Dart Programming Language Specification

Draft Version 0.11

The Dart Team

September 17, 2012

Contents

1 Notes

This is a work in progress. Expect the contents and language rules to change over time. Please mail comments to gbracha@google.com.

1.1 Licensing

1.2 Change Log

1.2.1 Changes Since Version 0.02

The following changes have been made in version 0.03 since version 0.02. In addition, various typographical errors have been corrected. The changes are listed by section number.

2: Expanded examples of grammar.

7.6.2: Corrected reference to undefined production *typeVariables* to *typeParameters*.

7.10: Removed static warning when imported superinterface of a class contains private members.

The former section on factories and constructors in interfaces (now removed): Removed redundant prohibition on default values.

8.1: Removed static warning when imported superinterface of an interface contains private members.

11: Fixed typo in grammar.

11.11.1, 11.11.2 : made explicit accessibility requirement for class being constructed.

11.11.2: make clear that referenced constructor must be marked **const**.

11.15.4: fixed botched sentence where superclass S is introduced.

11.28: qualified definition of $v++$ so it is clear that v is an identifier.

1.2.2 Changes Since Version 0.03

7.1, The former section on interface methods (now removed): Added missing requirement that overriding methods have same number of required parameters and all optional parameters as overridden method, in same order.

9: Added prohibition against cyclic type hierarchy for type parameters.

11.11: Clarified requirements on use of parameterized types in instance creation expressions.

11.14.2: Added requirement that q_i are distinct.

11.15.3. Static method invocation determines the function (which may involve evaluating a getter) before evaluating the arguments, so that static invocation and top-level function invocation agree.

11.31: Added missing test that type being tested against is in scope and is indeed a type.

12.6.1: Changed for loop to introduce fresh variable for each iteration.

14.8: Malformed parameterized types generate warnings, not errors(except when used in reified contexts like instance creation and superclasses/interfaces).

1.2.3 Changes Since Version 0.04

Added hyperlinks in PDF.

7.1.1: Removed unary plus operator. Clarified that operator formals must be required.

7.6.3: Filled in a lot of missing detail.

The former section on factories and constructors in interfaces (now removed): Allowed factory class to be declared via a qualified name.

11.3: Changed production for $Number$.

11.11.2: Added requirements that actuals be constant, rules for dealing with inappropriate types of actuals, and examples. Also explicitly prohibit type parameters.

11.14.4: Modified final bullet to keep it inline with similar clauses in other sections. Exact wording of these sections also tweaked slightly.

11.27: Specified ! operator. Eliminated section on prefix expressions and moved contents to section on unary expressions.

15.1: Specified unicode form of Dart source.

1.2.4 Changes Since Version 0.05

7.6.1: Clarified how initializing formals can act as optional parameters of generative constructors.

7.6.2: Treat factories as constructors, so type parameters are implicitly in scope.

The former section on factories and constructors in interfaces (now removed): Simplify rules for interface factory clauses. Use the keyword **default** instead of **factory**.

9: Mention that typedefs can have type parameters.

11.31: Added checked mode test that type arguments match generic type.

14.2: Added definition of malformed types, and requirement on their handling in checked mode.

1.2.5 Changes Since Version 0.06

5: library variable initializers must be constant.

7: Added **abstract** modifier to grammar.

7, 7.7, 7.8, 11.14.3,11.30: Superclass static members are not in scope in subclasses, and do not conflict with subclass members.

7.1.1: []= must return **void**. Operator **call** added to support function emulation. Removed operator >>>. Made explicit restriction on methods named **call** or **negate**.

11.1: Added !*e* as constant expression. Clarified what happens if evaluation of a constant fails.

11.7: Map keys need not be constants. However, they are always string literals.

11.10: State restrictions on use of **this**.

11.11, 11.11.1: Rules for bounds checking of constructor arguments when calling default constructors for interfaces refined.

11.15.1: Revised semantics to account for function emulation.

11.15.3: Revised semantics to account for function emulation.

11.15.4: Factory constructors cannot contain super invocations. Revised semantics to account for function emulation.

11.18: Specified assignment involving []= operator.

11.18.1: Removed operator >>>.

11.24: Removed operator >>>.

11.28: Postfix -- operator specified. Behavior of postfix operations on subscripted expressions specified.

11.30: Added built-in identifier **call**. Banned use of built-in identifiers as types made other uses warnings.

11.31: Moved specification of test that type arguments match generic type to 14.2.

12.9: Corrected evaluation of case clauses so that case expression is the receiver of ==. Revised specification to correctly deal with blank statements in case clauses.

12.15: Fixed bug in **assert** specification that could lead to puzzlers.

14.2: Consolidated definition of malformed types.

14.5: Revised semantics to account for function emulation.

1.2.6 Changes Since Version 0.07

5: Static variables are lazily initialized, but need not be constants. Orthogonal notion of constant variable introduced.

7.1.1: Added **equals** operator as part of revised == treatment.

7.6.1: Initializing formals have the same type as the field they correspond to.

7.8: Static variable getter rules revised to deal with lazy initialization.

11: Modified syntax to support cascaded method invocations.

11.1: Removed support for + operator on Strings. Extended string constants to support certain cases of string interpolation. Revised constants to deal with constant variables.

11.5: Corrected definition of HEX_DIGIT_SEQUENCE. Support implicit concatenation of adjacent single line strings.

11.14.2: Centralized and corrected type rules for function invocation.

11.15: Moved rules for checking function/method invocations to 11.14.2. Added definition of cascaded method invocations.

11.17, 11.18: Updated noSuchMethod() call for getters and setters to conform to planned API.

11.19: Modified syntax to support cascaded method invocations.

11.22: Revised semantics for ==.

11.30: Removed **import**, **library** and **source** from list of built-in identifiers and added **equals**. Revised rules for evaluating identifiers to deal with lazy static variable initialization.

12.14: Fixed bug that allowed **continue** labeled on non-loops.

13: Revised syntax so no space is permitted between # and directives. Introduced show: combinator. Describe prefix: as a combinator. Added initial discussion of namespaces. Preclude string interpolation in arguments to directives.

1.2.7 Changes Since Version 0.08

7.1, 7.4: Abstract methods may specify default values.

8, The former section on interface methods (now removed): Interface methods may specify default values.

11.1: The ~/ operator can apply to doubles.

11.11: Refined rules regarding abstract class instantiation, allowing factories to be used.

12.9: **switch** statement specification revised.

11.8: **throw** may not throw **null**.

13.1: Imports introduce a scope separate from the library scope. Multiple libraries may share prefix.

14.3.1: Recursive typedefs disallowed.

1.2.8 Changes Since Version 0.09

3.1: Consolidated discussion of namespaces and scopes. Started to tighten up definitions of scopes.

7: Overriding of fields allowed.

7.1.1: **call** is no longer an operator.

The former section on variables in interfaces (now removed): Added specification of variable declarations in interfaces.

11.1: Static methods and top-level functions are compile-time constants.

11.5: Multiline strings can be implicitly concatenated and contain interpolated expressions.

11.32: Type cast expression added.

12.2: Map literals cannot be expression statements.

12.10, 11.8: Clarified type of stack trace.

13.2, 13.1: Added re-export facility.

1.2.9 Changes Since Version 0.10

3: Discuss reified runtime types.

3.1: Removed shadowing warnings. Allow overloading of -.

5: Centralized discussion of implicit getters and setters.

6.4: External functions added.

7: Abstract classes must now be declared explicitly.

7.1.1: Eliminate **negate** in favor of overloaded unary minus. Eliminate **equals** in favor of operator ==.

7.3: Setter syntax no longer includes = sign after the name.

7.4: Clarify that getters and setters may be abstract. Eliminate **abstract** modifier for abstract members. Added static warning if abstract class has abstract member.

7.5: Instance variables can be initialized to non-constants. Moved discussion of implicit getters and setters to 5.

7.6.1: Clarify that finals can only be set once.

7.6.2: Added redirecting factories.

7.8: Moved discussion of implicit getters and setters to 5.

8: Eliminated interface declarations.

10: Added metadata.

11.1: Refined definition constant identity/caching.

11.8: Stack traces moved to 12.10.

11.11.1: Creating an instance via **new** using an undefined class or constructor is a dynamic error and a static warning, not a compile-time error. Evaluation rules to allow instance variables with non-constant initializers. Instantiating an abstract class via a generative constructor is now a dynamic error.

11.22: Replaced === with built-in function identical(). Equality does not special case identity.

11.30: Type names have meaning as expressions. **assert** is not a built-in identifier anymore, but **export**, **import**, **library** and **part** are. Removed warning on use of built-in identifiers as variable/function names.

11.32: type cast accepts **null**.

11.33: Added operation to determine if optional argument was actually passed.

12.4: Added section on local functions.

12.9: Allow switch on compile-time constants of any one type.

12.10: Revised syntax for **catch** clauses. Specification of stack traces moved from 11.8.

12.11: Made explicit that checked mode tests returns.

12.15: **assert** is now a reserved word.

13: Revised library syntax and semantics.

14.6: Renamed **Dynamic** to **dynamic**.

15.1.1: **assert** is a reserved word.

2 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 (11.30) 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 (the *not* combinator of PEGs) is represented by prefixing an element of a production with a tilde.

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. The longest applicable lexical token will always be chosen during the parsing of a Dart program.

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

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.

The notation $[x_1, \ldots, x_n/y_1, \ldots, y_n]E$ denotes a copy of E in which all occurrences of $y_i, 1 \leq i \leq n$ have been replaced with x_i.

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.

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 runtime types respectively.

3 Overview

Dart is a class-based, single-inheritance, pure object-oriented programming language. Dart is optionally typed (14) 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 (14.1) have absolutely no effect on execution. In checked mode, assignments are dynamically checked, and certain violations of the type system raise 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 runtime 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 runtime 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 runtime 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* (13). 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.

3.1 Scoping

A *namespace* is a mapping of identifiers to 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 field with the same name in Dart. Similarly one cannot declare a top-level function with the same name as a library variable or 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.

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 (discretely). 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.

3.2 Privacy

Dart supports two levels of privacy: *public* and *private*. A declaration is *private* iff its name 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 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.

3.3 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 (11.15.5). No state is ever shared between isolates. Isolates are created by spawning (11.12).

4 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 a compile-time error occurs within the code of a running isolate A, A is immediately suspended.

Typically, 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 embedders 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.

Dynamic type errors are type errors reported in checked mode.

Run-time errors are exceptions raised during execution. Whenever we say that an exception *ex* is *raised* or *thrown*, we mean that a throw expression (11.8) of the form: **throw** *ex;* was implicitly evaluated. 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.

5 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 initial value **null** (11.2).

A *static variable* is a variable that is not associated with a particular instance, but rather with an entire library or class. A variable declared at the top-level of a library is referred to as either a *library variable* or simply a top-level variable.

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 sections 5.1 and 11.30.

Current implementations give a compile-time error if static variables are not initialized with compile-time constants. These restrictions will be lifted in time.

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 declaration includes the modifier **final**.

It is a compile-time error 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 final instance variable that has is initialized by means of an initializing formal of a constructor is also initialized elsewhere in the same constructor.

It is a compile-time error if a library, static or local variable v is final and v is not initialized at its point of declaration.

Attempting to assign to a final variable elsewhere will cause a NoSuchMethodError to be thrown, because no setter is defined for it. The assignment will also give rise to a static warning for the same reason.

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 (11.1) or a compile-time error occurs.

If a variable declaration does not explicitly specify a type, the type of the declared variable(s) is **dynamic**, the unknown type (14.6).

A library variable is implicitly static. It is a compile-time error to preface a top-level variable declaration with the built-in identifier (11.30) **static**.

Variable declarations always induce implicit getters. Mutable variables also induce implicit setters. 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 (and if necessary a setter) into the top level scope of the enclosing library. A static class variable introduces a static getter (and if necessary a static setter) into the immediately enclosing class. An instance variable introduces an instance getter (and if necessary an instance setter) into the immediately enclosing class. Local variables cause a getter (and

if necessary a setter) to be added to the innermost enclosing scope at the point following the local variable declaration.

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 (7.2) with signature

T **get** v

whose invocation evaluates as described below (5.1).

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 (5.1).

A non-final variable declaration of the form $T\ v$; or the form $T\ v = e$; always induces an implicit setter function (7.3) 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.

5.1 Evaluation of Implicit Variable Getters

Let d be the declaration of a static or library variable v. If d is a local or 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:

- If d is of one of the forms **var** $v = e$; , **static** $T\ v = e$; , **static final** $v = e$; or **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 referenced, a CyclicInitializationError is thrown. If the evaluation succeeded yielding an object o, let $r = o$, otherwise let $r =$ **null**. In any case, r is stored into v. The result of executing the getter is r.
- If d is of one of the forms **const** $v = e$; or **const** $T\ v = e$; the result of the getter is the value of the compile time constant e. Otherwise
- The result of executing the getter method is the value stored in v.

6 Functions

Functions abstract over executable actions.

functionSignature:
metadata returnType? identifier formalParameterList
;

returnType:
  void |
 type
;

functionBody:
 '=>' expression ';' |
 block
;

block:
  '{' statements '}'
;

Functions include function declarations (6.1), methods (7.1, 7.7), getters (7.2), setters (7.3), constructors (7.6) and function literals (11.9).

All functions have a signature and a body. The signature describes the formal parameters of the function, and possibly its name and return type. The body is a block statement (12.1) containing the statements (12) executed by the function. A function body of the form $=> e$ is equivalent to a body of the form {**return** e;}.

If the last statement of a function is not a return statement, the statement **return null**; is implicitly appended to the function body.

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. See further discussion in section 12.11.

6.1 Function Declarations

A *function declaration* is a function that is not a method, getter, setter or 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, followed by a signature and body.

The scope of a library function is the scope of the enclosing library. The scope of a local function is described in section 12.4. In both cases, the name of the function is in scope in the formal parameters scope of the function.

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

6.2 Formal Parameters

Every function includes a *formal parameter list*, which consists of a list of required positional parameters (6.2.1), followed by any optional parameters. The optional parameters may be specified either as a set of named parameters or as a list of positional parameters, but not both.

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

The body of a function introduces a new scope known as the function's *body scope*. The body 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 (5).

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

normalFormalParameters:
  normalFormalParameter (', ' normalFormalParameter)*
;

optionalFormalParameters:
  optionalPositionalFormalParameters |
  namedFormalParameters
;

optionalPositionalFormalParameters:
  '[' defaultFormalParameter (', ' defaultFormalParameter)* ']'
;
namedFormalParameters:
  '{' defaultNamedParameter (', ' defaultNamedParameter)* '}'
;

6.2.1 Required Formals

A *required formal parameter* is a simple variable declaration (5).

normalFormalParameter:
  functionSignature |
  fieldFormalParameter |

simpleFormalParameter
;

simpleFormalParameter:
declaredIdentifier |
metadata identifier
;

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

6.2.2 Optional Formals

Optional parameters may be specified and provided with default values.

defaultFormalParameter:
normalFormalParameter ('=' expression)?
;

defaultNamedParameter:
normalFormalParameter (':' expression)?
;

It is a compile-time error if the default value of an optional parameter is not a compile-time constant (11.1). 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.

6.3 Type of a Function

If a function does not declare a return type explicitly, its return type is **dynamic**.

Let F be a function with 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 $(T_1, \ldots, T_n) \rightarrow T_0$.

Let F be a function with 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 $(T_1, \ldots, T_n, [T_{n+1}\ p_{n+1}, \ldots, T_{n+k}\ p_{n+k}]) \rightarrow T_0$.

Let F be a function with 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 $(T_1, \ldots, T_n, \{T_{n+1}\ p_{n+1}, \ldots, T_{n+k}\ p_{n+k}\}) \rightarrow T_0$.

6.4 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 (13), a method (7.1, 7.7), a getter (7.2), a setter (7.3) or a non-redirecting constructor (7.6.1). External functions are introduced via the built-in identifier **external** (11.30) 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 runtime), 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 raise a NoSuchMethodError or some subclass thereof.

The actual syntax is given in sections 7 and 13 below.

7 Classes

A *class* defines the form and behavior of a set of objects which are its *instances*.

classDefinition:
metadata **abstract**? **class** identifier typeParameters? superclass? interfaces?
'{' metadata classMemberDefinition* '}'
;

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** ?)| **abstract**)? getterSignature |
((**external static**?)| **abstract**)? setterSignature |
external? operatorSignature |
((**external static**?)| **abstract**)? functionSignature |
static (**final** | **const**) type? staticFinalDeclarationList |
const type? staticFinalDeclarationList |
final type? initializedIdentifierList |
static? (**var** | type) initializedIdentifierList
;

staticFinalDeclarationList:
:
staticFinalDeclaration (‘’ staticFinalDeclaration)*
;

staticFinalDeclaration:
identifier ‘=’ expression
;

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.

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 (7.10).

An *abstract class* is a class that is explicitly declared with the **abstract** modifier .

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 (7.10). 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.

What about a final instance variable and a setter? This case is illegal as well. If the setter is setting the variable, the variable should not be final.

It is a compile-time error if a class has an instance member and a static member with the same name.

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; // compile-time error: B has two variables with same name i
  static j; // compile-time error: B has two variables with same name j
   static f(x) => 3; // compile-time error: static method conflicts with instance method
```

It is a compile time error if a class C declares a member with the same name as C.

7.1 Instance Methods

Instance methods are functions (6) 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 compile-time error if an instance method m_1 overrides (7.9.1) an instance member m_2 and m_1 has a different number of required parameters than m_2. It is a compile-time error if an instance method m_1 overrides an instance member m_2 and m_1 has fewer optional positional parameters than m_2. It is a compile-time error 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 specifies a different default value for p. It is a static warning if a class C declares an instance method named n and an accessible static member named n is declared in a superclass of C.

7.1.1 Operators

Operators are instance methods with special names.

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

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

unaryOperator:
'!' |
'~'
;

binaryOperator:
multiplicativeOperator |
additiveOperator |
shiftOperator |
relationalOperator |
equalityOperator |
bitwiseOperator
;

prefixOperator:
'-' |
negateOperator
;

An operator declaration is identified using the built-in identifier (11.30) **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**.

7.2 Getters

Getters are functions (6) 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 instance getters of a class C are those instance getters declared by C and the instance getters inherited by C from its superclass. The static getters of a class C are those static getters declared by C.

It is a compile-time error if a class has both a getter and a method with the same name. This restriction holds regardless of whether the getter is defined explicitly or implicitly, or whether the getter or the method are inherited or not.

This implies that a getter can never override a method, and a method can never override a getter or field.

It is a static warning if a getter m_1 overrides (7.9.1) a getter 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 declares a static getter named v and also has a non-static setter named $v =$. It is a static warning if a class C declares an instance getter named v and an accessible static member named v or $v =$ is declared in a superclass of C.

7.3 Setters

Setters are functions (6) 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 **dynamic**.

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.

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 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 (7.9.1) 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. It is a static warning if a class declares a static setter named $v =$ and also has a non-static member named v. It is a static warning if a class C declares an instance setter named $v =$ and an accessible static member named $v =$ or v is declared in a superclass of C.

7.4 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.

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 makes abstract classes more concise, making abstract classes an attractive substitute for interface declarations.

Invoking an abstract method, getter or setter always results in a run-time error. This must be NoSuchMethodError or an instance of a subclass of NoSuchMethodError, such as AbstractMethodError.

These errors are ordinary objects and are therefore catchable.

It is a static warning if an abstract member is declared or inherited in a concrete class.

7.5 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.

7.6 Constructors

A *constructor* is a special member that is used in instance creation expressions (11.11) to produce objects. Constructors may be generative (7.6.1) or they may be factories (7.6.2).

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 *id* is the name of a member declared in the immediately enclosing class. It is a compile-time error if the name of a constructor is not a constructor name.

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

7.6.1 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 (6.2) 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 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 field named id in the immediately enclosing class. It is a static warning if the static type of id is not assignable to T_{id}.

Using an initializing formal **this**.id in a formal parameter list does not introduce a formal parameter name into the scope of the constructor. However, the initializing formal does effect the type of the constructor function exactly as if a formal parameter named id of the same type were introduced in the same position.

Initializing formals are executed during the execution of generative constructors detailed below. Executing an initializing formal **this**.id causes the field id of

the immediately surrounding class to be assigned the value of the corresponding actual parameter.

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 (11.11).

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:
  ':' superCallOrFieldInitializer (', ' superCallOrFieldInitializer)*
;

superCallOrFieldInitializer:
  super arguments |
  super '.' identifier arguments |
  fieldInitializer
;

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

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.

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 compile-time error 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 is always done with respect to a set of bindings for its formal parameters and with **this** bound to a fresh instance i and the type parameters of the immediately enclosing class bound to a set of actual type arguments $V_1, \ldots, V_m$.

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

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 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 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:

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

After all the initializers have completed, the body of k is executed in a scope where **this** is bound to i. Execution of the body begins with execution of the body of the superconstructor with **this** bound to i, the type parameters of the immediately enclosing class bound to a set of actual type arguments $V_1, \ldots, V_m$ and the formal parameters bindings determined by the argument list of the superinitializer of k.

*This process ensures that no uninitialized final field is ever seen by code. Note that **this** is not in scope on the right hand side of an initializer (see 11.10) 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.*

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 the object denoted by **this** is bound to o. In checked mode, it is a dynamic type error if o is not **null** and the interface of the class of o is not a subtype of the static type of the field v.

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

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.

Let C be the class in which the superinitializer appears and let S be the superclass of C. If S is generic (9), let $U_1, , \ldots, U_m$ be the actual type arguments passed to S in the superclass clause of C.

Then, the initializer list of the constructor S (respectively $S.id$) is executed with respect to the bindings that resulted from the evaluation of the argument list, with **this** bound to the current binding of **this**, and the type parameters (if any) of class S bound to the current bindings of $U_1, , \ldots, U_m$.

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

7.6.2 Factories

A *factory* is a constructor prefaced by the built-in identifier (11.30) **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 static warning if $M.id$ is not a constructor name.

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 an object whose type is not a subtype of its actual (14.8.1) return type.

It seems useless to allow a factory to return null. 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.

It follows that if *type* or *type.id* are not defined, a dynamic error occurs, as with any other undefined constructor call.

It is a compile-time error if k has more optional positional parameters than k'. It is a compile-time error if k has a named parameter that is not declared by k'.

This implies that the arguments to k are always legal arguments to k'.

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. This is a deliberate decision, so that k can easily determine what arguments were actually passed by the caller

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 constructor to which calls are being redirected can determine whether any user arguments were explicitly passed.*
- *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 (7.6.3).

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*.

7.6.3 Constant Constructors

A *constant constructor* may be used to create compile-time constant (11.1) 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.

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.

The difference between a potentially constant expression and a compile-time constant expression (11.11.2) 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 (11.11.1), 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 (11.6), maps (11.7) and objects (11.11.2), 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 (11.11.2).*

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.

7.7 Static Methods

Static methods are functions 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.

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.

7.8 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.

7.9 Superclasses

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
- The superclass of C is Object.

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

superclass:
  extends type
;

It is a compile-time error if the **extends** clause of a class C includes a type expression that does not denote a class available in the lexical scope of C.

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

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 a superclass of C.

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

7.9.1 Inheritance and Overriding

A class C *inherits* any accessible instance members of its superclass that are not overridden by members declared in C.

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

Let C be a class declared in library L, with superclass S and let C declare an instance member m, and assume S declares an instance member m' with the same name as m. Then m *overrides* m' iff m' is accessible (3.2) to L, m has the same name as m' and neither m nor m' are fields.

Fields never override each other. The getters and setters induced by fields do.

Whether an override is legal or not is described elsewhere in this specification (see 7.1, 7.2 and 7.3).

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

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 (3.1). A field f introduces a getter f, and if it is not const or final, a setter $f =$ (7.5, 7.8). When we speak of members here, we mean accessible fields, getters, setters and methods (7).
2. You cannot have two members with the same name in the same class - be they declared or inherited (3.1, 7).
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 (7.1, 7.2, 7.3).
5. It is a warning if you have a static setter $v =$, and an instance member v (7.3).
6. It is a warning if you have a static getter v and an instance setter $v =$ (7.2).
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 (7.1, 7.2, 7.3) (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 compile time error if they have a different number of required parameters (7.1).

10. If two members override each other, it is a compile time error if the overriding member has fewer optional positional parameters than the member being overridden (7.1).

11. If two members override each other, it is a compile time error if the overriding member does not have all the named parameters that the member being overridden has (7.1).

12. Setters, getters and operators never have optional parameters of any kind; it's a compile-time error (7.1.1, 7.2, 7.3).

13. It is a compile-time error if a member has the same name as its enclosing class (7).

14. A class has an implicit interface (7).

15. Interface members are not inherited by a class, but are inherited by its implicit interface. Interfaces have their own inheritance rules (8.1.1).

16. A member is abstract if it has no body and is not labeled **external** (7.4, 6.4).

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

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

19. It is a static warning and a dynamic error to call a non-factory constructor of an abstract class (11.11.1).

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 a type is a subtype of all the others. If there is no such member, then:

    - A static warning is given.

- If possible interface gets a member named m that has the same number of required parameters as all the members in the superinterfaces, the maximal number of optional positionals, and the superset of named parameters. The types of these are all **dynamic**. If this is impossible (because the superinterface members differ in their number of required parameters) then no member m appears in the interface.

(8.1.1)

23. Rules 8 applies to interfaces as well as classes (8.1.1).

24. It is a static warning if a concrete class does not have an implementation for a method in any of its superinterfaces (7.10).

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 (7.6).

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 non-abstract class inherits an abstract method.

7.10 Superinterfaces

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

interfaces:
  implements typeList
;

It is a compile-time error if the implements clause of a class C includes a type expression that does not denote a class available in the lexical scope of C.

In particular, one cannot inherit from a type parameter.

It is a compile-time error if the implements clause of a class includes type **dynamic**. It is a compile-time error if a type T appears more than once in the implements clause of a class.

One might argue that it is harmless to repeat a type in this way, 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.

It is a static warning if the implicit interface of a non-abstract class C includes an instance member m and C does not declare or inherit a corresponding instance member m.

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

8 Interfaces

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

8.1 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.

It is a compile-time error if an interface is a superinterface of itself.

8.1.1 Inheritance and Overriding

Let I be the implicit interface of a class C. I inherits any instance members of its superinterfaces that are not overridden by members declared in C.

However, if the above rule 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 the static types $T_1, \ldots, T_k$ of the members $m_1, \ldots, m_k$ are not identical, then there must be a member m_x such that $T_x <: T_i, 1 \leq x \leq k$ for all $i \in 1..k$, or a static type warning occurs. The member that is inherited is m_x, if it exists; otherwise:

- If all of $m_1, \ldots, m_k$ have the same number r of required parameters and the same set of named parameters s, then let

  $h = max(numberOfOptionalPositionals(m_i)), i \in 1..k$.

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

- Otherwise none of the members $m_1, \ldots, m_k$ is inherited.

The only situation where the runtime 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.

9 Generics

A class declaration (7) or type alias (14.3.1) G may be *generic*, that is, G may have formal type parameters declared. A generic declaration induces a family of declarations, one for each set of actual type parameters provided in the program.

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 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. It is however, a compile-time error to refer to a type parameter from within a static member.

The restriction is necessary since a type variable has no meaning in the context of a static member, because statics are shared among all instantiations of a generic. 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:

```
interface 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 (11.11).
- A type parameter cannot be used as a superclass or superinterface (7.9, 7.10, 8.1).

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

10 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 either a reference to a compile-time constant variable, or 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 runtime 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 runtime 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, class, typedef, type parameter, constructor, factory, function, field, parameter, or variable declaration and before an import or export directive.

11 Expressions

An *expression* is a fragment of Dart code that can be evaluated at run time to yield a *value*, which is always an object. Every expression has an associated static type (14.1). Every value has an associated dynamic type (14.2).

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

expressionWithoutCascade:
assignableExpression assignmentOperator expressionWithoutCascade |
conditionalExpression |
throwExpressionWithoutCascade
;

expressionList:
expression (‘, ’ expression)*
;

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

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

11.1 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 (11.3).

- A literal boolean (11.4).
- A literal string (11.5) where any interpolated expression (11.5.1) is a compile-time constant that evaluates to a numeric, string or boolean value or to **null**. *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.*
- **null** (11.2).
- A reference to a static constant variable(5).
- An identifier expression that denotes a constant variable, a class or a type parameter.
- A constant constructor invocation (11.11.2).
- A constant list literal (11.6).
- A constant map literal (11.7).
- A simple or qualified identifier denoting a top-level function (6) or a static method (7.7).
- 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 that returns true iff its two arguments are the same object.
- 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 that evaluate to a numeric, string or boolean value or to **null**.
- 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, $e_1 | e_2$, $e_1 >> e_2$ or $e_1 << e_2$, where e, e_1 and e_2 are constant expressions that evaluate to an integer value or to **null**.
- 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, $e_1 > e_2$, $e_1 < e_2$, $e_1 >= e_2$, $e_1 <= e_2$ or e_1 % e_2, where e, e_1 and e_2 are constant expressions that evaluate to a numeric value or to **null**.

It is a compile-time error if evaluation of a compile-time constant would raise an exception.

The above is not dependent on program control-flow. The mere presence of a 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 raise 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.*

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 |
mapLiteral |
listLiteral
;

Let c_1 and c_2 be a pair of constants. Then identical(c_1, c_2) iff:

- c_1 evaluates to either **null**, a constant map, an constant list, an instance of bool, num, String or Type and $c_1 == c_2$, OR
- c_1 and c_2 are constant objects of the same class C and for each member field of c_1 is identical to the corresponding field of c_2.

where identical() is the predefined dart function that returns **true** iff its two arguments are the same object.

11.2 Null

The reserved word **null** denotes 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 attempt to extend or implement Null. Invoking a method on **null** yields a NoSuchMethodError or its subclass NullPointerException unless the method is explicitly implemented by class Null.

The static type of **null** is $\bot$.

The decision to use $\bot$ instead of Null *allows* ***null*** *to be be assigned everywhere without complaint by the static checker.*

Here is one way in which one might implement class Null:

```
class Null {
        factory Null._()  throw "cannot be instantiated";
        noSuchMethod(InvocationMirror msg) {
            throw new NullPointerException();
        }
         /* other methods, such as ==  */
}
```

11.3 Numbers

A *numeric literal* is either a decimal or hexadecimal integer of arbitrary size, or a decimal double.

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
;

If a numeric literal begins with the prefix '0x' or '0X', it denotes the hexadecimal integer represented by the part of the literal following '0x' (respectively '0X'). Otherwise, if the numeric literal does not include a decimal point it denotes a decimal integer. Otherwise, the numeric literal denotes a 64 bit double precision floating point number as specified by the IEEE 754 standard.

An integer literal or a literal double may optionally be prefixed by a plus sign (+). This has no semantic effect.

There is no unary plus operator in Dart. However, we allow a leading plus in decimal numeric literals for clarity and to provide some compatibility with Javascript.

Integers are not restricted to a fixed range. Dart integers are true integers, not 32 bit or 64 bit or any other fixed range representation. Their size is limited only by the memory available to the implementation.

It is a compile-time error for a class to attempt to extend or implement int. It is a compile-time error for a class to attempt to extend or implement double.

It is a compile-time error for any type other than the types int and double to attempt to extend or implement num.

An *integer literal* is either a hexadecimal integer literal or a decimal integer literal. The static type of an integer literal is int. A *literal double* is a numeric literal that is not an integer literal. The static type of a literal double is double.

11.4 Booleans

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

booleanLiteral:
true |
false
;

Both **true** and **false** implement the built-in class bool. It is a compile-time error for a class to attempt to extend or implement bool.

It follows that the two boolean literals are the only two instances of bool.

The static type of a boolean literal is bool.

11.4.1 Boolean Conversion

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

```
(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).*

11.5 Strings

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

This decision was made for compatibility with web browsers and Javascript. Earlier version 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 either a sequence of single line strings or a multiline string.

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 single line strings are implicitly concatenated to form a single string literal, and so are adjacent multiline strings, but the two forms may not be mixed.

Here is an example

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

Early versions of Dart used the operator + for string concatenation. However, this was dropped, as it leads to 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’. Instead, the recommended Dart idiom is to use string interpolation.

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

String interpolation work 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 whitespace characters then that line is ignored, including the new line at its end.

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 scalar value 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 scalar value.
- $ 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.

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

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

;

NEWLINE:
\ n |
\ r
;

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

11.5.1 String Interpolation

It is possible to embed expressions within non-raw string literals, such that the 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 ${id}. An interpolated string 's_1{e}s_2$' is equivalent to the concatenation of the strings 's_1', e.toString() and 's_2'. Likewise an interpolated string "s_1{e}s_2$" is equivalent to the concatenation of the strings "s_1", e.toString() and "s_2".

11.6 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 runtime 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 (11.1) and therefore evaluated at compile-time. Otherwise, it is a *run-time list literal* and it is evaluated at run-time.

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 that 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, yielding objects $o_1 \ldots o_n$.
- A fresh instance (7.6.1) a, of size n, that 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 runtime 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 runtime, the runtime 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****.*

11.7 Maps

A *map literal* denotes a map from strings to objects.

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

mapLiteralEntry:
stringLiteral ':' expression
;

A *map literal* consists of zero or more entries. Each entry has a *key*, which is a string literal, and a *value*, which is an object.

If a map literal begins with the reserved word **const**, it is a *constant map literal* which is a compile-time constant (11.1) and therefore evaluated at compile-time. Otherwise, it is a *run-time map literal* and it is evaluated at run-time.

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 type arguments of a constant map literal include a type parameter.

The value of a constant map literal $\textbf{const} < String, V > \{k_1 : e_1 \ldots k_n : e_n\}$ is an object m that implements the built-in class $Map < String, V >$. 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 $\textbf{const} < String, \textbf{dynamic} > \{k_1 : e_1 \ldots k_n : e_n\}$.

Let $map_1 = \textbf{const} < String, V > \{k_{11} : e_{11} \ldots k_{1n} : e_{1n}\}$ and $map_2 = \textbf{const} < String, U > \{k_{21} : e_{21} \ldots k_{2n} : e_{2n}\}$ 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 $V = U$ then identical(map_1, map_2).

In other words, constant map literals are canonicalized.

A runtime map literal $< String, V > \{k_1 : e_1 \ldots k_n : e_n\}$ is evaluated as follows:

- First, the expressions $e_1 \ldots e_n$ are evaluated in left to right order, yielding objects $o_1 \ldots o_n$.
- A fresh instance (7.6.1) m that implements the built-in class $Map < String, V >$ is allocated.
- Let u_i be the value of the string literal specified by k_i. The operator []= is invoked on m with first argument u_i and second argument $o_i, i \in 0..n$.
- The result of the evaluation is m.

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

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**$< String, V > \{k_1 : e_1 \ldots k_n : e_n\}$ or the form $< String, V > \{k_1 : e_1 \ldots k_n : e_n\}$ is $Map < String, V >$. The static type 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 $Map < String, \textbf{dynamic} >$. It is a compile-time error if the first type argument to a map literal is not String.

11.8 Throw

The *throw expression* is used to raise or re-raise an exception.

throwExpression:
  throw expression?
;

throwExpressionWithoutCascade:
  throw expressionWithoutCascade?
;

The *current exception* is the last unhandled exception thrown.

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

The expression e is evaluated yielding a value v. If v evaluates to **null**, then a NullPointerException is thrown. Otherwise, control is transferred to the nearest dynamically enclosing exception handler (12.10), with the current exception set to v.

There is no requirement that the expression e evaluate to a special kind of exception or error object.

Evaluationtion of an expression of the form **throw**; proceeds as follows: Control is transferred to the nearest innermost enclosing exception handler (12.10).

No change is made to the current exception.

It is a compile-time error if an expression of the form **throw**; is not enclosed within an on-catch clause.

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

11.9 Function Expressions

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

functionExpression:
formalParameterList functionExpressionBody
;

functionExpressionBody:
'=>' expression |
block
;

A function literal implements the built-in class Function.

The static type of a function literal of the form $(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1} = d_1, \ldots, T_{n+k}\ x_{n+k} = d_k])$ => e is $(T_1 \ldots, T_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) \rightarrow T_0$, where T_0 is the static type of e. In any case where $T_i, 1 \leq i \leq n+k$, is not specified, it is considered to have been specified as **dynamic**.

The static type of a function literal of the form $(T_1\ a_1, \ldots, T_n\ a_n, \{T_{n+1}\ x_{n+1} : d_1, \ldots, T_{n+k}\ x_{n+k} : d_k\})$ => e is $(T_1 \ldots, T_n, \{T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}\}) \rightarrow T_0$, where T_0 is the static type of e. In any case where $T_i, 1 \leq i \leq n+k$, is not specified, it is considered to have been specified as **dynamic**.

The static type of a function literal of the form $(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1} = d_1, \ldots, T_{n+k}\ x_{n+k} = d_k])\{s\}$ is $(T_1 \ldots, T_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) \rightarrow$ **dynamic**. In any case where $T_i, 1 \leq i \leq n+k$, is not specified, it is considered to have been specified as **dynamic**.

The static type of a function literal of the form $(T_1\ a_1, \ldots, T_n\ a_n, \{T_{n+1}\ x_{n+1} : d_1, \ldots, T_{n+k}\ x_{n+k} : d_k\})\{s\}$ is $(T_1 \ldots, T_n, \{T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}\}) \rightarrow$ **dynamic**. In any case where $T_i, 1 \leq i \leq n+k$, is not specified, it is considered to have been specified as **dynamic**.

11.10 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 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.

11.11 Instance Creation

Instance creation expressions invoke constructors to produce instances.

It is a compile-time error if any of the type arguments to a constructor of a generic type invoked by a new expression or a constant object expression do not denote types in the enclosing lexical scope. It is a compile-time error if a constructor of a non-generic type invoked by a new expression or a constant object expression is passed any type arguments. It is a compile-time error if a constructor of a generic type with n type parameters invoked by a new expression or a constant object expression is passed m type arguments where $m \neq n$.

It is a static type warning if any of the type arguments to a constructor of a generic type G invoked by a new expression or a constant object expression are not subtypes of the bounds of the corresponding formal type parameters of G.

11.11.1 New

The *new expression* invokes a constructor (7.6).

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** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$. It is a static warning if T is not a class accessible in the current scope, optionally followed by type arguments.

If T is not a class 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 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 (14.8) $S < U_1, \ldots, U_m >$, let $R = S$. It is a compile-time error if S is not a generic (9) type with m type parameters. 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. Finally, if R is generic but T is not a parameterized type, then for $i \in 1..m$, let $V_i =$ **dynamic**, otherwise let $V_i = U_i$.

Evaluation of e proceeds as follows:

If q is a constructor of an abstract class then an AbstractClassInstantiationError is thrown.

If T is not a class accessible in the current scope, a dynamic error occurs. Otherwise, if q is not defined or not accessible, a NoSuchMethodError is thrown. Otherwise, if q is a generative or redirecting constructor (7.6.1), then:

Let T_i be the type parameters of R (if any) and let B_i be the bound of $T_i, 1 \leq i \leq m$. In checked mode, it is a dynamic type error if V_i is not a subtype of $[V_1, \ldots, V_m/T_1, \ldots, T_m]B_i, i \in 1..m$.

A fresh instance (7.6.1), i, of class R is allocated. For each instance variable f of i, if the variable declaration of f has an initializer expression e_f, then e_f is evaluated to an object o_f and f is bound to o_f. Otherwise f is bound to **null**.

Observe that **this** is not in scope in e_f. Hence, the initialization cannot depend on other properties of the object being instantiated.

Next, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. Then, q is executed with **this** bound to i, the type parameters (if any) of R bound to the actual type arguments $V_1, \ldots, V_m$ and the formal parameter bindings that resulted from the evaluation of the argument list. The result of the evaluation of e is i.

Otherwise, q is a factory constructor (7.6.2). Then:

Let T_i be the type parameters of R (if any) and let B_i be the bound of $T_i, 1 \leq i \leq m$. In checked mode, it is a dynamic type error if V_i is not a subtype of $[V_1, \ldots, V_m/T_1, \ldots, T_m]B_i, i \in 1..m$.

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 $[V_1, \ldots, V_m/T_1, \ldots, T_m](\textbf{new}\ c(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k}))$.

Otherwise, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. Then, the body of q is executed with respect to the bindings that resulted from the evaluation of the argument list and the type parameters (if any) of q bound to the actual type arguments $V_1, \ldots, V_m$ resulting in an object i. The result of the evaluation of e is i.

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).

11.11.2 Const

A *constant object expression* invokes a constant constructor (7.6.3).

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 is not a class accessible in the current scope, optionally followed by type arguments. It is a compile-time error if T includes any type parameters.

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.

If T is a parameterized type (14.8) $S < U_1, \ldots, U_m >$, let $R = S$. It is a compile-time error if S is not a generic (9) type with m type parameters. If T is not a parameterized type, let $R = T$. Finally, if R is generic but T is not a parameterized type, then for all $i \in 1..m$, let $V_i =$ **dynamic**, otherwise let $V_i = U_i$.

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 field f in i, and let v_{jf} be the value of the field f in j. If identical(v_{if}, v_{jf}) for all fields 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 fields and identical type arguments. 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 var x;
  const A(var 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 (7.4) but q is not a factory constructor.

11.12 Spawning an Isolate

Spawning an isolate is accomplished via what is syntactically an ordinary library call, invoking one of the functions spawnUri() or spawnFunction()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 4, the handling of a suspended isolate is the responsibility of the embedder.

11.13 Property Extraction

Property extraction allows for a member of an object to be concisely extracted

from the object. If o is an object, and if m is the name of a method member of o, then $o.m$ is defined to be equivalent to:

- $(r_1, \ldots, r_n, \{p_1 : d_1, \ldots, p_k : d_k\})\{$**return** $o.m(r_1, \ldots, r_n, p_1 : p_1, \ldots, p_k : p_k);\}$ if m has required parameters $r_1, \ldots, r_n$, and named parameters $p_1, \ldots, p_k$ with defaults $d_1, \ldots, d_k$.
- $(r_1, \ldots, r_n, [p_1 = d_1, \ldots, p_k = d_k])\{$**return** $o.m(r_1, \ldots, r_n, p_1, \ldots, p_k);\}$ if m has required parameters $r_1, \ldots, r_n$, and optional positional parameters $p_1, \ldots, p_k$ with defaults $d_1, \ldots, d_k$.

Otherwise, if m is the name of a getter (7.2) member of o (declared implicitly or explicitly) then $o.m$ evaluates to the result of invoking the getter.

Observations:

1. One cannot extract a getter or a setter.
2. One can tell whether one implemented a property via a method or via field/getter, which means that one has to plan ahead as to what construct to use, and that choice is reflected in the interface of the class.

11.14 Function Invocation

Function invocation occurs in the following cases: when a function expression (11.9) is invoked (11.14.4), when a method (11.15), getter (11.17) or setter (11.18) is invoked or when a constructor is invoked (either via instance creation (11.11), constructor redirection (7.6.1) 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** is bound. Once f has been determined, the formal parameters of f are bound to corresponding actual arguments. The body of f is then executed with the aforementioned bindings. Execution of the body terminates when the first of the following occurs:

- An uncaught exception is thrown.
- A return statement (12.11) immediately nested in the body of f is executed.
- The last statement of the body completes execution.

11.14.1 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
;

Evaluation of an actual argument list of the form $(a_1, \ldots, a_m, q_1 : a_{m+1}, \ldots, q_l : a_{m+l})$ proceeds as follows:

The arguments $a_1, \ldots, a_{m+l}$ are evaluated in the order they appear in the program, yielding objects $o_1, \ldots, o_{m+l}$.

Simply stated, an argument list consisting of m positional arguments and l named arguments is evaluated from left to right.

11.14.2 Binding Actuals to Formals

Let f be a function, let $p_1 \ldots p_n$ be the positional parameters of f and let $p_{n+1}, \ldots, p_{n+k}$ be the optional parameters declared by f.

An evaluated actual argument list $o_1 \ldots o_{m+l}$ derived from an actual argument list of the form $(a_1, \ldots, a_m, q_1 : a_{m+1}, \ldots, q_l : a_{m+l})$ is bound to the formal parameters of f as follows:

Again, we have an argument list consisting of m positional arguments and l named arguments. We have a function with h required parameters and k optional 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. You may not provide a given named argument 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 $m < h$, or $m > n$, a run-time error occurs. Furthermore, each $q_i, 1 \leq i \leq l$, must have a corresponding named parameter in the set $\{p_{n+1}, \ldots, p_{n+k}\}$ or a run time error occurs. 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.

If $l > 0$, then it is necessarily the case that $n = h$, because a method cannot have both optional positional parameters and named parameters.

In checked mode, it is a dynamic type error if o_i is not **null** and the actual type (14.8.1) 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 **null** and the actual type (14.8.1) 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, let S_i be the type of $p_i, i \in 1..n+k$ and let S_q be the type of the named parameter q of f. 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 $q_i, 1 \leq i \leq l$, must have a corresponding named parameter in the set $\{p_{n+1}, \ldots, p_{n+k}\}$ or a static warning occurs. It is a static warning if T_{m+j} may not be assigned to $S_{q_j}, j \in 1..l$.

11.14.3 Unqualified Invocation

An unqualified function invocation i has the form $id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$, where id is an identifier.

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

- 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 (11.14.4).
- Otherwise, if f_{id} is a static method of the enclosing class C, i is equivalent the static method invocation $C.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

Otherwise, i is equivalent to the ordinary method invocation $\textbf{this}.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

11.14.4 Function Expression Invocation

A function expression invocation i has the form $e_f(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$, where e_f is an expression. 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 property access expression, then i is treated as an ordinary method invocation (11.15.1). Otherwise:

A function expression invocation $e_f(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is equivalent to the ordinary method invocation $e_f.call(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

The implication of this definition, and the other definitions involving the method call(), is that user defined types can be used as function values provided iff they define a call() method. The method call() is special in this regard. The signature of the call() method determines the appropriate signature used when using the object via the built-in invocation syntax.

It is a static warning if the static type F of e_f may not be assigned to a function type. If F is not a function type, the static type of i is **dynamic**. Otherwise the static type of i is the declared return type of F.

11.15 Method Invocation

Method invocation can take several forms as specified below.

11.15.1 Ordinary Invocation

An ordinary method invocation i has the form $o.m(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$. Method invocation involves method lookup, defined next.

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 an instance method named m that is accessible to L, then that method is the result of the lookup. 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.

Evaluation of an ordinary method invocation i of the form $o.m(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ proceeds as follows:

First, the expression o is evaluated to a value v_o. Next, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated yielding actual argument objects $o_1, \ldots, o_{n+k}$. Let f be the result of looking up method m in v_o with respect to the current library L. 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 v_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 (11.16) m in v_o with respect to L. If the getter lookup succeeded, let v_g be the value of the getter invocation $o.m$. Then the value of i is the value of the method invocation $v_g.\mathbf{call}(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

If the getter lookup has also failed, then a new instance im of the predefined class InvocationMirror is created by calling its factory constructor with arguments m, **this**, $[o_1, \ldots, o_n]$ and $\{x_{n+1} : o_{n+1}, \ldots, x_{n+k} : o_{n+k}\}$. 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.

Notice that the wording carefully avoids re-evaluating the receiver o and the arguments a_i.

Let T be the static type of o. It is a static type warning if T does not have an accessible (3.2) instance member named m. 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 the static type of i is the declared return type of F.

11.15.2 Cascaded Invocations

A *cascaded method invocation* has the form *e..suffix* where *suffix* is a sequence of operator, method, getter or setter invocations.

cascadeSection:
'..' (cascadeSelector arguments*) (assignableSelector arguments*)*
(assignmentOperator expressionWithoutCascade)?

;

cascadeSelector:
'[' expression ']' |
identifier
;

A cascaded method invocation expression of the form *e..suffix* is equivalent to the expression (t){t.*suffix*; **return** t;}(*e*).

11.15.3 Static Invocation

A static method invocation i has the form $C.m(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

It is a static warning if C does not denote a class in the current scope. It is a static warning if C does not declare a static method or getter m.

Note that the absence of C or of $C.m$ is statically detectable. Nevertheless, we choose not to define this situation as an error. The goal is to allow coding to proceed in the order that suits the developer rather than eagerly insisting on consistency. The warnings are given statically at compile-time to help developers catch errors. However, developers need not correct these problems immediately in order to make progress.

As of this writing, the above situations are treated as compile time errors by the implementations. This will change over time.

Note the requirement that C *declare* the method. This means that static methods declared in superclasses of C cannot be invoked via C.

Evaluation of i proceeds as follows:

If C does not denote a class in the current scope, or if C does not declare a static method or getter m then a NoSuchMethodError is thrown.

Otherwise, evaluation proceeds as follows:

- If the member m declared by C is a getter, then i is equivalent to the expression $C.m.call(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.
- Otherwise, let f be the the method m declared in class C. Next, the argument list$(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. The body of f is then executed with respect to the bindings that resulted from the evaluation of the argument list. The value of i is the value returned after the body of f is executed.

It is a static type warning if the type F of $C.m$ may not be assigned to a function type. If F is not a function type, or if $C.m$ does not exist, the static type of i is **dynamic**. Otherwise the static type of i is the declared return type of F.

11.15.4 Super Invocation

A super method invocation i has the form

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

Evaluation of i proceeds as follows:

First, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated yielding actual argument objects $o_1, \ldots, o_{n+k}$. Let S be the superclass of the immediately enclosing class, and let f be the result of looking up method (11.15.1) m in S with respect to the current library L. 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 (11.16) m in S 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 value of the method invocation v_g.**call**$(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

If getter lookup has also failed, then a new instance im of the predefined class InvocationMirror is created by calling its factory constructor with arguments m, **this**, $[o_1, \ldots, o_n]$ and $\{x_{n+1} : o_{n+1}, \ldots, x_{n+k} : o_{n+k}\}$. Then the method noSuchMethod() is looked up in S and invoked 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, in an instance variable initializer, a constructor initializer or in a static method or variable initializer.

It is a static type warning if S does not have an accessible (3.2) instance member named m. If $S.m$ exists, it is a static type warning if the type F of $S.m$ may not be assigned to a function type. If $S.m$ does not exist, or if F is not a function type, the static type of i is **dynamic**; otherwise the static type of i is the declared return type of F.

11.15.5 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.

11.16 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 an instance getter (respectively setter)

named m that is accessible to L, then that getter (respectively setter) is the result of the lookup. 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.

11.17 Getter Invocation

A getter invocation provides access to the value of a property.

Evaluation of a getter invocation i of the form $e.m$ proceeds as follows:

First, the expression e is evaluated to an object o. Then, the getter function (7.2) m is looked up (11.16) in o with respect to the current library, and its body is executed with **this** bound to o. The value of the getter invocation expression 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 InvocationMirror is created, such that :

- im.isGetter evaluates to **true**.
- im.memberName evaluates to 'm'.
- im.arguments.positionalArguments evaluates to the value of [].
- im.arguments.namedArguments evaluates to the value of {}.

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.

Let T be the static type of e. It is a static type warning if T does not have a getter named m. The static type of i is the declared return type of $T.m$, if $T.m$ exists; otherwise the static type of i is **dynamic**.

Evaluation of a getter invocation i of the form $C.m$ proceeds as follows:

If there is no class C in the enclosing lexical scope of i, or if C does not declare, implicitly or explicitly, a getter named m, then a NoSuchMethodError is thrown. Otherwise, the getter function $C.m$ is invoked. The value of i is the result returned by the call to the getter function.

It is a static warning if there is no class C in the enclosing lexical scope of i, or if C does not declare, implicitly or explicitly, a getter named m. The static type of i is the declared return type of $C.m$ if it exists or **dynamic** otherwise.

As of this writing, implementations treat the above situation as a compile-time error.

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.

11.18 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:

If there is no declaration d with name $v =$ in the lexical scope enclosing a, then:

- If a occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer, evaluation of a causes a NoSuchMethodError to be thrown.
- Otherwise, the assignment is equivalent to the assignment **this**.$v = e$.

Otherwise, let d be the innermost declaration whose name is v, if it exists.

If d is the declaration of a local or library variable, the expression e is evaluated to an object o. Then, the variable v is bound to o. The value of the assignment expression is o.

Otherwise, if d is the declaration of a static variable in class C, then the assignment is equivalent to the assignment $C.v = e$.

Otherwise, the assignment is equivalent to the assignment **this**.$v = e$.

In checked mode, it is a dynamic type error if o is not **null** and the interface of the class of o is not a subtype of the actual type (14.8.1) 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 of the form $C.v = e$ proceeds as follows:

The expression e is evaluated to an object o. If there is no class C in the enclosing lexical scope of the assignment, or if C does not declare, implicitly or explicitly, a setter $v =$, then a NoSuchMethodError is thrown. Otherwise, the setter $C.v =$ is invoked with its formal parameter bound to o. The value of the assignment expression is o.

It is a static warning if there is no class C in the enclosing lexical scope of the assignment, or if C does not declare, implicitly or explicitly, a setter $v =$.

As of this writing, implementations treat the above situation as a compile-time error.

In checked mode, it is a dynamic type error if o is not **null** and the interface of the class of o is not a subtype of the static type of $C.v$.

It is a static type warning if the static type of e may not be assigned to the static type of $C.v$. The static type of the expression $C.v = e$ is the static type of e.

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 (11.16) in o_1 with respect to the current library, and its body 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 InvocationMirror is created, such that :

- im.isSetter evaluates to **true**.
- im.memberName evaluates to 'v='.
- im.arguments.positionalArguments evaluates to $[o_2]$.
- im.arguments.namedArguments evaluates to the value of {}.

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 **null** 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 $v =$. It is a static type warning if the static type of e_2 may not be assigned to T. 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 $e_1[e_2] = e_3$ is equivalent to the evaluation of the expression (a, i, e){a.[]=(i, e); **return** e; } (e_1, e_2, e_3). The static type of the expression $e_1[e_2] = e_3$ is the static type of e_3.

11.18.1 Compound Assignment

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. A compound assignment of the form $e_1.v$ *op* $= e_2$ is equivalent to $((x) => x.v = x.v$ *op* $e_2)(e_1)$ where x is a variable that is not used in e_2. A compound assignment of the form $e_1[e_2]$ *op*$=e_3$ is equivalent to $((a, i) => a[i] = a[i]$ *op* $e_3)(e_1, e_2)$ where a and i are a variables that are not used in e_3.

compoundAssignmentOperator:
'*=' |
'/=' |
'~/=' |
'%=' |
'+=' |
'-=' |
'<<=' |
'>>=' |
'&=' |
'^=' |

‘|=’
;

11.19 Conditional

A *conditional expression* evaluates one of two expressions based on a boolean condition.

conditionalExpression:
logicalOrExpression (‘?’ 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. In checked mode, it is a dynamic type error if o_1 is not of type bool. Otherwise, o_1 is then subjected to boolean conversion (11.4.1) 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.

It is a static type warning if the type of e_1 may not be assigned to bool. The static type of c is the least upper bound (14.8.2) of the static type of e_2 and the static type of e_3.

11.20 Logical Boolean Expressions

The logical boolean expressions combine boolean objects using the boolean conjunction and disjunction operators.

logicalOrExpression:
logicalAndExpression (‘||’ logicalAndExpression)*
;

logicalAndExpression:
bitwiseOrExpression (‘&&’ bitwiseOrExpression)*
;

A *logical boolean expression* is either a bitwise expression (11.21), 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; if e_1 evaluates to true, the result of evaluating b is true, otherwise e_2 is evaluated to an object o, which is then subjected to boolean conversion (11.4.1) 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; if e_1 does not evaluate to true, the result of evaluating b is false, otherwise e_2 is evaluated to an object o, which is then subjected to boolean conversion producing an object r, which is the value of b.

The static type of a logical boolean expression is bool.

11.21 Bitwise Expressions

Bitwise expressions invoke the bitwise operators on objects.

bitwiseOrExpression:
  bitwiseXorExpression (‘|’ bitwiseXorExpression)* |
  super (‘|’ bitwiseXorExpression)+
;

bitwiseXorExpression:
  bitwiseAndExpression (‘^’ bitwiseAndExpression)* |
  super (‘^’ bitwiseAndExpression)+
;

bitwiseAndExpression:
  equalityExpression (‘&’ equalityExpression)* |
  super (‘&’ equalityExpression)+
;

bitwiseOperator:
  ‘&’ |
  ‘^’ |
  ‘|’
;

A *bitwise expression* is either an equality expression (11.22), 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.

11.22 Equality

Equality expressions test objects for equality.

equalityExpression:
relationalExpression (equalityOperator relationalExpression)? |
super equalityOperator relationalExpression
;

equalityOperator:
'==' |
'!='
;

An *equality expression* is either a relational expression (11.23), 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 **null**, then ee evaluates to identical(o_1, o_2). Otherwise,
- 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 **null**, then ee evaluates to identical(**this**, o). Otherwise,
- 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.

11.23 Relational Expressions

Relational expressions invoke the relational operators on objects.

relationalExpression:
shiftExpression (typeTest | typeCast | relationalOperator shiftExpression)? |
super relationalOperator shiftExpression
;

relationalOperator:
'>=' |
'>' |
'<=' |
'<'
;

A *relational expression* is either a shift expression (11.24), 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**$.op(e_2)$.

11.24 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 (11.25), 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.

11.25 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 (11.26), 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**$.op(e_2)$.

11.26 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 (11.27), 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**$.op(e_2)$.

11.27 Unary Expressions

Unary expressions invoke unary operators on objects.

unaryExpression:
  prefixOperator unaryExpression |
  postfixExpression |
  unaryOperator **super** |
  '-' **super** |
  incrementOperator assignableExpression
;

A *unary expression* is either aa postfix expression (11.28), 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 $e \mathrel{+}= 1$. Evaluation of an expression of the form $--e$ is equivalent to $e \mathrel{-}= 1$.

An 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 **super**.$op()$.

11.28 Postfix Expressions

Postfix expressions invoke the postfix operators on objects.

postfixExpression:
  assignableExpression postfixOperator |
  primary selector*
;

postfixOperator:
  incrementOperator
;

selector:
  assignableSelector |
  arguments
;

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.

A postfix expression of the form v++, where v is an identifier, is equivalent to ()\{var r = v; v = r + 1; return r\}().

The above ensures that if v is a field, the getter gets called exactly once. Likewise in the cases below.

A postfix expression of the form $C.v$ ++ is equivalent to ()\{var r = $C.v$; $C.v$ = r + 1; return r\}().

A postfix expression of the form $e_1.v$++ is equivalent to (x)\{var r = x.v; x.v = r + 1; **return** r\}(e_1).

A postfix expression of the form $e_1[e_2]$++, is equivalent to (a, i)\{var r = a[i]; a[i] = r + 1; return r\}(e_1, e_2).

A postfix expression of the form v--, where v is an identifier, is equivalent to ()\{var r = v; v = r - 1; return r\}().

A postfix expression of the form $C.v$-- is equivalent to ()\{var r = $C.v$; $C.v$ = r - 1; return r\}().

A postfix expression of the form $e_1.v$-- is equivalent to (x)\{var r = x.v; x.v = r - 1; **return** r\}(e_1).

A postfix expression of the form $e_1[e_2]$--, is equivalent to (a, i)\{var r = a[i]; a[i] = r - 1; return r\}(e_1, e_2).

11.29 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 (argument* assignableSelector)+ |
  super assignableSelector |
  identifier
 ;

assignableSelector:
  '[' expression ']' |
  '.' identifier
 ;

An *assignable expression* is either:

- An identifier.

- An invocation of a method, getter (7.2) 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 (11.30).

An assignable expression of the form $e.id(a_1, \ldots, a_n)$ is evaluated as a method invocation (11.15).

An assignable expression of the form $e.id$ is evaluated as a getter invocation (11.17).

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 getter invocation.

An assignable expression of the form **super**$[e_2]$ is equivalent to the method invocation **super**.[](e_2).

11.30 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 |
  dynamic |
  export |
  external |
  factory |
  get |
  implements |
  import |
  library |

operator |
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 class, type parameter or type alias. It is a compile-time error to use a built-in identifier other than **dynamic** as a type annotation.

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. Fur-

thermore, types should begin with an uppercase letter (see the appendix) and so no clash should occur in any Dart user program anyway.

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. 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 class or type alias T, the value of e is the unique instance of class Type 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**. We are assured that **this** is well defined, because if we were in a static member the reference to T would be a compile-time error (9.)
- If d is a library variable then:
  - If d is of one of the forms **var** $v = e_i$; , T $v = e_i$; , **final** $v = e_i$; or **final** T $v = e_i$; and no value has yet been stored into v then the initializer expression e_i is evaluated. If, during the evaluation of e_i, the getter for v is referenced, a CyclicInitializationError is thrown. If the evaluation succeeded yielding an object o, let $r = o$, otherwise let $r =$ **null**. In any case, r is stored into v. The value of e is r.
  - If d is of one of the forms **const** $v = e_i$; or **const** T $v = e_i$; then the value id is the value of the compile-time constant e. Otherwise
  - e evaluates to the current binding of id.
- 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 defined by d.
- If d is the declaration of a static variable or static getter declared in class C, then e is equivalent to the getter invocation (11.17) $C.id$.
- If d is the declaration of a top-level getter, then e is equivalent to the getter invocation 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 aNoSuchMethod to be thrown.
- Otherwise, e is equivalent to the property extraction **this**.id.

11.31 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 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. Since the class Null is not exported by the core library, the latter will not occur in user code. The former two are useless, as is anything of the form e **is** Object or e **is dynamic**. Users should test for a null value directly rather than via type tests.

The is-expression e **is!** T is equivalent to !(e **is** T).

It is a run-time error if T does not denote a type available in the current lexical scope. It is a compile-time error if T is a parameterized type of the form $G < T_1, \ldots, T_n >$ and G is not a generic type with n type parameters.

Note, that, in checked mode, it is a dynamic type error if a malformed type is used in a type test as specified in 14.2.

It is a static warning if T does not denote a type available in the current lexical scope. The static type of an is-expression is bool.

11.32 Type Cast

The *cast expression* ensures that an object is a member of a type.

typeTest:
  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 the interface of the class of v is a subtype of T, the cast expression evaluates to v. Otherwise, if v is **null**, the cast expression evaluates to v. In all other cases, a CastError is thrown.

It is a run-time error if T does not denote a type available in the current lexical scope. It is a compile-time error if T is a parameterized type of the form $G < T_1, \ldots, T_n >$ and G is not a generic type with n type parameters.

Note, that, in checked mode, it is a dynamic type error if a malformed typed is used in a type cast as specified in 14.2.

It is a static warning if T does not denote a type available in the current lexical scope. The static type of a cast expression e **as** T is T.

11.33 Argument Definition Test

An *argument definition test* is an expression that tests whether a formal parameter is bound to an object explicitly passed to a method or function.

argumentDefinitionTest:
  '?' identifier
;

An argument definition test e of the form ?v evaluates to **true** iff the currently executing invocation of the function that declares v explicitly provided an argument for the formal parameter v; otherwise e evaluates to **false**.

It is a compile time error if v does not denote a formal parameter. The static type of an argument definition test is bool.

12 Statements

statements:
  statement*
;

statement:
  label* nonLabelledStatement
;

nonLabelledStatement:
  block |
  localVariableDeclaration |
  forStatement |
  whileStatement |
  doStatement |
  switchStatement |
  ifStatement |

tryStatement |
breakStatement |
continueStatement |
returnStatement |
expressionStatement |
assertStatement |
localFunctionDeclaration
;

12.1 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.

12.2 Expression Statements

An *expression statement* consists of an expression other than a non-constant map literal (11.7) that has no explicit type arguments.

The restriction on maps stems 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 appears in a place where a statement is expected.

12.3 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 **null**, 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.

It is a compile-time error if e refers to v.

This rule may seem redundant in light of the scoping rules above. After all:

```
f() { // a top level function
  var v = v* 2;
}
```

is already an error since v is not yet in scope in its own initializer. However, consider the more insidious:

```
var v = 0; // top level variable
f() { // a top level function
  var v = v* 2; // compile-time error
}
```

12.4 Local Function Declaration

A function declaration statement declares a new local function (6.1).

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 at the point immediately following the function declaration statement.

As for all function declarations, the function is also made available under its name in the function's formal parameters scope.

This implies that local functions can be directly recursive, but not mutually recursive. Consider these examples:

```
top() { // a top level function
  f(x) => x > 0? x*f(x-1): 1;  // recursion is legal
  g1(x) => h(x, 1); // error: h is not in scope 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 in scope. These cases are quite rare,

and can always be managed by defining a pair of variables first, then assigning them appropriate closures:

```
top2() { // a top level function
 var p, q;
  p = (x) => q(x,x); // illegal
  q = (a, b) => a > 0 ? p(a-1): b; // fine
}
```

12.5 If

The *if statement* allows for conditional execution of statements.

ifStatement:
  if '(' expression ')' statement (**else** statement)?
;

Execution of an if statement of the form **if** $(b)s_1$ **else** s_2 proceeds as follows:

First, the expression b is evaluated to an object o. In checked mode, it is a dynamic type error if o is not of type bool. Otherwise, o is then subjected to boolean conversion (11.4.1), producing an object r. If r is **true**, then the statement s_1 is executed, otherwise statement s_2 is executed.

It is a static type warning if the type of the expression b may not be assigned to bool.

An if statement of the form **if** $(b)s_1$ is equivalent to the if statement **if** $(b)s_1$ **else** $\{\}$.

12.6 For

The *for statement* supports iteration.

forStatement:
  for '(' forLoopParts ')' statement
;

forLoopParts:
  forInitializerStatement expression? ';' expressionList? |
  declaredIdentifier **in** expression |
  identifier **in** expression
;

forInitializerStatement:
  localVariableDeclaration ';' |
  expression? ';'
;

The for statement has two forms - the traditional for loop and the for-in statement.

12.6.1 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 4.
2. The expression $[v'/v]c$ is evaluated and subjected to boolean conversion (11.4). If the result is **false**, the for loop completes. Otherwise, execution continues at step 3.
3. The statement $[v'/v]s$ is executed.
4. Let v'' be a fresh variable. v'' is bound to the value of v'.
5. 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 closure 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 closures 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.

12.6.2 For-in

A for statement of the form **for** (*varOrType*? id **in** e) s is equivalent to the following code:

var n0 = e.iterator(); **while** (n0.hasNext()) { *varOrType*? id = n0.next(); s }

where n0 is an identifier that does not occur anywhere in the program.

12.7 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. In checked mode, it is a dynamic type error if o is not of type bool. Otherwise, o is then subjected to boolean conversion (11.4.1), producing an object r. If r is **true**, then s is executed and then the while statement is re-executed recursively. If r is **false**, execution of the while statement is complete.

It is a static type warning if the type of e may not be assigned to bool.

12.8 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. Then, the expression e is evaluated to an object o. In checked mode, it is a dynamic type error if o is not of type bool. Otherwise, o is then subjected to boolean conversion (11.4.1), producing an object r. If r is **false**, execution of the do statement is complete. If r is **true**, then the do statement is re-executed recursively.

It is a static type warning if the type of e may not be assigned to bool.

12.9 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) { **case** $label_{11} \ldots label_{1j_1}$ $e_1 : s_1 \ldots$ **case** $label_{n1} \ldots label_{nj_n}$ $e_n : s_n$ **default**: s_{n+1} } or the form **switch**

(*e*) { **case** $label_{11} \ldots label_{1j_1}$ $e_1 : s_1 \ldots$ **case** $label_{n1} \ldots label_{nj_n}$ $e_n : s_n$ }, 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 all instances of the same class C for all $k \in 1..n$.

In other words, all the expressions in the cases evaluate to constants of the exact same class. 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 implements the operator ==.

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 istead 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). In cases where more generality is desired, users can use **if** statements; eventually, literal maps will be more general, and one may be able to use patterns such as

```
{ 'a': () => 42; 'b': () => print('not an A'); 'c': () => foo(1, 2); }[e]();
```

instead of **switch**.

Execution of a switch statement of the form **switch** (*e*) { **case** $label_{11} \ldots label_{1j_1}$ $e_1 : s_1 \ldots$ **case** $label_{n1} \ldots label_{nj_n}$ $e_n : s_n$ **default**: s_{n+1} } or the form **switch** (*e*) { **case** $label_{11} \ldots label_{1j_1}$ $e_1 : s_1 \ldots$ **case** $label_{n1} \ldots label_{nj_n}$ $e_n : s_n$ } proceeds as follows:

The statement **var** id = *e*; is evaluated, where id is a variable whose name is distinct from any other variable in the program. 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 $e_1 \ldots e_n$.

Next, the case clause **case** $e_1 : s_1$ is executed if it exists. If **case** $e_1 : s_1$ does not exist, then if there is a **default** clause it is executed by executing s_{n+1}.

Note that if there are no case clauses ($n = 0$), the type of e does not matter.

A case clause introduces a new scope, nested in the lexically surrounding scope. The scope of a case clause ends immediately after the case clause's statement.

Execution of a **case** clause **case** $e_k : s_k$ of a switch statement **switch** (*e*) { $label_{11} \ldots label_{1j_1}$ **case** $e_1 : s_1 \ldots$ $label_{n1} \ldots label_{nj_n}$ **case** $e_n : s_n$ **default**: s_{n+1} } proceeds as follows:

The expression e_k == id is evaluated to an object o which is then subjected to boolean conversion yielding a value v. If v is not **true** the following case, **case** $e_{k+1} : s_{k+1}$ is executed if it exists. If **case** $e_{k+1} : s_{k+1}$ does not exist, then the **default** clause is executed by executing s_{n+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 sequence of statements s_h is then executed. If execution reaches the point after s_h then a runtime error occurs, unless $h = n + 1$.

Execution of a **case** clause **case** $e_k : s_k$ of a switch statement **switch** (*e*) { $label_{11} \ldots label_{1j_1}$ **case** $e_1 : s_1 \ldots$ $label_{n1} \ldots label_{nj_n}$ **case** $e_n : s_n$ } proceeds as follows:

The expression e_k == id is evaluated to an object o which is then subjected to boolean conversion yielding a value v. If v is not **true** the following case, **case** e_{k+1} : s_{k+1} is executed if it exists. If v is **true**, let h be the smallest integer such that $h \geq k$ and s_h is non-empty. The sequence of statements s_h is executed if it exists. If execution reaches the point after s_h then a runtime error occurs, unless $h = n$.

In other words, there is no implicit fall-through between cases. The last case in a switch (default or otherwise) can fall-through to the end of the statement.

It is a static warning if the type of e may not be assigned to the type of e_k. It is a static warning if the last statement of the statement sequence s_k is not a **break**, **continue**, **return** or **throw** statement.

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.

12.10 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.*

An **on-catch** clause of the form **on** T **catch** (p_1, p_2) s matches an object o if the type of o is a subtype of T. It is a compile-time error if T does not denote a type available in the lexical scope of the catch clause.

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 an identifier that does not occur anywhere else in the program.

An **on-catch** clause of the form **catch** (p) s is equivalent to an **on-catch** clause **on** Object **catch** (p) s. An **on-catch** clause of the form **catch** (p_1, p_2) s is equivalent to an **on-catch** clause **on** Object **catch** (p_1, p_2) s

The *active stack trace* is an object whose toString() method produces a string that is a record of exactly those function activations within the current isolate that had not completed execution at the point where the current exception was thrown.

This implies that no synthetic function activations may be added to the trace, nor may any source level activations be omitted. This means, for example, that any inlining of functions done as an optimization must not be visible in the trace. Similarly, any synthetic routines used by the implementation must not appear in the trace.

Nothing is said about how any native function calls may be represented in the trace.

For each such function activation, the active stack trace includes the name of the function, the bindings of all its formal parameters, local variables and **this**, and the position at which the function was executing.

The term position should not be interpreted as a line number, but rather as a precise position - the exact character index of the expression that raised the exception.

The definition below is an attempt to characterize exception handling without resorting to a normal/abrupt completion formulation. It has the advantage that one need not specify abrupt completion behavior for every compound statement. On the other hand, it is new and different and needs more thought.

A try statement **try** s_1 $on - catch_1 \ldots on - catch_n$ **finally** s_f defines an exception handler h that executes as follows:

The **on-catch** clauses are examined in order, starting with $catch_1$, until either an **on-catch** clause that matches the current exception (11.8) is found, or the list of **on-catch** clauses has been exhausted. If an **on-catch** clause $on - catch_k$ is found, then p_{k1} is bound to the current exception, p_{k2}, if declared, is bound to the active stack trace, and then $catch_k$ is executed. If no **on-catch** clause is found, the **finally** clause is executed. Then, execution resumes at the end of the try statement.

A finally clause **finally** s defines an exception handler h that executes by executing the finally clause. Then, execution resumes at the end of the try statement.

Execution of an **on-catch** clause **on** T **catch** (p_1, p_2) s of a try statement t proceeds as follows: The statement s is executed in the dynamic scope of the exception handler defined by the finally clause of t. Then, the current exception and active stack trace both become undefined.

Execution of a **finally** clause **finally** s of a try statement proceeds as follows:

The statement s is executed. Then, if the current exception is defined, control is transferred to the nearest dynamically enclosing exception handler.

Execution of a try statement of the form **try** s_1 $on - catch_1 \ldots on - catch_n$ **finally** s_f; proceeds as follows:

The statement s_1 is executed in the dynamic scope of the exception handler defined by the try statement. Then, the **finally** clause is executed.

Whether any of the **on-catch** clauses is executed depends on whether a matching exception has been raised by s_1 (see the specification of the throw statement).

If s_1 has raised an exception, it will transfer control to the try statements handler, which will examine the catch clauses in order for a match as specified above. If no matches are found, the handler will execute the **finally** clause.

If a matching **on-catch** was found, it will execute first, and then the **finally** clause will be executed.

If an exception is raised during execution of an **on-catch** clause, this will transfer control to the handler for the **finally** clause, causing the **finally** clause to execute in this case as well.

If no exception was raised, the **finally** clause is also executed. Execution of the **finally** clause could also raise an exception, which will cause transfer of control to the next enclosing handler.

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** {};.

12.11 Return

The *return statement* returns a result to the caller of a function.

returnStatement:
  return expression? ';'
;

Executing a return statement
return e;
first causes evaluation of the expression e, producing an object o. Next, control is transferred to the caller of the current function activation, and the object o is provided to the caller as the result of the function call.

It is a static type warning if the type of e may not be assigned to the declared return type of the immediately enclosing function.

In checked mode, it is a dynamic type error if o is not **null** and the runtime type of o is not a subtype of the actual return type (14.8.1) of the immediately enclosing function.

It is a compile-time error if a return statement of the form **return** e; appears in a generative constructor (7.6.1).

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.

Let f be the function immediately enclosing a return statement of the form **return**; It is a static warning if both of the following conditions hold:

- f is not a generative constructor.
- The return type of f may not be assigned to **void**.

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**. However, any 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.

A return statement of the form **return**; is executed by executing the statement **return null**; if it occurs inside a method, getter, setter or factory; otherwise, the return statement necessarily occurs inside a generative constructor, in which case it is executed by executing **return this**;.

Despite the fact that **return**; is executed as if by a **return** e;, it is important to understand that it is not a static warning to include a statement of the form

return; in a generative constructor. The rules relate only to the specific syntactic form **return** *e*;.

The motivation for formulating **return**; *in this way stems from the basic requirement that all function invocations indeed return a value. Function invocations are expressions, and we cannot rely on a mandatory typechecker to always prohibit use of* ***void*** *functions in expressions. Hence, a return statement must always return a value, even if no expression is specified.*

*The question then becomes, what value should a return statement return when no return expression is given. In a generative constructor, it is obviously the object being constructed (****this****). A void function is not expected to participate in an expression, which is why it is marked* ***void*** *in the first place. Hence, this situation is a mistake which should be detected as soon as possible. The static rules help here, but if the code is executed, using* ***null*** *leads to fast failure, which is desirable in this case. The same rationale applies for function bodies that do not contain a return statement at all.*

12.12 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 (12.9) prefixed by a label L.

The sole role of labels is to provide targets for the break (12.13) and continue (12.14) statements.

label:
  identifier ':'
;

The semantics of a labeled statement $L : s$ are identical to those of the statement s. 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.

12.13 Break

The *break statement* consists of the reserved word **break** and an optional label (12.12).

breakStatement:
  break identifier? ';'
;

Let s_b be a **break** statement. If s_b is of the form **break** L;, then let s_E be the the innermost labeled statement with label L enclosing s_b. If s_b is of the form **break**;, then let s_E be the the innermost do (12.8), for (12.6), switch (12.9) or while (12.7) statement enclosing s_b. It is a compile-time error if no such statement s_E exists within the innermost function in which s_b occurs. Furthermore, let $s_1, \ldots, s_n$ be those **try** statements that are both enclosed in s_E and that enclose s_b, and that have a **finally** clause. Lastly, let f_j be the **finally** clause of $s_j, 1 \leq j \leq n$. Executing s_b first executes $f_1, \ldots, f_n$ in innermost-clause-first order and then terminates s_E.

12.14 Continue

The *continue statement* consists of the reserved word **continue** and an optional label (12.12).

continueStatement:
  continue identifier? ';'
;

Let s_c be a **continue** statement. If s_c is of the form **continue** L;, then let s_E be the the innermost labeled do (12.8), for (12.6) or while (12.7) statement or case clause with label L enclosing s_c. If s_c is of the form **continue**; then let s_E be the the innermost do (12.8), for (12.6) or while (12.7) statement enclosing s_c. It is a compile-time error if no such statement or case clause s_E exists within the innermost function in which s_c occurs. Furthermore, let $s_1, \ldots, s_n$ be those **try** statements that are both enclosed in s_E and that enclose s_c, and that have a **finally** clause. Lastly, let f_j be the **finally** clause of $s_j, 1 \leq j \leq n$. Executing s_c first executes $f_1, \ldots, f_n$ in innermost-clause-first order. Then, if s_E is a case clause, control is transferred to the case clause. Otherwise, s_E is necessarily a loop and execution resumes after the last statement in the loop body.

In a while loop, that would be the boolean expression before the body. In a do loop, it would be the boolean expression after the body. In a for loop, it would be the increment clause. In other words, execution continues to the next iteration of the loop.

12.15 Assert

An *assert statement* is used to disrupt normal execution if a given boolean condition does not hold.

assertStatement:
  assert '(' conditionalExpression ')' ';'
;

The assert statement has no effect in production mode. In checked mode, execution of an assert statement **assert**(e); proceeds as follows:

The conditional expression e is evaluated to an object o. If the class of o is a subtype of Function then let r be the result of invoking o with no arguments. Otherwise, let r be o. It is a dynamic type error if o is not of type bool or of type Function, or if r is not of type bool. If r is **false**, we say that the assertion failed. If r is **true**, we say that the assertion succeeded. If the assertion succeeded, execution of the assert statement is complete. If the assertion failed, an AssertionError is thrown.

It is a static type warning if the type of e may not be assigned to either bool or $() \rightarrow$ 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). Overall, perhaps it could be defined as a function, and the overhead issue could be viewed as an optimization.

13 Libraries and Scripts

A library consists of (a possibly empty) set of imports, and a set of top-level declarations. A top-level declaration is either a class (7), a type alias declaration (14.3.1), a function (6) or a variable declaration (5).

topLevelDefinition:
  classDefinition |
  functionTypeAlias |
  external? functionSignature |
  external? getterSignature |
  external? setterSignature |
  functionSignature functionBody |
  returnType? getOrSet identifier formalParameterList functionBody |
  (**final** | **const**) type? staticFinalDeclarationList ';' |
  variableDeclaration ';'
;

getOrSet:
  get |
  set
;

libraryDefinition:
  libraryName importOrExport* partDirective* topLevelDefinition*
;

libraryName:
  metadata **library** qualified ';'
;

importOrExport:
  libraryImport |
  libraryExport

Libraries may be *named* or *anonymous*. A named library begins with the word **library** (possibly prefaced with any applicable metadata annotaions), followed by a qualified identifier that gives the name of the library.

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 (such as first class libraries) as well. In the future it may also serve to define a default prefix when importing.

Names of libraries intended for widespread use should follow the well known reverse internet domain name convention.

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 (13.1).

13.1 Imports

An *import* specifies a library to be used in the scope of another library.

libraryImport:
  metadata **import** uri (**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. It is a compile-time error if the compilation unit found at the specified URI is not a library declaration. The interpretation of URIs is described in section 13.5 below.

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.

Imports assume a global namespace of libraries (at least per isolate). They also assume the library is in control, rather than the other way around.

An import directive I may optionally include:

- A prefix clause of the form **as** id used to prefix names imported by I.
- 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:

First,

- If the URI that is the value of s_1 has not yet been accessed by an import or export (13.2) 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 (13.2) 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, and is undefined otherwise.
- 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 identfiers l and a namespace n, and produces a namespace that is identical to n except that it is undefined for each name in l.

Next, if I includes a prefix clause of the form **as** p, let $NS = prefix(p, NS_n)$ where $prefix(id, n)$, takes an identifier id and produces a namespace that has, for each entry mapping key k to declaration d in n, an entry mapping $id.k$ to d. Otherwise, let $NS = NS_n$. It is a compile-time error if the current library declares a top-level member named p,or if any other import directive in the current library includes a prefix clause with of the form **as** p .

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.

If a name N is referenced by a library L and N is introduced into the top level scope L by more than one import then:

- It is a static warning if N is used as a type annotation.
- In checked mode, it is a dynamic error if N is used as a type annotation and referenced during a subtype test.
- Otherwise, it is a compile-time error.

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.

On a related note, the provenance of the conflicting elements is not considered. An element that is imported via distinct paths may conflict with itself. This avoids variants of the well known "diamond" problem.

We say that the namespace NS *has been imported into* L.

It is a compile-time error to import two different libraries with the same name.

A widely disseminated library should be given a name using the common inverted internet domain name convention, as in other popular languages.

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.

13.2 Exports

A library L exports a namespace (3.1), meaning that the declarations in the namespace are made available to other libraries if they choose to import L (13.1). The namespace that L exports is known as its *exported namespace*.

libraryExport:
  metadata **export** uri combinator* ';'
;

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. Evaluation 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. 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.

13.3 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** qualified
;
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.

13.4 Scripts

A *script* is a library with a top-level function main().

scriptDefinition:
scriptTag? libraryName? libraryImport* partDirective* topLevelDefinition*
;

scriptTag:
'#!' (~NEWLINE)* NEWLINE
;

A script may optionally begin with a *script tag* which 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.

A script *S* may be executed as follows:

First, *S* is compiled as a library as specified above. Then, the top-level function main() that is in scope in *S* is invoked with no arguments. It is a run time error if *S* does not declare or import a top-level function main().

The names of scripts are optional, in the interests of interactive, informal use. However, any script of long term value should be given a name as a matter of good practice. Named scripts are composable: they can be used as libraries by other scripts and libraries.

13.5 URIs

URIs are specified by means of string literals:

uri:
  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.

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 library s that is part of the Dart implementation.

A URI of the form package:s is interpreted as a URI of the form packages/s relative to an implementation specified location.

This location will often be the location of the root library presented to the Dart compiler. However, implementations may supply means to override or replace this choice.

The intent is that, during development, Dart programmers can rely on a package manager to find elements of their program. Such package managers may provide a directory structure starting at a local directory packages *where they place the required dart code (or links thereto).*

Otherwise, any relative URI is interpreted as relative to the the location of the current library. All further interpretation of URIs is implementation dependent.

This means it is dependent on the embedder.

14 Types

Dart supports optional typing based on interface types.

The type system is unsound, due to the covariance of generic types. 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.

14.1 Static Types

Static type annotations are used in variable declarations (5) (including formal parameters (6.2)) and in the return types of functions (6). 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.

14.2 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.

A type T is *malformed* iff:

- T has the form id, and id does not denote a type available in the enclosing lexical scope.
- T is a parameterized type of the form $G < S_1, \ldots, S_n >$, and any of the following conditions hold:
  - Either G or $S_i, i \in 1..n$ are malformed.
  - G is not a generic type with n type parameters.
  - Let T_i be the type parameters of G (if any) and let B_i be the bound of $T_i, i \in 1..n$, and S_i is not a subtype of $[S_1, \ldots, S_n/T_1, \ldots, T_n]B_i, i \in 1..n$.

In checked mode, it is a dynamic type error if a malformed type is used in a subtype test. In production mode, an undeclared type is treated as an instance of type **dynamic**.

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. Running the program in production mode will print $yoyoma$. In checked mode, however, the program will fail when executing the type test on the first line of $main()$. A similar situation would arise if we wrote

```
f(foo x) => x;
main() {
    print(f("yoyoma"));
}
```

but the reason would be slightly different - the implicit type test triggered by passing "$yoyoma$" to f would fail. In contrast, the program

```
f(foo x) => x;
main() {
    print("yoyoma");
}
```

runs without incident in both production mode and checked mode (though it too gives rise to a static warning).

Note that subtype tests may occur implicitly in checked mode, as in

```
var i;
i  j; //  a variable j of type i (supposedly)
main() {
    j = new Object(); // fails in checked mode
}
```

Since i is not a type, a static warning will be issue at the declaration of j. However, the program can be executed in production mode without incident. In checked mode, the assignment to j implicitly introduces a subtype test that checks whether the the type of the newly allocated object, $Object$, is a subtype of the malformed type i, which will cause a run-time error. However, no runtime error would occur if j was not used, or if j was assigned null (since no subtype check is performed in that case).

One could have chosen to treat undeclared types in checked mode as type ***dynamic****, as is done in production mode. After all, a static warning has already been given. That is a legitimate design option, and it is ultimately a judgement call as to whether checked mode should be more or less aggressive in dealing with such a situation.*

Likewise, we could opt to ignore malformed types entirely in checked mode.

For now, we have opted to treat a malformed type as an error type that has no subtypes or supertypes, and which causes a runtime error when tested against any other type.

Here is a different example involving malformed types:

```
interface I<T extends num> {}
interface 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 malformed 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.

14.3 Type Declarations

14.3.1 Typedef

A *type alias* declares a name for a type expression.

functionTypeAlias:
metadata **typedef** 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 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 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**.

Currently, type aliases are restricted to function types. It is a compile-time error if any default values are specified in the signature of a function type alias. It is a compile-time error if a typedef refers to itself via a chain of references that does not include a class type.

Hence
typedef F F(F f);
is illegal, as are
typedef B A();
typedef A B();
but
typedef D C();
class D C foo(){}
is legal, because the references goes through a class declaration.

14.4 Interface Types

The implicit interface of class I is a direct supertype of the implicit interface of class J iff:

- If I is Object, and J has no **extends** clause
- If I is listed in the **extends** clause of J.

- If I is listed in the **implements** clause of J

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 $\bot$.
- 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 of the form $I < T_1, \ldots, T_n >$ and S is of the form $I < S_1, \ldots, S_n >$ and: $T_i << S_i, 1 \leq i \leq n$
- $T << U$ 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.

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.

A type T may be assigned to a type S, written $T \Longleftrightarrow S$, iff either $T <: S$ or $S <: T$.

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 runtime value happens to be a string.

14.5 Function Types

Function types come in three variations:

1. The types of functions that only have required parameters. These have the general form $(T_1, \ldots, T_n) \rightarrow T$.
2. The types of functions with optional positional parameters. These have the general form $(T_1, \ldots, T_n, [T_{n+1} \ldots, T_{n+k}]) \rightarrow T$.

3. The types of functions with named positional parameters. These have the general form $(T_1, \ldots, T_n, \{T_{x_1}\ x_1 \ldots, T_{x_k}\ x_k\}) \rightarrow T$.

A function type $(T_1, \ldots T_n) \rightarrow T$ is a subtype of the function type $(S_1, \ldots, S_n,) \rightarrow S$, if all of the following conditions are met:

1. Either
   - S is **void**, Or
   - $T \Longleftrightarrow S$.
2. $\forall i \in 1..n, T_i \Longleftrightarrow S_i$.

A function type $(T_1, \ldots T_n, [T_{n+1} \ldots, T_{n+k}]) \rightarrow T$ is a subtype of the function type $(S_1, \ldots, S_n, [S_{n+1} \ldots, S_{n+m}]) \rightarrow S$, if all of the following conditions are met:

1. Either
   - S is **void**, Or
   - $T \Longleftrightarrow S$.
2. $k \geq m$ and $\forall i \in 1..n+m, T_i \Longleftrightarrow S_i$.

A function type $(T_1, \ldots T_n, \{T_{x_1}\ x_1, \ldots, T_{x_k}\ x_k\}) \rightarrow T$ is a subtype of the function type $(S_1, \ldots, S_n, \{S_{y_1}\ y_1, \ldots, S_{y_m}\ y_m\}) \rightarrow S$, if all of the following conditions are met:

1. Either
   - S is **void**, Or
   - $T \Longleftrightarrow S$.
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 $y_i \in \{y_1, \ldots, y_m\}, y_i = x_j \Rightarrow T_j \Longleftrightarrow S_i$

In addition, the following subtype rules apply:
$(T_1, \ldots, T_n, []) \rightarrow T <: (T_1, \ldots, T_n) \rightarrow T$.
$(T_1, \ldots, T_n) \rightarrow T <: (T_1, \ldots, T_n, \{\}) \rightarrow T$.
$(T_1, \ldots, T_n, \{\}) \rightarrow T <: (T_1, \ldots, T_n) \rightarrow T$.
$(T_1, \ldots, T_n) \rightarrow T <: (T_1, \ldots, T_n, []) \rightarrow T$.
A function is always an instance of some class that implements the class Function. However not all function types are a subtype of Function. If a type I includes a method named **call**, and the type of **call** is the function type F, then I is considered to be a subtype of F.

14.6 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**, . . . , **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 runtime error in checked mode. This seems desirable from the perspective of error localization. However, when a dynamic error is raised 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.

14.7 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)

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 **null**, 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 **null**; 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 runtime type **dynamic**).

14.8 Parameterized Types

A *parameterized type* is an invocation of a generic type declaration.

Let $G < A_1, \ldots, A_n >$ be a parameterized type.

It is a static type warning if G is not an accessible generic type declaration with n type parameters. It is a static type warning if $A_i, i \in 1..n$ does not denote a type in the enclosing lexical scope.

If S is the static type of a member m of G, then the static type of the member m of $G < A_1, \ldots, A_n >$ is $[A_1, \ldots, A_n/T_1, \ldots, T_n]S$ where $T_1, \ldots, T_n$ are the formal type parameters of G. Let B_i, be the bounds of $T_i, 1 \leq i \leq n$. It is a static type warning if A_i is not a subtype of $[A_1, \ldots, A_n/T_1, \ldots, T_n]B_i, i \in 1..n$.

14.8.1 Actual Type of Declaration

A type T *depends on a type parameter* U iff:

- T is U.
- T is a parameterized type, and one of the type arguments of T depends on U.

Let T be the declared type of a declaration d, as it appears in the program source. The *actual type* of d is

- $[A_1, \ldots, A_n/U_1, \ldots, U_n]T$ if d depends on type parameters $U_1, \ldots, U_n$, and A_i is the value of $U_i, 1 \leq i \leq n$.
- T otherwise.

14.8.2 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 $S_n = \{T | T \in S \wedge depth(T) = n\}$ for any finite n, and $k = max(depth(T_1), \ldots, depth(T_m)), T_i \in S, i \in 1..m$, where $depth(T)$ is the number of steps in the shortest inheritance path from T to Object. Let q be the largest number such that S_q has cardinality one. The least upper bound of I and J is the sole element of S_q.

15 Reference

15.1 Lexical Rules

Dart source text is represented as a sequence of Unicode code points normalized to Unicode Normalization Form C.

15.1.1 Reserved Words

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

LETTER:
  'a' .. 'z' |
  'A' ..'Z'
;

DIGIT:
  '0' .. '9'
;

WHITESPACE:
  ('\t' | ' ' | NEWLINE)+
;

15.1.2 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.

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 multi-line comments that begin with the tokens /**. Inside a documentation comment, the Dart compiler ignores all text unless it is enclosed in brackets.

15.2 Operator Precedence

Operator precedence is given implicitly by the grammar.

We expect to have a table here anyway.

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.