared by f.

We have an actual argument list consisting of r type arguments, n positional arguments, and k named arguments. We have a function with s type parameters, h required parameters, and l optional parameters. The number of type arguments must match the number of type parameters. The number of positional arguments must be at least as large as the number of required parameters, and no larger than the number of positional parameters. All named arguments must have a corresponding named parameter. A given named argument cannot be provided more than once.

If $r = 0$ and $s > 0$ then replace the actual type argument list: let r be s and $t_i =$ **dynamic** for $i \in 1..s$. If $r \neq s$, the method lookup has failed. If $k > 0$ and $m \neq h$, the method lookup has failed. If $n < h$, or $n > m$, the method lookup has failed. Furthermore, each $x_i, i \in n + 1..n + k$, must have a corresponding named parameter in the set $\{p_{h+1}, \ldots, p_{h+l}\}$, or the method lookup also fails.

If o is an instance of Type but e is not a constant type literal, then if m is a method that forwards (9.1) to a static method, method lookup fails. Otherwise method lookup has succeeded.

If the method lookup succeeded, the body of f is executed with respect to the bindings that resulted from the evaluation of the argument list, and with **this** bound to o. The value of i is the value returned after f is executed.

If the method lookup has failed, then let g be the result of looking up getter (16.16.2) m in o with respect to L. If o is an instance of Type but e is not a constant type literal, then if g is a getter that forwards to a static getter, getter lookup fails. If the getter lookup succeeded, let v_g be the value of the getter invocation $e.m$. Then the value of i is the result of invoking the static method Function.apply() with arguments v_g, $[o_1, \ldots, o_n]$, $\{\#x_{n+1}: o_{n+1}, \ldots, \#x_{n+k}: o_{n+k}\}$, and $[t_1, \ldots, t_r]$.

If getter lookup has also failed, then a new instance im of the predefined class Invocation is created, such that:

- im.isMethod evaluates to **true**.
- im.memberName evaluates to the symbol m.
- im.positionalArguments evaluates to an unmodifiable list with the same values as <Object>$[o_1, \ldots, o_n]$.
- im.namedArguments evaluates to an unmodifiable map with the same keys and values as <Symbol, Object>$\{\#x_{n+1}: o_{n+1}, \ldots, \#x_{n+k}: o_{n+k}\}$.
- im.typeArguments evaluates to an unmodifiable list with the same values as <Type>$[t_1, \ldots, t_r]$.

Then the method noSuchMethod() is looked up in o and invoked with argument im, and the result of this invocation is the result of evaluating i.

It is not possible to override the noSuchMethod of class Object in such a way that it cannot be invoked with one argument of type Invocation.

Notice that the wording carefully avoids re-evaluating the receiver o and the arguments a_i.

Let T be the static type of e. It is a static type warning if T does not have an accessible (6.2) instance member named m, unless either:

- T is Type, e is a constant type literal, and the class corresponding to e has a static getter named m. Or
- T is **Function** and m is **call**. *This means that for invocations of an instance method named call, a receiver of type **Function** is treated like a receiver of type **dynamic**. The expectation is that any concrete subclass of **Function** will implement **call**, but there is no method signature which can be assumed for **call** in **Function** because every signature will conflict with some potential overriding declarations. Note that any use of **call** on a subclass of **Function** that fails to implement **call** will provoke a warning, as this exemption is limited to type **Function**, and does not apply to its subtypes.*

If $T.m$ exists, it is a static type warning if the type F of $T.m$ may not be assigned to a function type. If $T.m$ does not exist, or if F is not a function type, the static type of i is **dynamic**. Otherwise, let $X_1, \ldots, X_s$ be the formal type parameters of the type of F, and T_0 its declared return type. If $r \neq s$ the static type of i is **dynamic**; otherwise, the static type of i is $[A_1/X_1, \ldots, A_r/X_s]T_0$.

That is, the declared return type where each formal type parameter has been replaced by the corresponding actual type argument.

It is a compile-time error to invoke any of the methods of class Object on a prefix object (18.1) or on a constant type literal that is immediately followed by the token '.'.

16.18.2 Cascaded Invocations

A *cascaded method invocation* has the form *e..suffix* where e is an expression and *suffix* is a sequence of operator, method, getter or setter invocations.

cascadeSection:
'..' (cascadeSelector argumentPart*)
(assignableSelector argumentPart*)*
(assignmentOperator expressionWithoutCascade)?
;

cascadeSelector:
'[' expression ']' |
identifier

;

argumentPart:
typeArguments? arguments
;

Evaluation of a cascaded method invocation expression e of the form *e..suffix* proceeds as follows:

Evaluate e to an object o. Let t be a fresh variable bound to o. Evaluate *t.suffix* to an object. Then e evaluates to o.

*With the introduction of null-aware conditional assignable expressions (16.33), it would make sense to extend cascades with a null-aware conditional form as well. One might define e?..suffix to be equivalent to the expression t == **null** ? **null** : t.suffix where t is a fresh variable bound to the value of e.*

The present specification has not added such a construct, in the interests of simplicity and rapid language evolution. However, Dart implementations may experiment with such constructs, as noted in section 2.

16.18.3 Super Invocation

Note that non-generic invocations arise as the special case where the number of type arguments is zero, in which case the type argument list is omitted, and similarly for formal type parameter lists (14).

A super method invocation i has the form
super.m<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k}).

Evaluation of i proceeds as follows:

First, the argument part
<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k})
is evaluated producing actual type arguments $t_1, \ldots, t_r$ and actual argument objects $o_1, \ldots, o_{n+k}$. Let g be the method currently executing, and let C be the class in which g was looked up (16.16.1). Let $S_{dynamic}$ be the superclass of C, and let f be the result of looking up method (16.16.1) m in $S_{dynamic}$ with respect to the current library L.

Let $X_1, \ldots, X_s$ be the formal type parameters of f. Let $p_1, \ldots, p_h$ be the required parameters of f, let $p_1, \ldots, p_m$ be the positional parameters of f, and let $p_{h+1}, \ldots, p_{h+l}$ be the optional parameters declared by f.

We have an actual argument list consisting of r type arguments, n positional arguments, and k named arguments. We have a function with s type parameters, h required parameters, and l optional parameters. The number of type arguments must match the number of type parameters. The number of positional arguments must be at least as large as the number of required parameters, and no larger than the number of positional parameters. All named arguments must have a corresponding named parameter. A given named argument cannot be provided more than once.

If $r \neq s$, the method lookup has failed. If $k > 0$ and $m \neq h$, the method

lookup has failed. If $n < h$, or $n > m$, the method lookup has failed. Furthermore, each $x_i, i \in n+1..n+k$, must have a corresponding named parameter in the set $\{p_{h+1}, \ldots, p_{h+l}\}$, or the method lookup also fails. Otherwise, method lookup has succeeded.

If the method lookup succeeded, the body of f is executed with respect to the bindings that resulted from the evaluation of the argument list, and with **this** bound to the current value of **this**. The value of i is the value returned after f is executed.

If the method lookup has failed, then let g be the result of looking up getter (16.16.2) m in $S_{dynamic}$ with respect to L. If the getter lookup succeeded, let v_g be the value of the getter invocation **super**.m. Then the value of i is the result of invoking the static method Function.apply() with arguments $v_g, [o_1, \ldots, o_n], \{\#x_{n+1} : o_{n+1}, \ldots, \#x_{n+k} : o_{n+k}\}, [t_1, \ldots, t_r]$.

If getter lookup has also failed, then a new instance im of the predefined class Invocation is created, such that:

- im.isMethod evaluates to **true**.
- im.memberName evaluates to the symbol m.
- im.positionalArguments evaluates to an unmodifiable list with the same values as <Object>$[o_1, \ldots, o_n]$.
- im.namedArguments evaluates to an unmodifiable map with the same keys and values as <Symbol, Object>$\{\#x_{n+1} : o_{n+1}, \ldots, \#x_{n+k} : o_{n+k}\}$.
- im.typeArguments evaluates to an unmodifiable list with the same values as <Type>$[t_1, \ldots, t_r]$.

Then the method noSuchMethod() is looked up in $S_{dynamic}$ and invoked on **this** with argument im, and the result of this invocation is the result of evaluating i.

It is a compile-time error if a super method invocation occurs in a top-level function or variable initializer, in an instance variable initializer or initializer list, in class Object, in a factory constructor, or in a static method or variable initializer.

Let S_{static} be the superclass of the immediately enclosing class. It is a static type warning if S_{static} does not have an accessible (6.2) instance member named m.

If $S_{static}.m$ exists, it is a static type warning if the type F of $S_{static}.m$ may not be assigned to a function type. If $S_{static}.m$ does not exist, or if F is not a function type, the static type of i is **dynamic**; Otherwise, let $X_1, \ldots, X_s$ be the formal type parameters of the type of F, and T_0 its declared return type. If $r \neq s$ the static type of i is **dynamic**; otherwise, the static type of i is $[A_1/X_1, \ldots, A_r/X_s]T_0$.

That is, the declared return type where each formal type parameter has been replaced by the corresponding actual type argument.

16.18.4 Sending Messages

Messages are the sole means of communication among isolates. Messages are sent by invoking specific methods in the Dart libraries; there is no specific syntax for sending a message.

In other words, the methods supporting sending messages embody primitives of Dart that are not accessible to ordinary code, much like the methods that spawn isolates.

16.19 Property Extraction

Property extraction allows for a member to be accessed as a property rather than a function. A property extraction can be either:

1. An instance method closurization, which converts a method into a function object (16.19.3). Or
2. A getter invocation, which returns the result of invoking of a getter method (16.19.1).

Function objects derived via closurization are colloquially known as tear-offs.

Property extraction can be either *conditional* or *unconditional*.

Evaluation of a *conditional property extraction expression* e of the form $e_1?.id$ proceeds as follows:

If e_1 is a type literal, e is equivalent to $e_1.m$.

Otherwise evaluate e_1 to an object o. If o is the null object, e evaluates to the null object (16.2). Otherwise let x be a fresh variable bound to o and evaluate $x.id$ to a value r. Then e evaluates to r.

The static type of e is the same as the static type of $e_1.id$. Let T be the static type of e_1 and let y be a fresh variable of type T. Exactly the same static warnings that would be caused by $y.id$ are also generated in the case of $e_1?.id$.

Unconditional property extraction has one of two syntactic forms: $e.m$ (16.19.1) or **super**.m (16.19.2), where e is an expression and m is an identifier.

16.19.1 Getter Access and Method Extraction

Evaluation of a property extraction i of the form $e.m$ proceeds as follows:

First, the expression e is evaluated to an object o. Let f be the result of looking up (16.16.1) method (10.1) m in o with respect to the current library L. If o is an instance of Type but e is not a constant type literal, then if f is a method that forwards (9.1) to a static method, method lookup fails. If method lookup succeeds then i evaluates to the closurization of method f on object o (16.19.3).

Note that f is never an abstract method, because method lookup skips abstract methods. Hence, if m refers to an abstract method, we will continue to the next step. However, since methods and getters never override each other, getter lookup will necessarily fail as well, and noSuchMethod() will ultimately be invoked. The

regrettable implication is that the error will refer to a missing getter rather than an attempt to closurize an abstract method.

Otherwise, i is a getter invocation. Let f be the result of looking up (16.16.2) getter (10.2) m in o with respect to L. If o is an instance of Type but e is not a constant type literal, then if f is a getter that forwards to a static getter, getter lookup fails. Otherwise, the body of f is executed with **this** bound to o. The value of i is the result returned by the call to the getter function.

If the getter lookup has failed, then a new instance im of the predefined class Invocation is created, such that :

- im.isGetter evaluates to **true**.
- im.memberName evaluates to the symbol m.
- im.positionalArguments evaluates to an empty, unmodifiable instance of List<Object>.
- im.namedArguments evaluates to an empty, unmodifiable instance of Map<Symbol, Object>.
- im.typeArguments evaluates to an empty, unmodifiable instance of List<Type>.

Then the method noSuchMethod() is looked up in o and invoked with argument im, and the result of this invocation is the result of evaluating i.

It is a compile-time error if m is a member of class Object and e is either a prefix object (18.1) or a constant type literal.

This precludes int.toString but not (int).toString because in the latter case, e is a parenthesized expression.

Let T be the static type of e. It is a static type warning if T does not have a method or getter named m, unless T is Type, e is a constant type literal, and the class corresponding to e has a static method or getter named m.

The static type of i is:

- The declared return type of $T.m$, if T has an accessible instance getter named m.
- The declared return type of m, if T is Type, e is a constant type literal and the class corresponding to e declares an accessible static getter named m.
- The static type of function $T.m$ if T has an accessible instance method named m.
- The static type of function m, if T is Type, e is a constant type literal and the class corresponding to e declares an accessible static method named m.
- The type **dynamic** otherwise.

16.19.2 Super Getter Access and Method Closurization

Evaluation of a property extraction i of the form **super**.m proceeds as follows:

Let g be the method currently executing, and let C be the class in which g was looked up. Let $S_{dynamic}$ be the superclass of C. Let f be the result of looking up method m in $S_{dynamic}$ with respect to the current library L. If method lookup succeeds then i evaluates to the closurization of method f with respect to superclass $S_{dynamic}$ (16.19.4).

Otherwise, i is a getter invocation. Let f be the result of looking up getter m in $S_{dynamic}$ with respect to L. The body of f is executed with **this** bound to the current value of **this**. The value of i is the result returned by the call to the getter function.

If the getter lookup has failed, then a new instance im of the predefined class Invocation is created, such that:

- im.isGetter evaluates to **true**.
- im.memberName evaluates to the symbol m.
- im.positionalArguments evaluates to an empty, unmodifiable instance of List<Object>.
- im.namedArguments evaluates to an empty, unmodifiable instance of Map<Symbol, Object>.
- im.typeArguments evaluates to an empty, unmodifiable instance of List<Type>.

Then the method noSuchMethod() is looked up in $S_{dynamic}$ and invoked with argument im, and the result of this invocation is the result of evaluating i.

Let S_{static} be the superclass of the immediately enclosing class. It is a static type warning if S_{static} does not have an accessible instance method or getter named m.

The static type of i is:

- The declared return type of $S_{static}.m$, if S_{static} has an accessible instance getter named m.
- The static type of function $S_{static}.m$, if S_{static} has an accessible instance method named m.
- The type **dynamic** otherwise.

16.19.3 Ordinary Member Closurization

Note that the non-generic case is covered implicitly using $s = 0$, in which case the type parameter declarations are omitted (14).

An *instance method closurization* is a closurization of some method on some

object, defined below, or a super closurization (16.19.4).

Let o be an object, and let u be a fresh final variable bound to o. The *closurization of method f on object o* is defined to be equivalent (except for equality, as noted below) to:

- $<X_1$ **extends** $B'_1, \ldots, X_s$ **extends** $B'_s>$
  $(T_1\ r_1, \ldots, T_n\ r_n, \{T_{n+1}\ p_1\ =\ d_1, \ldots, T_{n+k}\ p_k\ =\ d_k\})$ =>
  $u.m<X_1, \ldots, X_s>(r_1, \ldots, r_n, p_1: p_1, \ldots, p_k: p_k)$;
  if f is named m and has type parameter declarations X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required parameters $r_1, \ldots, r_n$, and named parameters $p_1, \ldots, p_k$ with defaults $d_1, \ldots, d_k$.

- $<X_1$ **extends** $B'_1, \ldots, X_s$ **extends** $B'_s>$
  $(T_1\ r_1, \ldots, T_n\ r_n, [T_{n+1}\ p_1\ =\ d_1, \ldots, T_{n+k}\ p_k\ =\ d_k])$ =>
  $u.m<X_1, \ldots, X_s>(r_1, \ldots, r_n, p_1, \ldots, p_k)$;
  if f is named m and has type parameter declarations X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required parameters $r_1, \ldots, r_n$, and optional positional parameters $p_1, \ldots, p_k$ with defaults $d_1, \ldots, d_k$.

$B'_j, j \in 1..s$, are determined as follows: If o is an instance of a non-generic class, $B'_j = B_j, j \in 1..s$. Otherwise, let $X'_1, \ldots, X'_{s'}$ be the formal type parameters of the class of o, and $t'_1, \ldots, t'_{s'}$ be the actual type arguments. Then $B'_j = [t'_1/X'_1, \ldots, t'_{s'}/X'_{s'}]B_j, j \in 1..s$.

That is, we replace the formal type parameters of the enclosing class, if any, by the corresponding actual type arguments.

The parameter types $T_j, j \in 1..n+k$, are determined as follows: Let the method declaration D be the implementation of m which is invoked by the expression in the body. Let T be the class that contains D.

Note that T is the dynamic type of o, or a superclass thereof.

If T is a non-generic class then for $j \in 1..n+k$, T_j is a type annotation that denotes the same type as that which is denoted by the type annotation on the corresponding parameter declaration in D. If that parameter declaration has no type annotation then T_j is **dynamic**.

Otherwise T is a generic instantiation of a generic class G. Let $X''_1, \ldots, X''_{s''}$ be the formal type parameters of G, and $t''_1, \ldots, t''_{s''}$ be the actual type arguments of o at T. Then T_j is a type annotation that denotes $[t''_1/X''_1, \ldots, t''_{s''}/X''_{s''}]S_j$, where S_j is the type annotation of the corresponding parameter in D. If that parameter declaration has no type annotation then T_j is **dynamic**.

There is one way in which the function object yielded by the instance method closurization differs from the function object obtained by function closurization on the above mentioned function literal: Assume that o_1 and o_2 are objects, m is an identifier, and c_1 and c_2 are function objects obtained by closurization of m on o_1 respectively o_2. Then $c_1 == c_2$ evaluates to true if and only if o_1 and o_2 is the same object.

In particular, two closurizations of a method m from the same object are equal, and two closurizations of a method m from non-identical objects are not equal. Assuming that v_i is a fresh variable bound to an object, $i \in 1..2$, it also follows that identical($v_1.m, v_2.m$) must be false when v_1 and v_2 are not bound to the same object. However, Dart implementations are not required to canonicalize function objects, which means that identical($v_1.m, v_2.m$) is not guaranteed to be true, even when it is known that v_1 and v_2 are bound to the same object.

The special treatment of equality in this case facilitates the use of extracted property functions in APIs where callbacks such as event listeners must often be registered and later unregistered. A common example is the DOM API in web browsers.

16.19.4 Super Closurization

Note that the non-generic case is covered implicitly using $s = 0$, in which case the type parameter declarations are omitted (14).

Consider expressions in the body of a class T which is a subclass of a given class S, where a method declaration that implements f exists in S, and there is no class U which is a subclass of S and a superclass of T which implements f.

In short, consider a situation where a superinvocation of f will execute f as declared in S.

A *super closurization* is a closurization of a method with respect to a class, as defined next. The *closurization of the method f with respect to the class S* is defined to be equivalent (except for equality, as noted below) to:

- $<X_1$ **extends** $B'_1, \ldots, X_s$ **extends** $B'_s>$
  $(T_1\ r_1, \ldots, T_n\ r_n, \{T_{n+1}\ p_1 = d_1, \ldots, T_{n+k}\ p_k = d_k\}) =>$
  super.$m<X_1, \ldots, X_s>(r_1, \ldots, r_n, p_1: p_1, \ldots, p_k: p_k)$;
  if f is named m and has type parameter declarations X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required parameters $r_1, \ldots, r_n$, and named parameters $p_1, \ldots, p_k$ with defaults $d_1, \ldots, d_k$.
- $<X_1$ **extends** $B'_1, \ldots, X_s$ **extends** $B'_s>$
  $(T_1\ r_1, \ldots, T_n\ r_n, [T_{n+1}\ p_1 = d_1, \ldots, T_{n+k}\ p_k = d_k]) =>$
  super.$m<X_1, \ldots, X_s>(r_1, \ldots, r_n, p_1, \ldots, p_k)$;
  if f is named m and has type parameter declarations X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required parameters $r_1, \ldots, r_n$, and optional positional parameters $p_1, \ldots, p_k$ with defaults $d_1, \ldots, d_k$.

Note that a super closurization is an *instance method closurization*, as defined in (16.19.3).

$B'_j, j \in 1..s$, are determined as follows: If S is a non-generic class then $B'_j = B_j, j \in 1..s$. Otherwise, let $X'_1, \ldots, X'_{s'}$ be the formal type parameters

of S, and $t'_1, \ldots, t'_{s'}$ be the actual type arguments of **this** at S. Then $B'_j = [t'_1/X'_1, \ldots, t'_{s'}/X'_{s'}]B_j, j \in 1..s$.

That is, we replace the formal type parameters of the enclosing class, if any, by the corresponding actual type arguments. We need to consider the type arguments with respect to a specific class because it is possible for a class to pass different type arguments to its superclass than the ones it receives itself.

The parameter types $T_j, j \in 1..n+k$, are determined as follows: Let the method declaration D be the implementation of m in S.

If S is a non-generic class then for $j \in 1..n+k$, T_j is a type annotation that denotes the same type as that which is denoted by the type annotation on the corresponding parameter declaration in D. If that parameter declaration has no type annotation then T_j is **dynamic**.

Otherwise S is a generic instantiation of a generic class G. Let $X''_1, \ldots, X''_{s''}$ be the formal type parameters of G, and $t''_1, \ldots, t''_{s''}$ be the actual type arguments of o at S. Then T_j is a type annotation that denotes $[t''_1/X''_1, \ldots, t''_{s''}/X''_{s''}]S_j$, where S_j is the type annotation of the corresponding parameter in D. If that parameter declaration has no type annotation then T_j is **dynamic**.

There is one way in which the function object yielded by the super closurization differs from the function object obtained by function closurization on the above mentioned function literal: Assume that an occurrence of the expression **super**.m in a given class is evaluated on two occasions where **this** is bound to o_1 respectively o_2, and the resulting function objects are c_1 respectively c_2: c_1 == c_2 is then true if and only if o_1 and o_2 is the same object.

16.20 Assignment

An assignment changes the value associated with a mutable variable or property.

```
assignmentOperator:
  '=' |
  compoundAssignmentOperator
;
```

Evaluation of an assignment a of the form $v = e$ proceeds as follows:

Let d be the innermost declaration whose name is v or $v =$, if it exists. It is a compile-time error if d denotes a prefix object.

If d is the declaration of a local variable, the expression e is evaluated to an object o. Then, the variable v is bound to o unless v is **final** or **const**, in which case a dynamic error occurs. If no error occurs, the value of the assignment expression is o.

If d is the declaration of a library variable, top level getter or top level setter, the expression e is evaluated to an object o. Then the setter $v =$ is invoked with its formal parameter bound to o. The value of the assignment expression is o.

Otherwise, if d is the declaration of a static variable, static getter or static

setter in class C, then the assignment is equivalent to the assignment $C.v = e$.

Otherwise, If a occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer, evaluation of a causes e to be evaluated, after which a NoSuchMethodError is thrown.

Otherwise, the assignment is equivalent to the assignment **this**$.v = e$.

In checked mode, it is a dynamic type error if o is not the null object (16.2) and the interface of the class of o is not a subtype of the actual type (19.8.1) of v.

It is a static type warning if the static type of e may not be assigned to the static type of v. The static type of the expression $v = e$ is the static type of e.

Evaluation of an assignment a of the form $e_1?.v = e_2$ proceeds as follows:

f e_1 is a type literal, a is equivalent to $e_1.v = e_2$.

Otherwise evaluate e_1 to an object o. If o is the null object, a evaluates to the null object (16.2). Otherwise let x be a fresh variable bound to o and evaluate $x.v = e_2$ to an object r. Then a evaluates to r.

The static type of a is the static type of e_2. Let T be the static type of e_1 and let y be a fresh variable of type T. Exactly the same static warnings that would be caused by $y.v = e_2$ are also generated in the case of $e_1?.v = e_2$.

Evaluation of an assignment of the form $e_1.v = e_2$ proceeds as follows:

The expression e_1 is evaluated to an object o_1. Then, the expression e_2 is evaluated to an object o_2. Then, the setter $v =$ is looked up (16.16.2) in o_1 with respect to the current library. If o_1 is an instance of Type but e_1 is not a constant type literal, then if $v =$ is a setter that forwards (9.1) to a static setter, setter lookup fails. Otherwise, the body of $v =$ is executed with its formal parameter bound to o_2 and **this** bound to o_1.

If the setter lookup has failed, then a new instance im of the predefined class Invocation is created, such that:

- im.isSetter evaluates to **true**.
- im.memberName evaluates to the symbol v=.
- im.positionalArguments evaluates to an unmodifiable list with the same values as <Object>$[o_2]$.
- im.namedArguments evaluates to an empty, unmodifiable instance of Map<Symbol, Object>.
- im.typeArguments evaluates to an empty, unmodifiable instance of List<Type>. m

Then the method noSuchMethod() is looked up in o_1 and invoked with argument im.

The value of the assignment expression is o_2 irrespective of whether setter lookup has failed or succeeded.

In checked mode, it is a dynamic type error if o_2 is not the null object (16.2) and the interface of the class of o_2 is not a subtype of the actual type of $e_1.v$.

Let T be the static type of e_1. It is a static type warning if T does not have an accessible instance setter named $v =$ unless T is Type, e_1 is a constant type literal and the class corresponding to e_1 has a static setter named $v =$.

It is a static type warning if the static type of e_2 may not be assigned to the static type of the formal parameter of the setter $v =$. The static type of the expression $e_1.v = e_2$ is the static type of e_2.

Evaluation of an assignment of the form **super**.$v = e$ proceeds as follows:

Let g be the method currently executing, and let C be the class in which g was looked up. Let $S_{dynamic}$ be the superclass of C. The expression e is evaluated to an object o. Then, the setter $v =$ is looked up (16.16.2) in $S_{dynamic}$ with respect to the current library. The body of $v =$ is executed with its formal parameter bound to o and **this** bound to **this**.

If the setter lookup has failed, then a new instance im of the predefined class Invocation is created, such that:

- im.isSetter evaluates to **true**.
- im.memberName evaluates to the symbol v=.
- im.positionalArguments evaluates to an unmodifiable list with the same values as [o].
- im.namedArguments evaluates to an empty unmodifiable instance of Map<Symbol, Object>.
- im.typeArguments evaluates to an empty, unmodifiable instance of List<Type>.

Then the method noSuchMethod() is looked up in $S_{dynamic}$ and invoked with argument im.

The value of the assignment expression is o irrespective of whether setter lookup has failed or succeeded.

In checked mode, it is a dynamic type error if o is not the null object (16.2) and the interface of the class of o is not a subtype of the actual type of $S.v$.

Let S_{static} be the superclass of the immediately enclosing class. It is a static type warning if S_{static} does not have an accessible instance setter named $v =$ unless S_{static}.

It is a static type warning if the static type of e may not be assigned to the static type of the formal parameter of the setter $v =$. The static type of the expression **super**.$v = e$ is the static type of e.

Evaluation of an assignment e of the form $e_1[e_2] = e_3$ proceeds as follows:

Evaluate e_1 to an object a, then evaluate e_2 to an object i, and finally evaluate e_3 to an object v. Call the method []= on a with i as first argument and v as second argument. Then e evaluates to v.

The static type of the expression $e_1[e_2] = e_3$ is the static type of e_3.

An assignment of the form **super**$[e_1] = e_2$ is equivalent to the expression **super**.$[e_1] = e_2$. The static type of the expression **super**$[e_1] = e_2$ is the static type of e_2.

It is a static warning if an assignment of the form $v = e$ occurs inside a

top level or static function (be it function, method, getter, or setter) or variable initializer and there is neither a local variable declaration with name v nor setter declaration with name $v =$ in the lexical scope enclosing the assignment.

It is a compile-time error to invoke any of the setters of class Object on a prefix object (18.1) or on a constant type literal that is immediately followed by the token '.'.

16.20.1 Compound Assignment

Evaluation of a compound assignment a of the form v ??= e proceeds as follows:

Evaluate v to an object o. If o is not the null object (16.2), a evaluates to o. Otherwise evaluate $v = e$ to a value r, and then a evaluates to r.

Evaluation of a compound assignment, a of the form $C.v$??= e, where C is a type literal, proceeds as follow:

Evaluate $C.v$ to an object o. If o is not the null object (16.2), a evaluates to o. Otherwise evaluate $C.v = e$ to a value r, and then a evaluates to r.

The two rules above also apply when the variable v or the type C is prefixed.

Evaluation of a compound assignment a of the form $e_1.v$??= e_2 proceeds as follows:

Evaluate e_1 to an object u. Let x be a fresh variable bound to u. Evaluate $x.v$ to an object o. If o is not the null object (16.2), a evaluates to o. Otherwise evaluate $x.v = e_2$ to an object r, and then a evaluates to r.

Evaluation of a compound assignment a of the form $e_1[e_2]$??= e_3 proceeds as follows:

Evaluate e_1 to an object u and then evaluate e_2 to an object i. Call the [] method on u with argument i, and let o be the returned value. If o is not the null object (16.2), a evaluates to o. Otherwise evaluate e_3 to an object v and then call the []= method on u with i as first argument and v as second argument. Then a evaluates to v.

Evaluation of a compound assignment a of the form **super**.v ??= e proceeds as follows:

Evaluate **super**.v to an object o. If o is not the null object (16.2) then a evaluates to o. Otherwise evaluate **super**.$v = e$ to an object r, and then a evaluates to r.

Evaluation of a compound assignment a of the form e_1?.v ??= e_2 proceeds as follows:

Evaluate e_1 to an object u and let x be a fresh variable bound to u. Evaluate $x.v$ to an object o. If o is not the null object (16.2) then a evaluates to o. Otherwise evaluate $x.v = e_2$ to an object r, and then a evaluates to r.

A compound assignment of the form C?.v *??*= e_2 is equivalent to the expression $C.v$ *??*= e.

The static type of a compound assignment of the form v *??*= e is the least upper bound of the static type of v and the static type of e. Exactly the same static warnings that would be caused by $v = e$ are also generated in the case of v *??*= e.

The static type of a compound assignment of the form $C.v$ *??*= e is the least upper bound of the static type of $C.v$ and the static type of e. Exactly the same static warnings that would be caused by $C.v = e$ are also generated in the case of $C.v$ *??*= e.

The static type of a compound assignment of the form $e_1.v$ *??*= e_2 is the least upper bound of the static type of $e_1.v$ and the static type of e_2. Let T be the static type of e_1 and let z be a fresh variable of type T. Exactly the same static warnings that would be caused by $z.v = e_2$ are also generated in the case of $e_1.v$ *??*= e_2.

The static type of a compound assignment of the form $e_1[e_2]$ *??*= e_3 is the least upper bound of the static type of $e_1[e_2]$ and the static type of e_3. Exactly the same static warnings that would be caused by $e_1[e_2] = e_3$ are also generated in the case of $e_1[e_2]$ *??*= e_3.

The static type of a compound assignment of the form **super**.v *??*= e is the least upper bound of the static type of **super**.v and the static type of e. Exactly the same static warnings that would be caused by **super**.$v = e$ are also generated in the case of **super**.v *??*= e.

For any other valid operator *op*, a compound assignment of the form v *op*= e is equivalent to $v = v$ *op* e. A compound assignment of the form $C.v$ *op*= e is equivalent to $C.v = C.v$ *op* e.

Evaluation of a compound assignment a of the form $e_1.v$ *op*= e_2 proceeds as follows: Evaluate e_1 to an object u and let x be a fresh variable bound to u. Evaluate $x.v = x.v$ *op* e_2 to an object r and then a evaluates to r.

Evaluation of s compound assignment a of the form $e_1[e_2]$ *op*= e_3 proceeds as follows: Evaluate e_1 to an object u and evaluate e_2 to an object v. Let a and i be fresh variables bound to u and v respectively. Evaluate $a[i] = a[i]$ *op* e_3 to an object r, and then a evaluates to r.

Evaluation of a compound assignment a of the form $e_1?.v$ *op* $= e_2$ proceeds as follows:

Evaluate e_1 to an object u. If u is the null object, then a evaluates to the null object (16.2). Otherwise let x be a fresh variable bound to u. Evaluate $x.v$ *op*= e_2 to an object r. Then a evaluates to r.

The static type of $e_1?.v$ *op*= e_2 is the static type of $e_1.v$ *op* e_2. Exactly the same static warnings that would be caused by $e_1.v$ *op*= e_2 are also generated in the case of $e_1?.v$ *op*= e_2.

A compound assignment of the form $C?.v$ *op* $= e_2$ is equivalent to the expression $C.v$ *op* $= e_2$.

compoundAssignmentOperator:
‘*=’ |
‘/=’ |
‘~/=’ |
‘%=’ |
‘+=’ |
‘-=’ |
‘<<=’ |

```
  '>>=' |
  '>>>=' |
  '&=' |
  '^=' |
  '|=' |
  '??='
;
```

16.21 Conditional

A *conditional expression* evaluates one of two expressions based on a boolean condition.

```
conditionalExpression:
  ifNullExpression
      ('?' expressionWithoutCascade ':' expressionWithoutCascade)?
;
```

Evaluation of a conditional expression c of the form $e_1?e_2 : e_3$ proceeds as follows:

First, e_1 is evaluated to an object o_1. Then, o_1 is subjected to boolean conversion (16.4.1) producing an object r. If r is **true**, then the value of c is the result of evaluating the expression e_2. Otherwise the value of c is the result of evaluating the expression e_3.

If all of the following hold:

- e_1 shows that a variable v has type T.
- v is not potentially mutated in e_2 or within a function.
- If the variable v is accessed by a function in e_2 then the variable v is not potentially mutated anywhere in the scope of v.

then the type of v is known to be T in e_2.

It is a static type warning if the static type of e_1 may not be assigned to bool. The static type of c is the least upper bound (19.8.2) of the static type of e_2 and the static type of e_3.

16.22 If-null Expressions

An *if-null expression* evaluates an expression and if the result is the null object (16.2), evaluates another.

```
ifNullExpression:
  logicalOrExpression ('??' logicalOrExpression)*
;
```

Evaluation of an if-null expression e of the form e_1 ?? e_2 proceeds as follows:

Evaluate e_1 to an object o. If o is not the null object (16.2), then e evaluates to o. Otherwise evaluate e_2 to an object r, and then e evaluates to r.

The static type of e is the least upper bound (19.8.2) of the static type of e_1 and the static type of e_2.

16.23 Logical Boolean Expressions

The logical boolean expressions combine boolean objects using the boolean conjunction and disjunction operators.

logicalOrExpression:
  logicalAndExpression ('||' logicalAndExpression)*
;

logicalAndExpression:
  equalityExpression ('&&' equalityExpression)*
;

A *logical boolean expression* is either an equality expression (16.24), or an invocation of a logical boolean operator on an expression e_1 with argument e_2.

Evaluation of a logical boolean expression b of the form $e_1||e_2$ causes the evaluation of e_1 which is then subjected to boolean conversion, producing an object o_1; if o_1 is **true**, the result of evaluating b is **true**, otherwise e_2 is evaluated to an object o_2, which is then subjected to boolean conversion (16.4.1) producing an object r, which is the value of b.

Evaluation of a logical boolean expression b of the form $e_1\&\&e_2$ causes the evaluation of e_1 which is then subjected to boolean conversion, producing an object o_1; if o_1 is not **true**, the result of evaluating b is **false**, otherwise e_2 is evaluated to an object o_2, which is then subjected to boolean conversion evaluating to an object r, which is the value of b.

A logical boolean expression b of the form $e_1\&\&e_2$ shows that a variable v has type T if all of the following conditions hold:

- Either e_1 shows that v has type T or e_2 shows that v has type T.
- v is a local variable or formal parameter.
- The variable v is not mutated in e_2 or within a function.

Furthermore, if all of the following hold:

- e_1 shows that v has type T.
- v is not mutated in either e_1, e_2 or within a function.
- If the variable v is accessed by a function in e_2 then the variable v is not potentially mutated anywhere in the scope of v.

then the type of v is known to be T in e_2.

It is a static warning if the static type of e_1 may not be assigned to bool or if the static type of e_2 may not be assigned to bool. The static type of a logical boolean expression is bool.

16.24 Equality

Equality expressions test objects for equality.

equalityExpression:
  relationalExpression (equalityOperator relationalExpression)? |
  super equalityOperator relationalExpression
;

equalityOperator:
  '==' |
  '!='
;

An *equality expression* is either a relational expression (16.25), or an invocation of an equality operator on either **super** or an expression e_1, with argument e_2.

Evaluation of an equality expression ee of the form $e_1 == e_2$ proceeds as follows:

- The expression e_1 is evaluated to an object o_1.
- The expression e_2 is evaluated to an object o_2.
- If either o_1 or o_2 is the null object (16.2), then ee evaluates to **true** if both o_1 and o_2 are the null object and to **false** otherwise. Otherwise,
- evaluation of ee is equivalent to the method invocation $o_1.==(o_2)$.

Evaluation of an equality expression ee of the form **super** $== e$ proceeds as follows:

- The expression e is evaluated to an object o.
- If either **this** or o is the null object (16.2), then ee evaluates to evaluates to **true** if both **this** and o are the null object and to **false** otherwise. Otherwise,
- evaluation of ee is equivalent to the method invocation **super**.==(o).

As a result of the above definition, user defined == methods can assume that their argument is non-null, and avoid the standard boiler-plate prelude:

```
if (identical(null, arg)) return false;
```

Another implication is that there is never a need to use identical() to test against **null**, nor should anyone ever worry about whether to write **null** == e or e == **null**.

An equality expression of the form e_1 != e_2 is equivalent to the expression !(e_1 == e_2). An equality expression of the form **super** != e is equivalent to the expression !(**super** == e).

The static type of an equality expression is bool.

16.25 Relational Expressions

Relational expressions invoke the relational operators on objects.

relationalExpression:
  bitwiseOrExpression (typeTest | typeCast |
      relationalOperator bitwiseOrExpression)? |
  super relationalOperator bitwiseOrExpression
 ;

relationalOperator:
  '>=' |
  '>' |
  '<=' |
  '<'
 ;

A *relational expression* is either a bitwise expression (16.26), or an invocation of a relational operator on either **super** or an expression e_1, with argument e_2.

A relational expression of the form e_1 *op* e_2 is equivalent to the method invocation e_1.*op*(e_2). A relational expression of the form **super** *op* e_2 is equivalent to the method invocation **super**.*op*(e_2).

16.26 Bitwise Expressions

Bitwise expressions invoke the bitwise operators on objects.

bitwiseOrExpression:
  bitwiseXorExpression ('|' bitwiseXorExpression)* |
  super ('|' bitwiseXorExpression)+
 ;

bitwiseXorExpression:
  bitwiseAndExpression ('^' bitwiseAndExpression)* |
  super ('^' bitwiseAndExpression)+
 ;

bitwiseAndExpression:
shiftExpression (‘&’ shiftExpression)* |
super (‘&’ shiftExpression)+
;

bitwiseOperator:
‘&’ |
‘^’ |
‘|’
;

A *bitwise expression* is either a shift expression (16.27), or an invocation of a bitwise operator on either **super** or an expression e_1, with argument e_2.

A bitwise expression of the form e_1 *op* e_2 is equivalent to the method invocation $e_1.op(e_2)$. A bitwise expression of the form **super** *op* e_2 is equivalent to the method invocation **super**.$op(e_2)$.

It should be obvious that the static type rules for these expressions are defined by the equivalence above - ergo, by the type rules for method invocation and the signatures of the operators on the type e_1. The same holds in similar situations throughout this specification.

16.27 Shift

Shift expressions invoke the shift operators on objects.

shiftExpression:
additiveExpression (shiftOperator additiveExpression)* |
super (shiftOperator additiveExpression)+
;

shiftOperator:
‘<<’ |
‘>>’ |
‘>>>’
;

A *shift expression* is either an additive expression (16.28), or an invocation of a shift operator on either **super** or an expression e_1, with argument e_2.

A shift expression of the form e_1 *op* e_2 is equivalent to the method invocation $e_1.op(e_2)$. A shift expression of the form **super** *op* e_2 is equivalent to the method invocation **super**.$op(e_2)$.

Note that this definition implies left-to-right evaluation order among shift expressions:

$e_1 << e_2 << e_3$

is evaluated as $(e_1 << e_2).<<(e_3)$ which is equivalent to $(e_1 << e_2) << e_3$. The same holds for additive and multiplicative expressions.

16.28 Additive Expressions

Additive expressions invoke the addition operators on objects.

additiveExpression:
  multiplicativeExpression
      (additiveOperator multiplicativeExpression)* |
  super (additiveOperator multiplicativeExpression)+
;

additiveOperator:
  '+' |
  '-'
;

An *additive expression* is either a multiplicative expression (16.29), or an invocation of an additive operator on either **super** or an expression e_1, with argument e_2.

An additive expression of the form e_1 *op* e_2 is equivalent to the method invocation $e_1.op(e_2)$. An additive expression of the form **super** *op* e_2 is equivalent to the method invocation **super**$.op(e_2)$.

The static type of an additive expression is usually determined by the signature given in the declaration of the operator used. However, invocations of the operators + and - of class int are treated specially by the typechecker. The static type of an expression $e_1 + e_2$ where e_1 has static type int is int if the static type of e_2 is int, and double if the static type of e_2 is double. The static type of an expression $e_1 - e_2$ where e_1 has static type int is int if the static type of e_2 is int, and double if the static type of e_2 is double.

16.29 Multiplicative Expressions

Multiplicative expressions invoke the multiplication operators on objects.

multiplicativeExpression:
  unaryExpression (multiplicativeOperator unaryExpression)* |
  super (multiplicativeOperator unaryExpression)+
;

multiplicativeOperator:
  '*' |
  '/' |
  '%' |

```
  '~/'
;
```

A *multiplicative expression* is either a unary expression (16.30), or an invocation of a multiplicative operator on either **super** or an expression e_1, with argument e_2.

A multiplicative expression of the form e_1 *op* e_2 is equivalent to the method invocation $e_1.op(e_2)$. A multiplicative expression of the form **super** *op* e_2 is equivalent to the method invocation **super**.$op(e_2)$.

The static type of an multiplicative expression is usually determined by the signature given in the declaration of the operator used. However, invocations of the operators ***** and **%** of class int are treated specially by the typechecker. The static type of an expression $e_1 * e_2$ where e_1 has static type int is int if the static type of e_2 is int, and double if the static type of e_2 is double. The static type of an expression $e_1 \% e_2$ where e_1 has static type int is int if the static type of e_2 is int, and double if the static type of e_2 is double.

16.30 Unary Expressions

Unary expressions invoke unary operators on objects.

```
unaryExpression:
  prefixOperator unaryExpression |
  awaitExpression |
  postfixExpression |
  (minusOperator | tildeOperator) super |
  incrementOperator assignableExpression
;

prefixOperator:
  minusOperator |
  negationOperator |
  tildeOperator
;

minusOperator:
  '-'
;

negationOperator:
  '!'
;

tildeOperator:
  '~'
```

;

A *unary expression* is either a postfix expression (16.32), an await expression (16.31) or an invocation of a prefix operator on an expression or an invocation of a unary operator on either **super** or an expression e.

The expression $!e$ is equivalent to the expression e ? **false** : **true**.

Evaluation of an expression of the form $++e$ is equivalent to e += 1. Evaluation of an expression of the form $--e$ is equivalent to e -= 1.

If e is an expression is of the form -l where l is an integer literal (16.3) with numerical integer value i, then it is a compile-time error if $i > 2^{63}$. Otherwise, when $0 \leq i \leq 2^{63}$, the static type of e is int and e evaluates to an instance of the class int representing the integer value $-i$, and it is a compile-time error if the integer $-i$ cannot be represented exactly by an instance of int. *This treats -l where l is an integer literal as an atomic signed numeral. It does not* evaluate *l as an individual expression because -9223372036854775808 should represent a valid* int *even if 9223372036854775808 does not.*

Any other expression of the form *op* e is equivalent to the method invocation e.*op*(). An expression of the form *op* **super** is equivalent to the method invocation (16.18.3) **super**.*op*().

16.31 Await Expressions

An *await expression* allows code to yield control until an asynchronous operation (9) completes.

awaitExpression:
  await unaryExpression
;

Evaluation of an await expression a of the form **await** e proceeds as follows: First, the expression e is evaluated to an object o.

Then, if o is not an instance of Future, then let f be the result of creating a new object using the constructor Future.value() with o as its argument; otherwise let f be o.

Next, the stream associated with the innermost enclosing asynchronous for loop (17.6.3), if any, is paused. The current invocation of the function body immediately enclosing a is suspended until after f completes. At some time after f is completed, control returns to the current invocation. If f has completed with an error x and stack trace t, a throws x and t (16). If f completes with a value v, a evaluates to v.

It is a compile-time error if the function immediately enclosing a is not declared asynchronous. However, this error is simply a syntax error, because in the context of a normal function, **await** has no special meaning.

An await expression has no meaning in a synchronous function. If such a function were to suspend waiting for a future, it would no longer be synchronous.

It is not a static warning if the type of e is not a subtype of Future. Tools may choose to give a hint in such cases.

The static type of a is $flatten(T)$ (the $flatten$ function is defined in section 16.10) where T is the static type of e.

16.32 Postfix Expressions

Postfix expressions invoke the postfix operators on objects.

postfixExpression:
  assignableExpression postfixOperator |
  primary selector*
;

postfixOperator:
  incrementOperator
;

selector:
  assignableSelector |
  argumentPart
;

incrementOperator:
  '++' |
  '--'
;

A *postfix expression* is either a primary expression, a function, method or getter invocation, or an invocation of a postfix operator on an expression e.

Evaluation of a postfix expression e of the form $v{+}{+}$, where v is an identifier, proceeds as follows:

Evaluate v to an object r and let y be a fresh variable bound to r. Evaluate $v = y + 1$. Then e evaluates to r.

The static type of such an expression is the static type of v.

The above ensures that if v is a variable, the getter gets called exactly once. Likewise in the cases below.

Evaluation of a postfix expression e of the form $C.v{+}{+}$ proceeds as follows:

Evaluate $C.v$ to a value r and let y be a fresh variable bound to r. Evaluate $C.v = y + 1$. Then e evaluates to r.

The static type of such an expression is the static type of $C.v$.

Evaluation of a postfix expression e of the form $e_1.v{+}{+}$ proceeds as follows:

Evaluate e_1 to an object u and let x be a fresh variable bound to u. Evaluate $x.v$ to a value r and let y be a fresh variable bound to r. Evaluate $x.v = y + 1$. Then e evaluates to r.

The static type of such an expression is the static type of $e_1.v$.

Evaluation of a postfix expression e of the form $e_1[e_2]{+}{+}$ proceeds as follows:

Evaluate e_1 to an object u and e_2 to an object v. Let a and i be fresh variables bound to u and v respectively. Evaluate $a[i]$ to an object r and let y be a fresh variable bound to r. Evaluate $a[i] = y + 1$. Then e evaluates to r.

The static type of such an expression is the static type of $e_1[e_2]$.

Evaluation of a postfix expression e of the form v--, where v is an identifier, proceeds as follows:

Evaluate the expression v to an object r and let y be a fresh variable bound to r. Evaluate $v = y$ - 1. Then e evaluates to r.

The static type of such an expression is the static type of v.

Evaluation of a postfix expression e of the form $C.v$-- proceeds as follows:

Evaluate $C.v$ to a value r and let y be a fresh variable bound to r. Evaluate $C.v = y$ - 1. Then e evaluates to r.

The static type of such an expression is the static type of $C.v$.

Evaluation of a postfix expression of the form $e_1.v$-- proceeds as follows:

Evaluate e_1 to an object u and let x be a fresh variable bound to u. Evaluate $x.v$ to a value r and let y be a fresh variable bound to r. Evaluate $x.v = y$ - 1. Then e evaluates to r.

The static type of such an expression is the static type of $e_1.v$.

Evaluation of a postfix expression e of the form $e_1[e_2]$-- proceeds as follows:

Evaluate e_1 to an object u and e_2 to an object v. Let a and i be fresh variables bound to u and v respectively. Evaluate $a[i]$ to an object r and let y be a fresh variable bound to r. Evaluate $a[i] = y$ - 1. Then e evaluates to r.

The static type of such an expression is the static type of $e_1[e_2]$.

Evaluation of a postfix expression e of the form $e_1?.v{+}{+}$ proceeds as follows:

If e_1 is a type literal, evaluation of e is equivalent to evaluation of $e_1.v{+}{+}$.

Otherwise evaluate e_1 to an object u. if u is the null object, e evaluates to the null object (16.2). Otherwise let x be a fresh variable bound to u. Evaluate $x.v{+}{+}$ to an object o. Then e evaluates to o.

The static type of such an expression is the static type of $e_1.v$.

Evaluation of a postfix expression e of the form $e_1?.v$-- proceeds as follows:

If e_1 is a type literal, evaluation of e is equivalent to evaluation of $e_1.v$--.

Otherwise evaluate e_1 to an object u. If u is the null object, e evaluates to the null object (16.2). Otherwise let x be a fresh variable bound to u. Evaluate $x.v$-- to an object o. Then e evaluates to o.

The static type of such an expression is the static type of $e_1.v$.

16.33 Assignable Expressions

Assignable expressions are expressions that can appear on the left hand

side of an assignment. This section describes how to evaluate these expressions when they do not constitute the complete left hand side of an assignment.

Of course, if assignable expressions always appeared as *the left hand side, one would have no need for their value, and the rules for evaluating them would be unnecessary. However, assignable expressions can be subexpressions of other expressions and therefore must be evaluated.*

assignableExpression:
primary (argumentPart* assignableSelector)+ |
super unconditionalAssignableSelector |
identifier
;

unconditionalAssignableSelector:
'[' expression ']' |
'.' identifier
;

assignableSelector:
unconditionalAssignableSelector |
'?.' identifier
;

An *assignable expression* is either:

- An identifier.
- An invocation (possibly conditional) of a getter (10.2) or list access operator on an expression e.
- An invocation of a getter or list access operator on **super**.

An assignable expression of the form *id* is evaluated as an identifier expression (16.34).

An assignable expression of the form $e.id$ or $e?.id$ is evaluated as a property extraction (16.19).

An assignable expression of the form $e_1[e_2]$ is evaluated as a method invocation of the operator method [] on e_1 with argument e_2.

An assignable expression of the form **super**.*id* is evaluated as a property extraction.

Evaluation of an assignable expression of the form **super**$[e_2]$ is equivalent to evaluation of the method invocation **super**.[](e_2).

16.34 Identifier Reference

An *identifier expression* consists of a single identifier; it provides access to an object via an unqualified name.

identifier:
IDENTIFIER
;

IDENTIFIER_NO_DOLLAR:
IDENTIFIER_START_NO_DOLLAR
IDENTIFIER_PART_NO_DOLLAR*
;

IDENTIFIER:
IDENTIFIER_START IDENTIFIER_PART*
;

BUILT_IN_IDENTIFIER:
abstract |
as |
covariant |
deferred |
dynamic |
export |
external |
factory |
Function |
get |
implements |
import |
interface |
library |
operator |
mixin |
part |
set |
static |
typedef
;

IDENTIFIER_START:
IDENTIFIER_START_NO_DOLLAR |
'$'
;

IDENTIFIER_START_NO_DOLLAR:
LETTER |
'_'

;

IDENTIFIER_PART_NO_DOLLAR:
IDENTIFIER_START_NO_DOLLAR |
DIGIT
;

IDENTIFIER_PART:
IDENTIFIER_START |
DIGIT
;

qualified:
identifier ('.' identifier)?
;

A built-in identifier is one of the identifiers produced by the production *BUILT_IN_IDENTIFIER*. It is a compile-time error if a built-in identifier is used as the declared name of a prefix, class, type parameter or type alias. It is a compile-time error to use a built-in identifier other than **dynamic** in a type annotation or type parameter.

Built-in identifiers are identifiers that are used as keywords in Dart, but are not reserved words in Javascript. To minimize incompatibilities when porting Javascript code to Dart, we do not make these into reserved words. A built-in identifier may not be used to name a class or type. In other words, they are treated as reserved words when used as types. This eliminates many confusing situations without causing compatibility problems. After all, a Javascript program has no type declarations or annotations so no clash can occur. Furthermore, types should begin with an uppercase letter (see the appendix) and so no clash should occur in any Dart user program anyway.

It is a compile-time error if either of the identifiers **await** or **yield** is used as an identifier in a function body marked with either **async**, **async*** or **sync***.

For compatibility reasons, new constructs cannot rely upon new reserved words or even built-in identifiers. However, the constructs above are only usable in contexts that require special markers introduced concurrently with these constructs, so no old code could use them. Hence the restriction, which treats these names as reserved words in a limited context.

Evaluation of an identifier expression e of the form id proceeds as follows:

Let d be the innermost declaration in the enclosing lexical scope whose name is id or $id=$. If no such declaration exists in the lexical scope, let d be the declaration of the inherited member named id if it exists.

- if d is a prefix p, a compile-time error occurs unless the token immediately following d is '.'.

- If d is a class or type alias T, the value of e is an instance of class Type (or a subclass thereof) reifying T.
- If d is a type parameter T, then the value of e is the value of the actual type argument corresponding to T that was passed to the generative constructor that created the current binding of **this**. If, however, e occurs inside a static member, a compile-time error occurs.
- If d is a constant variable of one of the forms **const** $v = e$; or **const** T $v = e$; then the value id is the value of the compile-time constant e.
- If d is a local variable or formal parameter then e evaluates to the current binding of id.
- If d is a static method, top-level function or local function then e evaluates to the function object obtained by closurization (16.15) of the declaration denoted by d.
- If d is the declaration of a static variable, static getter or static setter declared in class C, then evaluation of e is equivalent to evaluation of the property extraction (16.19) $C.id$.
- If d is the declaration of a library variable, top-level getter or top-level setter, then evaluation of e is equivalent to evaluation of the top level getter invocation (16.17) id.
- Otherwise, if e occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer, evaluation of e causes a NoSuchMethod to be thrown.
- Otherwise, evaluation of e is equivalent to evaluation of the property extraction (16.19) **this**.id.

The static type of e is determined as follows:

- If d is a class, type alias or type parameter the static type of e is Type.
- If d is a local variable or formal parameter the static type of e is the type of the variable id, unless id is known to have some type T, in which case the static type of e is T, provided that T is more specific than any other type S such that v is known to have type S.
- If d is a static method, top-level function or local function the static type of e is the function type defined by d.
- If d is the declaration of a static variable, static getter or static setter declared in class C, the static type of e is the static type of the getter invocation (16.19) $C.id$.

- If d is the declaration of a library variable, top-level getter or top-level setter, the static type of e is the static type of the top level getter invocation id.
- Otherwise, if e occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer, the static type of e is **dynamic**.
- Otherwise, the static type of e is the type of the property extraction (16.19) **this**.id.

Note that if one declares a setter, we bind to the corresponding getter even if it does not exist.

This prevents situations where one uses uncorrelated setters and getters. The intent is to prevent errors when a getter in a surrounding scope is used accidentally.

It is a static warning if an identifier expression id occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer and there is no declaration d with name id in the lexical scope enclosing the expression.

16.35 Type Test

The *is-expression* tests if an object is a member of a type.

typeTest:
  isOperator type
;

isOperator:
  is '!'?
;

Evaluation of the is-expression e **is** T proceeds as follows:

The expression e is evaluated to a value v. Then, if T is a malformed or deferred type (19.1), a dynamic error occurs. Otherwise, if the interface of the class of v is a subtype of T, the is-expression evaluates to **true**. Otherwise it evaluates to **false**.

It follows that e **is** Object is always true. This makes sense in a language where everything is an object.

Also note that **null is** T is false unless T = Object, T = **dynamic** or T = Null. The former two are useless, as is anything of the form e **is** Object or e **is dynamic**. Users should test for the null object (16.2) directly rather than via type tests.

The is-expression e **is!** T is equivalent to !(e **is** T).

Let v be a local variable or a formal parameter. An is-expression of the

form v **is** T shows that v has type T iff T is more specific than the type S of the expression v and both $T \neq$ **dynamic** and $S \neq$ **dynamic**.

The motivation for the "shows that v has type T" relation is to reduce spurious warnings thereby enabling a more natural coding style. The rules in the current specification are deliberately kept simple. It would be upwardly compatible to refine these rules in the future; such a refinement would accept more code without warning, but not reject any code now warning-free.

The rule only applies to locals and parameters, as instance or static variables could be modified via side-effecting functions or methods that are not accessible to a local analysis.

It is pointless to deduce a weaker type than what is already known. Furthermore, this would lead to a situation where multiple types are associated with a variable at a given point, which complicates the specification. Hence the requirement that $T << S$ (we use $<<$ rather than subtyping because subtyping is not a partial order).

We do not want to refine the type of a variable of type ***dynamic****, as this could lead to more warnings rather than fewer. The opposite requirement, that $T \neq$* ***dynamic*** *is a safeguard lest S ever be $\bot$.*

The static type of an is-expression is bool.

16.36 Type Cast

The *cast expression* ensures that an object is a member of a type.

```
typeCast:
  asOperator type
 ;

asOperator:
  as
 ;
```

Evaluation of the cast expression e **as** T proceeds as follows:

The expression e is evaluated to a value v. Then, if T is a malformed or deferred type (19.1), a dynamic error occurs. Otherwise, if the interface of the class of v is a subtype of T, the cast expression evaluates to v. Otherwise, if v is the null object (16.2), the cast expression evaluates to v. In all other cases, a CastError is thrown.

The static type of a cast expression e **as** T is T.

17 Statements

A *statement* is a fragment of Dart code that can be executed at run time. Statements, unlike expressions, do not evaluate to a value, but are instead executed for their effect on the program state and control flow.

Execution of a statement *completes* in one of five ways: either it *completes normally*, it *breaks* or it *continues* (either to a label or without a label), it *returns* (with or without a value), or it *throws* an exception object and an associated stack trace.

In the description of statement execution, the default is that the execution completes normally unless otherwise stated.

If the execution of a statement, s, is defined in terms of executing another statement, and the execution of that other statement does not complete normally, then, unless otherwise stated, the execution of s stops at that point and completes in the same way. For example, if execution of the body of a **do** loop returns a value, so does execution of the **do** loop statement itself.

If the execution of a statement is defined in terms of evaluating an expression and the evaluation of that expression throws, then, unless otherwise stated, the execution of the statement stops at that point and throws the same exception object and stack trace. For example, if evaluation of the condition expression of an **if** statement throws, then so does execution of the **if** statement. Likewise, if evaluation of the expression of a **return** statement throws, so does execution of the **return** statement.

statements:
  statement*
 ;

statement:
  label* nonLabelledStatement
 ;

nonLabelledStatement:
  block |
  localVariableDeclaration |
  forStatement |
  whileStatement |
  doStatement |
  switchStatement |
  ifStatement |
  rethrowStatement |
  tryStatement |
  breakStatement |
  continueStatement |
  returnStatement |
  yieldStatement |
  yieldEachStatement |
  expressionStatement |
  assertStatement |

localFunctionDeclaration
;

17.1 Blocks

A *block statement* supports sequencing of code.

Execution of a block statement $\{s_1, \ldots, s_n\}$ proceeds as follows:

For $i \in 1..n$, s_i is executed.

A block statement introduces a new scope, which is nested in the lexically enclosing scope in which the block statement appears.

17.2 Expression Statements

An *expression statement* consists of an expression other than a non-constant map literal (16.8) that has no explicit type arguments.

The restriction on maps is designed to resolve an ambiguity in the grammar, when a statement begins with {.

expressionStatement:
expression? ';'
;

Execution of an expression statement e; proceeds by evaluating e.

It is a compile-time error if a non-constant map literal that has no explicit type arguments appears in a place where a statement is expected.

17.3 Local Variable Declaration

A *variable declaration statement* declares a new local variable.

localVariableDeclaration:
initializedVariableDeclaration ';'
;

Executing a variable declaration statement of one of the forms **var** $v = e$;, T $v = e$;, **const** $v = e$;, **const** T $v = e$;, **final** $v = e$; or **final** T $v = e$; proceeds as follows:

The expression e is evaluated to an object o. Then, the variable v is set to o.

A variable declaration statement of the form **var** v; is equivalent to **var** $v =$ **null**;. A variable declaration statement of the form T v; is equivalent to T $v =$ **null**;.

This holds regardless of the type T. For example, int i; does not cause i to be initialized to zero. Instead, i is initialized to the null object (16.2), just as if we had written **var** i; or Object i; or Collection<String> i;.

To do otherwise would undermine the optionally typed nature of Dart, causing type annotations to modify program behavior.

17.4 Local Function Declaration

A function declaration statement declares a new local function (9.1).

localFunctionDeclaration:
  functionSignature functionBody
;

A function declaration statement of one of the forms *id signature* { *statements* } or *T id signature* { *statements* } causes a new function named *id* to be added to the innermost enclosing scope. It is a compile-time error to reference a local function before its declaration.

This implies that local functions can be directly recursive, but not mutually recursive. Consider these examples:

```
f(x) => x++; // a top level function
top() { // another top level function
  f(3); // illegal
  f(x) => x > 0? x*f(x-1): 1; // recursion is legal
  g1(x) => h(x, 1); // error: h is not declared yet
  h(x, n) => x > 1? h(x-1, n*x): n; // again, recursion is fine
  g2(x) => h(x, 1); // legal
  p1(x) => q(x,x); // illegal
  q1(a, b) => a > 0 ? p1(a-1): b; // fine
  q2(a, b) => a > 0 ? p2(a-1): b; // illegal
  p1(x) => q2(x,x); // fine
}
```

There is no way to write a pair of mutually recursive local functions, because one always has to come before the other is declared. These cases are quite rare, and can always be managed by defining a pair of variables first, then assigning them appropriate function literals:

```
top2() { // a top level function
  var p, q;
  p = (x) => q(x,x);
  q = (a, b) => a > 0 ? p(a-1): b;
}
```

The rules for local functions differ slightly from those for local variables in that a function can be accessed within its declaration but a variable can only be accessed after its declaration. This is because recursive functions are useful whereas recursively defined variables are almost always errors. It therefore makes sense to harmonize the rules for local functions with those for functions in general rather than with the rules for local variables.

17.5 If

The *if statement* allows for conditional execution of statements.

ifStatement:
  if '(' expression ')' statement (**else** statement)?
;

An if statement of the form **if** (e) s_1 **else** s_2 where s_1 is not a block statement is equivalent to the statement **if** (e) $\{s_1\}$ **else** s_2. An if statement of the form **if** (e) s_1 **else** s_2 where s_2 is not a block statement is equivalent to the statement **if** (e) s_1 **else** $\{s_2\}$.

The reason for this equivalence is to catch errors such as

```
void main() {
 if (somePredicate)
   var v = 2;
 print(v);
}
```

Under reasonable scope rules such code is problematic. If we assume that v *is declared in the scope of the method* main(), *then when* somePredicate *is false,* v *will be uninitialized when accessed. The cleanest approach would be to require a block following the test, rather than an arbitrary statement. However, this goes against long standing custom, undermining Dart's goal of familiarity. Instead, we choose to insert a block, introducing a scope, around the statement following the predicate (and similarly for* ***else*** *and loops). This will cause both a warning and a run-time error in the case above. Of course, if there is a declaration of* v *in the surrounding scope, programmers might still be surprised. We expect tools to highlight cases of shadowing to help avoid such situations.*

Execution of an if statement of the form **if** (b) s_1 **else** s_2 where s_1 and s_2 are block statements, proceeds as follows:

First, the expression b is evaluated to an object o. Then, o is subjected to boolean conversion (16.4.1), producing an object r. If r is **true**, then the block statement s_1 is executed, otherwise the block statement s_2 is executed.

It is a static type warning if the type of the expression b may not be assigned to bool.

If:

- b shows that a variable v has type T.
- v is not potentially mutated in s_1 or within a function.
- If the variable v is accessed by a function in s_1 then the variable v is not potentially mutated anywhere in the scope of v.

then the type of v is known to be T in s_1.

An if statement of the form **if** (e) s is equivalent to the if statement **if** (e) s **else** $\{\}$.

17.6 For

The *for statement* supports iteration.

forStatement:
  await? **for** '(' forLoopParts ')' statement
;

forLoopParts:
  forInitializerStatement expression? ';' expressionList? |
  declaredIdentifier **in** expression |
  identifier **in** expression
;

forInitializerStatement:
  localVariableDeclaration |
  expression? ';'
;

The for statement has three forms - the traditional for loop and two forms of the for-in statement - synchronous and asynchronous.

17.6.1 For Loop

Execution of a for statement of the form **for** (**var** $v = e_0$; c; e) s proceeds as follows:

If c is empty then let c' be **true** otherwise let c' be c.

First the variable declaration statement **var** $v = e_0$ is executed. Then:

1. If this is the first iteration of the for loop, let v' be v. Otherwise, let v' be the variable v'' created in the previous execution of step 3.
2. The expression $[v'/v]c$ is evaluated and subjected to boolean conversion (16.4). If the result is **false**, the for loop completes normally. Otherwise, execution continues at step 3.
3. The statement $[v'/v]\{s\}$ is executed.

   If this execution completes normally, continues without a label, or continues to a label (17.13) that prefixes this **for** statement (17), then execution of the statement is treated as if it had completed normally.

   Let v'' be a fresh variable. v'' is bound to the value of v'.
4. The expression $[v''/v]e$ is evaluated, and the process recurses at step 1.

The definition above is intended to prevent the common error where users create a function object inside a for loop, intending to close over the current

binding of the loop variable, and find (usually after a painful process of debugging and learning) that all the created function objects have captured the same value—the one current in the last iteration executed.

Instead, each iteration has its own distinct variable. The first iteration uses the variable created by the initial declaration. The expression executed at the end of each iteration uses a fresh variable v'', bound to the value of the current iteration variable, and then modifies v'' as required for the next iteration.

It is a static warning if the static type of c may not be assigned to bool.

17.6.2 For-in

Let D be derived from finalConstVarOrType? and let $n0$ be an identifier that does not occur anywhere in the program. A for statement of the form **for** (D *id* **in** e) s is equivalent to the following code:

```
var n0 = e.iterator;
while (n0.moveNext()) {
  D id = n0.current;
  s
}
```

For purposes of static typechecking, this code is checked under the assumption that $n0$ is declared to be of type T, where T is the static type of e.iterator.

It follows that it is a static warning if D is empty and *id* is a final variable, and a dynamic error will then occur if the body is executed.

17.6.3 Asynchronous For-in

A for-in statement may be asynchronous. The asynchronous form is designed to iterate over streams. An asynchronous for loop is distinguished by the keyword **await** immediately preceding the keyword **for**.

Let D be derived from finalConstVarOrType?. Execution of a for-in statement, f, of the form **await for** (D *id* **in** e) s proceeds as follows:

The expression e is evaluated to an object o. It is a dynamic error if o is not an instance of a class that implements Stream. It is a static warning if D is empty and *id* is a final variable, and it is then a dynamic error if the body is executed.

The stream associated with the innermost enclosing asynchronous for loop, if any, is paused. The stream o is listened to, producing a stream subscription u, and execution of the asynchronous for-in loop is suspended until a stream event is available. This allows other asynchronous events to execute while this loop is waiting for stream events.

Pausing an asynchronous for loop means pausing the associated stream subscription. A stream subscription is paused by calling its pause method. If the subscription is already paused, an implementation may omit further calls to pause.

The pause call can throw, although that should never happen for a correctly implemented stream.

For each *data event* from u, the statement s is executed with id bound to the value of the current data event.

Either execution of s is completely synchronous, or it contains an asynchronous construct (**await**, **await for**, **yield** or **yield***) which will pause the stream subscription of its surrounding asynchronous loop. This ensures that no other event of u occurs before execution of s is complete, if o is a correctly implemented stream. If o doesn't act as a valid stream, for example by not respecting pause requests, the behavior of the asynchronous loop may become unpredictable.

If execution of s continues without a label, or to a label (17.13) that prefixes the asynchronous for statement (17), then the execution of s is treated as if it had completed normally.

If execution of s otherwise does not complete normally, the subscription u is canceled by evaluating **await** v.cancel() where v is a fresh variable referencing the stream subscription u. If that evaluation throws, execution of f throws the same exception and stack trace. Otherwise execution of f completes in the same way as the execution of s. Otherwise the execution of f is suspended again, waiting for the next stream subscription event, and u is resumed if it has been paused. The resume call can throw, in which case the asynchronous for loop also throws. That should never happen for a correctly implemented stream.

On an *error event* from u, with error object e and stack trace st, the subscription u is canceled by evaluating **await** v.cancel() where v is a fresh variable referencing the stream subscription u. If that evaluation throws, execution of f throws the same exception object and stack trace. Otherwise execution of f throws with e as exception object and st as stack trace.

When u is done, execution of f completes normally.

It is a compile-time error if an asynchronous for-in statement appears inside a synchronous function (9). It is a compile-time error if a traditional for loop (17.6.1) is prefixed by the **await** keyword.

An asynchronous loop would make no sense within a synchronous function, for the same reasons that an await expression makes no sense in a synchronous function.

17.7 While

The while statement supports conditional iteration, where the condition is evaluated prior to the loop.

whileStatement:
  while '(' expression ')' statement
;

Execution of a while statement of the form **while** (e) s; proceeds as follows:

The expression e is evaluated to an object o. Then, o is subjected to boolean conversion (16.4.1), producing an object r.

If r is **false**, then execution of the while statement completes normally (17).

Otherwise r is **true** and then the statement $\{s\}$ is executed. If that execution completes normally or it continues with no label or to a label (17.13) that prefixes the **while** statement (17), then the while statement is re-executed. If the execution breaks without a label, execution of the while statement completes normally. If the execution breaks with a label that prefixes the **while** statement, it does end execution of the loop, but the break itself is handled by the surrounding labeled statement (17.13).

It is a static type warning if the static type of e may not be assigned to bool.

17.8 Do

The do statement supports conditional iteration, where the condition is evaluated after the loop.

doStatement:
  do statement **while** '(' expression ')' ';'
;

Execution of a do statement of the form **do** s **while** (e); proceeds as follows:

The statement $\{s\}$ is executed. If that execution continues with no label, or to a label (17.13) that prefixes the do statement (17), then the execution of s is treated as if it had completed normally.

Then, the expression e is evaluated to an object o. Then, o is subjected to boolean conversion (16.4.1), producing an object r. If r is **false**, execution of the do statement completes normally (17). If r is **true**, then the do statement is re-executed.

It is a static type warning if the static type of e may not be assigned to bool.

17.9 Switch

The *switch statement* supports dispatching control among a large number of cases.

switchStatement:
  switch '(' expression ')' '{' switchCase* defaultCase? '}'
;

switchCase:
  label* **case** expression ':' statements
;

defaultCase:
  label* **default** ':' statements

;

Given a switch statement of the form
switch (e) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
  $label_{(n+1)1}$ ... $label_{(n+1)j_{n+1}}$ **default**: s_{n+1}
}
or the form
switch (e) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
}
it is a compile-time error if the expressions e_k are not compile-time constants for all $k \in 1..n$. It is a compile-time error if the values of the expressions e_k are not either:

- instances of the same class C, for all $k \in 1..n$, or
- instances of a class that implements int, for all $k \in 1..n$, or
- instances of a class that implements String, for all $k \in 1..n$.

In other words, all the expressions in the cases evaluate to constants of the exact same user defined class or are of certain known types. Note that the values of the expressions are known at compile time, and are independent of any static type annotations.

It is a compile-time error if the class C has an implementation of the operator == other than the one inherited from Object unless the value of the expression is a string, an integer, literal symbol or the result of invoking a constant constructor of class Symbol.

The prohibition on user defined equality allows us to implement the switch efficiently for user defined types. We could formulate matching in terms of identity instead with the same efficiency. However, if a type defines an equality operator, programmers would find it quite surprising that equal objects did not match.

The **switch** statement should only be used in very limited situations (e.g., interpreters or scanners).

Execution of a switch statement of the form
switch (e) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
  $label_{(n+1)1}$ ... $label_{(n+1)j_{n+1}}$ **default**: s_{n+1}

}
or the form
switch (e) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
}
proceeds as follows:

The statement **var** $id = e$; is evaluated, where id is a fresh variable. In checked mode, it is a run-time error if the value of e is not an instance of the same class as the constants $e_1, \ldots, e_n$.

Note that if there are no case clauses ($n = 0$), the type of e does not matter.

Next, the case clause **case** e_1: s_1 is matched against id, if $n > 0$. Otherwise if there is a **default** clause, the case statements s_{n+1} are executed (17.9.1).

Matching of a **case** clause **case** e_k : s_k of a switch statement
switch (e) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
  $label_{(n+1)1}$ ... $label_{(n+1)j_{n+1}}$ **default**: s_{n+1}
}
against the value of a variable id proceeds as follows:

The expression $e_k == id$ is evaluated to an object o which is then subjected to boolean conversion evaluating to a value v. If v is not **true** the following case, **case** e_{k+1} : s_{k+1} is matched against id if $k < n$. If $k = n$, then the **default** clause's statements are executed (17.9.1). If v is **true**, let h be the smallest number such that $h \geq k$ and s_h is non-empty. If no such h exists, let $h = n + 1$. The case statements s_h are then executed (17.9.1).

Matching of a **case** clause **case** e_k : s_k of a switch statement
switch (e) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
}
against the value of a variable id proceeds as follows:

The expression $e_k == id$ is evaluated to an object o which is then subjected to boolean conversion evaluating to a value v. If v is not **true** the following case, **case** e_{k+1} : s_{k+1} is matched against id if $k < n$. If v is **true**, let h be the smallest integer such that $h \geq k$ and s_h is non-empty. If such a h exists, the case statements s_h are executed (17.9.1). Otherwise the switch statement completes normally (17).

It is a static warning if the type of e may not be assigned to the type of e_k. Let s be the last statement of the statement sequence s_k. If s is a non-empty block statement, let s instead be the last statement of the block statement. It is a static warning s is not a **break**, **continue**, **rethrow** or **return** statement or an expression statement where the expression is a **throw** expression.

The behavior of switch cases intentionally differs from the C tradition. Implicit fall through is a known cause of programming errors and therefore disallowed. Why not simply break the flow implicitly at the end of every case, rather than requiring explicit code to do so? This would indeed be cleaner. It would also be cleaner to insist that each case have a single (possibly compound) statement. We have chosen not to do so in order to facilitate porting of switch statements from other languages. Implicitly breaking the control flow at the end of a case would silently alter the meaning of ported code that relied on fall-through, potentially forcing the programmer to deal with subtle bugs. Our design ensures that the difference is immediately brought to the coder's attention. The programmer will be notified at compile time if they forget to end a case with a statement that terminates the straight-line control flow. We could make this warning a compile-time error, but refrain from doing so because do not wish to force the programmer to deal with this issue immediately while porting code. If developers ignore the warning and run their code, a run-time error will prevent the program from misbehaving in hard-to-debug ways (at least with respect to this issue).

The sophistication of the analysis of fall-through is another issue. For now, we have opted for a very straightforward syntactic requirement. There are obviously situations where code does not fall through, and yet does not conform to these simple rules, e.g.:

switch (x) {
  case 1: **try** { ... **return**; } **finally** { ... **return**; }
}

Very elaborate code in a case clause is probably bad style in any case, and such code can always be refactored.

It is a static warning if all of the following conditions hold:

- The switch statement does not have a default clause.
- The static type of e is an enumerated typed with elements $id_1, \ldots, id_n$.
- The sets $\{e_1, \ldots, e_k\}$ and $\{id_1, \ldots, id_n\}$ are not the same.

In other words, a warning will be issued if a switch statement over an enum is not exhaustive.

17.9.1 Switch case statements

Execution of the case statements s_h of a switch statement

switch (e) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
}

or a switch statement

switch (e) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1

...
$label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
$label_{(n+1)1}$ ... $label_{(n+1)j_{n+1}}$ **default**: s_{n+1}
}
proceeds as follows:

Execute $\{s_h\}$. If this execution completes normally, and if s_h is not the statements of the last case of the switch ($h = n$ if there is no **default** clause, $h = n + 1$ if there is a **default** clause), then the execution of the switch case throws an error. Otherwise s_h are the last statements of the switch case, and execution of the switch case completes normally.

In other words, there is no implicit fall-through between non-empty cases. The last case in a switch (default or otherwise) can 'fall-through' to the end of the statement.

If execution of $\{s_h\}$ breaks with no label (17), then the execution of the switch statement completes normally.

If execution of $\{s_h\}$ continues to a label (17), and the label is $label_{ij}$, where $1 \leq i \leq n + 1$ if the **switch** statement has a **default**, or $1 \leq i \leq n$ if there is no **default**, and where $1 \leq j \leq j_i$, then let h be the smallest number such that $h \geq i$ and s_h is non-empty. If no such h exists, let $h = n + 1$ if the **switch** statement has a **default**, otherwise let $h = n$. The case statements s_h are then executed (17.9.1).

If execution of $\{s_h\}$ completes in any other way, execution of the **switch** statement completes in the same way.

17.10 Rethrow

The *rethrow statement* is used to re-throw an exception and its associated stack trace.

rethrowStatement:
rethrow ';'
;

Execution of a **rethrow** statement proceeds as follows:

Let f be the immediately enclosing function, and let **on** T **catch** (p_1, p_2) be the immediately enclosing catch clause (17.11).

*A **rethrow** statement always appears inside a **catch** clause, and any **catch** clause is semantically equivalent to some **catch** clause of the form **on** T **catch** (p1, p2). So we can assume that the **rethrow** is enclosed in a **catch** clause of that form.*

The **rethrow** statement *throws* (17) with p_1 as the exception object and p_2 as the stack trace.

It is a compile-time error if a **rethrow** statement is not enclosed within an **on-catch** clause.

17.11 Try

The try statement supports the definition of exception handling code in a structured way.

tryStatement:
  try block (onPart+ finallyPart? | finallyPart)
;

onPart:
  catchPart block |
  on type catchPart? block
;

catchPart:
  catch '(' identifier (',' identifier)? ')'
;

finallyPart:
  finally block
;

A try statement consists of a block statement, followed by at least one of:

1. A set of **on-catch** clauses, each of which specifies (either explicitly or implicitly) the type of exception object to be handled, one or two exception parameters, and a block statement.
2. A **finally** clause, which consists of a block statement.

The syntax is designed to be upward compatible with existing Javascript programs. The ***on*** *clause can be omitted, leaving what looks like a Javascript catch clause.*

A try statement of the form **try** s_1 $on-catch_1 \ldots on-catch_n$; is equivalent to the statement **try** s_1 $on-catch_1 \ldots on-catch_n$ **finally** {}.

An **on-catch** clause of the form **on** T **catch** (p_1) s is equivalent to an **on-catch** clause **on** T **catch** (p_1, p_2) s where p_2 is a fresh identifier.

An **on-catch** clause of the form **on** T s is equivalent to an **on-catch** clause **on** T **catch** (p_1, p_2) s where p_1 and p_2 are fresh identifiers.

An **on-catch** clause of the form **catch** (p) s is equivalent to an **on-catch** clause **on dynamic catch** (p, p_2) s where p_2 is a fresh identifier.

An **on-catch** clause of the form **catch** (p_1, p_2) s is equivalent to an **on-catch** clause **on dynamic catch** (p_1, p_2) s.

An **on-catch** clause of the form **on** T **catch** (p_1, p_2) s introduces a new scope CS in which final local variables specified by p_1 and p_2 are defined. The

statement s is enclosed within CS. The static type of p_1 is T and the static type of p_2 is StackTrace.

Execution of a **try** statement s of the form:

try b
on T_1 **catch** (e_1, t_1) c_1
...
on T_n **catch** (e_n, t_n) c_n
finally f

proceeds as follows:

First b is executed. If execution of b throws (17) with exception object e and stack trace t, then e and t are matched against the **on-catch** clauses to yield a new completion (17.11.1).

Then, even if execution of b did not complete normally or matching against the **on-catch** clauses did not complete normally, the f block is executed.

If execution of f does not complete normally, execution of the **try** statement completes in the same way. Otherwise if execution of b threw (17), the **try** statement completes in the same way as the matching against the **on-catch** clauses. Otherwise the **try** statement completes in the same way as the execution of b.

If T_1 is a malformed or deferred type (19.1), then performing a match causes a run-time error. It is a static warning if T_i, $1 \leq i \leq n$ is a deferred or malformed type.

17.11.1 on-catch clauses

Matching an exception object e and stack trace t against a (potentially empty) sequence of **on-catch** clauses of the form

on T_1 **catch** (e_1, st_1) { s_1 }
...
on T_n **catch** (e_n, st_n) { s_n }

proceeds as follows:

If there are no **on-catch** clauses ($n = 0$), matching throws the exception object e and stack trace t (17).

Otherwise the exception is matched against the first clause.

Otherwise, if the type of e is a subtype of T_1, then the first clause matches, and then e_1 is bound to the exception object e and t_1 is bound to the stack trace t, and s_1 is executed in this scope. The matching completes in the same way as this execution.

Otherwise, if the first clause did not match e, e and t are recursively matched against the remaining **on-catch** clauses:

on T_2 **catch** (e_2, t_2) { s_2 }
...
on T_n **catch** (e_n, t_n) { s_n }

17.12 Return

The *return statement* returns a result to the caller of a synchronous function,

completes the future associated with an asynchronous function or terminates the stream or iterable associated with a generator (9).

returnStatement:
  return expression? ';'
;

Executing a return statement **return** e; proceeds as follows:

Let T be the static type of e, let f be the immediately enclosing function, and let S be the actual return type (19.8.1) of f.

First the expression e is evaluated, producing an object o. If the body of f is marked **async** (9) and the run-time type of o is a subtype of Future<$flatten(S)$>, then let r be the result of evaluating await v where v is a fresh variable bound to o. Otherwise let r be o. Then the return statement returns the value r (17).

It is a static type warning if the body of f is marked **async** and the type Future<$flatten(T)$> (16.10) may not be assigned to the declared return type of f. Otherwise, it is a static type warning if T may not be assigned to the declared return type of f.

Let S be the run-time type of o. In checked mode:

- If the body of f is marked **async** (9) it is a dynamic type error if o is not the null object (16.2), the actual return type (19.8.1) of f is not **void**, and Future<$flatten(S)$> is not a subtype of the actual return type of f.
- Otherwise, it is a dynamic type error if o is not the null object (16.2), the actual return type of f is not **void**, and S is not a subtype of the actual return type of f.

It is a compile-time error if a return statement of the form **return** e; appears in a generative constructor (10.6.1).

It is quite easy to forget to add the factory prefix for a constructor, accidentally converting a factory into a generative constructor. The static checker may detect a type mismatch in some, but not all, of these cases. The rule above helps catch such errors, which can otherwise be very hard to recognize. There is no real downside to it, as returning a value from a generative constructor is meaningless.

It is a compile-time error if a return statement of the form **return** e; appears in a generator function.

In the case of a generator function, the value returned by the function is the iterable or stream associated with it, and individual elements are added to that iterable using yield statements, and so returning a value makes no sense.

Let f be the function immediately enclosing a return statement of the form **return**;. It is a static warning if f is neither a generator nor a generative constructor and either:

- f is synchronous and the return type of f may not be assigned to **void** (19.7) or,

- f is asynchronous and the return type of f may not be assigned to Future<Null>.

Hence, a static warning will not be issued if f has no declared return type, since the return type would be **dynamic** and **dynamic** may be assigned to **void** and to Future<Null>. However, any synchronous non-generator function that declares a return type must return an expression explicitly. *This helps catch situations where users forget to return a value in a return statement.*

An asynchronous non-generator always returns a future of some sort. If no expression is given, the future will be completed with the null object (16.2) and this motivates the requirement above.

Leaving the return type of a function marked **async** blank will be interpreted as **dynamic** as always, and cause no type error.

Executing a return statement with no expression, **return**; returns with no value (17).

It is a static warning if a function contains both one or more explicit return statements of the form **return**; and one or more return statements of the form **return** e;.

17.13 Labels

A *label* is an identifier followed by a colon. A *labeled statement* is a statement prefixed by a label L. A *labeled case clause* is a case clause within a switch statement (17.9) prefixed by a label L.

The sole role of labels is to provide targets for the break (17.14) and continue (17.15) statements.

label:
  identifier ':'
;

Execution a labeled statement s, $label : s_l$, consists of executing s_l. If execution of s_l breaks to the label $label$ (17), then execution of s completes normally, otherwise execution of s completes in the same ways as the execution of s_l.

The namespace of labels is distinct from the one used for types, functions and variables.

The scope of a label that labels a statement s is s. The scope of a label that labels a case clause of a switch statement s is s.

Labels should be avoided by programmers at all costs. The motivation for including labels in the language is primarily making Dart a better target for code generation.

17.14 Break

The *break statement* consists of the reserved word **break** and an optional

label (17.13).

breakStatement:
  break identifier? ';'
;

Let s_b be a **break** statement. If s_b is of the form **break** L;, then it is a compile-time error if s_b is not enclosed in a labeled statement with the label L within the innermost function in which s_b occurs. If s_b is of the form **break**;, then it is a compile-time error if s_b is not enclosed in an **await for** (17.6.3), **do** (17.8), **for** (17.6), **switch** (17.9) or **while** (17.7) statement within the innermost function in which s_b occurs.

Execution of a **break** statement **break** L; breaks to the label L (17). Execution of a **break** statement **break**; breaks without a label (17).

17.15 Continue

The *continue statement* consists of the reserved word **continue** and an optional label (17.13).

continueStatement:
  continue identifier? ';'
;

Let s_c be a **continue** statement. If s_c is of the form **continue** L;, then it is a compile-time error if s_c is not enclosed in either an **await for** (17.6.3), **do** (17.8), **for** (17.6), or **while** (17.7) statement labeled with L, or in a **switch** statement with a case clause labeled with L, within the innermost function in which s_c occurs. If s_c is of the form **continue**; then it is a compile-time error if s_c is not enclosed in an **await for** (17.6.3) **do** (17.8), **for** (17.6), or **while** (17.7) statement within the innermost function in which s_c occurs.

Execution of a **continue** statement **continue** L; continues to the label L (17). Execution of a **continue** statement **continue**; continues without a label (17).

17.16 Yield and Yield-Each

17.16.1 Yield

The *yield statement* adds an element to the result of a generator function (9).

yieldStatement:
  yield expression ';'
;

Execution of a statement s of the form **yield** e; proceeds as follows:

First, the expression e is evaluated to an object o. If the enclosing function m is marked **async*** (9) and the stream u associated with m has been paused, then the nearest enclosing asynchronous for loop (17.6.3), if any, is paused and execution of m is suspended until u is resumed or canceled.

Next, o is added to the iterable or stream associated with the immediately enclosing function.

If the enclosing function m is marked **async*** and the stream u associated with m has been canceled, then the **yield** statement returns without a value (17), otherwise it completes normally.

*The stream associated with an asynchronous generator could be canceled by any code with a reference to that stream at any point where the generator was passivated. Such a cancellation constitutes an irretrievable error for the generator. At this point, the only plausible action for the generator is to clean up after itself via its **finally** clauses.*

Otherwise, if the enclosing function m is marked **async*** (9) then the enclosing function may suspend, in which case the nearest enclosing asynchronous for loop (17.6.3), if any, is paused first.

*If a **yield** occurred inside an infinite loop and the enclosing function never suspended, there might not be an opportunity for consumers of the enclosing stream to run and access the data in the stream. The stream might then accumulate an unbounded number of elements. Such a situation is untenable. Therefore, we allow the enclosing function to be suspended when a new value is added to its associated stream. However, it is not essential (and in fact, can be quite costly) to suspend the function on every **yield**. The implementation is free to decide how often to suspend the enclosing function. The only requirement is that consumers are not blocked indefinitely.*

If the enclosing function m is marked **sync*** (9) then:

- Execution of the function m immediately enclosing s is suspended until the nullary method moveNext() is invoked upon the iterator used to initiate the current invocation of m.
- The current call to moveNext() returns **true**.

It is a compile-time error if a yield statement appears in a function that is not a generator function.

Let T be the static type of e and let f be the immediately enclosing function. It is a static type warning if either:

- the body of f is marked **async*** and the type Stream<T> may not be assigned to the declared return type of f.
- the body of f is marked **sync*** and the type Iterable<T> may not be assigned to the declared return type of f.

17.16.2 Yield-Each

The *yield-each statement* adds a series of values to the result of a generator function (9).

yieldEachStatement:
  yield* expression ';'
  ;

Execution of a statement s of the form **yield*** e; proceeds as follows:
First, the expression e is evaluated to an object o.
If the immediately enclosing function m is marked **sync*** (9), then:

1. It is a dynamic error if the class of o does not implement Iterable. Otherwise
2. The method iterator is invoked upon o returning an object i.
3. The moveNext method of i is invoked on it with no arguments. If moveNext returns **false** execution of s is complete. Otherwise
4. The getter current is invoked on i. If the invocation throws (16), execution of s throws the same exception object and stack trace (17). Otherwise, the result x of the getter invocation is added to the iterable associated with m. Execution of the function m immediately enclosing s is suspended until the nullary method moveNext() is invoked upon the iterator used to initiate the current invocation of m, at which point execution of s continues at 3.
5. The current call to moveNext() returns **true**.

If m is marked **async*** (9), then:

- It is a dynamic error if the class of o does not implement Stream. Otherwise
- The nearest enclosing asynchronous for loop (17.6.3), if any, is paused.
- The o stream is listened to, creating a subscription s, and for each event x, or error e with stack trace t, of s:
  - If the stream u associated with m has been paused, then execution of m is suspended until u is resumed or canceled.
  - If the stream u associated with m has been canceled, then s is canceled by evaluating **await** v.cancel() where v is a fresh variable referencing the stream subscription s. Then, if the cancel completed normally, the stream execution of s returns without a value (17).
  - Otherwise, x, or e with t, are added to the stream associated with m in the order they appear in o. The function m may suspend.
- If the stream o is done, execution of s completes normally.

It is a compile-time error if a yield-each statement appears in a function that is not a generator function.

Let T be the static type of e and let f be the immediately enclosing function. It is a static type warning if T may not be assigned to the declared return type of f. If f is synchronous it is a static type warning if T may not be assigned to Iterable. If f is asynchronous it is a static type warning if T may not be assigned to Stream.

17.17 Assert

An *assert statement* is used to disrupt normal execution if a given boolean condition does not hold.

assertStatement:
  assertion ';'
;

assertion:
  assert '(' expression (',' expression)? ','? ')'
;

The grammar allows a trailing comma before the closing parenthesis, similarly to an argument list. That comma, if present, has no effect. An assertion with a trailing comma is equivalent to one with that comma removed.

An assertion of the form **assert**(e)) is equivalent to an assertion of the form **assert**(e, **null**).

Execution of an assert statement executes the assertion as described below and completes in the same way as the assertion.

In production mode an assertion has no effect and its execution immediately completes normally (17). In checked mode, execution of an assertion **assert**(c, e) proceeds as follows:

The expression c is evaluated to an object r. It is a dynamic type error if r is not of type bool. Hence it is a compile-time error if that situation arises during evaluation of an assertion in a **const** constructor invocation. If r is **true** then execution of the assert statement completes normally (17). Otherwise, e is evaluated to an object m and then the execution of the assert statement throws (17) an AssertionError containing m and with a stack trace corresponding to the current execution state at the assertion.

It is a static type warning if the type of e may not be assigned to bool.

Why is this a statement, not a built in function call? Because it is handled magically so it has no effect and no overhead in production mode. Also, in the absence of final methods, one could not prevent it being overridden (though there is no real harm in that). It cannot be viewed as a function call that is being optimized away because the arguments might have side effects.

18 Libraries and Scripts

A Dart program consists of one or more libraries, and may be built out of one or more *compilation units*. A compilation unit may be a library or a part (18.3).

A library consists of (a possibly empty) set of imports, a set of exports, and a set of top-level declarations. A top-level declaration is either a class (10), a type alias declaration (19.3.1), a function (9) or a variable declaration (8). The members of a library L are those top level declarations given within L.

topLevelDefinition:
  classDefinition |
  enumType |
  typeAlias |
  external? functionSignature ';' |
  external? getterSignature ';' |
  external? setterSignature ';' |
  functionSignature functionBody |
  returnType? **get** identifier functionBody |
  returnType? **set** identifier formalParameterList functionBody |
  (**final** | **const**) type? staticFinalDeclarationList ';' |
  variableDeclaration ';'
;

getOrSet:
  get |
  set
;

libraryDefinition:
  scriptTag? libraryName? importOrExport* partDirective*
      topLevelDefinition*
;

scriptTag:
  '#!' (~NEWLINE)* NEWLINE
;

libraryName:
  metadata **library** dottedIdentifierList ';'
;

importOrExport:
  libraryImport |
  libraryExport

;

dottedIdentifierList:
identifier ('.' identifier)*
;

Libraries may be *explicitly named* or *implicitly named*. An explicitly named library begins with the word **library** (possibly prefaced with any applicable metadata annotations), followed by a qualified identifier that gives the name of the library.

Technically, each dot and identifier is a separate token and so spaces between them are acceptable. However, the actual library name is the concatenation of the simple identifiers and dots and contains no spaces.

An implicitly named library has the empty string as its name.

The name of a library is used to tie it to separately compiled parts of the library (called parts) and can be used for printing and, more generally, reflection. The name may be relevant for further language evolution.

Libraries intended for widespread use should avoid name collisions. Dart's pub package management system provides a mechanism for doing so. Each pub package is guaranteed a unique name, effectively enforcing a global namespace.

A library may optionally begin with a *script tag*. Script tags are intended for use with scripts (18.4). A script tag can be used to identify the interpreter of the script to whatever computing environment the script is embedded in. The script tag must appear before any whitespace or comments. A script tag begins with the characters #! and ends at the end of the line. Any characters that follow #! in the script tag are ignored by the Dart implementation.

Libraries are units of privacy. A private declaration declared within a library L can only be accessed by code within L. Any attempt to access a private member declaration from outside L will cause a method, getter or setter lookup failure.

Since top level privates are not imported, using the top level privates of another library is never possible.

The *public namespace* of library L is the mapping that maps the simple name of each public top-level member m of L to m. The scope of a library L consists of the names introduced by all top-level declarations declared in L, and the names added by L's imports (18.1).

18.1 Imports

An *import* specifies a library to be used in the scope of another library.

libraryImport:
metadata importSpecification
;

importSpecification:
  import configurableUri (**as** identifier)? combinator* ';' |
  import uri **deferred as** identifier combinator* ';'
;

combinator:
  show identifierList |
  hide identifierList
;

identifierList:
  identifier (, identifier)*
;

An import specifies a URI x where the declaration of an imported library is to be found.

Imports may be *deferred* or *immediate*. A deferred import is distinguished by the appearance of the built-in identifier **deferred** after the URI. Any import that is not deferred is immediate.

It is a compile-time error if the specified URI of an immediate import does not refer to a library declaration. The interpretation of URIs is described in section 18.5 below.

It is a static warning if the specified URI of a deferred import does not refer to a library declaration.

One cannot detect the problem at compile time because compilation often occurs during execution and one does not know what the URI refers to. However the development environment should detect the problem.

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.

An immediate import directive I may optionally include a prefix clause of the form **as** *id* used to prefix names imported by I. A deferred import must include a prefix clause or a compile-time error occurs. It is a compile-time error if a prefix used in a deferred import is used in another import clause.

An import directive I may optionally include a namespace combinator clauses used to restrict the set of names imported by I. Currently, two namespace combinators are supported: **hide** and **show**.

Let I be an import directive that refers to a URI via the string s_1. Evaluation of I proceeds as follows:

If I is a deferred import, no evaluation takes place. Instead, a mapping of the name of the prefix, p to a *deferred prefix object* is added to the scope of the current library L. The deferred prefix object has the following methods:

- loadLibrary. This method returns a future f. When called, the method causes an immediate import I' to be executed at some future time, where

I' is derived from I by eliding the word **deferred** and adding a **hide** loadLibrary combinator clause. When I' executes without error, f completes successfully. If I' executes without error, we say that the call to loadLibrary has succeeded, otherwise we say the call has failed.

- For every top level function f named id in the imported library B, a corresponding method named id with the same signature as f. Calling the method results in a run-time error.
- For every top level getter g named id in B, a corresponding getter named id with the same signature as g. Calling the method results in a run-time error.
- For every top level setter s named id in B, a corresponding setter named id with the same signature as s. Calling the method results in a run-time error.
- For every type T named id in B, a corresponding getter named id with return type Type. Calling the method results in a run-time error.

The purpose of adding members of B to p is to ensure that any warnings issued when using p are correct, and no spurious warnings are generated. In fact, at run time we cannot add these members until B is loaded; but any such invocations will fail at run time as specified by virtue of being completely absent.

The static type of the prefix object p is a unique interface type that has those members whose names and signatures are listed above.

After a call succeeds, the name p is mapped to a non-deferred prefix object as described below. In addition, the prefix object also supports the loadLibrary method, and so it is possible to call loadLibrary again. If a call fails, nothing happens, and one again has the option to call loadLibrary again. Whether a repeated call to loadLibrary succeeds will vary as described below.

The effect of a repeated call to p.loadLibrary is as follows:

- If another call to p.loadLibrary has already succeeded, the repeated call also succeeds. Otherwise,
- If another call to p.loadLibrary has failed:
  - If the failure is due to a compilation error, the repeated call fails for the same reason.
  - If the failure is due to other causes, the repeated call behaves as if no previous call had been made.

In other words, one can retry a deferred load after a network failure or because a file is absent, but once one finds some content and loads it, one can no longer reload.

We do not specify what value the future returned resolves to.

If I is an immediate import then, first

- If the URI that is the value of s_1 has not yet been accessed by an import or export (18.2) directive in the current isolate then the contents of the URI are compiled to yield a library B. Because libraries may have mutually recursive imports, care must be taken to avoid an infinite regress.
- Otherwise, the contents of the URI denoted by s_1 have been compiled into a library B within the current isolate.

Let NS_0 be the exported namespace (18.2) of B. Then, for each combinator clause $C_i, i \in 1..n$ in I:

- If C_i is of the form

  show $id_1, \ldots, id_k$

  then let $NS_i = \mathbf{show}([id_1, \ldots, id_k], NS_{i-1})$

  where $show(l, n)$ takes a list of identifiers l and a namespace n, and produces a namespace that maps each name in l to the same element that n does. Furthermore, for each name x in l, if n defines the name $x =$ then the new namespace maps $x =$ to the same element that n does. Otherwise the resulting mapping is undefined.

- If C_i is of the form

  hide $id_1, \ldots, id_k$

  then let $NS_i = \mathbf{hide}([id_1, \ldots, id_k], NS_{i-1})$

  where $hide(l, n)$ takes a list of identifiers l and a namespace n, and produces a namespace that is identical to n except that for each name k in l, k and $k =$ are undefined.

Next, if I includes a prefix clause of the form **as** p, let $NS = NS_n \cup \{p : prefixObject(NS_n)\}$ where $prefixObject(NS_n)$ is a *prefix object* for the namespace NS_n, which is an object that has the following members:

- For every top level function f named id in NS_n, a corresponding method with the same name and signature as f that forwards (9.1) to f.
- For every top level getter with the same name and signature as g named id in NS_n, a corresponding getter that forwards to g.
- For every top level setter s with the same name and signature as named id in NS_n, a corresponding setter that forwards to s.
- For every type T named id in NS_n, a corresponding getter named id with return type Type, that, when invoked, returns the type object for T.

Otherwise, let $NS = NS_n$. It is a compile-time error if the current library declares a top-level member named p.

The static type of the prefix object p is a unique interface type that has those members whose names and signatures are listed above.

Then, for each entry mapping key k to declaration d in NS, d is made available in the top level scope of L under the name k unless either:

- a top-level declaration with the name k exists in L, OR
- a prefix clause of the form **as** k is used in L.

The greatly increases the chance that a member can be added to a library without breaking its importers.

A *system library* is a library that is part of the Dart implementation. Any other library is a *non-system library.*

If a name N is referenced by a library L and N would be introduced into the top level scope of L by imports of two libraries, L_1 and L_2, the exported namespace of L_1 binds N to a declaration originating in a system library, and the exported namespace of L_2 binds N to a declaration that does not originate in a system library, then the import of L_1 is implicitly extended by a **hide** N clause.

Whereas normal conflicts are resolved at deployment time, the functionality of dart*: libraries is injected into an application at run time, and may vary over time as browsers are upgraded. Thus, conflicts with* dart*: libraries can arise at run time, outside the developer's control. To avoid breaking deployed applications in this way, conflicts with the* dart*: libraries are treated specially.*

It is recommended that tools that deploy Dart code produce output in which all imports use show clauses to ensure that additions to the namespace of a library never impact deployed code.

If a name N is referenced by a library L and N is introduced into the top level scope of L by more than one import, and not all the imports denote the same declaration, then:

- A static warning occurs.
- If N is referenced as a function, getter or setter, a NoSuchMethodError is thrown.
- If N is referenced as a type, it is treated as a malformed type.

We say that the namespace NS *has been imported into* L.

It is neither an error nor a warning if N is introduced by two or more imports but never referred to.

The policy above makes libraries more robust in the face of additions made to their imports.

A clear distinction needs to be made between this approach, and seemingly similar policies with respect to classes or interfaces. The use of a class or interface, and of its members, is separate from its declaration. The usage and declaration may occur in widely separated places in the code, and may in fact be authored by different people or organizations. It is important that errors are given at the offending declaration so that the party that receives the error can respond to it a meaningful way.

In contrast a library comprises both imports and their usage; the library is under the control of a single party and so any problem stemming from the import can be resolved even if it is reported at the use site.

It is a static warning to import two different libraries with the same name unless their name is the empty string.

A widely disseminated library should be given a name that will not conflict with other such libraries. The preferred mechanism for this is using pub, the Dart package manager, which provides a global namespace for libraries, and conventions that leverage that namespace.

Note that no errors or warnings are given if one hides or shows a name that is not in a namespace. *This prevents situations where removing a name from a library would cause breakage of a client library.*

The dart core library dart:core is implicitly imported into every dart library other than itself via an import clause of the form

import 'dart:core';

unless the importing library explicitly imports dart:core.

Any import of dart:core, even if restricted via **show**, **hide** or **as**, preempts the automatic import.

It would be nice if there was nothing special about dart:core. However, its use is pervasive, which leads to the decision to import it automatically. However, some library L may wish to define entities with names used by dart:core (which it can easily do, as the names declared by a library take precedence). Other libraries may wish to use L and may want to use members of L that conflict with the core library without having to use a prefix and without encountering warnings. The above rule makes this possible, essentially canceling dart:core's special treatment by means of yet another special rule.

18.2 Exports

A library L exports a namespace (6.1), meaning that the declarations in the namespace are made available to other libraries if they choose to import L (18.1). The namespace that L exports is known as its *exported namespace*.

libraryExport:
  metadata **export** configurableUri combinator* ';'
;

An export specifies a URI x where the declaration of an exported library is to be found. It is a compile-time error if the specified URI does not refer to a library declaration.

We say that a name *is exported by a library* (or equivalently, that a library *exports a name*) if the name is in the library's exported namespace. We say that a declaration *is exported by a library* (or equivalently, that a library *exports a declaration*) if the declaration is in the library's exported namespace.

A library always exports all names and all declarations in its public namespace. In addition, a library may choose to re-export additional libraries via *export directives*, often referred to simply as *exports*.

Let E be an export directive that refers to a URI via the string s_1. Evalu-

ation of E proceeds as follows:

First,

- If the URI that is the value of s_1 has not yet been accessed by an import or export directive in the current isolate then the contents of the URI are compiled to yield a library B.
- Otherwise, the contents of the URI denoted by s_1 have been compiled into a library B within the current isolate.

Let NS_0 be the exported namespace of B. Then, for each combinator clause $C_i, i \in 1..n$ in E:

- If C_i is of the form **show** $id_1, \ldots, id_k$ then let $NS_i = \mathbf{show}([id_1, \ldots, id_k], NS_{i-1})$.
- If C_i is of the form **hide** $id_1, \ldots, id_k$ then let $NS_i = \mathbf{hide}([id_1, \ldots, id_k], NS_{i-1})$.

For each entry mapping key k to declaration d in NS_n an entry mapping k to d is added to the exported namespace of L unless a top-level declaration with the name k exists in L.

If a name N is not declared by a library L and N would be introduced into the exported namespace of L by exports of two libraries, L_1 and L_2, the exported namespace of L_1 binds N to a declaration originating in a system library, and the exported namespace of L_2 binds N to a declaration that does not originate in a system library, then the export of L_1 is implicitly extended by a **hide** N clause.

See the discussion in section 18.1 for the reasoning behind this rule.

We say that L *re-exports library* B, and also that L *re-exports namespace* NS_n. When no confusion can arise, we may simply state that L *re-exports* B, or that L *re-exports* NS_n.

It is a compile-time error if a name N is re-exported by a library L and N is introduced into the export namespace of L by more than one export, unless all exports refer to same declaration for the name N. It is a static warning to export two different libraries with the same name unless their name is the empty string.

18.3 Parts

A library may be divided into *parts*, each of which can be stored in a separate location. A library identifies its parts by listing them via **part** directives.

A *part directive* specifies a URI where a Dart compilation unit that should be incorporated into the current library may be found.

partDirective:
  metadata **part** uri ';'

;

partHeader:
metadata **part of** identifier ('.' identifier)* ';'
;

partDeclaration:
partHeader topLevelDefinition* EOF
;

A *part header* begins with **part of** followed by the name of the library the part belongs to. A part declaration consists of a part header followed by a sequence of top-level declarations.

Compiling a part directive of the form **part** *s*; causes the Dart system to attempt to compile the contents of the URI that is the value of *s*. The top-level declarations at that URI are then compiled by the Dart compiler in the scope of the current library. It is a compile-time error if the contents of the URI are not a valid part declaration. It is a static warning if the referenced part declaration *p* names a library other than the current library as the library to which *p* belongs.

It's a compile-time error if the same library contains two part directives with the same URI.

18.4 Scripts

A *script* is a library whose exported namespace (18.2) includes a top-level function declaration named main that has either zero, one or two required arguments.

A script *S* is executed as follows:

First, *S* is compiled as a library as specified above. Then, the top-level function defined by main in the exported namespace of *S* is invoked (16.14) as follows: If main can be be called with with two positional arguments, it is invoked with the following two actual arguments:

1. An object whose run-time type implements List<String>.
2. An object specified when the current isolate *i* was created, for example through the invocation of Isolate.spawnUri that spawned *i*, or the null object (16.2) if no such object was supplied.

If main cannot be called with two positional arguments, but it can be called with one positional argument, it is invoked with an object whose run-time type implements List<String> as the only argument. If main cannot be called with one or two positional arguments, it is invoked with no arguments.

Note that if main requires more than two positional arguments, the library is not considered a script.

A Dart program will typically be executed by executing a script.

It is a compile-time error if a library's export scope contains a declaration named main, and the library is not a script. This restriction ensures that all top-level main declarations introduce a script main-function, so there cannot be a top-level getter or field named main, nor can it be a function that requires more than two arguments. The restriction allows tools to fail early on invalid main methods, without needing to know whether a library will be used as the entry point of a Dart program. It is possible that this restriction will be removed in the future.

18.5 URIs

URIs are specified by means of string literals:

uri:
  stringLiteral
;
configurableUri:
  uri configurationUri*
;
configurationUri:
  if '(' uriTest ')' uri
;
uriTest:
  dottedIdentifierList ('==' stringLiteral)?
;

It is a compile-time error if the string literal x that describes a URI is not a compile-time constant, or if x involves string interpolation.

It is a compile-time error if the string literal x that is used in a *uriTest* is not a compile-time constant, or if x involves string interpolation.

A *configurable URI* c of the form *uri configurationUri*$_1$ ... *configurationUri*$_n$ *specifies a URI* as follows:

- Let u be *uri*.
- For each of the following configuration URIs of the form **if** $(test_i)$ uri_i, in source order, do the following.
  - If $test_i$ is *ids* with no == clause, it is equivalent to *ids* == "true".
  - If $test_i$ is *ids* == *string*, then create a string, *key*, from *ids* by concatenating the identfiers and dots, omitting any spaces between them that may occur in the source.
  - Look up *key* in the available compilation *environment*. The compilation environment is provided by the platform. It maps some string keys to string values, and can be accessed programmatically using the const String.fromEnvironment constructor. Tools may choose to only

make some parts of the compilation environment available for choosing configuration URIs.

  - If the environment contains an entry for *key* and the associated value is equal, as a constant string value, to the value of the string literal *string*, then let u be uri_i and stop iterating the configuration URIs.
  - Otherwise proceed to the next configuration URI.

- The URI specified by c is u.

This specification does not discuss the interpretation of URIs, with the following exceptions.

The interpretation of URIs is mostly left to the surrounding computing environment. For example, if Dart is running in a web browser, that browser will likely interpret some URIs. While it might seem attractive to specify, say, that URIs are interpreted with respect to a standard such as IETF RFC 3986, in practice this will usually depend on the browser and cannot be relied upon.

A URI of the form dart:s is interpreted as a reference to a system library (18.1) s.

A URI of the form package:s is interpreted in an implementation specific manner.

The intent is that, during development, Dart programmers can rely on a package manager to find elements of their program.

Otherwise, any relative URI is interpreted as relative to the location of the current library. All further interpretation of URIs is implementation dependent.

This means it is dependent on the embedder.

19 Types

Dart supports optional typing based on interface types.

The type system is unsound, due to the covariance of generic classes. This is a deliberate choice (and undoubtedly controversial). Experience has shown that sound type rules for generics fly in the face of programmer intuition. It is easy for tools to provide a sound type analysis if they choose, which may be useful for tasks like refactoring.

19.1 Static Types

Static type annotations are used in variable declarations (8) (including formal parameters (9.2)), in the return types of functions (9) and in the bounds of type variables. Static type annotations are used during static checking and when running programs in checked mode. They have no effect whatsoever in production mode.

type:
  typeName typeArguments?

;

typeName:
qualified
;

typeArguments:
'<' typeList '>'
;

typeList:
type (',' type)*
;

A Dart implementation must provide a static checker that detects and reports exactly those situations this specification identifies as static warnings and only those situations. However:

- Running the static checker on a program P is not required for compiling and running P.
- Running the static checker on a program P must not prevent successful compilation of P nor may it prevent the execution of P, regardless of whether any static warnings occur.

Nothing precludes additional tools that implement alternative static analyses (e.g., interpreting the existing type annotations in a sound manner such as either non-variant generics, or inferring declaration based variance from the actual declarations). However, using these tools must not preclude successful compilation and execution of Dart code.

A type T is *malformed* iff:

- T has the form *id* or the form *prefix.id*, and in the enclosing lexical scope, the name *id* (respectively *prefix.id*) does not denote a type.
- T denotes a type variable in the enclosing lexical scope, but occurs in the signature or body of a static member.
- T is a parameterized type of the form $G\langle S_1, \ldots, S_n\rangle$, and G is malformed, or G is not a generic type, or G is a generic type, but it declares n' type parameters and $n' \neq n$, or S_j is malformed for some $j \in 1..n$.
- T is a function type of the form

  T_0 **Function**$\langle X_1$ **extends** $B_1, \ldots, X_m$ **extends** $B_m\rangle$
  $(T_1\ x_1, \ldots, T_k\ x_k, [T_{k+1}\ x_{k+1}, \ldots, T_n\ x_n])$

  or of the form

T_0 **Function**$<X_1$ **extends** $B_1, \ldots, X_m$ **extends** $B_m>$
$(T_1\ x_1, \ldots, T_k\ x_k, \{T_{k+1}\ x_{k+1}, \ldots, T_n\ x_n\})$
where each x_j which is not a named parameter may be omitted, and T_j is malformed for some $j \in 0..n$, or B_j is malformed for some $j \in 1..m$.

- T denotes declarations that were imported from multiple imports clauses.

Any use of a malformed type gives rise to a static warning. A malformed type is then interpreted as **dynamic** by the static type checker and the run-time system unless explicitly specified otherwise.

This ensures that the developer is spared a series of cascading warnings as the malformed type interacts with other types.

A type T is *deferred* iff it is of the form $p.T$ where p is a deferred prefix. It is a static warning to use a deferred type in a type annotation, type test, type cast or as a type parameter. However, all other static warnings must be issued under the assumption that all deferred libraries have successfully been loaded.

19.1.1 Type Promotion

The static type system ascribes a static type to every expression. In some cases, the types of local variables and formal parameters may be promoted from their declared types based on control flow.

We say that a variable v is known to have type T whenever we allow the type of v to be promoted. The exact circumstances when type promotion is allowed are given in the relevant sections of the specification (16.23, 16.21 and 17.5).

Type promotion for a variable v is allowed only when we can deduce that such promotion is valid based on an analysis of certain boolean expressions. In such cases, we say that the boolean expression b shows that v has type T. As a rule, for all variables v and types T, a boolean expression does not show that v has type T. Those situations where an expression does show that a variable has a type are mentioned explicitly in the relevant sections of this specification (16.35 and 16.23).

19.2 Dynamic Type System

A Dart implementation must support execution in both *production mode* and *checked mode*. Those dynamic checks specified as occurring specifically in checked mode must be performed iff the code is executed in checked mode.

Note that this is the case even if the deferred type belongs to a prefix that has already been loaded. This is regrettable, since it strongly discourages the use of type annotations that involve deferred types because Dart programmers use checked mode much of the time.

In practice, many scenarios involving deferred loading involve deferred loading of classes that implement eagerly loaded interfaces, so the situation is often less

onerous than it seems. The current semantics were adopted based on considerations of ease of implementation.

Clearly, if a deferred type has not yet been loaded, it is impossible to do a correct subtype test involving it, and one would expect a dynamic failure, as is the case with type tests and casts. By the same token, one would expect checked mode to work seamlessly once a type had been loaded. We hope to adopt these semantics in the future; such a change would be upwardly compatible.

In checked mode, it is a dynamic type error if a deferred, malformed or malbounded (19.8) type is used in a subtype test.

Consider the following program:

```
typedef F(bool x);
f(foo x) => x;
main() {
  if (f is F) {
    print("yoyoma");
  }
}
```

The type of the formal parameter of f is foo, which is undeclared in the lexical scope. This will lead to a static type warning. At run time the program will print yoyoma, because foo is treated as **dynamic**.

As another example take

```
var i;
i j; // a variable j of type i (supposedly)
main() {
    j = 'I am not an i';
}
```

Since i is not a type, a static warning will be issue at the declaration of j. However, the program can be executed without incident in production mode because the undeclared type i is treated as **dynamic**. However, in checked mode, the implicit subtype test at the assignment will trigger an error at run time.

Here is an example involving malbounded types:

```
class I<T extends num> {}
class J {}
class A<T> implements J, I<T> // type warning: T is not a subtype of num
{ ...
}
```

Given the declarations above, the following

```
I x = new A<String>();
```

will cause a dynamic type error in checked mode, because the assignment requires a subtype test A<String> <: I. To show that this holds, we need to show that A<String> << I<String>, but I<String> is a malbounded type, causing the dynamic error. No error is thrown in production mode. Note that

```
J x = new A<String>();
```

does not cause a dynamic error, as there is no need to test against I<String> in this case. Similarly, in production mode

```
A x = new A<String>();
```

bool b = x is I;

b is bound to **true**, but in checked mode the second line causes a dynamic type error.

19.3 Type Declarations

19.3.1 Typedef

A *type alias* declares a name for a type expression.

typeAlias:
  metadata **typedef** typeAliasBody
;

typeAliasBody:
  functionTypeAlias
;

functionTypeAlias:
  functionPrefix typeParameters? formalParameterList ';'
;

functionPrefix:
  returnType? identifier
;

The effect of a type alias of the form
typedef T $id(T_1\ p_1, \ldots,\ T_n\ p_n,\ [T_{n+1}\ p_{n+1}, \ldots,\ T_{n+k}\ p_{n+k}])$
declared in a library L is to introduce the name id into the scope of L, bound to the function type $(T_1, \ldots,\ T_n, [T_{n+1}\ p_{n+1}, \ldots,\ T_{n+k} p_{n+k}]) \rightarrow T$. The effect of a type alias of the form
typedef T $id(T_1\ p_1, \ldots,\ T_n\ p_n,\ \{T_{n+1}\ p_{n+1}, \ldots,\ T_{n+k}\ p_{n+k}\})$
declared in a library L is to introduce the name id into the scope of L, bound to the function type $(T_1, \ldots,\ T_n, \{T_{n+1}\ p_{n+1}, \ldots,\ T_{n+k}\ p_{n+k}\}) \rightarrow T$. In either case, iff no return type is specified, it is taken to be **dynamic**. Likewise, if a type annotation is omitted on a formal parameter, it is taken to be **dynamic**.

It is a compile-time error if any default values are specified in the signature of a function type alias. Any self reference in a typedef, either directly, or recursively via another typedef, is a compile-time error.

19.4 Interface Types

The implicit interface of class I is a direct supertype of the implicit interface of class J iff:

- I is Object, and J has no **extends** clause.
- I is listed in the **extends** clause of J.
- I is listed in the **implements** clause of J.
- I is listed in the **with** clause of J.
- J is a mixin application (12.1) of the mixin of I.

A type T is more specific than a type S, written $T << S$, if one of the following conditions is met:

- T is S.
- T is $\bot$.
- T is Null and S is not $\bot$.
- S is **dynamic**.
- S is a direct supertype of T.
- T is a type parameter and S is the upper bound of T.
- T is a type parameter and S is Object.
- T is of the form $I<T_1, \ldots, T_n>$ and S is of the form $I<S_1, \ldots, S_n>$ and: $T_i << S_i, 1 \leq i \leq n$
- T and S are both function types, and $T << S$ under the rules of section 19.5.
- T is a function type and S is **Function**.
- $T << U$ and $U << S$.

$<<$ is a partial order on types. T is a subtype of S, written $T <: S$, iff $[\bot/\textbf{dynamic}]T << S$.

Note that $<:$ is not a partial order on types, it is only binary relation on types. This is because $<:$ is not transitive. If it was, the subtype rule would have a cycle. For example: List $<:$ List < String > and List < int >$<:$ List, but List < int > is not a subtype of List < String >. Although $<:$ is not a partial order on types, it does contain a partial order, namely $<<$. This means that, barring raw types, intuition about classical subtype rules does apply.

The Null type is more specific than all non-$\bot$ types, even though it doesn't actually extend or implement those types. The other types are effectively treated as if they are *nullable*, which makes the null object (16.2) assignable to them.

S is a supertype of T, written $S :> T$, iff T is a subtype of S.

The supertypes of an interface are its direct supertypes and their supertypes.

An interface type T may be assigned to a type S, written $T \Longleftrightarrow S$, iff either $T <: S$, $S <: T$.

This rule may surprise readers accustomed to conventional typechecking. The intent of the $\Longleftrightarrow$ relation is not to ensure that an assignment is correct. Instead, it aims to only flag assignments that are almost certain to be erroneous, without precluding assignments that may work.

For example, assigning a value of static type Object to a variable with static type String, while not guaranteed to be correct, might be fine if the run-time value happens to be a string.

19.5 Function Types

Note that the non-generic case is covered by using $s = 0$, in which case the type parameter declarations are omitted (14).

Function types come in two variants:

1. The types of functions that only have positional parameters. These have the general form

   $<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$

   $(T_1, \ldots, T_n, [T_{n+1}, \ldots, T_{n+k}]) \rightarrow T.$

2. The types of functions with named parameters. These have the general form

   $<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$

   $(T_1, \ldots, T_n, \{T_{x_1}\ x_1, \ldots, T_{x_k}\ x_k\}) \rightarrow T.$

Two function types are considered equal if consistent renaming of type parameters can make them identical.

A common way to say this is that we do not distinguish function types which are alpha-equivalent. For the subtyping rule below this means we can assume that a suitable renaming has already taken place. In cases where this is not possible because the number of type parameters in the two types differ or the bounds are different, no subtype relationship exists.

The function type
$<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$
$(T_1, \ldots, T_k, [T_{k+1}, \ldots, T_{n+m}]) \rightarrow T$
is a subtype of the function type
$<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$
$(S_1, \ldots, S_{k+j}, [S_{k+j+1}, \ldots, S_n]) \rightarrow S$,
if all of the following conditions are met, assuming that X_j is a subtype of B_j, for all $j \in 1..s$:

1. Either

   - S is **void**, Or
   - $T \Longleftrightarrow S$.

2. $\forall i \in 1..n, T_i \Longleftrightarrow S_i$.

A function type
$<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$
$(T_1, \ldots, T_n, \{T_{x_1}\ x_1, \ldots, T_{x_k}\ x_k\}) \rightarrow T$
is a subtype of the function type
$<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$
$(S_1, \ldots, S_n, \{S_{y_1}\ y_1, \ldots, S_{y_m}\ y_m\}) \rightarrow S$,
if all of the following conditions are met, assuming that X_j is a subtype of B_j, for all $j \in 1..s$:

1. Either
   - S is **void**, Or
   - $T \Longleftrightarrow S$.
2. $\forall i \in 1..n, T_i \Longleftrightarrow S_i$.
3. $k \geq m$ and $y_i \in \{x_1, \ldots, x_k\}, i \in 1..m$.
4. For all $y_i \in \{y_1, \ldots, y_m\}, y_i = x_j \Rightarrow T_{x_j} \Longleftrightarrow S_{y_i}$

In addition, the following subtype rules apply:
$<X_1\ B_1, \ldots, X_s\ B_s>(T_1, \ldots, T_n, []) \rightarrow T \quad <:$
$<X_1\ B_1, \ldots, X_s\ B_s>(T_1, \ldots, T_n) \rightarrow T$.

$<X_1\ B_1, \ldots, X_s\ B_s>(T_1, \ldots, T_n, \{\}) \rightarrow T \quad <:$
$<X_1\ B_1, \ldots, X_s\ B_s>(T_1, \ldots, T_n) \rightarrow T$.

The naive reader might conclude that, since it is not legal to declare a function with an empty optional parameter list, these rules are pointless. However, they induce useful relationships between function types that declare no optional parameters and those that do.

A function type T may be assigned to a function type S, written $T \Longleftrightarrow S$, iff $T <: S$.

A function is always an instance of some class that implements the class **Function**. All function types are subtypes of **Function**. It is a static warning if a concrete class implements **Function** and does not have a concrete method named **call** unless that class has a concrete noSuchMethod() distinct from the one declared in class Object.

A function type
$<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$
$(T_1, \ldots, T_k, [T_{k+1}, \ldots, T_{n+m}]) \rightarrow T$
is more specific than the function type
$<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$
$(S_1, \ldots, S_{k+j}, [S_{k+j+1}, \ldots, S_n]) \rightarrow S$,
if all of the following conditions are met:

1. Either
   - S is **void**, Or

- $T << S$.

2. $\forall i \in 1..n, T_i << S_i$.

A function type
$<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$
$(T_1, \ldots, T_n, \{T_{x_1}\ x_1, \ldots, T_{x_k}\ x_k\}) \rightarrow T$
is more specific than the function type
$<X_1$ **extends** $B_1, \ldots, X_s$ **extends** $B_s>$
$(S_1, \ldots, S_n, \{S_{y_1}\ y_1, \ldots, S_{y_m}\ y_m\}) \rightarrow S$,
if all of the following conditions are met:

1. Either
   - S is **void**, Or
   - $T << S$.
2. $\forall i \in 1..n, T_i << S_i$.
3. $k \geq m$ and $y_i \in \{x_1, \ldots, x_k\}, i \in 1..m$.
4. For all $y_i \in \{y_1, \ldots, y_m\}, y_i = x_j \Rightarrow T_j << S_i$

Furthermore, if F is a function type, $F << $ **Function**.

19.6 Type dynamic

The type **dynamic** denotes the unknown type.

If no static type annotation has been provided the type system assumes the declaration has the unknown type. If a generic type is used but type arguments are not provided, then the type arguments default to the unknown type.

This means that given a generic declaration $G<T_1, \ldots, T_n>$, the type G is equivalent to $G<$**dynamic**$, \ldots,$ **dynamic**$>$.

Type **dynamic** has methods for every possible identifier and arity, with every possible combination of named parameters. These methods all have **dynamic** as their return type, and their formal parameters all have type **dynamic**. Type **dynamic** has properties for every possible identifier. These properties all have type **dynamic**.

From a usability perspective, we want to ensure that the checker does not issue errors everywhere an unknown type is used. The definitions above ensure that no secondary errors are reported when accessing an unknown type.

The current rules say that missing type arguments are treated as if they were the type ***dynamic****. An alternative is to consider them as meaning* Object*. This would lead to earlier error detection in checked mode, and more aggressive errors during static typechecking. For example:*

(1) typedAPI(G<String>g){...}

(2) typedAPI(new G());

Under the alternative rules, (2) would cause a run-time error in checked mode. This seems desirable from the perspective of error localization. However,

when a dynamic error is thrown at (2), the only way to keep running is rewriting (2) into

(3) typedAPI(new G<String>());

This forces users to write type information in their client code just because they are calling a typed API. We do not want to impose this on Dart programmers, some of which may be blissfully unaware of types in general, and genericity in particular.

What of static checking? Surely we would want to flag (2) when users have explicitly asked for static typechecking? Yes, but the reality is that the Dart static checker is likely to be running in the background by default. Engineering teams typically desire a "clean build" free of warnings and so the checker is designed to be extremely charitable. Other tools can interpret the type information more aggressively and warn about violations of conventional (and sound) static type discipline.

The name **dynamic** denotes a Type object even though **dynamic** is not a class.

The built-in type declaration FutureOr, which is declared in the library dart:async, defines a generic type with one type parameter (14).

The FutureOr<T> type is a non-class type with the following type relations:

- T <: FutureOr<T>.
- Future<T> <: FutureOr<T>.
- If T <: S and Future<T> <: S then FutureOr<T> <: S. In particular, FutureOr<T> <: Object.

. The last point guarantees that generic type FutureOr is *covariant* in its type parameter, just like class types. That is, if S <: T then FutureOr<S> <: FutureOr<T>. If the type arguments passed to FutureOr would issue static warnings if applied to a normal generic class with one type parameter, the same warnings are issued for FutureOr. The name FutureOr as an expression denotes a Type object representing the type FutureOr<dynamic>.

The FutureOr<type> type represents a case where a value can be either an instance of the type type or the type Future<type>. Such cases occur naturally in asynchronous code. Using FutureOr instead of ***dynamic*** *allows some tools to provide a more precise type analysis.*

The type FutureOr<T> has an interface that is identical to that of Object. The only members that can be invoked on a value with static type FutureOr<T> are members that are also on Object. *We only want to allow invocations of members that are inherited from a common supertype of both T and Future<T>. In most cases the only common supertype is Object. The exceptions, like FutureOr<Future<Object» which has Future<Object> as common supertype, are few and not practically useful, so for now we choose to only allow invocations of members inherited from Object.*

19.7 Type Void

The special type **void** may only be used as the return type of a function: it is a compile-time error to use **void** in any other context.

For example, as a type argument, or as the type of a variable or parameter

Void is not an interface type.

The only subtype relations that pertain to void are therefore:

- **void** <: **void** (by reflexivity)
- $\bot$ <: **void** (as bottom is a subtype of all types).
- **void** <: **dynamic** (as **dynamic** is a supertype of all types)

The analogous rules also hold for the $<<$ relation for similar reasons.

Hence, the static checker will issue warnings if one attempts to access a member of the result of a void method invocation (even for members of the null object (16.2), such as $==$). Likewise, passing the result of a void method as a parameter or assigning it to a variable will cause a warning unless the variable/formal parameter has type dynamic.

On the other hand, it is possible to return the result of a void method from within a void method. One can also return the null object (16.2); or a value of type **dynamic**. Returning any other result will cause a type warning. In checked mode, a dynamic type error would arise if a non-null object was returned from a void method (since no object has run-time type **dynamic**).

The name **void** does not denote a Type object.

*It is syntacticly illegal to use **void** as an expression, and it would make no sense to do so. Type objects reify the run-time types of instances. No instance ever has type **void**.*

19.8 Parameterized Types

A *parameterized type* is a syntactic construct where the name of a generic type declaration is applied to a list of actual type arguments. A *generic instantiation* is the operation where a generic type is applied to actual type arguments.

So a parameterized type is the syntactic concept that corresponds to the semantic concept of a generic instantiation. When using the former, we will often leave the latter implicit.

Let T be a parameterized type $G<S_1, \ldots, S_n>$. Assume that T is not malformed.

In particular, G denotes a generic type with n formal type parameters.

T is *malbounded* iff either S_i is malbounded for one or more $i \in 1..n$, or T is not well-bounded (14.2).

Any use of a malbounded type gives rise to a static warning.

Let T be a parameterized type of the form $G<A_1, \ldots, A_n>$ and assume that T is not malformed and not malbounded. If S is the static type of a member m declared by G, then the static type of the member m of an expression of type

T is $[A_1/X_1, \ldots, A_n/X_n]S$, where $X_1, \ldots, X_n$ are the formal type parameters of G.

Let T be a parameterized type of the form $G<A_1, \ldots, A_n>$ and assume that T is not malformed and not malbounded. T is then evaluated as follows:

For $j \in 1..n$, evaluate A_j to a type t_j. T then evaluates to the generic instantiation where G is applied to $t_1, \ldots, t_n$.

19.8.1 Actual Type of Declaration

Let T be the declared type of a declaration d, as it appears in the program source. Let $X_1, \ldots, X_n$ be the formal type parameters in scope at d. In a context where the actual type arguments corresponding to $X_1, \ldots, X_n$ are $t_1, \ldots, t_n$, the *actual type* of d is $[t_1/X_1, \ldots, t_n/X_n]T$.

In the non-generic case where $n = 0$ the actual type is equal to the declared type. Note that $X_1, \ldots, X_n$ may be declared by multiple entities, e.g., one or more enclosing generic functions and an enclosing generic class.

Let X **extends** B be a formal type parameter declaration. Let $X_1, \ldots, X_n$ be the formal type parameters in scope at the declaration of X. In a context where the actual type arguments corresponding to $X_1, \ldots, X_n$ are $t_1, \ldots, t_n$, the *actual bound* for X is $[t_1/X_1, \ldots, t_n/X_n]B$.

Note that there exists a j such that $X = X_j$, because each formal type parameter is in scope at its own declaration.

19.8.2 Least Upper Bounds

Given two interfaces I and J, let S_I be the set of superinterfaces of I, let S_J be the set of superinterfaces of J and let $S = (\{I\} \cup S_I) \cap (\{J\} \cup S_J)$. Furthermore, we define $S_n = \{T | T \in S \wedge \mathit{depth}(T) = n\}$ for any finite n where $\mathit{depth}(T)$ is the number of steps in the longest inheritance path from T to Object. Let q be the largest number such that S_q has cardinality one, which must exist because S_0 is {Object}. The least upper bound of I and J is the sole element of S_q.

The least upper bound of **dynamic** and any type T is **dynamic**. The least upper bound of **void** and any type $T \neq$ **dynamic** is **void**. The least upper bound of $\bot$ and any type T is T. Let U be a type variable with upper bound B. The least upper bound of U and a type $T \neq \bot$ is the least upper bound of B and T.

The least upper bound operation is commutative and idempotent, but it is not associative.

The least upper bound of a function type and an interface type T is the least upper bound of **Function** and T. Let F and G be function types. If F and G differ in their number of required parameters, then the least upper bound of F and G is **Function**. Otherwise:

- If

  $F = <X_1\ B_1, \ldots,\ X_s\ B_s>(T_1, \ldots,\ T_r, [T_{r+1}, \ldots,\ T_n]) \rightarrow T_0$ and

$G = <X_1\ B_1, \ldots,\ X_s\ B_s>(S_1, \ldots,\ S_r, [S_{r+1}, \ldots,\ S_k]) \rightarrow S_0$

where $k \leq n$ then the least upper bound of F and G is

$<X_1\ B_1, \ldots,\ X_s\ B_s>(L_1, \ldots,\ L_r, [L_{r+1}, \ldots,\ L_k]) \rightarrow L_0$

where L_i is the least upper bound of T_i and $S_i, i \in 0..k$.

- If

  $F = <X_1\ B_1, \ldots,\ X_s\ B_s>(T_1, \ldots,\ T_r, [T_{r+1}, \ldots,\ T_n]) \rightarrow T_0$,

  $G = <X_1\ B_1, \ldots,\ X_s\ B_s>(S_1, \ldots,\ S_r, \{\ \ldots\ \}) \rightarrow S_0$

  then the least upper bound of F and G is

  $<X_1\ B_1, \ldots,\ X_s\ B_s>(L_1, \ldots,\ L_r) \rightarrow L_0$

  where L_i is the least upper bound of T_i and $S_i, i \in 0..r$.

- If

  $F = <X_1\ B_1, \ldots,\ X_s\ B_s>(T_1, \ldots,\ T_r, \{T_{r+1}\ p_{r+1}, \ldots,\ T_f\ p_f\}) \rightarrow T_0$,

  $G = <X_1\ B_1, \ldots,\ X_s\ B_s>(S_1, \ldots,\ S_r, \{S_{r+1}\ q_{r+1}, \ldots,\ S_g\ q_g\}) \rightarrow S_0$

  then let $\{x_m, \ldots, x_n\} = \{p_{r+1}, \ldots, p_f\} \cap \{q_{r+1}, \ldots, q_g\}$ and let X_j be the least upper bound of the types of x_j in F and $G, j \in m..n$. Then the least upper bound of F and G is

  $<X_1\ B_1, \ldots,\ X_s\ B_s>(L_1, \ldots,\ L_r, \{X_m\ x_m, \ldots,\ X_n\ x_n\}) \rightarrow L_0$

  where L_i is the least upper bound of T_i and $S_i, i \in 0..r$

Note that the non-generic case is covered by using $s = 0$, in which case the type parameter declarations are omitted (14).

20 Reference

20.1 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.

20.1.1 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)+
;

20.1.2 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.

20.2 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–, e1[e2], e1() , ()	None	16
Unary prefix	-e, !e, ˜e, ++e, –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

Appendix: Naming Conventions

The following naming conventions are customary in Dart programs.

- The names of compile-time constant variables never use lower case letters. If they consist of multiple words, those words are separated by underscores. Examples: PI, I_AM_A_CONSTANT.
- The names of functions (including getters, setters, methods and local or library functions) and non-constant variables begin with a lowercase letter. If the name consists of multiple words, each word (except the first) begins with an uppercase letter. No other uppercase letters are used. Examples: camlCase, dart4TheWorld
- The names of types (including classes and type aliases) begin with an upper case letter. If the name consists of multiple words, each word begins with an uppercase letter. No other uppercase letters are used. Examples: CamlCase, Dart4TheWorld.
- The names of type variables are short (preferably single letter). Examples: T, S, K, V , E.
- The names of libraries or library prefixes never use upper case letters. If they consist of multiple words, those words are separated by underscores. Example: my_favorite_library.

Appendix: Integer Implementations

The int type represents integers. The specification is written with 64-bit two's complement integers as the intended implementation, but when Dart is compiled to JavaScript, the implementation of int will instead use the JavaScript number type.

This introduces a number of differencs:

- Valid values of JavaScript int are any IEEE-754 64-bit floating point number with no fractional part. This includes positive and negative *infinity*, which can be reached by overflowing (integer division by zero is still not allowed). Otherwise valid integer literals (including any leading minus sign) that represent invalid JavaScript int values cannot be compiled to JavaScript. Operations on integers may lose precision since 64-bit floating point numbers are limited to 53 significant bits.

- JavaScript int instances also implement double, and integer-valued double instances also implement int. The int and double class are still separate subclasses of the class num, but *instances* of either class that represent an integer, act as if they are actually instances of a common subclass implementing both int and double. Fractional numbers only implement double.

- Bitwise operations on integers (and, or, xor, negate and shifts) all truncate the operands to 32-bit values.

- The identical method cannot distinguish the values 0.0 and -0.0, and it cannot recognize any *NaN* value as identical to itself. For efficiency, the identical operation uses the JavaScript $===$ operator.