Dart Programming Language Specification

The Dart Team

September 15, 2011

Contents

1 Note

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

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 *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 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. *Rationale is intended to explain the motivation for language design choices, while commentary is more general.*

Open questions (**in this font**). Open questions are points that are unsettled in the mind of the spec writer; expect them to be eliminated in the final spec. **Should the text at the end of the previous bullet be rationale or commentary?**

Reserved words and built-in identifiers appear in **bold**.
Examples would be, **switch** or **class**.

3 Overview

Dart is a class-based, single-inheritance, pure object-oriented programming language. Dart is optionally typed (**??**) and supports reified generics and interfaces.

Dart programs may be statically typechecked. 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: scripting mode or checked mode. In scripting mode, static type annotations (**??**) 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 and interface declarations, the 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. Scripting mode respects optional typing. Static type annotations do not effect 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* (**??**). 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. The reasons for this need to be examined in depth. Possibly driven by the difficulties of compiling efficiently to Javascript?

3.1 Privacy

Dart supports two levels of privacy: *public* and *private*. A declaration is *private* iff it begins with an underscore (the _ character) otherwise it is *public.* Private declarations may only be accessed within the library in which they are declared.

A declaration m is *accessible to library* L if m is declared in L or if m is not private.

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

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

3.2 Concurrency

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

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

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 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).

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.*

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 statement (**??**) of the form: **throw** *ex*; was implicitly executed. When we say that *a C is thrown*, where *C* is an exception class, we mean that the result of evaluating **new** *C*() is thrown.

5 Variables

Variables are storage locations in memory.

```
variableDeclaration:
  declaredIdentifier (', ' identifier)*
 ;

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

initializedIdentifierList:
  initializedIdentifier (', ' initializedIdentifier)*
 ;

initializedIdentifier:
  identifier ('=' expression)?
 ;

declaredIdentifier:
  finalVarOrType identifier
 ;
```

finalVarOrType:
  final type? |
  var |
  type
 ;

An uninitialized variable has the value **null** (**??**). A *final variable* is a variable whose declaration includes the modifier **final**. A final variable can only be assigned once, when it is initialized.

A *static variable* is a variable that is not associated with a particular instance, but rather with an entire library or class.

A variable that is marked both **static** and **final** must be initialized to a compile-time constant (**??**).

If no type is explicitly specified, the type of the declared variable(s) is **Dynamic**, the unknown type (**??**).

6 Functions

Functions abstract over executable actions.

functionDeclaration:
  returnType? identifier formalParameterList
 ;
functionPrefix:
  returnType? identifier
 ;

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

block:
  '{' statements '}'
 ;

A function body of the form of the form => e is equivalent to a body of the form {**return** e;}.

6.1 Formal Parameters

Every function declaration includes a *formal parameter list*, which consists of a list of required parameters, followed by any optional parameters. Optional parameters consist of a set of keyword parameters . **The following can be simplified to:**

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

optionalFormalParameters: namedFormalParameters .

```
formalParameterList:
   '(' namedFormalParameters ')' |
  '(' normalFormalParameters normalFormalParameterTail? ')'
 ;

normalFormalParameters:
   normalFormalParameter (', ' normalFormalParameter)* ']'
 ;

normalFormalParameterTail:
   ', ' namedFormalParameters
 ;

namedFormalParameters:
   '[' defaultFormalParameter (', ' defaultFormalParameter)* ']'
 ;
```

Formal parameters are always **final**. A formal parameter is always considered to be initialized. *This is because it will always be initialized by the call - even if it is optional.*

6.1.1 Positional Formals

A positional formal parameter is a simple variable declaration (**??**).

```
normalFormalParameter:
   functionDeclaration fieldFormalParameter simpleFormalParameter
 ;

simpleFormalParameter:
   declaredIdentifier identifier
 ;

fieldFormalParameter:
   finalVarOrType? this '.' identifier
 ;
```

6.1.2 Keyword Formals

defaultFormalParameter:
  normalFormalParameter ('=' constantExpression)?
 ;

6.1.3 Type of a Function

Let F be a function with required formal parameters $T_1\ p_1 \ldots, T_n\ p_n$. Then the type of F is $(T_1, \ldots, T_n, [id_{n+1} : T_n + 1, \ldots, id_{n+k} : T_{n+k}])$ if F has named optional parameters $id_{n+1} : T_{n+1} p_{n+1}, \ldots, id_{n+k} : T_{n+k} p_{n+k}$.

7 Classes

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

classDefinition:
  class identifier typeParameters? superclass? interfaces? '{' class-MemberDefinition* '}'
 ;

classMemberDefinition:
  declaration '.' |
 methodDeclaration functionBody
 ;

// A method, operator, or constructor (which all should be followed by // a block of code)
 ;
methodDeclaration:
  factoryConstructorDeclaration |
 static functionDeclaration |
 specialSignatureDefinition |
 functionDeclaration initializers? |
 namedConstructorDeclaration initializers?
 ;

// An abstract method/operator, a field, or const constructor (which // all should be followed by a semicolon)
 ;
declaration:
  constantConstructorDeclaration (redirection | initializers)? |
 functionDeclaration redirection |
 namedConstructorDeclaration redirection |

abstract specialSignatureDefinition |
abstract functionDeclaration |
static final type? staticFinalDeclarationList |
static? variableDeclaration
;

staticFinalDeclarationList:
:
staticFinalDeclaration (', ' staticFinalDeclaration)*
;

staticFinalDeclaration:
identifier '=' constantExpression
;

specialSignatureDefinition:
getterDefinition |
setterDefinition |
operatorDefinition
;

getOrSet:
get |
set
;

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

unaryOperator:
negateOperator
;

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

```
  bitwiseOperator |

 ;
prefixOperator:
   additiveOperator |
  negateOperator
 ;

negateOperator:
   '!' ' '
 ;
```

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.

Every class has a single superclass except class Object which has no superclass. A class may implement a number of interfaces, either by declaring them in its implements clause or via interface injection declarations (**??**) outside the class declaration.

An *abstract class* is either a class that is explicitly declared with the **abstract** modifier, or a class that declares at least one abstract method.

It is a compile-time error if a class declares two members of the same name, except that a getter and a setter may be declared with the same name provided both are instance members or both are static members.

What about a final instance variable and a setter?

7.1 Instance Methods

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

It is a static warning if an instance method m_1 overrides an instance method m_2 and the type of m_1 is not a subtype of the type of m_2.

7.1.1 Abstract Methods

An *abstract method* is an instance method whose declaration is prefixed by the built in identifier **abstract**. **Well, no, its just a signature, not an actual method or function.** It is a compile time error to specify an implementation for an abstract method. It is a compile time error if any default values are specified in the signature of an abstract method. **This could all be enforced syntactically.** An *abstract class* is a class that declares an abstract method or is itself prefixed by the built in identifier **abstract**.

Invoking an abstract method always results in a run-time error. This may be NoSuchMethodError or a subclass, such as AbstractMethodError. **Are these errors catchable?**

It follows from the above that unless explicitly stated otherwise, all ordinary rules that apply to methods apply to abstract methods.

7.1.2 Operators

Operators are methods with special names.

operatorDeclaration:
  returnType? **operator** operator formalParameterList
 ;

An operator declaration is identified with built-in identifier operator.

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

It is a compile time error if the arity of a user-declared operator with one of the names: ==, <, >, <=, >=, -, +, *, ?, /, %, —, ^, &, <<, >>, >>>, []= 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.

7.2 Getters

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

getterDefinition:
  static? returnType? **get** identifier formalParameterList
 ;

Why does a getter have a formal parameter list at all?

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.

It is a compile-time error if a getters formal parameter list is not empty.

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.

It is a static warning if a getter m_1 overrides a getter m_2 and the type of m_1 is not a subtype of the type of m_2.

7.3 Setters

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

```
getterDefinition:
  static? 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 the setter is given by the identifier in the definition.

It is a compile-time error if a setters formal parameter list consists of a more than one formal parameter p. It is a compile-time error of p is not a required parameter. *We could enforce this via the grammar, but wed have to specify the evaluation rules in that case.*

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

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 a setter m_2 and the type of m_1 is not a subtype of the type of m_2.

7.4 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.

If an instance variable declaration has the form T $v = e$; or the form **var** $v = e$; then the expression e must be a compile-time constant (**??**).

*In Dart, all uninitialized variables have the value **null**, regardless of type. Numeric variables in particular must, therefore, be explicitly initialized; such variables will not be initialized to 0 by default. The form above is intended to ease the burden of such initialization.*

An instance variable declaration of the form T v; or the form T $v = e$; always induces an implicit getter function (**??**) with signature

T **get** v

whose execution returns the value stored in v.

An instance variable declaration of the form **var** v; or the form **var** $v = e$; always induces an implicit getter function with signature

get v

whose execution returns the value stored in v.

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

void set $v(T\ x)$

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

An instance 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.

7.5 Constructors

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

A constructor has a name, which always begins with the name of its immediately enclosing class, and may optionally be followed by a dot and an identifier.

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

7.5.1 Generative Constructors

A *generative constructor* consists of a constructor name, a constructor parameter list, a (possibly empty) initializer list and a body.

```
constructorDeclaration:
   identifier formalParameterList |
  namedConstructorDeclaration
 ;

namedConstructorDeclaration:
   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 (**??**) 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.

A *fresh instance* is an instance whose identity is distinct from any previously allocated instance of the class. A generative constructor always allocates 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 (**??**).

Redirecting Constructors A generative constructor may be *redirecting*, in which case its only action is to invoke another generative constructor.

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

We now have generative constructors with no bodies as well.

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* specifies a *superconstructor* - that is, a specific constructor of the superclass. Execution of the superinitializer causes all initializers of the superconstructor to be invoked.
- An *instance variable initializer* assigns a value to an individual instance variable. **May an instance variable initializer be repeated in a list, or over superclasses?**

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

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

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

  ambiguousFunctionInvocation:
    identifier arguments ('{' | '=>')
   ;

A generative constructor 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 the initializer list, unless the enclosing class is class `Object`. It is a compile-time error if a generative constructor of class `Object` includes a superinitializer.

It is a compile time error if more than one initializer corresponding to a given instance variable appears in the initializer list. There must be exactly one initializer for each final instance variable declared in the class, or a compile time error occurs. But they could be initialized via this.x formals

Additional initializers may be included.

Execution of a generative constructor proceeds as follows:

First, a fresh instance i of the immediately enclosing class is allocated. Next, the instance variable declarations of the immediately enclosing class are visited in the order they appear in the program text. For each such declaration d, if d has the form $finalVarOrType\ v = e;$ then the instance variable v of i is bound to the value of e (which is necessarily a compile-time constant). Next, any initializing formals declared in the constructor's parameter list are executed in the order they appear in the program text **Nail this down somewhere as left to right for the entire spec**. Then, the constructor's initializers are executed in the order they appear in the program text .

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 the constructor is executed in a scope where **this** is bound to i. Execution of the body begins with execution of the body of the superconstructor. **(explain how args are bound, value of this etc.)**

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

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 C. If S is generic, let $U_1, , \ldots, U_m$ be the actual parameters passed to S in the superclass clause of C.

Then, the initializer list of the constructor S (respectively $S.id$) is executed 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 binding of $U_1, , \ldots, U_m$.

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

7.5.2 Factories

A factory is a static method prefaced by the built-in identifier **factory**. The name of the method must be a constructor name.

factoryConstructorDeclaration:
  factory qualified ('.' identifier)? formalParameterList
;

In checked mode, it is a dynamic type error if a factory returns an object whose type is not a subtype of its immediately enclosing class.

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 (which of course should conform to the interface of the immediately enclosing class).

7.5.3 Constant Constructors

A *constant constructor* may be used to create compile-time constant (**??**) objects. A constant constructor is prefixed by the reserved word **const**. It is a compile-time error if a constant constructor has a body.

constantConstructorDeclaration:
  const qualified formalParameterList
;

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

7.6 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 and the static methods inherited by C from its superclass.

It is a compile-time error if a class has two static methods with the same name, regardless of where they are declared.

There is no hiding of static methods, or of static variables, unlike certain languages of the past.

7.7 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 and the static variables inherited by C from its superclass.

A static variable declaration of the form **static** T v; or the form **static** T v $= e$; always induces an implicit static getter function (**??**) with signature

static T **get** v

whose execution returns the value stored in v.

A static variable declaration of the form **static var** v; or the form **static var** $v = e$; always induces an implicit static getter function with signature

static get v

whose execution returns the value stored in v.

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

static void set $v(T\ x)$

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

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

static set $v(x)$

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

It is a compile-time error if a class has two static variables with the same name, regardless of where they are declared.

7.8 Superclasses

The **extends** clause of a class specifies its superclass. If no **extends** clause is specified, the superclass 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 an class C includes a type expression that does not denote a class available in the lexical scope of C.

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.

May static members be accessed via the subclass name?

7.8.1 Inheritance and Overriding

A class C declared in library L inherits any members of its superclass that are accessible to L 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 with superclass S that declares 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 an instance method.
- m is a getter and m is a getter or a method.
- m is a setter and m is a setter or a method.

Whether an override is legal or not is described elsewhere in this specification.

For example getters and setters may not legally override methods and vice versa. *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 dont override setters and vice versa. Finally, static members never override anything.

7.9 Superinterfaces

A class has a set of direct superinterfaces. This set includes the interface of its superclass, 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 an class C includes a type expression that does not denote a class or interface available in the lexical scope of C.

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

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

A class does not inherit members from its superinterfaces. *This avoids the issues with so-called miranda methods etc.*

8 Interfaces

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

interfaceDefinition:
  interface identifier typeParameters? superinterfaces? factorySpecification? '{' (interfaceMemberDefinition)* '}'
 ;

interfaceMemberDefinition:
  static final type? initializedIdentifierList ';' |
  functionDeclaration ';' |

```
    constantConstructorDeclaration ';' |
    namedConstructorDeclaration ';' |
    specialSignatureDefinition ';' |
    variableDeclaration ';'
   ;
```

```
factorySpecification:
   factory type
  ;
```

8.1 Methods

An interface method declaration specifies a method signature but no body. It is a static warning if an interface method m_1 overrides an interface method m_2 and the type of m_1 is not a subtype of the type of m_2.

8.1.1 Operators

Operators are methods with special names. Some, but not all, operators may be defined by user code.

8.2 Getters and Setters

An interface may contain getter and/or setter signatures. These are subject to the same compile-time and static checking rules as getters and setters in classes.

8.3 Factories and Constructors

An interface may specify a factory class, which is a class that will be used to provide instances when constructors are invoked via the interface.

```
factorySpecification:
   factory type
  ;
```

An interface can specify the signatures of constructors that are used to provide objects that conform to the interface.

A constructor k_I of interface I with name N_I corresponds to a constructor k_F of class F with name N_F iff both of the following conditions hold:

- F is the factory class of I.
- Either
  - k_I is named N_I and k_F is named N_F, OR

  - k_I is named $N_I.id$ and k_F is named $N_F.id$.

It is a compile-time error if an interface declares a constructor without declaring a factory class. It is a compile-time error if an interface I declares a constructor k_I and there is no constructor k_F in the factory class F such that k_I corresponds to k_F.

Let k_I be a constructor declared in an interface I, and let k_F be its corresponding constructor. Then:

- It is a compile-time error if k_I and k_F do not have the same number of required parameters.
- It is a compile-time error if k_I and k_F do not have identically named optional parameters.
- It is a static type warning if the type of the nth required formal parameter of k_I is not identical to the type of the nth required formal parameter of k_F.
- It is a static type warning if the types of named optional parameters with the same name differ between k_I and k_F.

8.4 Superinterfaces

An interface has a set of direct superinterfaces. This set consists of the interfaces specified in the **extends** clause of the interface .

```
superinterfaces:
  extends typeList
;
```

It is a compile-time error if the **extends** clause of an interface I includes a type expression that does not denote a class or interface available in the lexical scope of I. It is a compile-time error if the extends clause of an interface includes type **Dynamic**. It is a compile-time error if an interface is a superinterface of itself.

8.4.1 Inheritance and Overriding

An interface I declared in library L inherits any members of its superinterfaces that are accessible (**??**) to L that are not overridden by members declared in I.

However, if the above rule would cause multiple members $m_1, \ldots, m_n$ to be inherited (because identically named members existed in several superinterfaces) then only one member is inherited. If the static types $T_1, \ldots, T_n$ of the members $m_1, \ldots, m_n$ are not identical, then there must be a member m_x such that $T_x <: T_i, 1 \le x \le n$ for all $i, 1 \le i \le n$, or a static type warning occurs. The member that is inherited is m_x, if it exists; otherwise it is selected at random from $m_1, \ldots, m_n$.

Alternatives: (a) No member is inherited in case of conflict. (b) I gets a member mx with arguments and return type ***Dynamic****. But what if the arities disagree? Perhaps best is if the first m is selected (based on order of superinterface list)*

The only situation where the runtime would be concerned with this would be when a mirror attempted to obtain the signature of an interface member.

An interface may override instance members that would otherwise have been inherited from its superinterfaces.

Let I be an interface with superinterface S that declares 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 an instance method.
- m is a getter and m is a getter or a method.
- m is a setter and m is a setter or a method.

Whether an override is legal or not is described elsewhere in this specification.

8.5 Generics

A class (**??**) or interface (**??**) 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:
  identifier (extends type)?
;
typeParameters:
  '<' typeParameter (',' typeParameter)* '>'
;
```

The type parameters of a generic declaration are in scope in the non-static members of the declaration. 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.

9 Expressions

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

```
expression:
  assignableExpression assignmentOperator expression |
  conditionalExpression
 ;

expressionList:
   expression (', ' expression)*
 ;


primary:
   thisExpression |
  super assignableSelector |
  functionExpression |
  literal |
  identifier |
  newExpression constantObjectExpression |
  '(' expression ')' |

 ;
```

9.1 Constants

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

```
constantExpression:
   expression
 ;
```

The constant expression production is used to mark certain expressions as only being allowed to hold a compile-time constant. The grammar cannot express these restrictions (yet), so this will have to be enforced by a separate analysis phase.

A constant expression is one of the following:

- A literal number (**??**).
- A literal boolean (**??**).
- A literal string (**??**) that does not involve string interpolation (**??**).
- **null** (**??**).
- A reference to a static final variable (**??**).
- A constant constructor invocation (**??**).
- A constant array literal (**??**).

- A constant map literal (**??**).
- An expression of one of the forms e_1 == e_2, e_1 != e_2, 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.
- 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 boolean value.
- An expression of one of the forms e, e_1 / e_2, e_1 ˆ e_2, e_1 & e_2, e_1|e_2, e_1 >> e_2 or e_1 << e_2, where e_1 and e_2 are constant expressions that evaluate to an integer value.
- An expression of one of the forms $e_1 + e_2$, 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_1 and e_2 are constant expressions that evaluate to a numeric value.

  literal:
    nullLiteral |
   booleanLiteral |
   numericLiteral |
   stringLiteral |
   mapLiteral |
   arrayLiteral
  ;

9.2 Null

The reserved word **null** denotes the *null object.* **Any methods, such as isNull?**

  nullLiteral:
    null
  ;

The null object is the sole instance of the built-in class Null. Attempting to instantiate Null causes a runtime error. It is a compile-time error for a class or interface attempt to extend or implement Null. **Shouldn't a warning suffice?** Invoking a method on **null** usually yields a NullPointerException unless the method is explicitly implemented by class Null.

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

The decision to use $\perp$ instead of Null *allows* ***null*** *to be be assigned everywhere without complaint by the static checker.* class Null { factory Null._() throw "cannot be instantiated";

noSuchMethod(String name, ...arguments) { throw new NullPointerException(); // What methods does this implement? isNull? }}

9.3 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', it denotes the hexadecimal integer represented by the part of the literal following '0x'. Otherwise, if the numeric literal does not include a decimal point denotes 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.

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.

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

9.4 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 interface bool. It is a compile-time error for a class or interface 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.

9.4.1 Boolean Conversion

Boolean conversion maps any object *o* into a boolean defined as

(bool v){ assert(null !== v); return true === v; }(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**. Darts 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 truth values. Indeed, there is no way to derive a truth value 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 coerced into **true** (as it is a non-null object).*

9.5 Strings

A *string* is sequence of valid unicode code points.

```
stringLiteral:
   '@'? MULTI_LINE_STRING |
  SINGLE_LINE_STRING
  ;
```

A string can either a single line string or a multiline string.

```
SINGLE_LINE_STRING:
   '$'? '"' STRING_CONTENT_DQ* '"' |
  '$'? '
'' STRING_CONTENT_SQ* '
'' |
  '@' '
'' ( ( '
'' | NEWLINE ))* '
'' |
  '@' '"' ( ( '"' | NEWLINE ))* '"'
 ;
```

A single line string may not span more than one line of source code. 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".

```
MULTI_LINE_STRING:
   '"""' .* '"""' |
  '
'
'
'' .* '
'
'
''
 ;

ESCAPE_SEQUENCE:
   \ n |
  \ r |
  \ f |
  \ b |
  \ t |
  \ v |
  \ x HEX_DIGIT HEX_DIGIT |
  \ u HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT |
  \ u{ HexDigitSequence }
 ;

HEX_DIGIT_SEQUENCE:
  HEX_DIGIT 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.

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 $HexDigit_1$ $HexDigit_2$, equivalent to \u{$HexDigit_1$ $HexDigit_2$}.
- \u $HexDigit_1$ $HexDigit_2$ $HexDigit_3$ $HexDigit_4$, equivalent to \u{$HexDigit_1$ $HexDigit_2$ $HexDigit_3$ $HexDigit_4$}.
- \u{*HexDigitSequence*} is the unicode scalar value represented by the HexDigitSequence. It is a compile-time error if the value of the *HexDigitSequence* 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 $\{n, r, f, b, t, v, x, u\}$.

It is a compile-time error if a string literal to 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 string literal to 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.

However, any string may be prefixed with the character @, indicating that it is a *raw string*, in which case no escapes are recognized.

```
    STRING_CONTENT_DQ:
       ( '

    ' | '"' | '$' | NEWLINE ) |
      '


    '  ( NEWLINE ) |
      '$' IDENTIFIER_NO_DOLLAR |
      '$' '' IDENTIFIER ''
```

```
;

STRING_CONTENT_SQ:
    ( '

' | '
'' | '$' | NEWLINE ) |
  '


'  ( NEWLINE ) |
  '$' IDENTIFIER_NO_DOLLAR |
  '$' '{' IDENTIFIER '}'
 ;

NEWLINE:
   '\ n' |
  '\ r'
 ;
```

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

9.5.1 String Interpolation

It is possible to embed expressions within 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*.

```
STRING_INTERPOLATION:
   '$' IDENTIFIER_NO_DOLLAR |
  '$' '{' 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 's1 ${$e$} s2' is equivalent to 's1' + e.toString + 's2'. Likewise an interpolated string "s1 ${e} s2" is equivalent to "s1" + e.toString + "s2", assuming + is the string concatenation operator.

The string interpolation syntax is designed to be familiar and easy to use, if somewhat awkward to parse. The intent is to encourage its use over alternatives such as $s1 + s2$. In a dynamically typed language, the use of the + operator requires dynamic dispatch. In contrast, in the case of string interpolation we can statically determine that the string concatenation operation is required, making the operation more efficient. Even more importantly, it helps the system to determine if other uses of + are numeric, helping the implementation speed up those operations. This is especially crucial for a language that must be efficiently compiled into Javascript.

9.6 Arrays

An *array literal* denotes an array, which is an integer indexed collection of objects.

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

An array may contain zero or more objects. The number of elements in an array is its size. An array has an associated set of indices. An empty array has an empty set of indices. A non-empty array has the index set $\{0 \ldots n-1\}$ where n is the size of the array. It is a runtime error to attempt to access an array using an index that is not a member of its set of indices. **Do we really want to specify arrays? Isnt this just a library issue?**

If an array literal begins with the reserved word **const**, it is a *constant array literal* and it is computed at compile-time. Otherwise, it is a *run-time array literal* and it is evaluated at run-time.

It is a compile time error if an element of a constant array literal is not a compile-time constant. It is a compile time error if the type argument of a constant array literal includes a type variable.

The value of a constant array literal **const** $< E > [e_1 \ldots e_n]$ is an object a that implements the built-in interface *Array* $< E >$. 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 array literal **const** $[e_1 \ldots e_n]$ is defined as the value of a constant array literal **const**< **Dynamic** > $[e_1 \ldots e_n]$.

Let $array_1$ = **const** $< V > [e_{11} \ldots e_{1n}]$ and $array_2$ = **const** $< U > [e_{21} \ldots e_{2n}]$ be two constant array literals and let the elements of $array_1$ and $array_2$ evaluate to $o_{11} \ldots o_{1n}$ and $o_{21} \ldots o_{2n}$ respectively. Iff $o_{1i} === o_{2i}$ for $1 \leq i \leq n$ and $V = U$ then $array_1 === array_2$.

In other words, constant array literals are canonicalized.

A run-time array literal $< E > [e_1 \ldots e_n]$ 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 a that implements the built-in interface $Array < E >$ is allocated.
- The ith element of a is set to $o_{i+1}, 0 \leq i \leq n$.
- The result of the evaluation is a.

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

A runtime array literal $[e_1 \ldots e_n]$ is evaluated as $<$ **Dynamic** $>$ $[e_1 \ldots e_n]$. **Any restriction on implementing this interface in user code? There shouldnt be.**

There is no restriction precluding nesting of array literals. It follows from the rules above that $< Array < int >> [[1, 2, 3][4, 5, 6]]$ is an array with type parameter int, containing two arrays with type parameter **Dynamic**.

The static type of an array literal of the form **const**$< E >$ $[e_1 \ldots e_n]$ or the form $< E > [e_1 \ldots e_n]$ is $Array < E >$. The static type an array literal of the form **const** $[e_1 \ldots e_n]$ or the form $[e_1 \ldots e_n]$ is $Array <$ **Dynamic** $>$.

It is tempting to assume that the type of the array literal would be computed based on the types of its elements. However, for mutable arrays this may be unwarranted. Even for constant arrays, 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****.*

9.7 Maps

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

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

mapLiteralEntry:
  identifier ':' expression |
  stringLiteral ':' expression
;
```

A *map literal* consists of zero or more entries. Each entry has a *key*, which is a string, and a *value*, which is an object. The key of an entry may be specified via an identifier or via a compile-time constant string. If the key is specified via an identifier id, the specification is interpreted as if it was the string 'id'.

If a map literal begins with the reserved word **const**, it is a *constant map literal* and it is computed at compile-time. Otherwise, it is a *runtime map literal* and it is evaluated at runtime.

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 either type argument of a constant map literal includes a type variable.

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

As specified, a typed map literal takes only one type parameter. If we generalize literal maps so they can have keys that are not strings, we would need two parameters.

Let $map_1 =$ **const**$< V > \{k_{11} : e_{11} \ldots k_{1n} : e_{1n}\}$ and $map_2 =$ **const**$< U > \{k_{21} : e_{21} \ldots k_{2n} : e_{2n}\}$ 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 $o_{1i} === o_{2i}$ and $s_{1i} === s_{2i}$ for $1 \leq i \leq n$, and $V = U$ then $map_1 === map_2$.

In other words, constant map literals are canonicalized.

A runtime map literal $< V > \{k_1 : e_1 \ldots k_n : e_n\}$ 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 m that implements the built-in interface $Map < String, V >$ is allocated.
- Let u_i be the value of the compile-time constant string specified by k_i. An entry with key u_i and value o_i is added to $m, 0 \leq i \leq n$.
- The result of the evaluation is m.

A runtime map literal $\{k_1 : e_1 \ldots k_n : e_n\}$ is evaluated as $<$ **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**$< V > \{k_1 : e_1 \ldots k_n : e_n\}$ or the form $< V > \{k_1 : e_1 \ldots k_n : e_n\}$ 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 $Map < String,$ **Dynamic** $>$.

9.8 Function Expressions

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

```
functionExpression:
   (returnType? identifier)? formalParameterList functionExpres-
sionBody
 ;

functionExpressionBody:
   '=>' expression |
  block
 ;
```

A function literal implements the built-in interface Function.

The static type of a function literal $(a_1, \ldots, a_n, x_{n+1} \ldots x_{n+k}])$ $=>$ 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 is not specified, it is considered to have been specified as **Dynamic**.

9.9 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.

9.10 Instance Creation

Instance creation expressions invoke constructors to produce instances.

It is a static warning to instantiate an abstract class (**??**). It is a static type warning if any of the type arguments to a constructor of a generic type invoked by a new expression or a constant object expression are not subtypes of the bounds of the corresponding formal type parameters.

9.10.1 New

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

```
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 compile-time error if T is not a class or interface optionally followed by type arguments.

If e 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 compile-time error if id is not the name of a constructor declared by the type T. If e of the form **new** $T(a_1, \ldots, a_n)$ it is a compile-time error if the type T does not declare a constructor with the same name as the declaration of T.

If T is a parameterized type $S < U_1, \ldots, U_m >$, let $R = S$. It is a compile time error if S is not a generic type with m type parameters. If T is not a parameterized type, let $R = T$. If R is an interface, let C be the factory class of R. Otherwise let $C = R$. 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 k be the constructor $C.id$, otherwise let k be the constructor C. Finally, if C 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:

First, if k is a generative constructor, then:

A fresh instance, i, of class C is allocated. For each instance variable f of i, if the variable declaration of f has an initializer, then f is bound to that value (which is necessarily a compile-time constant). Otherwise f is bound to **null**.

Next, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k}])$ is evaluated. Next, the initializer list of k is executed with respect to the bindings that resulted from the evaluation of the argument list, and with **this** bound to i and the type parameters (if any) of C bound to the actual type arguments $U_1, \ldots, U_m$. Then, the body of k is executed with respect to the bindings that resulted from the evaluation of the argument list. The result of the evaluation of e is i.

Otherwise, if k is a redirecting constructor, then: 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 k is executed resulting in an object i that is necessarily the result of another constructor call. The result of the evaluation of e is i.

Otherwise, k is a factory constructor. then: 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 k is executed resulting in an object i. The result of the evaluation of e is i.

The static type of a new 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.

9.10.2 Const

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

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

Let e be a constant object 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 compile-time error if T is not a class or interface optionally followed by type arguments.

If e 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 id is not the name of a constructor declared by the type T. If e of the form **const** $T(a_1, \ldots, a_n)$ it is a compile-time error if the type T does not declare a constructor with the same name as the declaration of T.

If T is a parameterized type $S < U_1, \ldots, U_m >$, let $R = S$. It is a compile time error if S is not a generic type with m type parameters. If T is not a parameterized type, let $R = T$. If R is an interface, let C be the factory class of R. Otherwise let $C = R$. Furthermore, 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 k be the constructor $C.id$, otherwise let k be the constructor C. Finally, if C 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:

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 C with type arguments $V_i, 1 \le i \le n$, then:
  - For each instance variable f of i, let v_{if} be the value of the f in i, and let v_{jf} be the value of the field f in j. If $v_{if} === v_{jf}$ 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 of 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 of 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.

9.10.3 Spawning an Isolate

9.11 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:

- $(r_1, \ldots, r_n,)\{$**return** $o.m(r_1, \ldots, r_n);\}$ if m has only required parameters $r_1, \ldots r_n$.
- $(r1, \ldots, rn, [p1 : r_{n+1}, \ldots, p_k : r_{n+k}])\{$**return** $o.m(r_1, \ldots, r_n, [p_1 : r_1, \ldots, p_k : r_k]);\}$ if m has required parameters $r_1, \ldots r_n$, and named parameters $p_1 \ldots p_k$.

Otherwise, if m is the name of a getter (**??**) member of o (declared implicitly or explicitly) then $o.m$ evaluates to the result of invoking the getter. **How does one extract a getter? Or a setter?** This means 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. Will we see style guides for Dart telling people to always define getters for nullary methods, and setters for methods with one required argument?

9.12 Function Invocation

Function invocation occurs in the following cases: when a function expression (**??**) is invoked(**??**), when a method is invoked (**??**)) or when a constructor is invoked (either via instance creation (**??**), constructor redirection (**??**) or super initialization). The various kinds of function invocation differ as to how the function to be invoked, f, is determined **Also the binding of this**. Once f has been determined, the actual argument list (**??**) of the invocation is evaluated and 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 (**??**) immediately nested in the body of f is executed.
- The last statement of the body completes execution.

9.12.1 Actual Argument List Evaluation

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

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

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

namedArgument:
  label expression
 ;
```

Let $a_0 \dots a_m$ be a list of actual arguments, and let f be the function being invoked, and let $p_0 \dots p_{k-1}$ be the positional parameters of f. It is necessarily true that $m \geq k-1$ and $a_0 \dots a_{k-1}$ are necessarily expressions $e_0 \dots e_{k-1}$. Then p_i is bound to the value of $e_i, 0 \leq i \leq k-1$. It is a dynamic type error if, in checked mode, e_i is not **null** and the type of p_i is not a supertype of the value of e_i.

If f declares named parameters $p_k \dots p_l$, then $a_{k+1} \dots a_m$ must be named arguments, or a runtime error occurs. For each $p_i, k \leq i \leq l$, if there exists an $a_j, k \leq j \leq m$, such that a_j is a named argument whose name is the same as p_i, then p_i is bound to a_j. All other formal parameters are bound to their default values. It is a dynamic type error if, in checked mode, e_i is not **null** and the type of p_i is not a supertype of the value of a_j.

9.12.2 Unqualified Invocation

If there exists a lexically visible function named id, let f_{id} be the innermost such function. Then: If f_{id} is a local function, a library function or a variable then i is interpreted as a function expression invocation. Otherwise, if f_{id} is a static method of the enclosing class C, i is equivalent the static method invocation $C.id(a_1, \dots, a_n, x_{n+1} : a_{n+1}, \dots, x_{n+k} : a_{n+k}])$.

Otherwise, i is equivalent to the ordinary method invocation **this**.$id((a_1, \dots, a_n, x_{n+1} : a_{n+1}, \dots, x_{n+k} : a_{n+k}])$. **What about inherited static methods?**

9.12.3 Function Expression Invocation

A function expression invocation i has the form $e_f(a_1, \dots, a_n, x_{n+1} : a_{n+1}, \dots, x_{n+k} : a_{n+k}])$, where e_f is an expression. If e_f is an identifier id, then id must necessarily denote a local function, a library function 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. Otherwise:

Evaluation of a function expression invocation $e_f(a_1, \dots, a_n, x_{n+1} : a_{n+1}, \dots, x_{n+k} : a_{n+k}])$ proceeds as follows:

First, e_f is evaluated to a value v_f of type T_f. It is a run-time error if T_f is not of a function type. 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 v_f is then executed with respect to the bindings that resulted from the evaluation of the argument list.

The static type of i is the declared return type of v_f.

9.13 Method Invocation

Method invocation can take several forms as specified below.

9.13.1 Ordinary Invocation

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

The result of looking up a method m in object o is the result of looking up method m in class C, where C is the class of o.

The result of looking up method m in class C is: If C declares a method named m, 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. 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. Let f be the result of looking up method m in v_o. 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.

If the method lookup has failed, then the method noSuchMethod() is looked up in o and invoked with arguments m, $[e_1, \ldots, e_n]$ and $\{x_{n+1} : e_{n+1}, \ldots, x_{n+k} : e_{n+k}\}$, 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 instance member named m. **Check arguments if $T.m$ exists**. 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**.

9.13.2 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 compile-time error if C does not denote a class in the current scope. It is a compile-time error if C does not declare a static method m.

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:

First, 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 the method m declared in class C is then executed with respect to the bindings that resulted from the evaluation of the argument list.

The static type of i is the declared return type of the method $C.m$.

9.13.3 Super Invocation

A super method invocation 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 an super method invocation i of the form

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

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 f be the result of looking up method m in S, the superclass of the class of **this**. 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**.

If the method lookup has failed, then the method noSuchMethod() is looked up in **this** and invoked with arguments m, $[e_1, \ldots, e_n]$ and $\{x_{n+1} : e_{n+1}, \ldots, x_{n+k} : e_{n+k}\}$, and the result of this invocation is the result of evaluating i.

It is a static type warning if S does not have an instance member named m. **Check arguments if** $S.m$ **exists.** The static type of i is the declared return type of $S.m$, if $S.m$ exists; otherwise the static type of i is **Dynamic**.

9.13.4 Sending Messages

9.14 Getter Invocation

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

The result of looking up a getter (respectively setter) m in object o is the result of looking up getter (respectively setter) m in class C, where C is the class of o.

The result of looking up getter (respecitively setter) m in class C is: If C declares a getter (respectively setter) named m, 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. Otherwise, we say that the lookup has failed.

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 (**??**) m is looked up in o, and invoked. 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 the method noSuchMethod() is looked up in o and invoked with arguments get:v , [] and $\{\}$, 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:

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 compile-time error 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 $C.m$.

9.15 Assignment

An assignment changes the value associated with a mutable variable or property.

```
assignmentOperator:
  '=' |
 compoundAssignmentOperator
;
```

Evaluation of an assignment of the form $v = e$ proceeds as follows:

If there is no declaration d with name v in the lexical scope enclosing the assignment, then 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**.$v = e$.

It is a dynamic type error if, in checked mode, o is not **null** and the interface induced by the class of o is not a subtype of the static type of v.

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

Evaluation of an assignment of the form $C.v = e$ proceeds as follows:

The expression e is evaluated to an object o. Then, the setter $C.v$ is invoked with its formal parameter bound to o. The value of the assignment expression is o.

It is a compile-time error if there is no class C in the enclosing lexical scope of assignment, or if C does not declare, implicitly or explicitly, a setter v. It is a dynamic type error if, in checked mode, o is not **null** and the interface induced by 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 $C.v$.

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 in o_1, and invoked with its formal parameter bound to o_2. If the setter lookup has failed, then the method noSuchMethod() is looked up in o_1 and invoked with arguments set:v ,

[] and $\{o_2\}$.The value of the assignment expression is o_2 irrespective of whether setter lookup has failed or succeeded.

It is a dynamic type error if, in checked mode, o_2 is not **null** and the interface induced by the class of o_2 is not a subtype of the static type of $e_1.v$.

It is a static type warning if the static type of e_2 may not be assigned to the static type of $e_1.v$.

9.15.1 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).

```
compoundAssignmentOperator:
   *=' |
  '/=' |
  ' /=' |
  '%=' |
  '+=' |
  '-=' |
  '<<=' |
  '>' '>' '>' '=' ? |
  '<' '<' '=' ? |
  '&=' |
  '≙' |
  '|='
;
```

9.16 Conditional

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

```
conditionalExpression:
  logicalOrExpression ('?' expression ':' expression)?
;
```

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 (**??**) 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 is not bool. The static type of c is the least upper bound of the static type of e_2 and the static type of e_3.

Define LUB

9.17 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 (**??**), 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 (**??**) 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.

9.18 Bitwise Expressions

Bitwise expressions invoke the bitwise operators on objects.

```
bitwiseOrExpression:
  bitwiseXorExpression ('|' bitwiseXorExpression)* |
 super ('|' bitwiseXorExpression)+
 ;

bitwiseXorExpression:
  bitwiseAndExpression ('ˆ' bitwiseAndExpression)* |
 super ('ˆ' bitwiseAndExpression)+
 ;

bitwiseAndExpression:
  equalityExpression ('&' equalityExpression)* |
  super ('&' equalityExpression)+
 ;

bitwiseOperator:
   '&' |
  '^' |
  '|'
 ;
```

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

9.19 Equality

Equality expressions test objects for identity or equality.

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

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

An *equality expression* is either a relational expression (**??**), or an invocation of a equality operator on either **super** or an expression e_1, with argument e_2.

An equality expression of the form $e_1 == e_2$ is equivalent to the method invocation $e_1.==(e_2)$. An equality expression of the form **super** $== e_2$ is equivalent to the method invocation **super**.==(e_2).

An equality expression of the form $e_1 == e_2$ is equivalent to the method invocation $e_1.==(e_2)$. An equality expression of the form **super** $== e$ is equivalent to the method invocation **super**.==(e).

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)).

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; then the expression e_2 is evaluated to an object o_2. Next, if o_1 and o_2 are the same object, then ee evaluates to **true**, otherwise ee evaluates to **false**.

An equality expression of the form **super** === e is equivalent to the expression **this** === e. An equality expression of the form **super** !== e is equivalent to the expression !(**super** === e).

An equality expression of the form e_1 !== e_2 is equivalent to the expression !(e_1 === e_2).

The static type of an equality expression of the form e_1 == e_2 is bool.

The static types of other equality expressions follow from the definitions above. The forms e_1 != e_2, e_1 !== e_2 and **super** != e are negations and have static type bool. The expression e_1 == e_2 is typed as a method invocation so its static type depends on the operator method declaration.

9.20 Relational Expressions

Relational expressions invoke the relational operators on objects.

```
relationalExpression:
  shiftExpression (isOperator type | relationalOperator shiftExpres-
sion)? |
 super relationalOperator shiftExpression
 ;

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

A *relational expression* is either an shift expression (**??**), 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**.$op(e_2)$.

9.21 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 (**??**), 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.

9.22 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 (**??**), 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**.$op(e_2)$.

9.23 Multiplicative Expressions

Multiplicative expressions invoke the multiplication operators on objects.

```
multiplicativeExpression:
  unaryExpression (multiplicativeOperator unaryExpression)* |
 super (multiplicativeOperator unaryExpression)+
 ;

multiplicativeOperator:
  ’*’ |
 ’/’ |
 ’%’ |
 ’ /’
 ;
```

A *multiplicative expression* is either an unary expression (**??**), 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**.$op(e_2)$.

9.24 Unary Expressions

Unary expressions invoke unary operators on objects.

```
unaryExpression:
  prefixExpression |
 postfixExpression |
 unaryOperator super |
 ’-’ super
 ;
|
 incrementOperator assignableExpression
 ;
```

A *unary expression* is either a prefix expression (**??**), a postfix expression (**??**), or an invocation of a unary operator on on either **super** or an expression e.

9.25 Prefix Expressions

Prefix Expressions invoke prefix operators on objects.

```
prefixExpression:
  prefixOperator unaryExpression
 ;
```

Evaluation of a prefix expression of the form $++e$ is equivalent to e += 1. Evaluation of a prefix expression of the form $-e$ is equivalent to e -= 1.

9.26 Postfix Expressions

Postfix expressions invoke the postfix operators on objects.

```
postfixExpression:
  assignableExpression postfixOperator |
 primary selector*
 ;

postfixOperator:
  incrementOperator
 ;

incrementOperator:
  '++' |
 '–'
 ;
```

A *postfix expression* is either a primary expression, a function, method or getter invocation, or an invocation of a postfix operator on an expression e.

Evaluation of a postfix expression of the form v++ 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.

Evaluation of 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).

Evaluation of a postfix expression of the form e– is equivalent to e ++ (-1).

9.27 Assignable Expressions

Assignable expressions are expressions that can appear on the left hand side of an assignment.

Of course, if they 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 (arguments* assignableSelector)+ |
  super assignableSelector |
  identifier
 ;

selector:
   assignableSelector |
  arguments
 ;

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

An assignable expression is either:

- An identifier.
- An invocation of a method, getter (**??**) or array access operator on an expression *e*.
- An invocation of a getter or array access operator on **super**.

An assignable expression of the form *id* is evaluated as an identifier expression (**??**).

An assignable expression of the form $e.id(a1, , an)$ is evaluated as a method invocation (**??**).

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

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**.$[e_2]$.

9.28 Identifier Reference

An identifier expression consists of a single identifier; it provides access to an object via an unqualified name.

```
identifier:
  x IDENTIFIER_NO_DOLLAR |
  IDENTIFIER |
  ABSTRACT |
  ASSERT |
  CLASS |
  EXTENDS |
  FACTORY |
  FUNCTION |
  GET |
  IMPLEMENTS |
  IMPORT |
  INSTANCEOF |
  INTERFACE |
  IS |
  LIBRARY |
  NATIVE |
  NEGATE |
  OPERATOR |
  SET |
  SOURCE |
  STATIC |
  TYPEDEF
 ;

IDENTIFIER_NO_DOLLAR:
  IDENTIFIER_START_NO_DOLLAR IDENTIFIER_PART_NO_DOLLAR*
 ;

IDENTIFIER:
   IDENTIFIER_START IDENTIFIER_PART*
 ;

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)?
;
```

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 d is a library variable, local variable, or formal parameter, then e evaluates the current binding of id. Otherwise, if d is the declaration of a static variable declared in class C, then e is equivalent to the getter invocation $C.id$. Otherwise, d is necessarily the declaration of an instance variable, and e is equivalent to the getter invocation **this**.id.

9.29 Type Test

The is-expression tests if an object is a member of a 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 induced by 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 or T = Null. Since the class Null is not exported by the core library, the latter will not occur in user code. The former is useless, as is anything of the form e **is** Object. 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).

The static type of an is-expression is bool.

10 Statements

```
statements:
  statement* |
statement:
   label* nonLabelledStatement
 ;

nonLabelledStatement:
   block |
  initializedVariableDeclaration ';' |
  forStatement |
  whileStatement |
  doStatement |
  switchStatement |
  ifStatement |
  tryStatement |
  breakStatement |
  continueStatement |
  returnStatement |
  throwStatement |
  expressionStatement |
  assertStatement |
  functionDeclaration functionBody
 ;
```

10.1 Blocks

A block statement supports sequencing of code.

Execution of a block statement $\{s_1 s_n\}$ proceeds as follows:

For $i \in 1 \rightarrow n, s_i$ is executed.

10.2 Expression Statements

An expression statement consists of an expression.

```
expressionStatement:
   expression? ';' |


 ;
```

Execution of an expression statement e; proceeds by evaluating e.

10.3 Variable Declaration

A variable declaration statement declares a new local variable.

A variable declaration statement T id; or T $id = e$; introduces a new variable id with static type T into the innermost enclosing scope. A variable declaration statement **var** id; or **var** $id = e$; introduces a new variable named id with static type **Dynamic** into the innermost enclosing scope. In all cases, iff the variable declaration is prefixed with the **final** modifier, the variable is marked as final.

Rules about shadowing etc.

Executing a variable declaration statement T $id = e$; is equivalent to evaluating the assignment expression $id = e$, except that the assignment is considered legal even if the variable is final.

However, it is still illegal to assign to a final variable from within its initializer.

A variable declaration statement of the form Tid; is equivalent to T $id =$ **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.

10.4 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 (**??**), 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 is not bool.

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

10.5 For

The for statement supports iteration.

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

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

forInitializerStatement:
  initializedVariableDeclaration ';' |
  expression? ';'
 ;
```

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

10.5.1 For Loop

A for statement of the form **for** $(e_1\ ;\ c;\ e_3)\ s$ is equivalent to the the following code:
$\{e_1;$ **while** $(c)\ \{\ s\ e_3;\ \}\}$

10.5.2 Foreach

A for statement of the form **for** $(finalVarOrType$ id **in** $e)\ s$ is equivalent to the the following code:
var n0 = e.iterator(); **while** (n0.hasNext()) $\{\ finalVarOrType$ id = n0.next(); $s\ \}$ where n0 is an identifier that does not occur anywhere in the program.

10.6 While

The while statement supports conditional iteration, where the condition is evaluated prior to the loop.

```
  whileStatement:
     while '(' expression ')' statement
   ;
```

Execution of a while statement of the form **while** $(e)\ s$; proceeds as follows:
The expression e is evaluated to an object o. Then, 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 (**??**), 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 is not bool.

10.7 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 (**??**), 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 is not bool.

10.8 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? (case expression ':')* default ':' statements
 ;
```

Execution of a switch statement **switch** (e) $\{$ **case** e_1 : $s_1 \ldots$ **case** e_n : s_n **default** s_{n+1} $\}$ proceeds as follows:

The statement **var** n = e; is evaluated, where n is a variable whose name is distinct from any other variable in the program. Next, the case clause **case** e_1 : s_1 is executed if it exists. If **case** e_1 : s_1 does not exist, then the default clause is executed by executing s_{n+1}.

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

The expression n $== e_k$ is evaluated to a value v.

If v is **false**, or if s_k is empty, 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**, the statement sequence s_k is executed.

A switch statement **switch** (e) $\{$ **case** $e_1 : s_1 \ldots$ **case** $e_n : s_n$ $\}$ is equivalent to switch statement **switch** (e) $\{$ **case** $e_1 : s_1 \ldots$ **case** $e_n : s_n$ **default** $\}$

It is a static warning if the type of e is may not be assigned to the type of e_k for $0 \leq k \leq n$.

10.9 Try

The try statement supports the definition of exception handling code in a structured way.

```
tryStatement:
  try block (catchPart+ finallyPart? | finallyPart)
 ;

catchPart:
  catch '(' simpleFormalParameter (', ' simpleFormalParameter)?
')' block
 ;

finallyPart:
  finally block
 ;
```

A try statement consists of a block statement, followed by at least one of:

1. A set of **catch** clauses, each of which specifies one or two exception parameters and a block statement.
2. An **finally** clause, which consists of a block statement.

A **catch** clause **catch** $(T\ p_1,\ T\ p_2)$ s matches an object o if o is **null** or if the type of o is a subtype of T.

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

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

The **catch** clauses are examined in order, starting with $catch_1$, until either a **catch** clause that matches the current exception is found, or the list of **catch** clauses has been exhausted. If a **catch** clause $catch_k$ is found, then p_{k1} is bound to the current exception, p_{k2} is bound to the current stack trace, and then $catch_k$

is executed. If no **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 a **catch** clause **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 current 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 $catch_1 \ldots catch_n$ **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 **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 **catch** was found, it will execute first, and then the **finally** clause will be executed.

If an exception is raised during execution of a **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.

10.10 Return

The return statement consists of the reserved word **return** and an optional expression.

```
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.

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

It is quite easy to forget to 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 relay on a typechecker to prohibit use of **void** functions in expressions. Hence, a return statement must always return a value, even if no expression is specified.*

*The question than 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.*

10.11 Labels

A label is an identifier. A labeled statement is a statement prefixed by a label L. A *labeled case clause* is a case clause within a switch statement (**??**) prefixed by a label L.

The sole role of labels is to provide targets for the break (??) and continue (??) 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 member, variables and declarations. **Bug 4974299**

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 useful target for code generation.

10.12 Break

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

```
breakStatement:
  break identifier? ';'
;
```

If a label L is provided, a break statement transfers control to the innermost enclosing labeled statement with the label L. It is a compile-time error if no such a statement exists within the innermost function in which the break statement occurs. **Is it actually a runtime error?**

If no label is provided, a break statement transfers control to the innermost enclosing do (??), for (??), switch (??) or while (??) statement. It is a compile-time error if no such a statement exists within the innermost function in which the break statement occurs.

Need to deal with complications due to try-finally etc.?

10.13 Continue

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

```
continueStatement:
  continue identifier? ';'
;
```

A continue statement with label L behaves exactly like a break statement (**??**) with label L except that:

- A compile time time error occurs if the statement to which control would be transferred is not a do (**??**), for (**??**), or while (**??**) statement, or a case clause of a switch (**??**) statement.
- If label L marks a case clause, the control is transferred to that clause.

A continue statement with no label behaves exactly like a break statement with no label, except that it is a compile time error if the statement to which control would be transferred is a switch statement.

10.14 Throw

The throw statement consists of the reserved word **throw** and an optional expression.

```
throwStatement:
  throw expression? ';'
 ;
```

The current exception is the last unhandled exception thrown. The current stack trace is a record of all the function activations within the current isolate that had not completed execution at the point where the current exception was thrown. For each such function activation, the current 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.

Should we make it unambiguous that line numbers aren't good enough?

Execution of a throw statement of the form **throw** e; proceeds as follows:

The expression e is evaluated yielding a value v. Then, control is transferred to the nearest dynamically enclosing exception handler (**??**), with the current exception set to v and the current stack trace set to the series of activations that led to execution of the current function.

Need to specify what that means. Also, we need a consistent story on abrupt completion. Rather than scatter if x completes normally .. everywhere, we should clearly state that is the default interpretation of evaluating things.

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

Execution of a statement of the form **throw**; proceeds as follows: Control is transferred to the nearest innermost enclosing exception handler (**??**).

No change is made to the current stack trace or the current exception.

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

10.15 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 scripting mode. In checked mode, execution of an assert statement assert(e); proceeds as follows:

The conditional expression e is evaluated to an object o. In checked mode, it is a dynamic error if o is not of type bool. Otherwise, o is subjected to boolean conversion (**??**). If the resulting value 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.

Might be cleaner to define it as if (!e) {**throw new** AssertionError();} **(in checked mode only). What about an error message as part of the assert?**

It is a static type warning if the type of e is not 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 scripting 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.

11 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 (**??**), an interface (**??**), a type declaration, a function (**??**)or a variable declaration (**??**).

topLevelDefinition:
  classDefinition |
  interfaceDefinition |
  functionTypeAlias |
  functionDeclaration functionBody |
  returnType? getOrSet identifier formalParameterList function-Body |
  final type? staticFinalDeclarationList ';' |
  variableDeclaration ';'
  ;

libraryDefinition:
  scriptTag? libraryName import* include* topLevelDefinition*
  ;

```
scriptTag:
   '#!'
  ;
* NEWLINE  |

libraryName:
   '#' 'library' '(' stringLiteral ')' ';'  |
```

A library 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. A script tag begins with the characters #! and ends at the end of the line. Any characters after #! are ignored by the Dart implementation.

Libraries are units of privacy. A private declaration declared within a library L can only be accessed by code within L. Any attempt to access such a declaration from outside L will cause a run-time error.

Which one? Since top level privates are not imported, using them is a compile time error and not an issue here. Any other name will cause a NoSuchMethodError?

The scope of a library L consists of the names introduced of all top level declarations declared in L, and the names of all public top level declarations declared in any library imported by L.

11.1 Imports

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

```
libraryImport:
   '#' 'import' '(' stringLiteral (', ' 'prefix:
 ' stringLiteral)? ') ';'
  ;
```

An import specifies a URI where the declaration of the imported library is to be found. The effect of an *import of library B with prefix P within the declaration of library A* is"

- If P is the empty string, each non-private top level declaration d of B is added to the scope of A.
- Otherwise, each non-private top level declaration d of B is added to the scope of A under the name $P.d$.

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. If we reify libs, we need to avoid these assumptions. An import will simply be a familiar syntax for something more like named parameters.

An imported library must not define a name that is defined by the importing library, or a compile-time error occurs. *This implies that it is a load-time error for a library to import itself, as the names of its members will be duplicated.*

The *current library* is the library currently being compiled.

Compiling an include directive of the form #import(s_1, prefix: s_2); proceeds as follows:

- If the contents of the URI that is value of s_1 have not yet been compiled in the current isolate then they are compiled to yield a library B. It is a compile-time error if s_1 does not denote a URI that contains the source code for a Dart library.
- Otherwise, the contents of URI denoted by s_1 have been compiled into a library B within the current library.
- Then, the library B is imported into the current library with prefix p, where p is the value of s_2.

Compiling an include directive of the form #import(s) is equivalent to compiling the directive #import(s, prefix: '');.

It is a compile-time error to import two or more libraries that define the same name.

It is a compile-time error if either s_1 or s_2 is not a compile-time constant.

11.2 Includes

An *include directive* specifies a URI where a Dart compilation unit that should be incorporated into the current library may be found.

```
include:
  '#' 'source' '(' stringLiteral ')' ';'
 ;

compilationUnit:
  topLevelDefinition* EOF
 ;
```

A *compilation unit* is a sequence of top level declarations.

Compiling an include directive of the form #source(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 compilation unit.

It is a compile-time error if s is not a compile-time constant.

11.3 Scripts

A *script* is a library with a top level function main(). A script S may be executed as follows:

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

12 Types

Dart has a built-in optional type system 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.

12.1 Static Types

Static type annotations are used in variable declarations (**??**) (including formal parameters (**??**)) and in the return types of functions (**??**). Static type annotations are used during static checking and when running programs in checked mode. They have no effect whatsoever in scripting mode.

```
type:
  qualified typeArguments?
 ;

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. 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 does not preclude successful compilation and execution of Dart code.

Can we do something with non-nullable types?

It is a static type warning if a generic type is not provided the correct number of type arguments, or if any of those type arguments are not subtypes of the bounds of the corresponding formal type parameters.

12.2 Dynamic Type System

A Dart implementation must support execution in both scripting 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.

Need to specify what the dynamic type of a variable is; also (dynamic) type of an object is; dynamic type of a function

12.3 Type Declarations

12.3.1 Typedef

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

```
functionTypeAlias:
  typedef functionPrefix typeParameters? formalParameterList ';'
;
```

The effect of a type alias of the form **typedef** $T\ id(T_1 a_1, \ldots, T_n a_n, [x_{n+1} : T_{n+1} a_{n+1}, \ldots, x_{n+k} : T_{n+k} a_{n+k}])$ declared in a library L is is to introduce the name id into the scope of L, bound to the function type $(T_1, \ldots, T_n, [x_{n+1} : T_{n+1}, \ldots, x_{n+k} : T_{n+k}]) \to T$.

Currently, type aliases are restricted to function types.

12.4 Interface Types

An interface I is a direct supertype of an interface J iff:

- If I is Object, and J has no **extends** clause and no interface injection declaration has extended J.
- if I is listed in the **extends** clause of J or an interface injection declaration has extended J with I.

Can wacky stuff happen with interface injection, e.g., a direct superinterface becomes indirect? What about side effects - loading

order can effect type relationships. The supertypes of an interface are its direct supertypes and their supertypes.

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 $\perp$.
- S is *dynamic*.
- S is a direct supertype of T.
- T is a type variable 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 $[\perp/dynamic]T << S$.

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: Array $<:$ Array $<$ String $>$ and Array $<$ int $><:$ Array, but Array $<$ int $>$ is not a subtype of Array $<$ 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.

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.

12.5 Function Types

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. $\forall 1 \leq i \leq n, T_i \iff S_i$.

3. $\{x_1, \ldots, x_k\}$ is a superset of $\{y_1, \ldots, y_m\}$.

4. For all $y \in \{y_1, \ldots, y_m\} S_y \iff T_y$

In addition, a function type $(T_1, \ldots, Tn, [T_{n+1}x_{n+1}, \ldots, T_{n+k}x_{n+k}]) \rightarrow T$ is a subtype of the function type $(T_1, \ldots, T_n, T_{n+1}, [T_{n+2}x_{n+2}, \ldots, T_{n+k}x_{n+k}]) \rightarrow T$.

This second rule is attractive to web developers, who are used to this sort of flexibility from Javascript. However, it may be costly to implement efficiently. **Should we do this or not?**

We write $(T_1, \ldots, T_n) \rightarrow T$ as a shorthand for the type $(T_1, \ldots, T_n, []) \rightarrow T$.

All functions implement the interface Function, so all function types are a subtype of Function.

Need to specify how a function values dynamic type is derived from its static signature.

12.6 Type Dynamic

The type **Dynamic** is 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 the corresponding type arguments are not provided, then the missing type arguments default to the unknown type.

This means that given a generic declaration $G < T_1, \ldots, T_n >$, the type G is equivalent to $G <$ **Dynamic**$, \ldots,$ **Dynamic** $>$.

Type **Dynamic** has methods for every possible identifier and arity. 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.

12.7 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) $\perp$ <: **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**; returning any other result will cause a type error (what about dynamic? that should is not an error by current rules? What if it is undeclared but intended as a void? On the other hand, we will miss real errors this way. But it would need a special rule to do otherwise).

13 Reference

13.1 Lexical Rules

13.1.1 Reserved Words

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

13.1.2 Built-in Identifiers

abstract, **assert**, **class**, **Dynamic**, **extends**, **factory**, **get**, **implements**, **import**, **interface**, **is**, **library**, **native**, **negate**, **operator**, **set**, **source**, **static**, **typedef**.

Unicode characters.

```
LETTER:
   'a'
 ;

 ;
'z' |
  'A'
 ;

 ;
'Z'
 ;

DIGIT:
   '0'
 ;

 ;
'9'
 ;

WHITESPACE:
   ('⁀' — ' ' — NEWLINE)+
 ;
```

13.1.3 Comments

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

```
SINGLE_LINE_COMMENT:
   '//'  (NEWLINE)* (NEWLINE)?
 ;

MULTI_LINE_COMMENT:
   '/*' (
 ;
)*
 ;
```

Dart supports both single-line and multi-line comments. A single line comments begins with the token //. Everything between // 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.

This needs to be specified in detail.

13.2 Grammar

13.3 Operator Precedence

Operator precedence is given implicitly by the grammar.

We expect to have a table here anyway.

13.4 Glossary