Dartプログラミング言語仕様書（第3版）
Dart Programming Language Specification

Version 3
ECMA-408
June 2015

翻訳経歴:

2012年 1月 10日（0.06版）
2012年 1月 30日（0.07版）
2012年 4月 02日（0.08版）
2012年 5月 21日（0.09版）
2012年 6月 11日（0.10版）
2012年 10月 09日（0.11版）
2012年 12月 03日(0.12 M1リリース版）
2013年 1月 05日（0.20 M2リリース版）
2013年 2月 25日（0.30 M3リリース版）
2013年 5月 07日（0.40 M4リリース版）
2013年 7月 01日（0.42 M4+ リリース版、M5とも称している）
2013年 9月 10日（0.51版）
2013年 9月 30日（0.60 M6リリース版）
2013年 10月 28日（0.70版）
2013年 11月 25日(1.0版）
2014年 01月 27日（1.1版）
2014年 03月 17日（1.2版）
2014年 09月 01日（ECMA-408 第1版） これは書式がECMAの標準に合わせただけで、基本的には1.2版と同じものである
2015年 01月 26日（1.3版 ECMA-408 第2版）
2015年 12月 14日（ECMA-408 第2版）。
第2版からの変更箇所はこの色の背景色で示してある。

0.12版ではM1（マイルストン1リリース）と表現されている。これは2012年9月のMicrosoftからのTypeScript発表に反応し、一応一般使用が可能になったとの意味をもたせたものである。

2013年11月に1.0版が出版されdraftという言葉が消えた。1.0版以降は変更記録が無いので、翻訳する側としては全文突合せをしなければならず、迷惑である。

2014年6月末にECMAの総会はECMA-408 第1版を承認した。
2014年12月にECMAの総会はECMA-408 第2版を承認した。
2015年6月17日にMontreuxで開催されたECMAの総会はECMA-408 第3版を承認した。

翻訳の精度は保証しないので、不明な箇所は原文を見て頂きたい。

Dart 言語の標準化

ECMA 第 1 版

2013 年 12 月 13 日に Google は ECMA が Dart の標準化のための新しい技術委員会 TC52 を設立したと発表した。Google はこの TC52 を介してウェブのコミュニティと協働しこの言語の発展を推進すると述べている。TC52 の委員長は Google デンマークの Anders Thorhauge Sandholm である。

2014 年 3 月 13 日に開催された TC52 会合の報告として、言語仕様書担当の Gilad Bracha は次のように報告している:

- 我々は 1.2 仕様書に少し手を加えたものをこの委員会に標準化のための言語仕様書案として提出する計画である。この委員会が承認すれば、6 月末までに公式の標準ができる。
- 次回会合はデンマークの Aarhus で 5 月 1 日に開催される。
- 彼としては年末までに enums 及び deferred loading を付加した改定版をまとめたいと考えており、これらの機能提案を 5 月の会合で提出する。

2014 年 5 月 2 日に Gilad Bracha は次のように報告している:

- TC52 は正式に現在の仕様(1.3 版)を承認した。
- 6 月末には批准されよう。
- 1.3 版仕様は ECMA のサイトで取得できるが、これは Dart のサイトにある 1.2 版に ECMA の書式を先頭に付加しただけのものである。
- TC52 の別の会合では enums、deferred loading および非同期対応の付加に関して議論した。これらの機能追加の批准は時間がかかり 12 月になりそうである。
- 次回会合は 7 月 1 日および 9 月 16 日に仮設定(in Zurich)。

2014 年 7 月 2 日に Gilad Bracha は次のように報告している:

- ECMA は 6 月末の第 107 回全体会議で正式に 1.3 版を承認した。
- TC52 の第 3 回会合では enums, deferred loading, async, and minor bug fixes が討議された。
- 次回は 9 月 16 日にスイスで開催される。

Dart チームからの発表はここにある。

ECMA 第 2 版

2014 年 11 月 21 日に 1.6 版のものが Draft: Dart Programming Language Specification, 2nd Edition として公開されている。これは TC52 で承認されたものだが、まだ全体会議では未承認のものである。第 2 版では以下のものが追加されている:

- 列挙型 (enum)　1.8 版で実装済み
- 非同期関係(async, await ほか)　部分的に 1.8 版で第 1 フェーズとして実装されている。
- 後回しのロード(import ... deferred as)　1.6 版で実装済み

2014 年 12 月 11 日に Sapporo で開催された第 108 回全体会議で ECMA-408 2nd edition が承認された。

ECMA 第 3 版

2015 年 1 月 30 日に Dart の広報役の Seth Ladd が、1 月 14 日に開催された Ecma TC52 会合の報告をしている。

この会合では以下に記す幾つかの更なる改善事項が討議され、更に詳細を詰めることとした。今後数カ月かけて提案がなされる模様である。

- 設定可能インポート(Configurable Imports)

ライブラリによってはサーバでのみまたはクライアント（特にもし dart:html がインポートされてしまっているとサーバでの実行は出来なくなる）でのみ機能するものがある。条件付きコンパイルの類の機能が付加されると、どの環境に対しどのライブラリをインポートするかを選べるようになる。ただ条件付きコンパイルの機能は‘part’のメカニズムで実現されているとの指摘があった。

もうひとつの提案はパラメタ化されたライブラリ(parameterized libraries)というより複雑で強力なもので、ライブラリに型引数をもたせたり、他のライブラリを引数にしたり出来るようにする。

- 共用体(Union Types)

Dart の型システムは柔軟なので他の言語ほど共用体は重要でなくこれまで何度かこの提案は後回しにされて来ていた。しかし JavaScript との相互運用性が重要になってきており、JavaScript と同じ機能を実現する為に Dart にこの機能を待たせたほうが良い場合がある。また型 T または Future<T>を引数とする関数で、その関数がどちらかを判断させるケースが考えられる。

もうひとつの例として、異なった引数の数（アリティ）をもつ関数である。共用体の問題は構文解析で不一致を起こしがちなことで、これは typedef 宣言で制限できるが、総称体ではそうはいかない（ローカルな typedef で解決できるかも）。静的型警告を出さない操作は共用体演算子の総てのオペランドに共通した副型にたいするものとなる。つまり総ての型で共有される操作に限られる。これにより一連の型関連コンパイルを避けることができる。

Dart の型システムにおける代入可能性規則（副型規則ではなく）ではある共用体を持った値がいろんなオペランド型で使えるので、Dart では共用型は良く機能する。

- 総称メソッド(Generic Methods)

これは多相型メソッド(polymorphic methods)とも呼ばれる型引数をとるメソッドである。これには負荷がかかる型推論が必要で、Dart では現在対応していない。Dart に導入する際は型推論を使わないよう制限される。即ち総ての型引数は明示的に指定するか、コンパイラが<dynamic..dynamic>として暗示的に定められるようになる。

- 一般化されたティアオフ(Generalized Tear-Offs)

属性抽出メカニズム（クロージャ化またはティアオフとも呼ばれる）では現在幾つかの問題がある：

現在の文法では演算子、ゲッタ、セッタ、コンストラクタでは機能しないし、更に‘a.m’が属性抽出やゲッタ呼びだしとして実行されるべきかどうかが実行時に分からないので最適化ができない。提案されている文法では‘.’の代わりに‘#’を使ってレシーバとセレクタを切り分けるものである。互換性の為これまでの文法も共存させる。

- Null ベースの演算子(Null-Aware Operators)

これは以前から要求されていた機能である。a?.b 演算子(Elvis operator)では a が null のとき結果は null になり、a??b では a が null のときデフォルトとして b の値をとり、また a ?= b では a が null のときに限り a に b の値が入る。

- 型プロモーションの改善(Improvements to Type Promotion)

これは純粋に静的型システムの問題で、ある変数の型テストが先行した制御フローの中で存在しているときの型づけに関する。現在型プロモーションに対しては幾つかの規則をかけてきている。これは窮屈だという意見があるので緩和を検討している。

- その他のマイナーな変更
    - ‘await throw’を throw immediately に変更
    - ‘noSuchMethod’が宣言されているクラスのオブジェクトたちの型チェックの使用に関する規則は一般化が必要
    - 'main'という名前はトップ・レベルの関数以外にも使えるようにする
    - doc コメントがインポートされたスコープにアクセスできるようにする

015 年 6 月 17 日に Montreux で開催された ECMA の総会では第 3 版が承認され、TC52 の委員長の Anders Sandholm は次のように説明している:

> 「第 3 版での主たる追加は null ベースの演算子たちと一般化されたティアオフである。null ベースの演算子たちで文法の短縮化をもたらす。例えば「安全なナビゲーション(safe navigation)」演算子の?.を導入したが、これは o?.m ではもし o が null と計算されるときは null を返し、そうでないときは o.m を返す。
> 一般化されたティアオフでは、我々は明示的な文法(#)を追加したが、これはこれた単にメソッドだけでなくコンストラクタ、演算子、ゲッタ、及びセッタもまたクロジュア化を可能とするものである。」

著作権に関する注意

本 ECMA 標準（第 3 版）は 2015 年 6 月の全体会議で採択された。

目次

Dart 言語の標準化 2
ECMA 第 1 版.....2
ECMA 第 2 版.....2
ECMA 第 3 版.....3
著作権に関する注意 5
目次.....6
1. 適用範囲(Scope).....10
2. 適合性(Conformance).....10
3. 引用文書(Normative References).....10
4. 用語と定義(Terms and Defnitions).....10
5. 本仕様書の表記(Notation).....11
6. 概要(Overview).....13
6.1 スコープづけ(Scoping)（適用範囲化）.....14
6.2 プライバシ(Privacy).....15
6.3 並行処理(concurrency).....16
7. エラーと警告(Errors and Warnings).....17
8. 変数(variables).....19
8.1 暗示的変数ゲッタの計算(Evaluation of Implicit Variable Getters).....23
9. 関数(Functions).....24
9.1 関数宣言(Function Declarations).....25
9.2 仮パラメタ(Formal Parameters).....26
9.2.1 必要とされる仮パラメタ(Required Formals).....27
9.2.2 オプショナル仮パラメタ(Optional Formals).....28
9.3 関数の型(Type of a Function).....28
9.4 外部関数(External Functions).....29
10. クラス(Classes).....31
10.1 インスタンス・メソッド(Instance Methods).....33
10.1.1 演算子(Operators).....34
10.2 ゲッタ(Getters).....35
10.3 セッタ(Setters).....36
10.4 抽象インスタンス・メンバ(Abstract Instance Members).....37
10.5 インスタンス変数(Instance Variables).....38
10.6 コンストラクタ(Constructors).....39
10.6.1 生成的コンストラクタ(Generative Constructors).....39
10.6.2 ファクトリ(Factories).....43
10.6.3 常数コンストラクタ(Constant Constructors).....45
10.7 static メソッド(Static Methods).....48
10.8 static 変数(Static Variables).....48
10.9 スーパークラス(Superclasses).....48

10.9.1 継承とオーバライド(Inheritance and Overriding)....49
10.10 スーパーインターフェイス(Superinterfaces)....52
11. インターフェイス(Interfaces)....54
11.1 スーパーインターフェイス(Superinterfaces)....54
11.1.1 継承とオーバライド(Inheritance and Overriding)....54
12. ミクスイン(Mixins)....56
12.1 ミクスインのアプリケーション(Mixin Application)....56
12.2 ミクスイン構成(Mixin Composition)....57
13. 列挙型(Enums)....59
14. 総称型(Generics)....60
15. メタデータ(MetaData)....62
16. 式(Expressions)....63
16.1 定数(Constants)....65
16.2 ヌル(Null)....67
16.3 数(Numbers)....68
16.4 ブール値(Booleans)....69
16.4.1 ブール変換(Boolean Conversion)....69
16.5 文字列 (Strings)....70
16.5.1 文字列内挿入(String Interpolation)....74
16.6 シンボル(Symbols)....74
16.7 リスト(Lists)....75
16.8 マップ(Maps)....77
16.9 スロー(Throw)....78
16.10 関数式(Function Expressions)....79
16.11 This....81
16.12 インスタンス生成(Instance Creation)....82
16.12.1 New....82
16.12.2 Const....84
16.13 アイソレートの産み付け(Spawning an Isolate)....86
16.14 関数呼び出し(Function Invocation)....86
16.14.1 実引数リスト計算(Actual Argument List Evaluation)....88
16.14.2 実引数たちの仮パラメタたちへのバインド(Binding Actuals to Formals)....89
16.14.3 無修飾呼び出し（Unqualified Invocation）....90
16.14.4 関数式呼び出し(Function Expression Invocation)....90
16.15 検索(Lookop)....91
16.15.1 メソッド検索(Method Lookup)....91
16.15.2 ゲッタとセッタの検索 Getter and Setter Lookup()....92
16.16 トップ・レベル・ゲッタ呼び出し(Top Level Getter Incocation)....92
16.17 メソッド呼び出し(Method Invocation)....92
16.17.1 通常呼び出し(Ordinary Invocation)....92
16.17.2 カスケードされた呼び出し(Cascaded Invocations)....95
16.17.3 スーパー呼び出し(Super Invocation)....95

16.17.4 メッセージ送信(Sending Messages)....97
16.18 属性の抽出(Poperty Extraction)....97
16.18.1 ゲッタ・アクセスとメソッド抽出(Getter Access and Method Extraction)....98
16.18.2 スーパー・ゲッタ・アクセスとメソッドのクロージャ化(Super Getter Access and Method Closurization)....99
16.18.3 一般的クロージャ化(General Closurization)....100
16.18.4 指名コンストラクタ抽出(Named Constructor Extraction)....102
16.18.5 匿名コンストラクタ抽出(Anonymous Constructor Extraction)....102
16.18.6 一般的スーパー属性抽出(General Super Property Extraction)....102
16.18.7 通常のメンバー・クロージャ化 (Ordinary Member Closurization)....103
16.18.8 指名コンストラクタのクロージャ化(Named Constructor Closurization)....104
16.18.9 匿名コンストラクタのクロージャ化(Anonymous Constructor Closurization)....105
16.18.10 Super のクロージャ化 (Super Closurization)....105
16.19 代入(Assignment)....106
16.19.1 複合代入(Compound Assignment)....109
16.20 条件(Conditional)....111
16.21 If-null 式(If-null Expressions)....111
16.22 論理ブール式(Logical Boolean Expressions)....112
16.23 等価性(Equality)....113
16.24 関係式(Relational Expressions)....114
16.25 ビット単位式(Bitwise Expressions)....114
16.26 シフト(Shift)....115
16.27 加減算式(Additive Expressions)....116
16.28 乗除算式(Multiplicative Expressions)....116
16.29 単項式(Unary Expressions)....117
16.30 アウェイト式 (Await Expressions)....118
16.31 後置式(Postfix Expressions)....119
16.32 代入可能式(Assignable Expressions)....120
16.33 識別子参照(Identifier Reference)....121
16.34 型テスト(Type Test)....124
16.35 型キャスト(Type Cast)....126
17. 文(Statements)....127
17.1 ブロック(Blocks)....127
17.2 式文(Expression Statements)....128
17.3 ローカル変数宣言(Variable Declaration Statement)....128
17.4 ローカル関数宣言(Local Function Declaration)....129
17.5 If....130
17.6 For....131
17.6.1 for ループ(For Loop)....131
17.6.2 For-in....132
17.6.3 非同期 For-in....132
17.7 While....133
17.8 Do....133

17.9 Switch....134
17.10 Rethrow....137
17.11 Try....138
17.12 Return....141
17.13 ラベル(Labels)....144
17.14 Break....145
17.15 Continue....145
17.16 Yield と Yield-Each（Yield and Yield-Each）....146
17.16.1 Yield....146
17.16.2 Yield-Each....147
17.17 Assert....148
18. ライブラリとスクリプト(Libraries and Scripts)....150
18.1 インポート(Imports)....152
18.2 エクスポート(Exports)....157
18.3 パート(Parts)....158
18.4 スクリプト(Scripts)....159
18.5 URI....159
19. 型(Types)....161
19.1 静的型(Static Types)....161
19.1.1 型プロモーション(Type Promotion)....162
19.2 動的型システム(Dynamic Type System)....163
19.3 型宣言(Type Declarations)....164
19.3.1 Typedef....164
19.4 インターフェイス型(Interface Types)....165
19.5 関数型(Function Types)....167
19.6 dynamic 型(Type dynamic)....168
19.7 型 void (Type Void)....169
19.8 パラメタ化された型たち（Parameterized Types）....170
19.8.1 宣言の実際の型 Actual Type of a Declaration()....171
19.8.2 最小上界(Least Upper Bounds)....171
20. 参照(Reference)....173
20.1 構文規則(Lexical Rules)....173
20.1.1 予約語(Reserved Words)....173
20.1.2 コメント(Comments)....173
20.2 演算子の順位(Operator Precedence)....174
21. 付録：名前付け規約(Naming Conventions)....176
22. 参考（訳者追加）....177
22.1 和英対照表....177

1. 適用範囲(Scope)

本 Ecma 標準は Dart プログラミング言語の文法と意味を規定する。本仕様書は、それらのライブラリ要素たちが本言語それ自身の機能を正確に機能するに不可欠な場合（例えば noSuchMethod, runtimeType といったメソッドを持った Object クラスの存在）を除き、Dart ライブラリたちの API を規定してはいない。

2. 適合性(Conformance)

Dart プログラミング言語に適合した実装は、本仕様書で義務化されている API たちのすべて（トップ・レベル、スタティック、インスタンス、またはローカルにかかわらず、ライブラリ、型、関数、ゲッタ、セッタ）を用意しサポートしなければならない。

本言語に適合した実装は付加的な API たちを用意することは許されるが、本仕様書の次の版で導入され得る null 対応カスケードとティアオフに対応した実験的な機能を除き、付加的な文法を備えてはいけない。

3. 引用文書(Normative References)

以下の参照ドキュメントは本ドキュメントの適用にとって不可欠である。日付がつけられた参照ドキュメントは指定された版のみが適用される。日付がつけられていないドキュメントの場合は、本参照ドキュメント（何らかの修正を含め）の最新版が適用される。

1. Unicode 標準第 5 版、Unicode 5.1.0 またはその後継版で修正されたもの
2. Dart API 参照 https://api.dartlang.org/

4. 用語と定義(Terms and Defnitions)

本仕様書のなかで使われている用語と定義は本仕様書のしかるべき場所の中で示されている。そのような用語は導入された箇所でイタリックでハイライトしてある。

例 `we use the term *verbosity* to refer to the property of excess verbiage' 「我々は過剰な冗長を示すのに*冗長性（verbosity）*という用語を使用する」

5. 本仕様書の表記(Notation)

本仕様書では我々は標準のテキストと非標準のテキストを区別している。標準のテキストは Dart の規則を示している。それにはこのフォントが使われている。非標準のテキストは現時点では以下のものがある:

論理的根拠(Rationale)　言語設計決定の動機に関する議論はイタリック文字(斜め文字)で示してある。*非標準と標準の区別は、このテキストのどの部分が拘束するものでどの部分が単なる説明かを明確化するのに寄与する。*

コメント(Commentary)　**「注意深い読者は Dart という名前は 4 文字であることに気がついているだろう」**といったコメントはこの仕様を説明あるいは明確化するのに寄与するが、規定するテキストには冗長なものである。**コメントと論理的根拠のテキスト間の差は微妙である。***コメントは根拠以上に一般的であり、説明の為の例や明確化の為のテキストが含まれる。*

オープンな問題(Open questions)(**このフォントを使用**)　本仕様の著者たちが決めかねている箇所のことをいう;これら(問題点であって、著者たちではない、仕様書では精密さが重要である)は最終仕様には削除される。**さっきの印の最後のところのテキストは根拠なのかコメントなのか？**

予約語と組み込み識別子(built-in identifiers)(16.33 節)は**太文字**で示される。
例えば、switch または class。

文法構文(grammar productions)には EBNF(Extended Backus–Naur Form)の一般的な共通変種(common variant)が使われている。ある構文(production)(訳者注:あるいは書換規則(rewrite rule)ともいう)の左辺は:で終了する。右辺においては、代替(訳者注:「これらのうちどれか」の意味)は縦棒(|)で示され、空白を置いて並べられる。ある構文のオプション要素は疑問符が後につく。例えば anElephant?　ある構文のある要素の最後にスター文字を付すとそれはゼロまたはそれ以上の繰り返しが可能なことを意味する。ある構文のある要素の最後にプラス文字を付すと回数は 1 またはそれ以上の繰り返しが可能なことを意味する。否定(PEG(訳者注:Parsing expression grammar:解析表現文法)の not 組合せ子)はある構文のある要素の前にチルド(tilde: ~)を付すことで示される。

以下はその例である:

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

;

文法の構文(syntactic productions)と語彙的構文(lexical productions)の双方がこのように表現される。語彙的構文はその名前で識別される。語彙的構文の名前は大文字とアンダースコア(_)でのみ構成される。文法の構文内では常にある構文の要素間のホワイトスペース（訳者注：語間の空白をを示す文字）とコメントは、特に想起されていない限り、暗示的に無視される。句読トーケン(Punctuation tokens）は引用符たちの中で存在する。

構文は、それが表現する構成概念(constructs)の議論の中では、極力多く埋め込まれる。

$x_1,\ldots,x_n$というリストは x_i, $1 \leqq i \leqq n$ の形の n 個の要素からなるリスト（要素並び）であることを意味する。n はゼロであっても良いことに注意。この場合はこのリストは空となる。本仕様ではリストが頻繁に使用されている。

$[x_1,\ldots,x_n / y_1,\ldots,y_n]E$ という表記は、xi, $0 \leqq i \leqq n$ の総ての値が yi によって置き換えられている E のコピーであることを意味する。

演算子の仕様においては、しばしば x.op y は x.op(y)メソッド呼びだしと等価であるといった記述になっている。そのような仕様は次のことの簡略表記であると解釈しなければならない：

- x.op y は、演算子 op と同じ関数を定義している op'という名前の非演算子メソッドを x のクラスが宣言されているとすれば、x.op'(y)メソッド呼び出しと等価である。

この回りくどい表現は、op が演算子である x.op(y)は合法な文法でないので、必要になっている。しかしながら、これは面倒な詳細事項で、このルールをここで使うことを我々は好み、また我々は本仕様にわたっては簡潔な表記を使用したい。

本仕様がそのプログラムで与えられた順序だというときは、それはそのプログラムのソース・コードのテキストを左から右、上から下にスキャンする順序であることを意味する。

そうでなければプログラム実体(program entities)（クラスとか関数とか）の指定されていない名前となる名前への参照は、Dart コア・ライブラリのメンバたちの名前として解釈される。
例としては、クラス階層のルートであることを表現している Object と、実行時の型を具象化している Type のクラスがあげられる。

6. 概要(Overview)

Dart はクラス・ベース、単一継承、そして純粋なオブジェクト指向プログラミング言語である。Dart はオプション的に型付けされ（19 章）、そして具象化された総称型(reified generics)とインターフェイス(interfaces)に対応している。各オブジェクトの実行時の型は **Type** というクラスのインスタンスとして表現され、これは Dart のクラス階層のルートにある **Object** クラスで定義されているゲッタの **runtimeType** を呼ぶことで取得できる。

Dart のプログラムは静的(statically)にチェックされ得る。静的なチェッカは型の規則たちの一部の違反を報告するが、そのような違反によってコンパイルの放棄(abort)あるいは実行の阻止(preclude)はもたらされない。

Dart のプログラムは 2 つのモード、即ち生産モード(production mode)とチェック・モード(checked mode)のどれかひとつによって実行される。生産モードでは、静的な型アノテーションたち(19.1 節)は実行には全く影響を与えない。

定義により、リフレクション(Reflection)はプログラム構造を調べる。我々がある宣言の型への、あるいはソース・コードへの反射的アクセスを提供するときは、そのもととなっているコードの中で使われている型に依存する結果をそれが作り出すことは避けられない。

型テストはまたあるプログラムの型を明示的に調べる。それにも拘らず、殆どの場合、これらは型アノテーションに依存しないだろう。この規則の例外は関数の型が関わるテストである。関数の型は構造的なものであり、従ってそれらのパラメタたちに宣言された型に、およびそれらの戻りの型に依存する。

チェック・モードでは、代入(assignments)は動的にチェックされ、この型システムの一部の違反は実行時（ランタイム）で例外を発生させる。

オプショナルな型づけと具象化の共存は、以下の事項に基づいている：

1. **具象化された型情報は実行時におけるオブジェクトの型を反映し、またダイナミックな型チェック構文たち(dynamic typechecking constructs)からのクエリを常に受け得る（他の言語における instanceOf, casts, typecase などと類似）。具象化された型情報には、クラス宣言、あるオブジェクトの実行時型(class)、及びコンストラクタへの型引数がある。**
2. **静的な型アノテーションたちは、変数と関数（メソッドとコンストラクタ）の宣言の型を決める。**
3. **生産モードはオプショナルな型付けを尊重する。静的な型アノテーションは実行時の振る舞いには影響を与えない。**
4. **チェック・モードでは、開発中に於ける早期誤り検出の為に、静的な型アノテーションと動的な型情報を選択的ではあるが積極的に活用する。**

Dart のプログラムたちはライブラリ(libraries)（第 18 章）と呼ばれる構成単位で構成されている。ライブラリたちはカプセル化の単位であり、また相互に再帰的である。

しかしながらこれらはファースト・クラスではない。あるライブラリの同時に走る複数のコピーを得る為には、アイソレート(Isolate)を多数発生させる必要がある。（訳者注：プログラミング言語におけるファースト・クラスとはそのオブジェクトの使用に何らの制限もないことをいう。）

6.1 スコープづけ(Scoping)（適用範囲化）

*名前空間(namespace)*は宣言を表示する名前たちの実際の宣言たちへのマッピングである。*NS*をある名前空間としよう。我々はもし*n*が*NS*のひとつのキーであるなら*名前n はNS 内にある*という。*NS*のあるキーが宣言*d*にマッピングしているなら、*宣言d はNS 内にある*という。

スコープS_0は名前空間NS_0を誘導(induce)し、それはS_0内で宣言された各変数、型、または関数の宣言*d*の単純な名前を*d*にマッピングするものである。ラベルたちはあるスコープの誘導された名前空間には含まれず、それらはそれら自身の専用の名前空間を持つ。

従って例えば、Dart 内では同じ名前を持ったメソッド及びフィールドを宣言しているクラスを定義することは出来ない。同じように、トップ・レベルの変数、クラス、あるいはインターフェイスとして同じ名前を持ったトップ・レベルの関数を定義することはできない。

同じスコープ内に宣言された同じ名前を持ったエンティティがひとつ以上あるときはコンパイル時エラーである。

一部の場合では、その宣言の名前がそれを宣言するのに使われた識別子と異なる。セッタはそれに対応したゲッタとは異なる名前を持つ。何故ならそれらは常にその終わりに自動的に = が付加され、単項マイナスは特別な名前の単項 - を持つからである。

Dart は構文的にスコープ付け(lexically scoped)されている。スコープたちはネスト（訳者注：入れ子）できる。名前または宣言*d*は、もし*d*が*S*によって誘導された名前空間にある、あるいはもし*d*が*S*の構文的に包含しているスコープ内で使えるならば、*スコープS 内で使える*(*available in scope S*)。もし*d*が現在のスコープ内で使えるなら、我々は名前または宣言*d*が*スコープ内にある*という。

もし*n*という名前の宣言*d*があるスコープ*S*によって誘導された名前空間内にあるときは、*d*は*S*の構文的に包含するスコープ内で使えるどの*n*という名前の宣言をも遮蔽する。

これらの規則のひとつの結果としては、メソッドまたは変数で型を隠すことが可能であるということである。名前付け規約は通常そのような乱用を防いでいる。それにも拘らず、次のプログラムは違反ではない。

```
class HighlyStrung {
  String() => "?";
}
```

そのスコープ内での宣言により、あるいはインポートまたは継承(imports or inheritance)といった他のメカニズムにより、名前たちはあるスコープ内に組み入れられる。

構文的スコーピングと継承との関係は微妙である。最終的には、問題は構文的スコーピングが継承より優先されるかあるいはその逆かと言うことである。Dart は前者を選択している。

継承した名前たちがローカルに宣言されたなメタ値より優先するようにすることで、コードが変わってゆくなかで予期せぬ状況を作り出し得る。特に、あるサブクラス内でのコードの振る舞いが、もしスーパークラス内で新しい名前が導入されたときに、変わってしまう可能性が出る。例えば:

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

*ここで*S *にメソッド*foo()*が付加されたとしよう。*

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

*もし継承が構文的スコープより優先されるとすると、*C *の振る舞いは予期せぬかたちで変わってしまう。*S *の作者も*C *の作者も必ずしもころを認識していない。Dart では、構文的に可視なメソッド* foo()*が存在すれば、それは常に呼び出される。*

*次に逆のシナリオを考えてみよう。我々は*foo()*を含むバージョンで、ライブラリ*L2 *内で*foo()*を宣言していない場合から始める。ここでも、振る舞いに変化が起きる-しかし*L2 *の著者は自分たちのコードに影響を与える食い違いを持ちこんだひとりであり、また新しいコードは構文的に可視である。これらの要素の双方がこの問題が検出される可能性を高めている。*

これらの考察は、もしネストしたクラスたちのような構成を導入する際により重要になっており、この言語の今後のバージョンの中で検討される可能性がある。

良いツーリングは無論そのような状況を(個別的に)プログラマたちに知らせるよう勤めるべきである。例えば、継承した及び構文的に可視なものの双方がハイライト(下線や色付け)されるような。一層のこと、言語認識型のツールを持ったソース・コントロールのきちんとした組み入れが、それらが発生したときにそのような変更を検出しよう。

6.2 プライバシ(Privacy)

Dart はプライバシの 2 つのレベル、即ち *public* と *private* に対応している。その名前がプライベートであるときに限りその宣言は *private* であり、そうでないときはその宣言は *public* である。*q* を構成す

る識別子たちのどれかひとつがプライベートであるときに限り名前 *q* はプライベートであり、そうでないときは名前 *q* は *public* である。ある識別子アンダスコア(_文字)で始まるときに限りその識別子はプライベートであり、そうでないときは *public* である。

ある宣言 *m* がライブラリ *L* 内で宣言されているとき、あるいは *m* がパブリックとして宣言されているときは、*m* という宣言は*ライブラリ L に対しアクセス可能*である。

このことは private 宣言たちはそれらが宣言されているライブラリ内でのみアクセスされることを意味する。

現時点では、プライバシは特定のコード(ライブラリ)に結び付けられた静的な概念である。これはセキュリティの危惧というよりはソフトウエア技術の危惧に対処するよう設計されている。信頼されないコードは常に別のアイソレート(isolate)内で実行されねばならない。ライブラリたちがファースト・クラスのオブジェクトになり、プライバシがあるライブラリのインスタンスに結び付けられた動的な概念になることは可能である。

プライバシはある宣言の名前により示され、従ってプライバシと名前づけは直交したものではない。このことは人間とマシンの双方が、そこからその宣言が引き出されているコンテキストについて知ることなく、その使用場所でプライベート宣言物へのアクセスを認識できるという利点がある。

6.3 並行処理(concurrency)

Dart は常に単一スレッドである。Dart では状態共有並行性(shared-state concurrency)は存在しない。並行性は*アイソレート(isolate)*と呼ばれるアクタ・ライクなものを介して対応される。

アイソレートというのは並行処置のひとつの単位である。これは自分のためのメモリを所有しまたそれ自身の為の制御スレッドを持つ。アイソレートたちはメッセージ渡し(message passing)により通信する(16.17.4 項)。アイソレート間では状態は共有されることはないことに注意のこと。アイソレートは産み付け(spawning)により生成される(16.13 節)

7. エラーと警告(Errors and Warnings)

本仕様ではエラーは幾つかのタイプに区別されている。

*コンパイル時エラー(compile-time errors)*は実行を阻むエラーである。コンパイル時エラーはその誤ったコードが実行される前にDartコンパイラによって報告されねばならない。

Dart実装は何時コンパイルするかに関しては相当な自由度を持っている。当今のプログラミング言語実装物はしばしばコンパイルと実行が交互になされ、あるメソッドのコンパイルが遅れる、即ちそれが最初に呼び出されるまで遅れることがある。その結果、あるメソッドmのコンパイル時エラーはmが最初に呼び出されるまで遅れて報告されることがあり得る。

ウェブ言語として(as a web language)、Dartはしばしば中間的なバイナリ表現なしでソースから直接ロードされる(訳者注:DartのVMは、Javaのようにバイト・コードに変換しない言語VMである)。ロードの高速化のために、Dart実装物は例えばメソッドのボディ部を完全に解析しないことを選択しても良い。これは入力をトーケン化してメソッドのボディの波括弧(curly brace)のバランスのチェックをすることで出来得る。そのような実装では、構文エラーであっても、それがコンパイルされる時間に、そのメソッドを実行する必要がある時(JITed :JITはJust In Time Compilationの意味)においてのみ、検出されることになる。

開発環境においてはコンパイラは無論、そのプログラマに最善を尽くすべく、積極的にコンパイル・エラーを報告すべきである。

もし実行中のアイソレートAのコード内で捕捉されないコンパイル時エラー(uncaught compile-time error)が発生するときは、Aは直ちに停止(suspend)する。コンパイル時エラーが捕捉でき得る唯一の状況は、反射的(reflectively)に走るコードを介したものであり、そこではミラー・システムがそれを捕捉できる。

一般的に、一旦コンパイル時エラーがスローされ、Aが停止すれば、Aは次に終了する。しかしながら、これは全体の環境に依存する。Dartのエンジンはエンベッダ(embedder:エンジンとそれを取り巻くコンピューティング環境間のインターフェイス)のコンテキストのなかで走る。このエンベッダはしばしばウェブ・ブラウザであり得るがそうである必要はない;例えばサーバ上のC++プログラムであり得る。上記のようにあるアイソレートがコンパイル時エラーを起こしたときは、制御はエンベッダに戻り、そのエンベッダがリソース等をクリーンアップできる、等々。したがってそのアイソレートを終了させるかどうかはエンベッダの判断ということになる。

*静的警告(static warnings)*は静的なチェッカ(static checker)によって報告されるエラーである。これらは実行には影響を与えない。総てではないが、多くの静的な警告は型に関連するものであり、この場合はこれらは*静的型警告(static type warnings)*として知られるものである。

*動的型エラー(dynamic type errors)*はチェック・モード(checked mode)で報告される型のエラーである。

*実行時エラー(run-time errors)*は実行中に発生された例外である。我々が例外*ex*が生起された

(raised)あるいはスローされた(thrown)というときはいつも我々は、**throw** *ex;*形式の throw 式（16.9 節）が暗示的に実行された、または **rethrow** 形式の rethrow 式（17.10 節）が実行されたということを意味している。*あるクラス C がスローされた (a C is thrown)*と我々が言うときは、クラス *C* のインスタンスがスローされたということを言う。

もし実行中のアイソレート *A* によって捕捉されない例外がスローされたときは、*A* は直ちに停止される。

8. 変数(variables)

変数はメモリ内の蓄積場所である。

variableDeclaration（変数の宣言）:
declaredIdentier（宣言された識別子） (', ' identier（識別子）)*
;

declaredIdentifier（宣言された識別子）:
metadata（メタデータ） finalConstVarOrType（final、Const、又は VarOrType）
identifier（識別子）
;
finalConstVarOrType（final、Const、又は VarOrType）:
final type?
| **const** type?
| varOrType（var または型）
;
varOrType（var または型）:
var
| type
;

initializedVariableDeclaration（初期化された変数宣言）:
declaredIdentifier（宣言された識別子） ('=' expression（式）)? (', ' initializedIdentier（初期化された識別子）)*
;

initializedIdentifier（初期化された識別子）:
identifier（識別子） ('=' expression（式）)?
;

initializedIdentierList（初期化された識別子のリスト）:
initializedIdentier（初期化された識別子） (', ' initializedIdentier（初期化された識別子）)*
;

初期化されていない変数は初期値 **null** を持つ（16.2 節）。

あるライブラリのトップ・レベルで宣言された変数は*ライブラリ変数(library variable)*あるいは単にトップ・レベル変数と呼ばれる。

*static 変数(static variable)*は特定のインスタンスに結び付けられていないで、ライブラリまたはクラスに結び付けられている変数である。static 変数にはライブラリ変数とクラス変数がある。クラス変数はその宣言があるクラス宣言のなかでただちにネストされている変数であり、修飾子の **static** を含む。

ライブラリ変数は暗示的に static である。組み込み識別子（16.33 節）の **static** をトップ・レベル変数につけるとコンパイル時エラーとなる。

静的変数宣言は後回しで初期化(lazily initialized)される。静的変数 *v* が読みだされたときに、それが未だ代入されていないときに限り、それにはそのイニシャライザの計算結果がセットされる。詳細規則は 8.1 節に記されている。

この後回し初期化という考えかたは、過剰な初期化計算が定義されていて、それがアプリケーションの立ち上がり時間を長くしてしまうような言語を我々が望んでいないから導入されている。クライアント・アプリケーションをコーディングするよう設計されている Dart にとって、このことはとりわけ重要である。

*final 変数*とはそのバインディングが初期化で固定されている変数であり、final 変数 *v* それが初期化されたあとは常に同じオブジェとを参照する。final 変数の宣言は常に修飾子 **final** が含まれていなければならない。

宣言の時点で初期化されてしまっている final なインスタンス変数がまたコンストラクタのなかで初期化されているときは静的警告となる。もしあるローカルな変数の *v* が final であり、その *v* が宣言の時点で初期化されていないときは静的警告となる。

ライブラリまたは static 変数は、文法によりその宣言においてイニシャライザを持っていることが保障されている。

その宣言またはコンストラクタ・ヘッダの中でを除いてどこかで final 変数に代入しようとすれば以下に論じるように実行時エラーがスローされる。この代入は常に静的警告を生起させる。final 変数の繰り返し代入しようとすれば実行時エラーを発生させてしまう。

全体として、これらの規則により final 変数に対する複数回の代入にたいしては静的警告がだされ、繰り返しの代入は動的に失敗することになる。

constant 変数（定数変数）は修飾子 **const** を宣言に含む変数のことである。定数変数は常に暗示的に final である。定数変数はコンパイル時定数（16.1 節）で初期化されねばならず、そうでないとコンパイル時エラーが発生する。

我々は、ある変数 v が final または常数でなく、v への代入があるスコープ s のなかで行われているときは、その変数 v はスコープ s のなかで潜在的に変異している(*potentially mutated*)という。

ある変数宣言が明示的にある型を指定していないときは、その宣言された変数の型は **dynamic** であり、これは未知の型（19.6 節）である。

final でない変数は *可変(mutable)*である。static およびインスタンス変数宣言は常に暗示的ゲッタ(getters)とセッタ(setters)を誘発させる。もしその変数が」可変であるなら、それはまた暗示的なゲッタとセッタを誘発させる。暗示的なゲッタとセッタが導入されるスコープは、関与している変数宣言の種類に依存する。

ライブラリ変数は包含しているライブラリのトップ・レベルのスコープにゲッタをもたらす。Static なクラス変数は即座に包含しているクラスに static なゲッタ及び static なセッタをもたらす。インスタンス変数は即座に包含しているクラスにインスタンス・ゲッタ及びインスタンス・セッタをもたらす。

可変ライブラリ変数(mutable library variable)は、包含しているライブラリのトップ・レベル・スコープにセッタをもたらす。可変 static クラス変数(mutable static class variable)は、直ちに包含しているクラスに static なセッタをもたらす。可変インスタンス変数は(mutable instance variable)、直ちに包含しているクラスにインスタンス・セッタをもたらす。

ローカル変数たちは最も内側で包含しているスコープに付加される。これらはゲッタおよびセッタを誘発しない。ローカル変数はそのローカル変数宣言が完了したあとでのソース・コードの場所でのみ参照され得る。そうでないときはコンパイル時エラーが発生する。このエラーはその不完全参照(premature reference)が生じる場所で、あるいはその変数宣言の場所で報告される。

我々は実装において追加の処理フェーズを回避することができるために、宣言の時点でこのエラーを報告することを許している。

以下に示す事例は期待される振る舞いを示したものである。変数 x があるライブラリのレベルで宣言されており、もう一つの x は関数 f の内部で宣言されている。

```
var x = 0;
f(y) {
  var z = x; // コンパイル時エラー
  if (y) {
    x = x + 1; // 2つのコンパイル時エラー
    print(x); // コンパイル時エラーたち
  }
 var x = x++; // コンパイル時エラー
 print(x);
}
```

f の内側にある宣言は包含している宣言を隠している。従って f の内部の x へのすべての参照は x の内部宣言を参照する。しかしながら、これらの参照の多くは、それがその宣言の前にあるので違反となっている。z への代入はそのケースにあたる。if 文のなかの x への代入は複数の問題を抱えている。右側ではその宣言の前に x を読みだそうとしており、左側ではその宣言の前に x に代入しようとしている。これらの各々は独立してコンパイル時エラーとなっている。if 文の中の print 文もまた違反である。

x の内部宣言は、右側でその宣言が終了する前に x を読みだそうとしているのでそれ自身間違っている。左側は技術的には参照あるいは代入ではないのが、宣言はそうなので、従って違反となる。最後の print 文も完全なエラーである。

別な例として、***var** x = 3, y =x;* は、*x* がそのイニシャライザのあとで参照されているので、合法である。

特に意地の悪い(perverse)例としては、ローカル変数名がある型を隠すものがある。Dart は型たち、関数たち、および変数たちにたいし単一の名前空間となっているので、これは可能である。

```
class C {}
perverse() {
   var v = new C(); // コンパイル時エラー
   C aC; // コンパイル時エラー
   var C = 10;
}
```

perverse()のなかでは、C はローカル変数を示している。型 C は同じ名前の子の変数によって隠されている。C をインスタンス化しようとすると、その宣言の前にあるローカル変数を参照しているので、コンパイル時エラーとなる。同じように、 aC の宣言でもそうである（例えそれが単に型あのテーションであるにも拘らず）。

一般に、型アノテーションは運用モード(production mode)では無視される。しかしながら、我々あるプログラムではあるモードではコンパイルで違反になるが別のモードではそうならないことを許すことは望まず、そしてこのきわめておかしな状況では、この考察が優先される。

以下の規則がすべての static およびインスタンス変数に適用される。

T v;, *T v = e;*, ***const*** *T v = e;*, ***final*** *T v;* または ***final*** *T v = e;*のどれかの形式の変数宣言は、常に次のシグネチュアを持った暗示的なゲッタ関数(10.2 節)を導入させる:

T get v

その呼び出しは以下(8.1 節)に示すように計算される。

var *v;*, ***var*** *v = e;*, ***const*** *v = e;*, ***final*** *v;* または ***final*** *v = e;*のどれかの形式の変数宣言は、常に次のシグネチュアを持った暗示的なゲッタ関数を導入させる:

get *v*

その呼び出しは以下(8.1 節)に示すように計算される。

final *T v;*, ***final*** *T v = e;* または ***const*** *T v = e;* の形式を持った final または定数変数宣言は常に以下のシグネチュアを持った暗示的なセッタ関数(10.3 節)を導入させる:

void set *v=(T x)*

その実行は *v* の値を到来引数 *x* でセットする。

var *v;*または ***var*** *v = e;* の形式を持った非 final 変数宣言は常に以下のシグネチュアを持った暗示的なセッタ関数を引き起こす:

***set** v=(x)*

その実行は *v* の値を到来引数 *x* でセットする。

8.1 暗示的変数ゲッタの計算(Evaluation of Implicit Variable Getters)

d を static またはインスタンス変数 *d* の宣言だとする。もし *d* がインスタンス変数のときは、*v* の暗示的ゲッタの呼び出しで *v* にストアされている値の計算が行われる。

d が static またはライブラリ変数のときは、*v* の暗示的ゲッタ・メソッドの実行は以下のようになる:

- 初期子をもった非コンスタント変数宣言。
  d が ***var** v = e; , T v = e; , **final** v = e; , **final** T v = e; , **static** v = e; , **static** T v = e; , **static final** v = e;* または ***static final** T v = e;*のどれかの形式で、*v* に未だ値がセットされていないときは、イニシャライザ式 *e* が計算される。もし *e* の計算過程で *v* に対するゲッタが参照されたときは、**CyclicInitializationError** がスローされる。もしその計算が成功しオブジェクト *o* が得られたら *r = o* とし、そうでないときは *r = **null*** とする。どの場合でも *r* は *v* にストアされる。該ゲッタの計算結果は *r* である。

- **定数変数宣言**
  d が ***const** v = e; , **const** T v = e; , **static const** v = e;* または ***static const** T v = e;*の形式のとき、該ゲッタの結果はコンパイル時定数 *e* の値となる。
  コンパイル時定数はそれ自身に依存できないのでサイクリックな参照は起きえないことに注意のこと。
  そうでないときは、

- **初期化子がない変数宣言**
  該ゲッタ・メソッドの実行結果は *v* にストアされた値となる。

9. 関数(Functions)

関数は実行可能なアクションの抽象化である。

functionSignature（関数シグネチュア：訳者注：シグネチュアは特にオブジェクト指向言語で使われる用語で狭い意味の構文）：
metadata（メタデータ） returnType（戻りの型）? identifier（識別子） formalParameterList（仮パラメタ・リスト）
;

returnType（戻りの型）：
void |
type（型）
;

functionBody（関数ボディ）：
'=>' expression（式） ';' |
block（ブロック）
;

block（ブロック）：
'{' statements（文たち） '}'
;

関数には関数宣言(function declarations)（9.1 節）、メソッド(methods)（10.1 節、10.7 節）、ゲッタ(getters)（10.2 節）、セッタ(setters)（10.3 節）、コンストラクタ(constructors)（10.6 節）、及び関数リテラル(function literals)（16.10 節）がある。

総ての関数はひとつのシグネチュアとひとつのボディを持つ。シグネチュアはその関数の仮パラメタたち(formal parameters)、及びおそらくはその名前と戻りの型を記述する。関数ボディは以下のどれかである：

- その関数によって実行される文たち(statements)（17 章）が入っているひとつのブロック文(block statement)（17.1 節）、あるいはオプションとして **async**、**async*** または **sync*** のどれかの修飾子でマークされているブロック文。この場合、ある関数の最後の文が return 文（17.12 節）でないときは、文 **return**；がその関数ボディの最後に暗示的に付加される。

  *Dart は選択的に型付けできる言語なので、ある値を返さないある関数はある式のコンテキストの中で使われないということを我々は保障できない。従って各関数は、ある値を返さなければならない。式なしの **return** は **null** を返す。更なる詳細は 17.12 節を参照のこと。*

または、

- その関数ボディが{**return** *e*;}の形式のボディと等価な => *e* の形式、あるいはその関数ボ

ディが async {**return** *e;*}と等価な **async** => e の形式。*以下で論じるように発生器(generators)にのみ提供され、発生器たちは* ***return*** *e;の形式を許さず、値たちはyield を使って発生されたストリームまたは iterable に付加されるので、他の修飾子はここでは適用されない。*

もしそのボディが **async** または **async***修飾子でマークされている関数は非同期(asynchronous)である。そうでないときはその関数は*同期(synchronous)*である。もしそのボディが **sync*** または **async*** 修飾子でマークされているときは、その関数は*発生器(generator)*である。

ある関数が同期か非同期かということと、その関数が発生器かどうかということは直交している（関連しない）。発生器関数(Generator functions)はある collection の個々の要素を発生させる関数を後回しで適用させる(lazily applying)ことでシステマティックなやり方で collection を生成する関数の為のものである。Dart では iterable を返す同期の場合、および stream を返す非同期の場合の双方でそのような利便性を提供している。Dart はまた単一の値を生成する同期及び非同期の関数を許している。

async、**async***または **sync*** 修飾子がセッタまたはコンストラクタのボディに付されていいる場合はコンパイル時エラーである。

セッタは代入(16.19 節)のコンテキストの中でのみ使われ、代入式は常にその代入の右側の値に対して計算されるので、非同期のセッタというのは殆ど使われないいだろう。もしそのセッタが実際に非同期にその仕事をこなしたら、そのセッタがその仕事をした後でその代入の右側への代入を解決するfuture を返したいと思うだろう。しかしながら、その場合は代入ごとに動的テストが必要になり、これは面倒すぎ使えないだろう。

非同期コンストラクタは、定義により、決してコンストラクトしようとしているそのクラスのインスタンスを返すことはせず、代わりにfuture を返す。そのような新機能を ***new*** *を使って呼ぶのは多分混乱を起こそう。あるオブジェクトを非同期で生成する必要がある場合はメソッドを使うことが好ましい。*

ファクトリに対し修飾子を付すことも可能である。 ***Future*** *の為のファクトリは* ***async*** *によって修飾でき、* ***Stream*** *の為のファクトリは* ***async**** *で修飾でき、* ***Iterable*** *の為のファクトリは* ***sync**** *で修飾できる。ファクトリで返されるオブジェクトは間違った型になってしまうので、その他のシナリオは意味がない。この状況はとても正常ではないのでそれを許すのにコンストラクタの為の一般規則に例外を設ける価値はなかろう。*

async とマークされた或る関数の宣言された戻りの型が **Future** に代入できない可能性があるときは静的警告である。**sync*** とマークされた或る関数の宣言された戻りの型が **Iterable** に代入できない可能性がある場合は静的警告である。**async*** とマークされた或る関数の宣言された戻りの型が **Stream** に代入されない可能性があるときは静的警告である。

9.1 関数宣言(Function Declarations)

*関数宣言(function declaration)*はクラスのメンバたちでも関数リテラルでない関数のことを言う。関数宣言にはあるライブラリのトップ・レベルにある関数宣言である*ライブラリ関数(library functions)*、及び他の関数の内部で宣言された関数宣言である*ローカル関数(local functions)*がある。ライブラリ関数はまた単にトップ・レベル関数とも呼ばれる。

関数宣言はその関数の名前を示す識別子（戻り型がその前に付されていることが好ましい）で構成される。関数名のあとにシグネチュアとボディが続く。ゲッタの場合はシグネチュアは空である。外部関数のボディは空である。

ライブラリ関数のスコープはそれを包含しているライブラリのスコープである。ローカル関数のスコープは 17.4 節に記されている。いずれの場合でも、該関数の名前はその関数の仮パラメタたちのスコープ内にある（9.2 節）。

組込み識別子(built-in identifier)である **static** をある関数宣言の前に付すのはコンパイル時エラーになる。

或る関数 f_1 が別の関数 f_2 に転送する(forwards)と我々が言うときは、f_1 を呼び出すと同じ引数及び/または同じレシーバで f_2 が実行され、f_2 の実行結果が例外を除いて（その場合は f_1 は同じ例外をスローする）f_1 を呼んだ側に返されることを意味する。更に、我々はこの用語をこの仕様書で導入されている合成関数たち(synthetic functions)に対してのみ使用する。

9.2 仮パラメタ(Formal Parameters)

各関数宣言には*仮パラメタ・リスト(formal parameter list)*が含まれ、これは必要な位置的パラメタたち（positional parameters： 9.2.1 節）のリストと、それに続く何らかのオプショナルなパラメタたちで（9.2.2 節）構成される（訳者注：formal parameter とは関数定義時の変数即ちパラメタのことを言う。実パラメタは実際に渡される値即ち引数のことを言う）。オプショナルなパラメタたちは名前つきパラメタ（named parameters 訳者注：指名パラメタともいう。named parameters あるいは keyword arguments というのは、関数呼び出しそれ自身の中で各パラメタの名前を明確にしていることをいう）たちのセット、または位置的パラメタたちのリストで指定されるが双方ともでは指定されない。

ある関数の仮パラメタ・リストは、その関数の*仮パラメタ・スコープ(formal parameter scope)*として知られる新しいスコープをもたらす。ある関数 f の仮パラメタ・スコープは f が宣言されているスコープ内に包含される。各仮パラメタはこの仮パラメタのスコープのなかにあるローカル変数をもたらす。しかしながら、ある関数のシグネチュアのスコープはこの関数の包含しているスコープであって、この仮パラメタのスコープではない。

ある関数のボディはその関数の*ボディ・スコープ(body scope)*として知られる新しいスコープをもたらす。ある関数 f のボディは f の仮パラメタ・スコープによってもたらされたスコープ内に包含される。

仮パラメタが定数変数（第 8 章）として宣言されているときはコンパイル時エラーである。

formalParameterList（仮パラメタ・リスト）:
'(' ')' |
'(' normalFormalParameters（通常の仮パラメタたち） (', ' optionalFormalParameters オプショナルな仮パラメタたち）)? ')' |
(optionalFormalParameters（オプショナルな仮パラメタたち）)
;

normalFormalParameters（通常の仮パラメタたち）:
normalFormalParameter（通常の仮パラメタ） (', ' normalFormalParameter（通常の仮パラメタ）)*
;

optionalFormalParameters（オプショナルな仮パラメタたち）:
optionalPositionalFormalParameters（オプショナルな位置的仮パラメタたち） |
namedFormalParameters（名前付き仮パラメタたち）
;
optionalPositionalFormalParameters（オプショナルな位置的仮パラメタたち）:
'[' defaultFormalParameter（デフォルト仮パラメタ） (',' defaultFormalParameter)* ']'

namedFormalParameters（名前付き仮パラメタたち）:
'[' defaultFormalParameter（デフォルトの仮パラメタ） (', ' defaultFormalParameter（デフォルトの仮パラメタ）)* ']'
;

9.2.1 必要とされる仮パラメタ(Required Formals)

*必要とされる仮パラメタ(Required Formals)*は以下の 3 つの手段のひとつとして規定される:

- パラメタたちの名前たちを指名しその関数型（19.5 節）たちを記述する関数シグネチュアによる手段。そのような関数型のシグネチュアの中で何らかのデフォルトの値が指定されているときはコンパイル時エラーである。
- 生成的コンストラクタ（10.6.1 節）へのパラメタとしてのみ有効な初期化仮パラメタとして。
- 通常の変数宣言（第 8 章）を介して。

normalFormalParameter（通常の仮パラメタ）:
functionSignature（関数シグネチュア） |
fieldFormalParameter（フィールド仮パラメタ） |
simpleFormalParameter（シンプルな仮パラメタ）
;
simpleFormalParameter（シンプルな仮パラメタ）:
declaredIdentier（宣言された識別子） |

identifier（識別子）
;

fieldFormalParameter（フィールド仮パラメタ）:
metadata finalVarOrType（final または var または型）? **this** '.' identifier（識別子）
formalParameterList（仮パラメタリスト）?
;

9.2.2 オプショナル仮パラメタ(Optional Formals)

オプショナルなパラメタが指定でき、またデフォルト値つきで指定できる。

defaultFormalParameter（デフォルトの仮パラメタ）:
normalFormalParameter（通常の仮パラメタ） ('=' constantExpression（定数の式）)?
;

defaultNamedParameter（デフォルトの名前付きパラメタ）:
normalFormalParameter（通常の仮パラメタ） (':' constantExpression（定数の式）)?
;

名前つきのパラメタがコンパイル時の常数（16.1 節）でないときはコンパイル時エラーである。あるオプショナルな仮パラメタに対しデフォルトが明示的に指定されていないが、デフォルトが合法的に提供できるときは、暗示的なデフォルトの **null** が用意される。

名前つきのオプショナルな仮パラメタの名前が '_' 文字で始まるときはコンパイル時エラーとなる。

この制約の必要性は名前付けとプライバシが直交していないという事実の直接の結果から来ている。もし我々がアンダスコアで始まる名前つき仮パラメタを認めたら、それらは private であるとみなされ、それが定義されたライブラリの外部からの呼び出し者からはアクセスできなくなる。もしそのライブラリの外部のあるメソッドがプライベートなオプショナルな名前をオーバロードしたときは、それはオリジナルのメソッドの副型(subtype)にはならない。無論静的チェッカがそのような状況に対しフラグを立てるが、その結果はあるプライベートな名前つきの仮パラメタを付加することで容易に修正できないようなやり方でそのライブラリの外部のクライアントたちを阻止する(break)ことになろう。

9.3 関数の型(Type of a Function)

ある関数が明示的に戻り値の型を宣言していないときは、コンストラクタ関数以外ではその戻り値の型は **dynamic**（19.6 節）である。コンストラクタ関数の戻り値の型は直ちに包含しているクラスのある型である。

F の必要な仮パラメタが $T_1\ p_1, \ldots, T_n\ p_n$、戻り値の型が T_0、オプショナルな仮パラメタが無かったとする。そのときは F の型は $(T_1, \ldots, T_m) \rightarrow T_0$ となる。

F の必要な仮パラメタが $T_1\ p_1,\ \ldots,\ T_n\ p_n$、戻り値の型が T_0、そして位置的なオプショナルな仮パラメタが $T_{n+1}\ p_{n+1},\ \ldots,\ T_{n+k}\ p_{n+k}$だったとする。そのときは F の型は$(T_1,\ \ldots,\ T_n,\ [T_{n+1},\ \ldots,\ T_{n+k}]) \rightarrow T_0$となる。

F の必要な仮パラメタが $T_1\ p_1,\ \ldots,\ T_n\ p_n$、戻り値の型が T_0、そして名前つきのオプショナルな仮パラメタが $T_{n+1}\ p_{n+1},\ \ldots,\ T_{n+k}\ p_{n+k}$だったとする。そのときは F の型は$(T_1,\ \ldots,\ T_n,\ [p_{n+1}:T_{n+1},\ \ldots,\ p_{n+k}:T_{n+k}]) \rightarrow T_0$となる。

ある関数オブジェクトの実行時の型は常にクラス **Function** を実装する。

上記に基づき、与えられた関数 f に対し f.runtimeType が実際は Function である、あるいは２つの別々の関数オブジェクトは必然的に同じ実行時型をもつと考えてはいけない。

関数に対し然るべき表現を選択するのは実装次第である。

*たとえば、然るべき抽出を介して作られたあるクロージャが対等性を通常のクロージャと異なって扱っており、従って違ったクラスである可能性がある場合を考えてみよう。実装に於いてはアリティおよび/または型に基づいて関数たちに対して異なったクラスたちを使用できる。アリティはある関数がインスタンス・メソッド(暗示的なレシーバ・パラメタで)であるかどうかによって明示的に影響を受けよう。これらの変体たちは奇形であり、従って本仕様書では関数オブジェクトたちは **Function** を実装したと考えられる何らかのクラスのインスタンスであることだけを保証している*

9.4 外部関数(External Functions)

*外部関数(External Functions)*はその宣言とは分離してそのボディが提供される関数である。外部関数はトップ・レベル関数(18 章)、メソッド(10.1、10.7 節)、ゲッタ(10.2 節)、セッタ(10.3 節)、または非リダイレクト・コンストラクタ(10.6.1、10.6.2 節)であり得る。外部関数は関数シグネチュアを伴った組込み識別子 **external**(16.32 節)で導入される。

外部関数により Dart コンパイラが静的に判らないコードの為の型情報を我々が与えることができる。

外部関数の例としては、別言語の関数（C または Javascript 等で定義された）、実装のためのプリミティブ（Dart ランタイムで定義された）、あるいは動的に生成されるがそのインターフェイスが静的に判らないコードがある。しかしながら、抽象メソッドはボディが無いので、外部関数とは異なる。

外部関数は実装固有のメカニズムによりそのボディと接続される。そのボディとまだ接続されていない外部関数を呼び出そうとすると **NoSuchMethodError** あるいはそのクラスの例外が生起される。

実際の文法は以下の第 10 章及び第 18 章で示されている。

10.クラス(Classes)

*クラス(class)*はその*インスタンスたち(instances)*であるオブジェクトたちのセットの形式と振る舞いを規定する。クラスたちは以下に示すようにクラス宣言によって、またはミクスイン・アプリケーション（12.1 節）を介して定義される。

classDefinition（クラス定義）:
　metadata（メタデータ） **abstract**? **class** identifier（識別子） typeParameters（型パラメタたち）? (superclass（スーパークラス） mixins（mixin たち）?)? interfaces（インターフェイスたち）?
　'{' (metadata classMemberDefinition)* '}'
　| metadata **abstract**? **class** mixinApplicationClass
　;

mixins（mixin たち）:
　with typeList（型リスト）
　;

classMemberDefinition（クラスのメンバの定義）:
　declaration（宣言） ';'
　| methodSignature（メソッド・シグネチュア） functionBody（関数ボディ）
　;
methodSignature（メソッド・シグネチュア）:
　constructorSignature initializers（コンストラクタ・シグネチュア・イニシャライザたち）?
　| factoryConstructorSignature（ファクトリ・コンストラクタ・シグネチュア）
　| **static** functionSignature（関数シグネチュア）
　| **static**? getterSignature（ゲッタ・シグネチュア）
　| **static**? setterSignature（セッタ・シグネチュア）
　| operatorSignature（演算子シグネチュア）
　;
declaration（宣言）:
　constantConstructorSignature（定数コンストラクタ・シグネチュア） (redirection（リダイレクション） | initializers（イニシャライザたち）)?
　| constructorSignature（コンストラクタ・シグネチュア） (redirection | initializers)?
　| **external** constantConstructorSignature
　| **external** constructorSignature
　| **external** factoryConstructorSignature（ファクトリ・コンストラクチャ・シグネチュア）
　| ((**external static**?))? getterSignature（ゲッタ・シグネチュア）
　| ((**external static**?))? setterSignature（セッタ・シグネチュア）
　| **external**? operatorSignature（演算子シグネチュア）
　| ((**external static**?))? functionSignature（関数シグネチュア）
　| **static** (**final** | **const**) type? staticFinalDeclarationList（static final な宣言リスト）

| **const** type? staticFinalDeclarationList（イニシャライザ識別子リスト）
| **final** type? initializedIdentifierList
| **static**? (**var** | type) initializedIdentifierList
;

staticFinalDeclarationList（static final な宣言のリスト）:
: staticFinalDeclaration（static fainal の宣言） (',' staticFinalDeclaration（static fainal の宣言）)*
;

staticFinalDeclaration（static final 宣言）:
identifier（識別子） '=' expression（式）
;

クラスはコンストラクタ(constructors)、インスタンス・メンバたち(instance methods)、及び static メンバたち(static members)を持つ。あるクラスの static メンバは、その static メソッドたち(static methods)、ゲッタたち(getters)、セッタたち(setters)、及び static 変数(static variables)たちである。あるクラスのメンバたちはその static メンバ及びインスタンス・メンバたちである。

クラスは幾つかのスコープを持つ:

- *型パラメタ・スコープ(type-parameter scope)*はそのクラスが総称（第 14 章）でないときは空である。あるクラスの型パラメタ・スコープの包含スコープは、そのクラス宣言の包含スコープである。
- *static スコープ(static scope)*。あるクラスの static スコープの包含スコープは、そのクラスの型パラメタ・スコープ（第 14 章）の包含スコープである。
- インスタンス・スコープ(instance scope)。あるクラスのインスタンス・スコープの包含スコープは、そのクラスの static スコープである。

あるインスタンス・メンバ宣言の包含スコープはそれが宣言されている該クラスのインスタンス・スコープである。

static メンバ宣言の包含スコープはそれが宣言されている該クラスの static スコープである。

各クラスはスーパークラスがない **Object** クラスを除いて単一のスーパークラスを持つ。クラスはその inplements 節(implements clause)（10.10 節）のなかで宣言することで幾つかのインターフェイスを実装できる。

*抽象クラス(abstract class)*は、クラス宣言の手段に依るかミクスイン・アプリケーション（12.1 節）の為の型エイリアス(19.3.1 項)を介するかのどちらかにより、**abstract** 修飾子で明示的に宣言されたクラスである。

我々は具体クラスと抽象クラスには異なった振る舞いを持たせたい。もし A が抽象クラスであることを意図したものなら、我々は A をインスタンス化しようとすることに対し静的チェッカが警告するよう

にしたいし、A のなかの未実装のメソッドがあることに対しチェッカが文句を言わないようにしたい。これに比べて、もし A が具体クラスであることを意図したものなら、チェッカは総ての未実装のメソッドたちに対し警告すべきだが、クライアントがそれを自由にインスタンス化させるようにしなければならない。

*クラス C のインターフェイス*は暗示的なインターフェイス(implicit interface)であって、*C* によって宣言されたインスタンス・メンバたちに対応したインスタンス・メンバたちを宣言しており、その直接的なスーパーインターフェイスたちは *C* の直接のスーパーインターフェイスである(10.10 節)。型あるいはインターフェイスとしてあるクラス名がある場合は、その名前はそのクラスのインターフェイスを意味する。

あるクラスが同じ名前の 2 つのメンバを宣言しているときはコンパイル時エラーである。

final なインスタンス変数とセッタではどうだろうか？この場合は同様に違反である。もしそのセッタがその変数をセットしているときは、その変数は final であってはならない。

あるクラスが同じ名前のインスタンス・メソッドと static なメンバ・メソッドを持っているときはコンパイル時エラーとなる。

以下に、「メンバを持つ」と「メンバを宣言する」の相違を示す例をあげる。例えば、B が f という名前のあるメンバを宣言しているが 2 つのそのようなメンバを持っている場合である。継承のルールがあるクラスがどのメンバを持っているかを決めている。

```
class A {
  var i = 0;
  var j;
  f(x) => 3;
}
class B extends A {
  int i = 1; // コンパイル時エラー、B は追ない i という名前の 2 つの変数を持っている
  static j; // コンパイル時エラー、B は追ない j という名前の 2 つの変数を持ってい
  static f(x) => 3; // コンパイル時エラー、static メソッドはインスタンス・メソッドとぶつかっている
}
```

あるクラス *C* が *C* と同じ名前を持ったメンバを宣言しているときはコンパイル時エラーである。ある総称クラスがそのクラスまたはそのメンバたちのどれかまたはコンストラクタたちの名前と同じ名前の型変数を宣言しているときはコンパイル時エラーである。

10.1 インスタンス・メソッド(Instance Methods)

インスタンス・メソッドはその宣言があるクラス宣言のなかに直ちに含まれている(immediately contained)もので、**static** と宣言されていない関数たち(第 9 章)のことを言う。あるクラス *C* のインス

タンス・メソッドはそれらのインスタンス・メソッドが *C* によって宣言されているもの、及びそのインスタンス・メソッドが *C* によってそのスーパークラスから継承されているものをいう。

あるインスタンス・メソッド m_1 がインスタンス・メンバ m_2 をオーバライド（10.9.1 節）し、m_1 が m_2 と異なった数のメンバが必要としている場合は静的警告である。

あるインスタンス・メソッド m_1 がインスタンス・メンバ m_2 をオーバライドし、m_1 が m_2 よりも少ないオプショナルな位置的パラメタであるときは静的警告である。

あるインスタンス・メソッド m_1 がインスタンス・メンバ m_2 をオーバライドし、m_1 が同じ順序で m_2 で宣言された総ての名前つきのパラメタたちを宣言していないときは静的警告である。

あるインスタンス・メソッド m_1 がインスタンス・メソッド m_2 をオーバライドし、m_1 の型が m_2 の型の副型（継承型）でないときは、静的警告となる。

あるインスタンス・メソッド m_1 がインスタンス・メソッド m_2 をオーバライドし、m_2 のシグネチュアが明示的に p の為の仮パラメタのためのデフォルト値を指定し、また m_1 のシグネチュアが p の為の仮パラメタのための異なったデフォルト値を指定しているときは静的警告となる。

あるクラス *C* が *n* という名前のインスタンス・メソッドを宣言し、*n* という名前の静的メンバが *C* のスーパークラスで宣言されているときは静的警告となる。

10.1.1 演算子(Operators)

*演算子(operators)*たちは特別な名前を持つインスタンス・メソッドたちである。

operatorSignature（演算子シグネチュア）:
returnType（戻りの型）? **operator** operator（演算子） formalParameterList（仮パラメタ・リスト）
;

operator（演算子）:
'~'
| binaryOperator（二項演算子）
| '[' ']'
| '[' ']' '='
;

binaryOperator（2 項演算子）:
multiplicativeOperator（積算演算子）
| additiveOperator（加算演算子）
| shiftOperator（シフト演算子）

| relationalOperator（関係演算子）
| '=='（イコール性演算子）
| bitwiseOperator（ビット演算子）
;

演算子宣言は組み込み識別識別子（16.32 節）の **operator** で識別される。

以下の名前たちはユーザ定義の演算子として許される: <, >, <=, >=, ==, -, +, /, ~/, *, %, |, ^, &, <<, >>, []=, [], ~.。

ユーザ定義の演算子の[]=の仮パラメタの数（arity:アリティ）が 2 でないときはコンパイル時エラーとなる。<, >, <=, >=, ==, -, +, /, ~/, *, %, |, ^, &, <<, >>, []の名前のひとつの名前のユーザ定義の演算子のアリティが 1 でないときはコンパイル時エラーとなる。~の名前を持ったユーザ定義演算子のアリティが 0 または 1 でないときはコンパイル時エラーとなる。

- 演算子は 2 つのオーバロードされたバージョンが許されるという点でユニークである。もしこの演算子が引数を持っていないときは、それは単項マイナスを意味する。引数を持っているときは、2 項減算を意味する。

単項演算子の - の名前は単項マイナス(**unary-**)である。

これにより 2 つのメソッドがメソッド検索(method lookup)、オーバライド、及びリフレクションの目的のために識別するように出来る。

ユーザ定義の演算子~のアリティが 0 でないときはコンパイル時エラーである。

ある演算子のなかでオプショナルなパラメタを宣言するのはコンパイル時エラーである。

ユーザ宣言の演算子[]= の戻りの型が明示的に宣言されていて **void** でないときは静的警告となる。

10.2 ゲッタ(Getters)

ゲッタはオブジェクトの属性たちの値を取得する為に使われる関数（第 9 章）である。

getterSignature（ゲッタ・シグネチュア）:
type（型）? **get** identifier（識別子）
;

戻り値が指定されていないときは、そのゲッタの型は **dynamic** である。

static 修飾子が先行しているゲッタ定義は static なゲッタを定義している。そうでないときは、それは

インスタンス・ゲッタを定義する。ゲッタの名前はその定義のなかの識別子(identifier)によって与えられる。

クラス *C* 内に置ける static ゲッタ宣言の効果は、該 static ゲッタに転送する（9.1 節）クラス *C* の為の **Type** オブジェクトへの同じ名前とシグネチュアを持ったインスタンス・ゲッタを付加することである。

あるクラス *C* のインスタンス・ゲッタたちは、*C* によって宣言されているインスタンス・ゲッタたち及び *C* がそのスーパークラスから継承しているインスタンス・ゲッタたちである。あるクラス *C* の static ゲッタたちは *C* によって宣言されている static ゲッタたちである。

あるクラスが同じ名前のゲッタとメソッドを有する場合はコンパイル時エラーとなる。この制約はそのゲッタが明示的または暗示的に定義されていようといまいが、あるいはそのゲッタまたはメソッドが継承したものであろうとなかろうと、維持される。

このことはゲッタは決してメソッドをオーバライドできず、またメソッドは決してゲッタまたはフィールドをオーバライド出来ないことを意味している。

ゲッタ m_1 がゲッタ m_2 をオーバライド（10.9.1 節）し、m_1 の型が m_2 の型の副型でないときは、静的な警告となる。あるクラスが *g* という名前の static ゲッタを宣言し、また *v*=という名前の非 static セッタを持っているときは静的警告となる。あるクラス *C* が *n* という名前のインスタンス・ゲッタを宣言し、*C* のスーパークラスの中で *v* または *v*= という名前のアクセス可能な static メンバが宣言されているときは静的警告となる。ゲッタまたはセッタが明示的に宣言されているかあるいは暗示的に宣言されているかにかかわらず、これらの警告は出されねばならない。

10.3 セッタ(Setters)

セッタはオブジェクトの属性たちの値をセットする為に使われる関数（第 9 章）である。

setterSignature（セッタ・シグネチュア）:
　　static? returnType（戻りの型）? **set** identifier（識別子）
;

戻りの型が指定されていないときは、そのセッタの戻りの方は **dynamic** である。

static 修飾子が先行しているセッタ定義はスタティックなセッタを定義する。そうでないときは、それはインスタンス・セッタを定義する。そのセッタの名前はその定義の識別子(identifier)によって与えられる。クラス *C* 内に置ける static セッタ宣言の効果は、該 static セットに転送する（9.1 節）クラス *C* の為の **Type** オブジェクトへの同じ名前とシグネチュアを持ったインスタンス・セッタを付加することである。

従ってセッタ名は、ゲッタまたはメソッドをオーバライドするあるいはゲッタまたはメ

ソッドによってオーバライドされることと決してぶつかることは無い。

あるクラス *C* のインスタンス・セッタたちは *C* によって宣言されたインスタンス・セッタたち及び C によってそのスーパークラスから継承したインスタンス・セッタたちである。

もしあるセッタの仮パラメタ・リストがまさしくひとつの要求された仮パラメタ *p* を含んでいないときはコンパイル時エラーである。*我々はこれは文法を介して施行できようが、その場合は我々は判定規則を規定することになる。*

あるセッタが **void** 以外の戻りの型を宣言するときは静的な警告となる。あるセッタ m_1 がセッタ m_2 をオーバライド（10.9.1 節）し、m_1 の型が m_2 の型の副型でないときは、静的な警告となる。

もしあるクラスが引数の型が *T* である *v*=という名前のセッタと、戻りの型が *S* である *v* という名前のゲッタをもち、*T* が *S* に代入出来ない可能性があるときは静的警告となる。

あるクラスが *v*=という名前のスタティックなセッタと、また *v* という名前の非スタティックのメンバを持っているときは静的警告となる。もしあるクラスが *v*=という名前のインスタンス・セッタと *v*= という名前のアクセス可能なスタティック・メンバを持っている、あるいは *v* が *C* のスーパークラスの中で宣言されているときは、静的警告となる。

ゲッタまたはセッタが明示的に宣言されているかあるいは暗示的に宣言されているかにかかわらず、これらの警告は出されねばならない。

10.4 抽象インスタンス・メンバ(Abstract Instance Members)

抽象メソッド(abstract method)（順に抽象ゲッタまたは抽象セッタ）は **external** と宣言されておらず実装を提供しないインスタンス・メソッド、インスタンス・ゲッタ、またはインスタンス・セッタである。*具体メソッド(concrete method)*（順に具体ゲッタまたは具体セッタ）は抽象でないインスタンス・メソッド、インスタンス・ゲッタ、またはインスタンス・セッタである。

*Dart の以前のバージョンでは抽象メンバたちは修飾子 **abstract** を前置することで識別されることが求められていた。この要求を削除する動機は抽象クラスをインターフェイスとして使えるようにしたい為である。Dart の各クラスは暗示的なインターフェイスを誘導する。*

*インターフェイス宣言の代わりに抽象クラスを使うことは重要な優位性を持つ。抽象クラスはデフォルト実装を提供できる;これはまたスタティックなメソッドたちを提供でき、その目的全体が与えられた型に関連するユーティリティたちをグループ化する **Collections** または **Lists** のようなサービス・クラスの必要性を無くしている。*

メンバたちへの明示的な就職子要求を削除したことにより、抽象クラスはより簡明となり、抽象クラスはインターフェイス宣言よりも魅力的な代替物としている。

抽象メソッド、抽象ゲッタ、あるいは抽象セッタを呼び出すと、適切なメンバａがスー

パークラスのなかで得られる場合を除き（その場合はａが呼びだされる）、まさしくあたかもその宣言が無かったかの如く NoSuchMethod の呼び出しが行われる。この規範的仕様はメソッド、ゲッタ、およびセッタたちの検索の定義のもとで出てくる。

抽象メソッドの目的は型チェックとリフレクションのような目的の為の宣言を提供することである。ミクスインとして使われるクラスに於いてはしばしば、そのミクスインが適用されるスーパークラスによってそのミクスインが予定しているメソッドが提供されるようにそのメソッドを宣言するのに有用である。

もし抽象メンバが具体クラス *C* の中で宣言されているまたは継承されているときは以下を除き静的警告となる:

- *m* が具体メンバをオーバライドしている、または
- C がクラス **Object** のなかで宣言されているものとは異なった **noSuchMethod()**を有している。

我々は抽象メンバたちを持った具体クラスが宣言されたときに警告をしたいと思う。しかしながら、以下のようなコードは警告なしで機能しなければならない:

```
class Base {
   int get one => 1;
 }

 abstract class Mix {
   int get one;
   int get two => one + one;
 }

 class C extends Base with Mix {
 }
```

*実行時には **Base** のなかで宣言された具体メソッド one が実行され、問題が起きてはいけない。従って警告は出してはならず、従ってこの階層のなかで対応する具体的メンバが存在している場合は警告を出さない。*

10.5 インスタンス変数(Instance Variables)

インスタンス変数は、あるクラス宣言のなかにすぐに含められていて **static** と宣言されていない変数たちである。あるクラス *C* のインスタンス変数は、*C* によって宣言されたインスタンス変数たち及びそのスーパークラスから *C* によって継承されたインスタンス変数たちである。

もしインスタンス変数が constant であると宣言されているときはコンパイル時エラーである。

constant なインスタンス変数という概念は捕えがたいものでありプログラマたちを混乱させる。

インスタンス変数はインスタンスごとに異なることを意図したものである。constant なインスタンス変数はすべてのインスタンスに対し同じ値を持たせることになり、従ってそれはすでにおかしなアイデアである。

この言語は const インスタンス変数宣言を定数を返すインスタンス・ゲッタだと解釈することになる。しかしながら、constant なインスタンス変数はそのゲッタがオーバライドの対象になるので真のコンパイル時定数として取り扱われないことになりかねない。

その値がそのインスタンスに依存しないときは、static クラス変数を使うほうが良い。必要ならインスタンス・ゲッタはマニュアルで常に定義できる。

10.6 コンストラクタ(Constructors)

*コンストラクタ(constructor)*はオブジェクト生成の為にインスタンス生成式(instanceCreation:16.12節)のなかでつくられる特別なメンバである。コンストラクタは生成的(generative)(10.6.1節)であるかまたはファクトリ(factories)(10.6.2節)になる。

*コンストラクタ名(constructor name)*は常にそれを最初に包含している(immediately enclosing)クラスまたはインターフェイスの名前で始まり、そしてオプション的にドットと識別子 *id* が続く。もし *id* が直ちに包含しているクラスの中で宣言されているメンバの名前のときは実行時エラーとなる。コンストラクタの名前がコンストラクタ名でないときはコンパイル時エラーである。

あるクラス *C* に対しコンストラクタが指定されていないときは、*C* がクラス **Object** で無い限りそれは暗示的なコンストラクタ *C*() : **super**() {}を持つ。

10.6.1 生成的コンストラクタ(Generative Constructors)

*生成的コンストラクタ(generative constructor)*はコンストラクタ名、コンストラクタ・パラメタ・リスト、及びリダイレクト句またはイニシャライザ・リストのどれか、及びオプショナルなボディからなる。

constructorSignature(コンストラクチャ・シグネチュア):
identifier(識別子) ('.' identifier)? formalParameterList(仮パラメタ・リスト)
;

*コンストラクチャ・パラメタ・リスト(constructor parameter list)*は丸カッコで括られ、仮コンストラクタ・パラメタたちのカンマで区切られたリストである。*仮コンストラクタ・パラメタ(formal constructor parameter)*は仮パラメタ(9.2節)または初期化仮パラメタかのどちらかである。*初期化仮パラメタ(initializing formal)*は **this**.*id* の形式をとる。*id* が最初に包含している(immediately enclosing)クラスのインスタンス変数の名前でないときはコンパイル時エラーとなる。初期化仮パラメタが非リダイレクトの生成的コンストラクタ以外の関数で使われているときはコンパイル時エラーとなる。

その初期化仮パラメタにたいし明示的な型が貼られているときは、それはその静的型である。そうでない場合、*id* という名前が付けられた初期化仮パラメタの型は T_{id} である。ここに T_{id} はそれを直ちに包含しているクラスの中の *id* という名前のフィールドの型である。*id* の静的型が T_{id} に代入出来ないときは静的警告となる。

仮パラメタ・リストの中に初期化仮パラメタ **this**.*id* を使うことはこのコンストラクタのスコープに仮パラメタ名を誘導させない。しかしながら、この初期化仮パラメタはまさしくあたかも同じ場所に *id* という名前の仮パラメタが誘導されたごとく、このコンストラクタ関数の型に影響を与える。

初期化仮パラメタたちは以下に詳述するように生成的コンストラクタの実行中に実行される。**this.**id という初期化仮パラメタの実行により、*id* が既に初期化されている final 変数で無い限り(この場合は実行時エラーが発生する)、直ちに包含しているクラスのフィールド *id* が対応した実パラメタの値に代入される。

上記の規則により初期化仮パラメタたちをオプショナルなパラメタとして使えるようになる：

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

は合法で、次と同じ効果を持つ：

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

*新規インスタンス(fresh instance)*というのは、その識別がそのクラスのこれまでに割り当てられた(assigned)インスタンスのどれとも区別されているものをいう。*生成的コンストラクタ*はその最初に包含しているクラスの新規インスタンスを常に割り当てる。

上記のことはそのコンストラクタが実際に走っているとき、new で走っているがごとく成り立つ。あるコンストラクタ *c* が const によって参照されているとき、*c* は走らないで、その代り基準的オブジェクト(canonical object)が検索されよう。インスタンス生成の節（16.12 節）を見られたい。

ある生成的コンストラクタ *c* がリダイレクト・コンストラクタでなく、ボディが指定されていないときは、その *c* は空のボディ{}をもつ。

リダイレクト・コンストラクタ(Redirecting Constructors)

生成的コンストラクタは*リダイレクト(redirecting)*であり得、その場合は別の生成的コンストラクタを呼び出すだけである。リダイレクト・コンストラクタはボディ部を持たず、その代りそのリダイレクトでどの

コンストラクタを呼び出すか、そしてどんな引数で呼び出すのかを指定するリダイレクト節(redirect clause)を持つ。

redirection（リダイレクト）:
':' **this** ('.' identifier（識別子）)? arguments（引数たち）
;

イニシャライザ・リスト(Initializer Lists)

*イニシャライザ・リスト(Initializer Lists)*はコロン':'で始まり、カンマで区切られた個々の*イニシャライザ（訳者注：初期化子ともいう）*のリストで構成される。イニシャライザには 2 つの種類がある。

- *スーパーイニシャライザ(superinitializer)*は*スーパーコンストラクタ(superconstructor)*、即ちスーパークラスの特定のコンストラクタを指定する。スーパーイニシャライザの実行によりスーパーコンストラクタのイニシャライザ・リストが実行される。

- インスタンス変数イニシャライザ(instance variable initializer)は個々のインスタンス変数にある値を代入する。

initializers（イニシャライザたち）:
':' superCallOrFieldInitializer（super 呼出しまたはフィールドのイニシャライザ） (',' superCallOrFieldInitializer（super 呼出しまたはフィールドのイニシャライザ）)*
;

superCallOrFieldInitializer（スーパー呼び出しまたはフィールド・イニシャライザ）:
super arguments（引数）
| **super** '.' identifier（識別子） arguments（引数たち）
| fieldInitializer（フィールド・イニシャライザ）
;

fieldInitializer（フィールド・イニシャライザ）:
(**this** '.')? identifier（識別子） '=' conditionalExpression（条件式） cascadeSection（カスケード区間）*
;

k が生成的コンストラクタだとする。そうすると *k* はそのイニシャライザ・リストにたかだかひとつのスーパーイニシャライザを含むが、そうでないとコンパイル時エラーを起こす。スーパーイニシャライザが用意されていないときは、それを包含するクラスが **Object** でない限り super()様式の暗示的スーパーイニシャライザが付加される。与えられたインスタンス変数に対応したイニシャライザが *k* のリストにひとつより多い場合にはコンパイル時エラーとなる。*k* のイニシャライザ・リストの中に *k* の初期化仮パラメタ(initializing formal)の手段で初期化されているある変数のイニシャライザが含まれているときはコンパイル時エラーとなる。

最初に包含しているクラスの中で宣言されている各 final なインスタンス変数 *f* は、以下の手段のどれかによって既に初期化されていない限り、*k* のイニシャライザ・リストのなかにイニシャライザを含んでいなければならない:

- *f* の宣言での初期化
- *k* の初期化パラメタの手段による初期化

そうでなければ静的警告が発生する。もし *k* のイニシャライザ・リストが最初に包含しているクラスの中で宣言されたインスタンス変数でない変数のためのイニシャライザを含んでいる場合はコンパイル時エラーとなる。

イニシャライザ・リストは無論、例えそれが final でないとしても、最初にそれを包含しているクラスによって宣言されているどのインスタンス変数の為のイニシャライザをも含むことが出来る。

クラス Object の生成的コンストラクタがスーパーイニシャライザを含んでいるときはコンパイル時エラーとなる。

生成的コンストラクタの実行は常に、その仮パラメタたちの為のバインディングたちのセットに対応して、及び新規インスタンス *i* への **this** バウンドと実型引数たち $V_1, \ldots, V_m$ のセットに対する直ちに包含するクラス・バウンドの型パラメタたちで、なされる。

これらのバインディングたちは通常そのコンストラクタを呼び出した（直接的にまたは間接的に）インスタンス生成式によって決定される。しかしながら、これらはまた反射的呼び出し(reflective call)によって、決定され得る。

もし *k* がリダイレクト・コンストラクタの場合は、そのリダイレクト句は **this**.$g(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式をとり、ここに *g* は直ちに包含しているクラスの別の生成的コンストラクタを識別するものである。そうすると k は引数リスト$(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の計算から始まり、次に *g* を$a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の計算から得られるバインディングたちと、$V_1, \ldots, V_m$ に対しバインドされた直ちに包含するクラスの型パラメタたちに対する this バインドで、実行される。

そうでない場合は、実行は次のように進行する:

k のパラメタ・リストの中で宣言されている初期化パラメタたちはそのプログラムの中で出現した順に実行される。次に *k* のイニシャライザたちがそのプログラムの中で出現した順に実行される。

副作用を与える外部ルーチンたちが呼び出される順序があり得る。従って我々はこの順番を必要としている。

総てのイニシャライザが完了したら、そのコンストラクタのボディが **this** が *i* にバインドされているスコープ内で実行される。ボディの実行は *k* のスーパーイニシャライザの引数によって決まるバインディングたちに対応したスーパーコンストラクタのボディ、実型引数のセット $V_1, \ldots, V_m$ にバインドされた直ちに包含するクラス・バウンドの型パラメタたち、及び *k* のスーパーイニシャライザの引数リス

トで決まる仮パラメタのバインディングたちで、なされる。

*このプロセスにより、このコードで初期化されていないファイナルなフィールドがないことが確保される。**this** はあるイニシャライザの右辺上のスコープ内にはなく(16.11 節参照)、従って初期化中にインスタンス・メソッドのどれも実行できないことに注意されたい:インスタンス・メソッドは直接呼び出すことは出来ず、また this はこのイニシャライザ内で呼び出されているどの他のコードにも渡されない。*

this.*v* = *e* の形式のイニシャライザの実行は以下のように進行する:

最初に、あるオブジェクト *o* として式(expression) *e* が計算される。次に **this** で示されたこのオブジェクトのインスタンス変数 *v* が *o* にバインドされる。チェック・モードでは、*o* が **null** でなくまた o のクラスのインターフェイスがフィールド v の静的型の副型でないときは動的型エラーとなる。

v = *e* 形式のイニシャライザは **this**.*v* = *e* の形式のイニシャライザと等価である。

super(a_1, …, a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})(各々 **super**.*id*(a_1, …, a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k}))の形式のスーパーイニシャライザの実行は以下のように進行する:

最初に、引数リストの(a_1, …, a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})が評価される。

そのスーパーイニシャライザが存在するクラスを *C* とし、*C* のスーパークラスを *S* とする。もし S が総称型(generics:第 14 章)のときは、U_1, ,.., U_m を *C* のスーパークラス節のなかの *S* に渡された実際の型パラメタたちだとする。

そうすると、コンストラクタ *S*(各々 *S.id*)のイニシャライザ・リストが引数リストの計算からもたらされたバインディングたちに対応して、現在の **this** のバインディングに対する **this** バインドで、及び U_1, ,.., U_m の現在のバインディングに対しバインドされたクラス *S* の型パラメタたち(もしあれば)で、実行される。

クラス *S* が *S*(各々 *s.id*)という名前のコンストラクタを持っていないときはコンパイル時エラーとなる。

10.6.2 ファクトリ(Factories)

*ファクトリ(factory)*は組み込み識別子(16.33 節)である **factory** が先行したコンストラクタである。

> **factoryConstructorSignature(ファクトリ・コンストラクチャ・シグネチュア):**
> **factory** qualified(修飾) ('.' identifier(識別子))? formalParameterList(仮パラメタ・リスト)
> ;

もし *M* が総称型でないときは、そのシグネチュアが **factory** *M* の形式または **factory** *M.id*の形式のファクトリの *戻りの型(return type)*は *M* であり、そうでないときは戻りの型は $M<T_1,...,T_n>$である。ここ

に $T_1,...,T_n$ は包含しているクラスの型パラメタたちである。

M が直ちに包含しているクラスの名前で無いときはコンパイル時エラーである。

チェック・モードに置いては、あるファクトリがその型が実際（19.8.1 項）の戻りの型の副型で無いオブジェクトを返すときは動的型エラーとなる。

ファクトリが null を返すのを認めることは無用であるかに見える。しかしこのルールが現在しているように、それを認めるほうがより一様化される。

ファクトリは他の言語でのコンストラクタに関わる古典的弱点に対処している。ファクトリは新規に割り当てられたものではないインスタンスを生成できる：これらはキャッシュから得られる。同様に、ファクトリは異なったクラスのインスタンスを返すことが出来る。

リダイレクト・ファクトリ・コンストラクタ(Redirecting Factory Constructors)

*リダイレクト・ファクトリ・コンストラクタ(Redirecting Factory Constructors)*は、リダイレクト・コンストラクタが呼ばれたときはいつでも使われることになる別のクラスのコンストラクタに対する呼び出しを指定する。

> **redirectingFactoryConstructorSignature（リダイレクト・ファクトリ・コンストラクチャ・シグナトリ）:**
>
> **const**? **factory** identifier（識別子） ('.' identifier)? formalParameterList（仮パラメタ・リスト）`=' type（型） ('.' identifier)?
>
> ;

リダイレクト・ファクトリ・コンストラクタ *k* を呼び出すと、*type*（各 *type.identifier*）で指定されたコンストラクタ *k'*が *k* に渡された実引数で呼び出され、*k'*の結果が *k* の結果として返される。結果としてのコンストラクタ呼び出しは、**new**（16.12 節）を使ったインスタンス生成式と同じ規則に従う。

結果としてもし type または type.id が指定されていないとき、あるいはクラスまたはコンストラクタを参照していないときは、他の未定義のコンストラクタ呼び出しと同様に動的エラーが発生する。同じことは、もし要求されているパラメタたちより少ないパラメタで、あるいは k'が予定しているよりも多い位置的パラメタで呼ばれた時、あるいは k'で宣言していない名前付きパラメタで呼ばれた時にも言える。

もし *k* があるオプショナルなパラメタのためにあるデフォルトの値を明示的に指定しているときはコンパイル時エラーである。

***k* のなかで指定されているデフォルト値たちは、*k'*に渡されるのは実パラメタたちであるので、無視される。従って、デフォルト値は許されない。**

リダイレクト・ファクトリ・コンストラクタが、直接的にまたは間接的にリダイレクションのシーケンスを介して自分自身をリダイレクトするのは実行時エラーである。

もしリダイレクト・ファクトリ F_1 が別のリダイレクト・ファクトリ F_2 にリダイレクトし、F_2 が次に F_1 にリダイレクトすると、F_1 と F_2 双方が間違って定義される。従ってそのようなサイクルは違法とされている。

もし *type* が現在のスコープ内でアクセス可能なクラスを示していないときは静的警告である； もし *type* がそのようなクラス *C* を示しているときは、参照されたコンストラクタ（*type* または *type.id* のかたちで）が *C* のコンストラクタでないときは静的警告となる。

k'に渡された引数たちを加工することは出来ないことに注意のこと。

一見して通常のファクトリ・コンストラクタが他のクラスのインスタンスを生成でき、リダイレクト・ファクトリは不要だと人は考えるかもしれない。しかしながら、リダイレクト・ファクトリには幾つかの利点がある：

- *抽象クラスは別のクラスの定数コンストラクタを活用した定数コンストラクタを提供できる。*
- *リダイレクト・ファクトリ・コンストラクタはフォワード側がそのシグネチュアの中の仮パラメタたちのデフォルト値を繰り返す必要性を回避する。*

もし *k* が **const** 修飾子で前置されているが *k'*が定数コンストラクタ（10.6.3 節）で無いときはコンパイル時エラーである。

*k'*の関数型が *k* の型の副型で無いときは静的警告である。

このことは結果となるオブジェクトは *k* の直ちに包含するクラスのインターフェイスを満たすことを意味する。

もし *k'*に対する型引数のどれかが対応する *type* の仮型パラメタたちのバウンドたちの副型で無いときは静的型警告である。

10.6.3　常数コンストラクタ(Constant Constructors)

*常数コンストラクタ(constant constructor)*はコンパイル時常数（16.1 節）オブジェクトを生成するのに使える。常数コンストラクタは予約語である **const** が先行している。

constantConstructorSignature（常数コンストラクチャ・シグネチュア）:
　　const qualified（修飾） formalParameterList（仮パラメタ・リスト）
;

常数コンストラクタの仕事の総てがそのイニシャライザたちを介してとり扱われねばならない。

非ファイナルなインスタンス変数を持つクラスによってある常数コンストラクタが宣言されているときはコンパイル時エラーとなる。

上記はローカルに宣言された、及び継承したインスタンス変数の双方に適用される。

C の中で宣言されたあるインスタンス変数が定数式でない式で初期化されているときは、定数コンストラクタが該クラス C で宣言されているときはコンパイル時エラーである。

必然的に同じく定数コンストラクタを宣言しなければならない（インスタンス変数を宣言しない **Object** を除き）ので、C のスーパークラスはそのようなイニシャライザを宣言できない。

定数コンストラクタの初期化リストのなかで、明示的または暗示的に、出現するスーパ・イニシャライザは直ちに包含しているクラスのスーパークラスの定数コンストラクタを指定しなければならない。そうでないときはコンパイル時エラーが発生する。

ある常数コンストラクタのイニシャライザ・リスト内にあるどの式も*潜在的常数式(potentially constant expression)*で無ければならず、そうでないとコンパイル時エラーが発生する。

*潜在的常数式*とは、もし最初に包含している定数コンストラクタの総ての仮パラメタたちが、それらの最初に包含している super 式(super expression)によって要求されているような整数、ブール値、あるいは文字列の値として計算することが保証されているコンパイル時定数たちとして取り扱われているなら、有効な定数式であるような式 *e* のことを言う。

一部の型に制限されているスーパー式のなかで使われていないパラメタは、任意の型の定数であり得る。例えば:

```
class A {
  final m;
  const A(this.m);
}
```

は A(const[]);でインスタンス化され得る。

潜在的常数式とコンパイル時常数式(compile-time constant expression)（16.12.2 節）との相違には少々説明が必要である。

問題はあるコンストラクタの仮パラメタたちをコンパイル時常数（compile-time constants）として取り扱うかどうかということである。

常数オブジェクト式(constant object expression)からある常数コンストラクタが呼び出されているときは、実際の引数はコンパイル時常数であることが必要になろう。従って、もし我々が常数コンストラクタたちが常数オブジェクト式から常に呼び出されるようにしていれば、我々はあるコンストラクタの仮パラメタたちがコンパイル時常数であることが保証できる。

しかしながら、常数コンストラクタはまた通常のインスタンス生成式（ordinary instance

creation expressions：16.12.1 項）から呼び出され得る、従って上記の仮定は一般的に有効ではない。

それにもかかわらず、コンストラクタ内で常数コンストラクタの仮パラメタたちを使用することは相当有用なものである。潜在的常数式の概念はそのような仮パラメタたちの制限された仕様を促進させる為に導入されている。特に、我々は組み込み演算子が絡む式の為に常数コンストラクタの仮パラメタたちの使用を許しているが、常数オブジェクト、リスト、及びマップの為には許していない。これによりユーザには以下のようなコンストラクタが許される：

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

このｘへの代入は、ｐがコンパイル時定数（たとえｐがそうでなくても、一般にはコンパイル時定数）であるとの仮定の下で許される。ｙへの代入は同じようではあるが、更なる問題を提起する。この場合、ｐのスーパ式はｐ + 100 であり、式全体が定数とみなされる為にはｐはコンパイル時定数であることが必要である。この仕様書の記述ではｐを整数として評価することを我々が想定することを許している。同じ主張がｚへの代入のなかでのｐとｑに対しても成立する。

しかしながら、以下のコンストラクタは許されない：

```
class D {
  final w;
  const D.makeList(p): w = const [p];  // コンパイル時エラー
  const D.makeMap(p): w = const {“help”: p}; // コンパイル時エラー
  const D.makeC(p): w = const C(p, 12); // コンパイル時エラー
}
```

ｗへの代入が潜在的に定数でないことが問題ではない；それらは定数である。しかしながらこれらの総てが定数リスト（16.7 節）、マップ（16.8 節）、及びオブジェクト（16.12.2 節）の規則に従っておらず、これらの総てが独立に定数式（constant expressions）への複式(subexpressions)を必要としている。

*上記の違反したＤのコンストラクタたちの総てが**new**を介して呼び出すことは出来なかった、何故なら定数でなければならない式が定数であることもそうでないこともあり得る仮パラメタに依存出来なかったからである。これに対比して、違反していない例ではそのコンストラクタが**const**を介してまたは**new**を介しているかに関わらず合理的である。*

慎重な読者たちは無論Ｃ()に対する実際の引数たちが定数ではあるがしかるべき型でない場合に関し心配するであろう。これは定数オブジェクトたちの評価の為の規則（15.12.2 節）にあわせて、以下の規則によって排除される。

定数オブジェクト式から呼び出されたとき、潜在的な定数式たちのひとつの要因となる値となるべき実際のパラメタたちのどれかが有効なコンパイル時定数とならないような場合は定数コンストラクタは例外をスローしなければならない。

10.7 static メソッド(Static Methods)

*static メソッド(static method)*とはその宣言があるクラス宣言のなかにすぐに含まれていて、**static** と宣言されている関数のことを言う。クラス *C* の static メソッドとは *C* によって宣言されている static メソッドのことである。

クラス *C* のなかの static メソッド宣言の効果は、その static メソッドに転送する(9.1 節)クラス *C* の為の **Type** オブジェクトと同じ名前とシグネチュアを持ったインスタンス・メソッドを付加することである。

Dart における static メソッドの継承は余り有用性は無い。必要な何らかの static 関数はそれを宣言しているライブラリから取得でき、継承を介してスコープ内に持ち込む必要は無い。経験によればデベロッパたちは継承されたメソッドはインスタンス・メソッドではないという発想に混乱している。

無論 static メソッドの概念には議論があるが、多くのプログラマたちがこれになじんでいるのでここでは保持されたままにしてある。Dart の static メソッドは包含しているライブラリの関数として見て良い。
もしクラス *C* が *n* という名前の static メソッドを宣言しており、*n* =という名前のセッタを有しているときは静的警告である。

10.8 static 変数(Static Variables)

*static 変数(static variable)*とはその宣言があるクラス宣言のなかにすぐに含まれていて、**static** と宣言されている変数のことを言う。クラス *C* の static 変数とは *C* によって宣言されている static 変数のことである。
(訳者注:この節は 0.11 版で殆どが削除されている)

10.9 スーパークラス(Superclasses)

あるクラス *C* の **extends** 節はそのスーパークラスを指定する。width 句 **with** M_1, ..., M_k を持ち extends 句 **extends** S を持ったあるクラス C のスーパークラスは、S に対するミクスイン(第 12 章) M_k* .. * M_1 のアプリケーションである。width 句が指定されていないときはあるクラス C の extends 句 **extends** S は、そのスーパークラスを指定する。**extends** 節が指定されていないときは以下のいずれかである:

- *C* はスーパークラスを持っていない **Object** である、または
- クラス *C* は **extends** Object の形式の **extends** 節をもっていると見做され、上記規則が適用される。

クラス **Object** に対し **extends** 節を指定するとコンパイル時エラーとなる。

superclass（スーパークラス）:
extends type（型）
;

あるクラス *C* の **extends** 句及び **width** 句のスコープは *C* の型パラメタ・スコープである。

クラス *C* の **extends 句**がスーパークラスとして列挙型(enumerated type)（13 章）、異形(malformed)型、または後回し型(deferred type)（19.1 節）を指定しているときはコンパイル時エラーとなる。

総称クラスの型パラメタはスーパークラス句の構文スコープ内で使え、潜在的な包含しているスコープのなかのクラスたちをシャドウイングする。従って以下のコードは許されず、コンパイル時エラーにしなければならない。

```
class T{}
class G<T> extends T {} // コンパイル・エラー：型パラメタをサブクラス化しようとしている
```

以下のいずれかの場合はクラス *S* はクラス *C* の*スーパークラス*である:

- *S* は *C* のスーパークラスである、または
- *S* がクラス *S'*のスーパークラスで、*S'*が *C* のスーパークラスである

クラス *C* が自分自身のスーパークラスのときはコンパイル時エラーとなる。

10.9.1 継承とオーバライド(Inheritance and Overriding)

C をあるクラスとし、*A* が *C* のスーパークラスだとし、$S_1 \ldots S_k$ が *A* のサブクラスでもある *C* のスーパークラスたちだとしよう。*C* は *C* または $S_1 \ldots S_k$ の少なくともひとつで宣言によりオーバライドされていない A のすべてのアクセス可能なインスタンス・メンバたちを継承する。

直接のスーパークラス S のメンバたちにのみ依存するような継承の純粋にローカルな定義を与えることはもっと魅力的かもしれない。しかしながら、あるクラス C はそのスーパークラス S のメンバでないメンバ m を継承できる。このことはメンバ m が C のライブラリ L_1 にたいしプライベートであり、一方 S が別のライブラリ L_2 から来ているものの、S のスーパークラスのチェインが L_1 で宣言されたクラスを含んでいる場合に起きえる。

クラスはそうでなければそのスーパークラスから継承していたであろうインスタンス・メンバたちをオーバライドできる。

$C = S_0$がライブラリLのなかで宣言されたあるクラスだとし、$\{S_1 \ldots S_k\}$をCのすべてのスーパークラスたちのセットだとする。ここにS_iは$1 .. k$のなかのiに対するS_{i-1}のスーパークラスである。Cがあるメンバmを宣言しており、m'はS_iは$1 .. k$のなかのjに対するS_jのメンバで、かつmと同じ名前を有しており、m'がLに対しアクセス可能だとする。そうすると、m'が$S_1 \ldots S_{j-1}$の少なくともひとつのメンバによって既にオーバライドされておらず、mとm'のどれもフィールドでないならば、mはm'をオーバライドする。

フィールドたちは決して相互にオーバライドしない。フィールドによって誘導されたゲッタとセッタたちは相互にオーバライドする。

ここでもオーバライドのローカルな定義は好ましいものであるが、ライブラリのプライバシに対処できない。

そのオーバライドが違反していないかどうかは本仕様書の別の個所に記載されている(10.1 節のインスタンス・メソッド、10.2 節のゲッタ、及び 10.3 節のセッタ)。

例えばゲッタとセッタは合法にメソッドたちをオーバライドしないことがありまたその逆もある。セッタとメソッドは、それらの名前が常に一致しないので相互にオーバライドすることは決してない。

それにも拘らずこのようにメンバたち間のオーバライド関係を定義して、違反したケースを簡潔に記述できることは便利なことである。

インスタンス変数たちはこのオーバライドの関係に加わっていないが、それらが誘発しているゲッタたちやセッタたちは関わっていることに注意されたい。同じく、ゲッタたちはセッタたちをオーバライドせず、またその逆もそうである。最後に static なメンバたちは決してなにをもオーバライドしない。

非抽象クラスが抽象メソッドをオーバライドすると静的警告となる。

利便性の為に以下に関連規則の要約を示す。これは規範的なもので無いことに注意。正規の文言は本仕様書の関連した節にある。

1. ゲッタ、セッタ、メソッド、及びコンストラクタ(6.1 節)にはただひとつの名前空間がある。フィールドfはゲッタfをもたらし、非 **final** なフィールドfはまたセッタ$f=$(10.5、10.8 節)をもたらす。ここで我々がメンバというときは、それはアクセスできる(accessible)フィールド、ゲッタ、セッタ、及びメソッド(第 10 章)のことをいう。
2. 宣言されたものまたは継承したもの(6.1 節、第 10 章)でも、同じクラスの中に同じ名前を持った 2 つのメンバを持つことはできない。
3. スタティックなメンバは決して継承されない。
4. 自分のクラスまたはスーパークラス(継承していないものであっても)のなかにmという名前

のスタティック・メンバがある、及びおなじく同じ名前のインスタンス・メンバがあると警告となる（10.1、10.2、10.3 節）。

5. スタティックなセッタ *v*=とインスタンス・メンバ *v* があると警告となる（10.3 節）。
6. スタティックなゲッタ *v*=とインスタンス・セッタ *v*=があると警告となる（10.2 節）。
7. もし m という名前のインスタンス・メンバを定義し、自分のスーパークラスが同じ名前のインスタンス・メンバを持っていると、それらは互いにオーバライドする。これは違反な場合もそうでない場合もある。
8. 2 つのメンバたちが相互にオーバライドしているとき、それらの型シグネチュアが互いに代入出来ないときは静的警告となる（10.1、10.2、10.3 節）（そしてこれらは関数型なので、このことは”相互に副型”ということと同じことを意味する）。
9. 2 つのメンバたちが相互にオーバライドしているとき、それらが必要とするパラメタたちの数が異なっているときはコンパイル時エラーである（10.1 節）。
10. 2 つのメンバたちが相互にオーバライドしているとき、オーバライドしているメンバぼオプショナルな位置的パラメタの数がオーバライドされているメンバのそれよりも少ないときはコンパイル時エラーである（10.1 節）。
11. 2 つのメンバたちが相互にオーバライドしているとき、オーバライドしているメンバがオーバライドされているメンバの総ての名前付きパラメタたちの持っていないときはコンパイル時エラーである（10.1 節）。
12. セッタ、ゲッタ、及び演算子たちはどの種のオプショナルなパラメタを決して持たない；これはコンパイル時エラーである（10.1.1、10.2、10.3 節）。
13. あるメンバがそれを包含しているクラスと同じ名前を持っているときはコンパイル時エラーである（第 10 章）。
14. クラスは暗示的なインターフェイスを持っている（第 10 章）。
15. インターフェイスのメンバたちはクラスによって継承されないが、その暗示的なインターフェイスによって継承される。インターフェイスはそれ自身の継承規則を持っている（11.1.1 節）
16. メンバはそれがボディを持たず **externel** とラベルが付られていないときは抽象メンバである（10.4、9.4 節）。
17. クラスは明示的に **abstract** とラベルが付されているときに限り抽象クラスである。
18. 具体クラスが抽象メンバ（宣言または継承して）を持っているときは静的警告となる。
19. 抽象クラスの非ファクトリなコンストラクタを呼ぶと静的警告と動的エラーとなる（16.12.1 節）。
20. あるクラスが *m* という名前のインスタンス・メンバを定義し、そのスーパークラスのどれかが *m* という名前のメンバを持っているとき、そのクラスのインターフェイスが *m* をオーバライドする。
21. インターフェイスはそのスーパーインターフェイスのオーバライドされておらずまた複数のスーパーインターフェイスのメンバたちで無い総てのメンバを継承する。
22. もしあるインターフェイスの複数のスーパーインターフェイスが同じ m という名前のメンバを定義しているとき、たかだかひとつのメンバが継承される。そのメンバは（もし存在すれば）他の総ての副型の型である。そのようなメンバ存在しないときは：
    1. 静的警告が出される。
    2. 可能なら、それらスーパーインターフェイスたちの総てのメンバたちと同じ数の同じ要求されるパラメタたち、オプショナルな位置的パラメタたちの最大数、及び名前付きパラメタたちのスーパーセットを持った *m* という名前のメンバをインターフェイスは得る。こ

れらの型たちは総て **dynamic** である。もしそれが可能なら(そのスーパーインターフェイスのメンバたちは必要なパラメタたちの数が異なるので)、m という名前のメンバはそのインターフェイスには出てこない。
(11.1.1 節)

23. 規則 8 はクラスとともにインターフェイスにも適用される(11.1.1 節)。
24. ある具体クラスがそのどのスーパーインターフェイス内のメソッド為の実装を持っていないときは、それがそれ自身の noSuchMethod メソッドを宣言していない限り、静的警告となる(10.10 節)。
25. 名前付きコンストラクタの識別子は同じクラス内で宣言された(継承した場合とは反対に)メンバの名前とは同じにはなれない(10.6 節)。

10.10 スーパーインターフェイス(Superinterfaces)

クラスは直接のスーパーインターフェイスのセットを持つ。このセットというのはそのスーパークラスのインターフェイス及びこのクラスの **implements** 節の中で指定されたインターフェイスである。

interfaces(インターフェイス):
implements typeList(型リスト)
;

あるクラス *C* の **implements** 句のスコープは *C* の型パラメタ・スコープである。

あるクラス *C* の **implements** 節がスーパーインターフェイスとして型変数を指定しているときはコンパイル時エラーである。あるクラス *C* の **implements** 節がスーパーインターフェイスとして奇形の型(malformed type)、列挙型(enumerated type)、または後回し型(deffered type)を指定しているときはコンパイル時エラーとなる。あるクラスの **implements** 節が型 **dynamic** を指定しているときはコンパイル時エラーである。あるクラス *C* の **implements** 節がスーパーインターフェイスとして型 T を一回以上指定しているときはコンパイル時エラーである。あるクラス *C* のスーパーインターフェイスが *C* のスーパーインターフェイスとして指定されているときはコンパイル時エラーである。

このようにある型を繰り返すのは有害で、どうしてそれをエラーとするのか?と主張する人もいよう。問題はプログラム・ソースに書かれた状況がエラーが多いということはそれほど問題ではなく、それが要領を得ないものだということである。従ってそれはそのプログラマが何か別のことを意味し、そのプログラマに指摘すべきミステークたということを示している可能性が高い。それでもわれわれは単に警告を出すだけにできよう、そして多分我々はそうすべきである。しかしながら、この種の問題はローカルでありその場で容易に修正されるので、強硬路線をとることは正当だと我々は思う。

あるクラス *C* から誘導されたインターフェイスがそれ自身のスーパーインターフェイスのときはコンパイル時エラーとなる。

C をクラス **Object** で定義されているものとは別の **noSuchMethod**()メソッドを有していない *具体クラス(conclete class)*だとする。もし *C* の暗示的なインターフェイスが型 *F* のあるインスタンス・メンバを

含んでおり、*C* が対応する *F* <: *F* のような型 *F'* のインスタンス・メンバ *m* を宣言または継承していないときは静的警告である。

クラスはそのスーパーインターフェイスたちからのメンバたちを継承しない。しかしながらその暗示的なインターフェイス(implicit interface)は継承する。

我々は具体クラスに対してのみ警告することを選択している;抽象クラスは具体副クラスがそのインターフェイスの一部を実装するだろうことを想定して合法的に設計され得る。我々はまた ***noSuchMethod()*** *宣言が存在する、あるいは* ***Object*** *以外のクラスから継承しているときにはこれらの警告を出さないようにしている。そのような場合には、対応されるインターフェイスが* ***noSuchMethod()*** *を介して実装されるものであり、実装されたインターフェイスのメンバたちの実際の宣言は必要とされない。これにより特定の型たちの為のプロキシ・クラスたちが型警告を起こすことなく実装されるようになる。*

クラス *C* の暗示的インターフェイスが型 *F* のインスタンス・メンバ *m* を含み、*F'*が *F* の副型で無いときに *C* が対応する型 *F'*の *m* のインスタンス・メンバを宣言または継承していないときは静的警告である。

しかしながら、あるクラスがそのスーパーインターフェイスとぶつかるメンバを明示的に宣言している場合、このときは常に静的警告となる。

11. インターフェイス(Interfaces)

*インターフェイス(interface)*はユーザがあるオブジェクトとどのように関わり合えるかを定義する。インターフェイスはメソッドたち、ゲッタたち、セッタたち及びコンストラクタたち、及びスーパーインターフェイスたちのセットを持つ。
（訳者注：0.11 版からインターフェイス定義シグネチュア、11.2 及び 11.3 節は削除されている）

11.1 スーパーインターフェイス(Superinterfaces)

インターフェイスは直接のスーパーインターフェイスたち(direct supeinterfaces)のセットを持つ。

J が *I* の直接のスーパーインターフェイスであるか、*J* が *I* の直接のスーパーインターフェイスのスーパーインターフェイスであるときにかぎり、インターフェイス *J* はインターフェイス *I* のスーパーインターフェイスである。

11.1.1 継承とオーバライド(Inheritance and Overriding)

J があるインターフェイスで *K* があるライブラリだとしよう。我々は*継承した(J, K) (inherited(J, K))*のことを以下の総てが成り立つメンバたち *m* のセットだと定義する：

- *m* は *K* にアクセスでき、また
- *A* は *J* の直接のスーパーインターフェイスであり、以下のいずれかである：
  - *m* は *A* のメンバである、または
  - *m* は*継承した(A, K)*のメンバである
- *m* は *J* によってオーバライドされていない。

更に我々は*オーバライドたち(J, K)*（*overrides(J, K)*）のことを以下の総てが成立するようなメンバたち *m'* のセットだと定義する：

- *J* はあるクラス *C* の暗示的なインターフェイスである。
- *C* はあるメンバ *m* を宣言している。
- *m'* は *m* と同じ名前を有している。
- *m'* は *K* にアクセス可能である

- *A* は *J* の直接のスーパーインターフェイスである。そして以下のいずれかである:
  - *m'*は *A* のメンバである、または
  - *m'*は*継承した(A, K)*のメンバである。

I がライブラリ *L* のなかで宣言されたクラス *C* の暗示的インターフェイスだとする。*I* は*継承した(I, L)* の総てのメンバたちを継承し、もし *m'*が*オーバライドたち(I, L)*のなかに *m* がいるとき *I* は *m'*をオーバライドする。

上記 7 章のなかで与えられているインスタンス・メンバたちのオーバライドに関するすべての静的警告は、インターフェイス間のオーバライドにも適用される。

もし *m* がメソッドであり、*m'*がゲッタの時、またはもし *m* がゲッタで *m'*がメソッドの時は静的警告である。

しかしながら、上記の規則により継承されるであろう同じ名前 *n* を持った複数のメンバたち m_1, …, m_k が存在するとき（何故なら幾つかのスーパーインターフェイス内に同じ名前のメンバが存在したため）は、最大 1 個のメンバが継承される。

もし m_i, i, $1 <= i <= k$ の総てではない幾つかがゲッタであるとき、m_i のうちのどれも継承されず、静的警告が出される。

そうでない場合、もしメンバたち m_1, …, m_k の静的型たち T_1, …, T_k が同じでないときは、総ての i, $1 <= i <= k$ に対し $T_x <: T_i$, $1 <= x <= k$ なるメンバ m_x が存在しなければならず、そうでなければ静的警告が発生する。継承されるメンバはもし存在すれば m_x で、そうでなければ:

- numberOfPositionals(*f*)はある関数 *f* の位置的パラメタたちの数を意味し、numberOfRequiredParams(*f*)はある関数 *f* の要求パラメタたちの数を意味するとしよう。更に、*s* は m_1, …, m_k の総ての名前付きパラメタたちのセットだとしよう。次に $h = max(numberOfPositionals(mi))$, $r = numberOfRequiredParams(mi)$, $1 <= i <= k$ としよう。もし $r <= h$ なら、*I* は *n* という名前、**dynamic** 型の必要とするパラメタたち *r*、**dynamic** 型の名前が付けられたパラメタたち *s*、及び **dynamic** 型の戻りの型を持つ。

- そうでないときは、m_1, …, m_k のどれも継承されない。

ランタイムがこれを問題とするであろう唯一の状況は、ミラーがあるインターフェイス・メンバのシグネチュアを取得しようとしたときのリフレクションの最中であろう。

現在のソリューションはいささか複雑ではあるが、型アノテーション変更の面では堅牢なものである。代替手段としては:(a)矛盾したときはどのメンバも継承しない。(b)最初の m が選択される（スーパーインターフェイスのリストの順に基づいて）。(c)継承メンバがランダムに選択される。

(a)はインターフェイスの継承したあるメンバの存在は型シグネチュアによって異なることを意味する。(b)はその宣言の無関係な詳細に敏感であり、(c)は実装間あるいは異なったコンパイル・セッション間でさえも予測できない結果をもたらしがちである。

12.ミクスイン(Mixins)

ミクスインはあるクラスとそのスーパーインターフェイス間の相違を記述したものである。ミクスインは直接宣言されるかまたは既存のクラス宣言から引き出される。

もし宣言されたまたは得られたミクスインが **super** を参照しているときはコンパイル時エラーである。もし宣言されたまたは引き出されたミクスインが明示的にあるコンストラクタを宣言しているときはコンパイル時エラーである。そのスーパークラスが **Object** でないクラスから引き出されているときはコンパイル時エラーである。

これらの制約は暫定的なものである。我々は Dart の後の版でこれらを削除する予定である。

***super** の使用に関する本制約により、該ミクスインが異なったスーパークラスたちにバインドされるときの **super** への再バインドの問題を回避している。*

コンストラクタに関するこの制約は、インスタンス生成プロセスがシンプルになるので、ミクスインのアプリケーションのインスタンス化(コンストラクション)を簡素化する。

スーパークラスに関するこの制約は、そこからあるミクスインが引き出されるあるクラスの型が常にそれをミックス・インするどのクラスによっても常に実装されるということを意味する。これにより我々は如何にそのスーパークラスとスーパー・インターフェイスの型たちと独立して該ミクスインの型を表現するかあるはすべきかどうかの問題を後回しにできる。

これらの総ての問題のしかるべき答えは存在するが、それらの実装は容易ではない。

12.1 ミクスインのアプリケーション(Mixin Application)

ミクスインはあるスーパークラスに適用され、新しいクラスをもたらす。ミクスインのアプリケーションは、あるミクスインがその **with** 句を介してあるクラス宣言に混ぜ合わされる(ミックス・インされる)とき生じる。ミクスインのアプリケーションはセクションあたりあるクラスを拡張する為に使われ得る(第 10 章);言い換えればあるクラスはこの節で記されているようにミクスインのアプリケーションとして定義されても良い。

mixinApplicationClass(ミクスイン・アプリケーション・クラス):
  identifier(識別子) typeParameters(型パラメタたち)?'=' mixinApplication(ミクスイン・アプリケーション) ';'
mixinApplication(ミクスイン・アプリケーション):
    type(型) mixins(ミクインたち) interfaces(インターフェイスたち)?
  ;

S **with** *M*;の形式のミクスイン・アプリケーションは、スーパークラスがスーパクラス *S* を持ったあるクラス *C* を定義する。

S **with** $M_1, ..., M_k$;の形式のミクスイン・アプリケーションは、そのスーパークラスが *S* に対するミクスイン・コンポジション（12.2 節）$M_{k-1}, *... * M_1$ のアプリケーションであるクラス *C* を定義する。

上記の 2 つのケースともに、*C* は *M* と同じインスタンス・メンバたちを宣言している。もし *M* のインスタンス・フィールドたちのどれかがイニシャライザを持っているときは、これらは *C* の対応するフィールドたちの初期化は *M* のスコープ内で実行される。

S の $q_i(T_{i1}\ a_{i1}, ...\ , T_{iki}\ a_{iki})$, $1 <= i <= n$ という名前の各生成的コンストラクタにたいし、*C* は $q'_i(a_{i1}, ...\ , a_{iki})$:**super**$(a_{i1}, ...\ , a_{iki})$;の形式の $q'_i = [C/S]q_i$ という名前の暗示的に宣言されたコンストラクタを有する。

もしそのミクスインのアプリケーションがインターフェイスたちに対応することを宣言しているときは、結果としてのクラスはこれらのインターフェイスたちを実装する。

もし *S* が列挙型（13 章）または奇形の型のときはコンパイル時エラーである。もし *M*（各々、$M_1, ..., M_k$ のどれか）が奇形の型のときはコンパイル時エラーである。もし良く構成されたミクスインが *M*（または $M_1, ..., M_k$）から引き出せないときはコンパイル時エラーである。

K が *C* と同じスーパーインターフェイスとスーパークラス、及び *M*（または $M_1, ..., M_k$）で宣言されたインスタンス・メンバたちを持ったあるクラス宣言だとしよう。もし *K* の宣言が静的警告を引き起こす場合はコンパイル時エラーである。もし *K* の宣言がコンパイル時エラーを引き起こす場合はコンパイル時エラーである。

もし、例えば、S のなかの同じ名前のあるメンバの型と合致しないインスタンス・メンバ *im* を *M* が宣言しているときは、あたかも *im* を含めたボディを持った S を継承する通常のクラス宣言によって我々が *K* を定義したかのごとく、これは静的警告をもたらす。

ライブラリ *L* のなかでの **class** $C = M;$の形式、または **class** $C<T_1, ..., T_n> = M;$の形式のクラス宣言の効果は、ミクスイン・アプリケーション *M* で定義されたクラス（第 10 章）にバインドされた名前 *C* を *L* のスコープに導入することである。そのクラスが組込み識別子 abstract で前置されているときに限り、そのクラスは抽象クラスとして定義されている。

12.2 ミクスイン構成(Mixin Composition)

Dart はミクスイン構成を直接サポートしていないが、この概念はミクスイン句を持ったあるクラスのスーパークラスがどのように生成されるかを定義する際に有用である。

2 つのミクスイン $M_1<T_1 ... T_kM_1>$ 及び $M_1<U_1 ... U_kM_2>$ からなる*ミクスイン構成*は $M_1<T_1 ... T_kM_1> * M_1<U_1 ... U_kM_2>$と書かれ、これはどのクラス $S<V_1 ... V_{ks}>$に於いても $S<V_1 ... V_{ks}>$に対する $M_1<T_1 ... T_kM_1> * M_1<U_1 ... U_kM_2>$のアプリケーションは以下と等価であるような匿名ミクスイン(anonymous mixin)を定義する:

abstract class $Id_1<T_1 \ldots T_{kM1}, U_1 \ldots U_{kM2}, V_1 \ldots V_{kS}> = I_{d2}<U_1 \ldots U_{kM2}, V_1 \ldots V_{kS}>$ **with** $M_1 <T_1 \ldots T_{kM1}>$;

ここで I_{d2} は以下のものを示す:

abstract class $Id_2<U_1 \ldots U_{kM2}, V_1 \ldots V_{kS}> = S<V_1 \ldots V_{kS}>$ **with** $M_2<U_1 \ldots U_{kM2}>$;

また Id_1 と I_{d2} は該プログラム中のどこにも存在しないユニークな識別子である。

ミクスイン構成で作られたクラスたちは、独立してインスタンス化できないので抽象であると見做される。これらは通常のクラス宣言とミクスイン・アプリケーションの匿名スーパークラスとしてのみ導入される。従って、もしあるミクスイン構成が抽象メンバたちを含んでいたり、あるいは不完全にあるインターフェイスを実装していたときには警告は出されない。

ミクスイン構成は結合的(associative)である。

M_1、M_2、及び S のどのサブセットも総称である場合もない場合もあることに注意。非総称宣言にたいして、それに対応する型パラメタたちは省略され得、また引き出された宣言 Id_1 及び/または Id_2 のなかに型パラメタが残っていないときは、これらの宣言はいずれも総称である必要はない。

13.列挙型(Enums)

*列挙型(enumerated type)*または*enum*は固定数の定数値たちを表現するのに使われる。

enumType（列挙型）:
metadata（メタデータ） **enum** id（識別子） ‘ {’ id [‘, ’ id]* [‘, ’] ‘}’
;

metadata **enum** E { id_0, . . . id_{n-1}};の形式の宣言は次のクラス宣言と同じ効果を持つ:

```
metadata class E {
final int index;
const E(this.index);
static const E id0 = const E(0);
. . .
static const E id_{n-1} = const E(n - 1);
static const List<E> values = const <E>[id0 . . . idn−1];
String toString() => { 0: ‘E.id0’, . . ., n-1: ‘E.idn−1’}[index]
}
```

enumをサブクラス化する、ミクスインする、または実装する、またはあるenumを明示的にインスタンス化するのはコンパイル時エラーである。これらの制約はの10.9, 10.10, 12.1及び16.12節に規範的に記されている。

14.総称型(Generics)

クラス宣言（第 10 章）または型エイリアス（19.3.1 節）*G* は*総称型(generic)*でありえる、即ち *G* は宣言された仮型パラメタたち（formal type parameters）を持つことが出来る。総称型宣言は宣言たちのファミリを誘導(induce)し、そのプログラムのなかで用意された実際の型パラメタたち(actual type parameters)の各セットにひとつの宣言をもたらす。

typeParameter（型パラメタ）:
metedata（メタデータ） identifier（識別子） (**extends** type（型）)?
;

typeParameters（型パラメタたち）:
'<' typeParameter（型パラメタ） (',' typeParameter（型パラメタ）)* '>'
;

型パラメタ *T* は *T* の*上界(upper bound)（上バウンドともいう）*を指定する **extends** 節でサフィックスを付けることが出来る。**extends** 節が付いていないときはその上界は **Object** である。ある型変数がその上界のスーパー型のときは静的型警告である。型変数たちのバウンドたちは型アノテーションの形式であり、運用モードでは実行に何の効果も持たない。型パラメタたちはあるクラスの型パラメタのスコープ内で宣言される。総称型クラス宣言 *G* の型パラメタたちは、*G* の型パラメタたちの総てのバウンドたち内のスコープ内にある。総称型クラス宣言 *G* の型パラメタたちはまた、G の **extends** 及び **implements** 句（もし存在すれば）内のスコープ内にある。しかしながら、スタティック・メンバ内から型パラメタを参照するのはコンパイル時エラーである。しかしながら、ある static メンバにより参照されているときは、型パラメタは奇形型と考えられる。

*Static たちは汎用体の総てのインスタンス化で共有されるため、static なメンバのコンテキストの中では型変数は意味が無いのでこの制約が必要となる。しかしながら、型変数は例え **this** が得られない場合でもインスタンス・イニシャライザから参照され得る。*

型パラメタたちがそれらのバウンドたちのスコープ内にあるので、我々はF-バウンド定量化(F-bounded quantification)をサポートしている（これが何かを知っていないときは質問しないで欲しい）。これにより次のような型チェックのコードが可能となる。

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

型パラメタたちがスコープ内にあったとしても、現時点では多くの制約がある：

- 型パラメタたちはインスタンス生成式の中ではコンストラクタの名前付けに使うことはできない（16.12 節）

- 型パラメタたちはスーパークラスまたはスーパーインターフェイスとして使えない（10.9、10.10、11.1 節）
- 型パラメタは総称型としては使えない。

これらの規範的なバージョンは本仕様書の然るべき節のなかで与えられている。これらの制限の幾つかは将来外される可能性がある。

15.メタデータ(MetaData)

Dart はユーザが定義したアノテーションをプログラム構造に付加する為に使われるメタデータに対応している。

metadata（メタデータ）:
('@' qualified（修飾された） ('.' identifier（識別子）)? (arguments（引数たち）)?)*
;

メタデータは一連のアノテーションたちで構成され、その各々は文字@で始まり、そのあとにある識別子で始まる常数式が続く。もしその式が以下のどれかでないときはコンパイル時エラーである:

- コンパイル時常数変数への参照
- 常数コンストラクタの呼び出し

メタデータはプログラム構成要素(program construct) *p* の抽象文法ツリーに関連付けられ、*p* がそれ自身またデータまたはコメントで無いとして、その直後にメタデータが続く。メタデータは、そのアノテートされたプログラム構成要素 *p* がリフレクションによってアクセス可能だとして、リフレクション的呼び出し(reflective call)を介して実行時に復元され得る。

明らかに、メタデータはまたしかるべきインタープリタを介してそのプログラムを解析し定数たちを計算することで静的にできる。実際殆どで無いにしてもメタデータの多くの使用法は完全に静的である。

実際に使われていないメタデータの導入によって実行時のオーバヘッドが重くなることは無いということは重要である。メタデータは定数たちのみが関与するので、それが計算される時間は重要でなく、実装物は通常の解析と実行時にはメタデータをスキップして、その計算を後回しにしても良い。

メタデータをローカル変数のように、リフレクションを介してアクセス可能で無いかもしれない定数たちとメタデータを結び付けることは可能である（将来考えうることではあるが、よりリッチなリフレクション的ライブラリもそれらへのアクセスを提供できる可能性がある）。これは一見有用で無いようであるがそうではなく、ソース・コードが取得できれば静的にデータが復元できる。

メタデータはライブラリ、part ヘッダ、クラス、typedef、型パラメタ、コンストラクタ、ファクトリ、関数、パラメタ、あるいは変数宣言の前に、及びインポートまたはエクスポート指令の後に置かれ得る。

16.式(Expressions)

*式(expression)*は Dart コードの一部であり、ある値(*value*:常にオブジェクトである)を引き出す為に実行時に計算(evaluate:評価とも訳す)される。各式はそれに関わる(associated)ある静的な型(static type)(19.1 節)を持つ。各値はそれに関わるある動的な型(dynamic type)(19.2 節)を持つ。

expression(式):
assignableExpression(代入可能式) assignmentOperator(代入演算子) expression(式)
| conditionalExpression(条件式) cascadeSection*(カスケード区間)
| throwExpression(throw 式)
;

expressionWithoutCascade(カスケードなしの式):
assignableExpression(代入可能式) assignmentOperator(代入演算子) expressionWithoutCascade(カスケードなしの式)
| conditionalExpression(条件式)
| throwExpressionWithoutCascade(カスケードなしの throw 式)
;

expressionList(式リスト):
expression(式) (',' expression(式))* ;

primary(プライマリ):
thisExpression(this 式)
| **super** assignableSelector(代入可能セレクタ)
| functionExpression(関数式)
| literal(リテラル)
| identifier(識別子)
| newExpression(new 式)
| **new** type '#' ('.' identifier)?
| constantObjectExpression(定数オブジェクト式)
| '(' expression(式) ')'
;

式 *e* は常に括弧で囲まれ得るが、これは決して *e* に対し意味的効果を持たない。

残念ながらこれはそれを囲む式に対し効果を与えうる。static なメソッド *m* => 42 を持つあるクラス *C* があったとすると、*C*:*m*()は 42 を返すが、(*C*):*m*()は NoSuchMethodError を発生させる。この異常は型 Type の各インスタンスが確実にその static なメンバたちに対応したインスタンス・メンバを持つようにすれば修正できよう。この問題は Dart の将来の版で対処されよう。

16.0.1 オブジェクトの同一性(Object Identity)

Dart であらかじめ定義されている **identical**()という関数は以下の場合に限り **identical**(c_1, c_2)だと定義されている:

- c_1 が **null** または **bool** のインスタンスと計算され $c_1 == c_2$ である。

または

- c_1 と c_2 が **int** のインスタンスであり、$c_1 == c_2$ である。

または

- c_1 と c_2 が定数文字列であり、$c_1 == c_2$ である。

または

- c_1 と c_2 が **double** のインスタンスであって、以下の一つがなりたつ:
  - c_1 と c_2 が非ゼロで $c_1 == c_2$
  - c_1 と c_2 がともに+0.0 を表現している
  - c_1 と c_2 がともに－0.0 を表現している
  - c_1 と c_2 がともに同じ下位層のビット・パタンを持った NaN の値を表現している

または

- c_1 と c_2 がリテラル・リスト式の仕様（16.7 節）の中で identical だと定義されている定数リストである。

または

- c_1 と c_2 がリテラル・マップ式の仕様（16.8 節）の中で identical だと定義されている定数マップである

または

- c_1 と c_2 が同じクラス C のオブジェクトであって、c_1 の各メンバが c_2 のそれに対応したフィールドと同じである。

または

- c_1 と c_2 が同じオブジェクトである

double に対する同一性の定義は NaN がそれ自身と等しい、そして符号に関わらずゼロはゼロと等しいという同一性とは異なる。

double に対する同一性の定義は IEEE 754 に規定されており、それでは NaN は柔軟性則に従わないと断定している。ハードウエアがこれらの規則に準拠しているので、効率性の理由からこれらに対応する必要がある。

この同一性の定義は同じようには制約されない。その代わりビット並びが同じ double は同一だと想定している。

同一性のこれらの規則は Dart のプログラマたちがブール値または数値がボックスされているかボックスされていないかを見ることを出来なくしている。

16.1 定数(Constants)

*定数式(constant expression)*はその値が変わることが無く、コンパイル時に完全に計算(evaluate)できる式である。

定数式は以下のもののどれかである:

- リテラル数値(literal number)(16.3 節)
- リテラル・ブール値(literal boolean)(16.4 節)
- 文字列内挿入式(16.5.1 節)が数値、文字列、ブール値、または **null** として計算されるコンパイル時定数であるようなリテラル文字列(16.5 節)。*挿入された値がコンパイル時定数となる文字列内挿入を認めたいと思われるだろう。しかしながらその為には定数オブジェクトのための **toString()** メソッドが走らねばならず、これは専横なコードを含むことになる。*
- リテラル・シンボル(literal symbol)(16.6 節)
- **null**(16.2 節)
- static な定数変数(第 8 章)への正規参照(qualified reference)。
  例えば、もしクラス C が定数 static 変数 v を宣言しているときは C.v は定数である。もし C が前置詞 p を介してアクセスされているときは同じことが言え、p.C.v は定数である。
- 定数変数を示す識別子式
- クラスまたは型エイリアスを示す単純または修飾識別子。例えば、もし *C* がクラスまたは **typedef** なら *C* は常数であり、また *C* がプレフィックス *p* でインポートされているときは *p.C* は常数である。
- 定数コンストラクタ呼び出し(16.12.2 節)
- 定数リスト・リテラル(16.7 節)
- 定数マップ・リテラル(16.8 節)
- トップ・レベル関数(第 9 章)または static メソッド(10.7 節)を示すシンプルなまたは修飾された識別子
- 括弧で括った式(*e*)、ここに *e* は定数式
- **identical**(e_1, e_2)の形式の式で、ここに e_1 と e_2 は定数式でまた(16.0.1 節で示したように)**identical()** は 2 つの引数が同じオブジェクトであるときに限り **true** を返すあらかじめ定められた Dart 関数に静的にバインドされている
- e_1 ==e_2,または e_1 != e_2, の形式のひとつの式で、ここで e_1 と e_2 は数値、文字列、ブール値、または **null** を計算結果とする定数式
- !*e*, e_1 && e_2, または e_1 || e_2 形式のひとつの式で、ここに *e*, e_1 と e_2 はブール値を計算結果とする定数式
- ~ *e*, e_1 ^ e_2, e_1 & e_2, e_1 | e_2, e_1 >> e_2 または e_1 << *e2* の形式のひとつの式で、ここで e_1 と e_2 は整数値、または **null** を計算結果とする定数式
- e_1 + e_2 形式の式で、ここで e_1 と e_2 は数値、文字列、ブール値、または **null** を計算結果とする定数式
- -*e*, e_1 + e_2, e_1 - e_2, e_1 * e_2, e_1 / e_2, e_1 ~/ e_2, e_1 >e_2, e_1 < e_2, e_1 >= e_2, e_1 <= e_2 または e_1 % e_2 の形式のひとつの式で、ここに e_1 と e_2 は数値または **null** を計算結果とする定数式

- e_1,e_2及びe_3が定数式で、e_1がブール値として計算される$e_1?e_2:e_3$の形式の式
- e.**length** の形式の式で、ここで e は文字列の値として計算される定数式。

ある式が定数式である必要があるにもかかわらずその計算に於いて例外を生起するときはコンパイル時エラーである:

各定数式の計算が正しく行われることという要求はないことに注意のこと。ある定数式が必要な場合にのみ（例えば定数変数を初期化するため、あるいは仮パラメタのデフォルト値として、あるいはメタデータとして）コンパイル時にある定数式が実際に成功裏に計算されることを我々はこだわるべきだろうか。

上記はプログラムの制御フローには依存していない。あるプログラム内でその計算が失敗するようなコンパイル時定数がたまたま存在するのはエラーである。これはまた再帰性(recursively)を持つ：複合定数は定数たちで構成されるので、ある定数の副部(subpart)が計算中に例外を生起するなら、それはエラーである。

一方、実装においてはあとでコードをコンパイルすることは自由であるので、一部のコンパイル時エラーたちはかなり遅れて起き得る。

```
const x = 1/0;
final y = 1/0;
class K {
  m1(){
    var z = false;
    if (z) {return x;}
    else {return 2;}
  }
 m2() {
   if (true) {return y;}
   else {return 3;}
 }
}
```

実装ではxに対するコンパイル・エラーを直ちに出すことは自由であるが、そうすることは要求されていない。xを参照する宣言を即座にコンパイルしないのなら、エラーの生起を遅らせることができる。例えば、xに対するコンパイル・エラーを出すのをm1の最初の呼び出しまで延期することが出来よう。しかしながら、m1の実行を選択できない、それはxを参照するブランチを通らず2を成功裏に返すことで判る。

m2呼び出しの場合はこの状況は異なる。yはコンパイル時定数（その値はそうであっても）でないので、m2のコンパイルでコンパイル時エラーを出す必要はない。実装はそのコードを実行でき、それはyに対するゲッタを呼び出させる。その時点で、yの初期化が起きねばならず、それはそのイニシャライザのコンパイルを必要とし、それがコンパイル・エラーを生起させる。

__null__ の取り扱いは何らかの議論に値する。__null__ + 2 を考えてみよう。この式は常にエラーを生起させる。我々はそれを定数式として扱わない(そして一般的には、null を数値の副式(subexpression)、あるいはブール定数式として許さない)ことを選択できただろう。それを含めるには2 つの主張がある:

1. *これは定数である。我々はそれをコンパイル時に計算できる。*
2. *その計算に由来するエラーに明確性を与えるのがより有用と思われる。*

コンパイル時定数式の値がそれ自身に依存しているときはコンパイル時エラーである。

例として、以下のコード片を考えよう:

```
class CircularConsts{
// 違反したコード - 相互に再帰するコンパイル時定数
  static const i = j; // コンパイル時定数
  static const j = i; // コンパイル時定数
}
```

リテラル(literals)は以下のもので構成される:

```
literal(リテラル):
    nullLiteral(nullリテラル)
  | booleanLiteral(ブール値リテラル)
  | numericLiteral(数値リテラル)
  | stringLiteral(文字列リテラル)
  | mapLiteral(マップ・リテラル)
  | listLiteral(リスト・リテラル)
  ;
```

16.2 ヌル(Null)

予約語の **null** は *null オブジェクト*を意味する。

```
nullLiteral(nullリテラル):
    null
;
```

null オブジェクトは組み込みクラス **Null** の唯一のインスタンスである。**Null** をインスタンス化しようとすると実行時エラーを引き起こす。あるクラスやインターフェイスが **Null** を継承または実装しようとするとコンパイル時エラーとなる。**null** 上のメソッドを呼び出すと、そのメソッドがクラス **Null** で明示的に実装されているものでないと **NullPointerException** が起きる。

null の静的型は bottom 型（訳者注: ⊥: 値を持たない空の型）である。

Null *の代わりに bottom を使うという決定により、***null** *は静的チェッカが苦情を出すことなく何所でも代入出来るようにしている。*

16.3 数(Numbers)

*数リテラル(numeric literals)*はサイズが固定されていない(arbitrary size)10 進または 16 進の整数、または倍精度の 10 進数である。

```
numericLiteral（数リテラル）:
    NUMBER
  | HEX_NUMBER
  ;

NUMBER（数）:
    DIGIT+ ('.' DIGIT+)? EXPONENT?
  | '.' DIGIT+ EXPONENT?
  ;

EXPONENT（指数部）:
    ('e' | 'E') ('+' | '-')? DIGIT+
  ;


HEX_NUMBER（16 進数）:
    '0x' HEX_DIGIT+
  | '0X' HEX_DIGIT+
  ;

HEX_DIGIT（16 進桁）:
    'a'..'f'
  | 'A'..'F'
  | DIGIT
  ;
```

数リテラルがプレフィックス‘0x’または‘0X’で始まるときは、それは 16 進整数リテラルで、‘0x’（以下‘0X’もおなじ）に続くリテラルの部分により表現された 16 進整数を意味する。そうでないときは、もしその数リテラルが小数点を含んでいないときは 10 進整数リテラルであることを意味し、10 進整数を意味する。そうでないときは、その数リテラルは IEEE 754 標準で規定された 64 ビット倍精度浮動小数点数を意味する。

原則として Dart 実装が対応している整数の域（range）は無制限である。実際これは使えるメモリに

よって制限される。実装によっては他の事項によって制限を受け得る。

例えば、実装によってはJavaScriptへの変換を効率化するためにこの域に制限を設けることも可能である。これらの制限は技術的に可能になれば直ちに緩和されるべきである。

あるクラスまたはインターフェイスが **int** を拡張または実装しようとするのはコンパイル時エラーである。あるクラスまたはインターフェイスが **double** を拡張または実装しようとするのはコンパイル時エラーである。**int** と **double** 以外の型が **num** を拡張または実装しようとするのはコンパイル時エラーである。

*整数リテラル(integer literal)*は16進整数リテラルか10進整数リテラルかのいずれかである。ある整数リテラル上でゲッタ **runtimeType** を呼び出すと式 **int** が値である **Type** オブジェクトが返される。整数リテラルの静的型はintである。

*倍精度リテラル(literal double)*は整数リテラルでない数リテラルである。ある倍精度リテラル上でゲッタ **runtimeType** を呼び出すと式 **double** が値である **Type** オブジェクトが返される。*倍精度リテラル(literal double)*の静的型は **double** である。

16.4 ブール値(Booleans)

予約語の **true** と **false** は各々ブール値の真と偽を表現するオブジェクトを意味する。これらは*ブール値リテラル(boolean literal)*である。

```
booleanLiteral（ブール値リテラル）:
    true
  | false
  ;
```

true と **false** の双方とも組込みインターフェイス **bool** を実装している。あるクラスまたはインターフェイスが **bool** を拡張または実装しようとするのはコンパイル時エラーである。

これは2つのブール値リテラルがboolのただ2つのインスタンスであることによる。

あるブール値リテラル上でゲッタ **runtimeType** を呼び出すと式 **bool** が値である **Type** オブジェクトが返される。ブール値リテラルの静的型は **bool** である。

16.4.1 ブール変換(Boolean Conversion)

*ブール変換(boolean conversion)*は以下に定めるように何らかのオブジェクト *o* をブール値にマップする:

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

ブール変換は制御－フロー構成(control-flow constructs)とブール式の一部として使われる。理想的には、人は制御－フロー決定はまさしくブール値そのものに基づくべきだと主張しよう。これは静的に型づけられた設定ではそのとうりである。ダイナミックな型づけの言語では、それにはダイナミックなチェックが必要である。洗練された仮想マシンはそれに関わる負担を最小化出来る。残念ながらDart はJavascript にコンパイルされねばならない。ブール変換はこれを効率的にさせてくれる。

*同時に、この定式化がJavascript とは劇的に異なっていて、Javascript では殆どの数とオブジェクトは**true** と解釈(interpreted)される。Dart のアプローチでは**if** (a-b) ... ;といった使用法を許さない。何故ならこれは非**null** のオブジェクトまたは非ゼロの数を**true** として取り扱う低レベルの規約と合致しないからである。実際、ブール変換を介して非ブールのオブジェクトからtrue を引き出す手段がなく、従ってこの種の低レベルの不正侵入は早いうちに芽をつんである。*

*Dart はまたJavascript にあるオートボクシング(autoboxing)とブール変換間の相互関与によって生じ得る不思議な振る舞いを避けている。ひどい事例では**false** が**true** と解釈され得る。Javascript では、ブール値はオブジェクトではなく、その代り「必要に応じ」オブジェクトにオートボックスされる。もし**false** があるオブジェクトにオートボックスされたら、そのオブジェクトは**true** にさせてしまうことが出来る（それが非**null** のオブジェクトとして）。*

ブール変換ではそのパラメタがブール値でなければならないので、ブール変換を使用する構文は、もし変換される値がブール値でないときは、チェック・モードに於いて動的型エラーを引き起こす。

16.5 文字列 (Strings)

*文字列(string)*はUTF-16 のコード単位(code units)の並びである。

この決定はウェブ・ブラウザ及びJavascript との互換性の為になされている。本仕様書の初期の版では有効なユニコード(unicode)のコード位置(code points)の並びであることが要求されていた。プログラマたちはこの区別に依存すべきではない。

```
stringLiteral（文字列リテラル）:
    (MULTI_LINE_STRING（複行文字列）
  | SINGLE_LINE_STRING（単行文字列）)+
  ;
```

文字列は単行文字列または複行文字列のどちらかになれる。

SINGLE_LINE_STRING（単行文字列）:

```
    ' " ' STRING_CONTENT_DQ* ' " '
  | ' ' ' STRING_CONTENT_SQ* ' ' '
  | 'r' ' ' ' (~( ' ' ' | NEWLINE ))* ' ' '
  | 'r' ' " ' (~( ' " ' | NEWLINE ))* ' " '
  ;
```

単行文字列はソース・コードの 1 行以上にわたれない。単行文字列は相互に対応したシングル・クオート（' ' '）またはダブル・クオート（' " '）で終端される。

従って'abc'と"abc"はともに合法な文字列である。同じく'He said "To be or not to be" did he not?'と"He said 'To be or not to be' didn't he?"もともに合法な文字列である。しかしながら"This'は無効な文字列だし、'this"もそうである。

本文法により、複数行にわたる文字列内挿入された式を含んでいるとき以外は、単行文字列が確実にソース・コードの 1 行を超えないことが確保できる。

隣接した単行文字列は単一の文字リテラルを構成するよう暗示的に連結される。

以下はその例である:

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

Dart ではまた文字列連結の為の+演算子を認めている。

String に対する演算子+は String の引数が要求される。その引数が文字列であることを強制してはいない。これにより次のようなパズル的コードを回避させる:

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

この場合は'A simple sum: 2+ 2 = 4'ではなくて、'A simple sum: 2 + 2 = 22' と印刷される。そうではなくて以下のように文字列内挿入を使うことが推奨される:

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

文字列内挿入は殆どの場合良く機能する。完全な満足を満たさない主たる状況は、1 行に収めるには長すぎる文字列の場合である。複行文字列は有用であるが、ある場合では、我々はコードを整列して見える形にしたい。これはホワイト・スペースで区切ったもっと小さな文字列を書くことで次のように表現できるようになる:

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

MULTI_LINE_STRING（複行文字列）:'

```
"""  stringContentTDQ* """"
  | """ stringContentTSQ* """
   | 'r' """"  (~("""))* """"
   | 'r' """ (~("))* """
  ;
```

ESCAPE_SEQUENCE（エスケープ・シーケンス）:

```
    '\'
  | '\r'
  | '\f'
  | '\b'
 | '\t'
 | '\v'
 | '\x' HEX_DIGIT（16 進桁） HEX_DIGIT
 | '\u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
 | '\u{' HEX_DIGIT_SEQUENCE '}'
 :
```

HEX_DIGIT_SEQUENCE（16 進桁シーケンス）:

```
    HEX_DIGIT HEX_DIGIT? HEX_DIGIT? HEX_DIGIT? HEX_DIGIT?
HEX_DIGIT? HEX_DIGIT?
   ;
```

複数行文字列は相互に対応したシングル・クオート 3 つまたは相互に対応したダブル・クオート 3 つによって終端される。

もしある複行文字列の最初の行がプロダクション WHITESPACE（20.1 節）で定義されたホワイトスペース文字のみで構成されているときは（\でプレフィックスされていることもあり）、その行はその終りにある改行(newlines)を含めて無視される。

これはタブ、スペース、および改行として定義されているホワイトスペースを無視するという発想である。これらは直接的に表現できようが、殆どの文字にとってバックスラッシュをプレフィックスすることと同じであるので、我々もその様式を許している。

文字列は特別な文字の為のエスケープ・シーケンスに対応している。これらのエスケープは：

- \n は改行(newline)を示し、\x0A と等しい
- \r はキャリッジ・リターンを示し、\x0D と等しい
- \f はフォーム・フィードを示し、\x0C と等しい
- \b f はバックスペースを示し、\x08 と等しい
- \t はタブを示し、\x09 と等しい
- \v は垂直タブを示し、\x0B と等しい

- \x*HEX_DIGIT*$_1$ *HEX_DIGIT*$_2$、は\u{ *HEX_DIGIT*$_1$ *HEX_DIGIT*$_2$}と等しい
- \u*HEX_DIGIT*$_1$ *HEX_DIGIT*$_2$ *HEX_DIGIT*$_3$ *HEX_DIGIT4*、は \u{ *HEX_DIGIT*$_1$ *HEX_DIGIT*$_2$ *HEX_DIGIT*$_3$ *HEX_DIGIT*$_4$}と等しい
- \u{*HEX_DIGIT_SEQUENCE*} は *HEX_DIGIT_SEQUENCE* で表現されるユニコードのスカラ値である。*HEX_DIGIT_SEQUENCE* の値が有効なユニコード・スカラ値でないときはランタイム時エラーである
- $ は文字列内挿入（interpolate）された式(interpolated expression)の始まりであることを意味する
- それ以外は、*k* はその文字が{*n, r, f, b, t, v, x, u*}のなかに無いどの文字 *k* であることを示す

どの文字列も'r'をその先頭にプレフィックスが付けられ、これは*生の文字列(raw string)*であることを示し、この場合はエスケープと文字列内挿入は認識されない。

もし生で無い文字列リテラルが 2 つの 16 進桁を伴っていない\x 形式の文字列シーケンスを含むときはコンパイル時エラーである。もし生で無い文字列リテラルが 4 つの 16 進桁または中括弧で区切られた 16 進桁のシーケンスを伴っていない\u 形式の文字列シーケンスを含むときはコンパイル時エラーである。

stringContentDQ（ダブル・クオート文字列コンテント）:
```
    ~( '\\' | '  "  ' | '$' | NEWLINE )
  | '\\' ~( NEWLINE )
  | STRING_INTERPOLATION
  ;
```

stringContentSQ（シングル・クオート文字列コンテント）:
```
    ~( '\\' | '\'' | '$' | NEWLINE )
  | '\\' ~( NEWLINE )
  | STRING_INTERPOLATION
  ;
```

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

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

NEWLINE（改行）:

```
  \n
| \r
;
```

総ての文字列リテラルは組込みインターフェイスの **String** を実装している。あるクラスまたはインターフェイスが **String** を拡張または実装しようとするのはコンパイル時エラーである。ある文字列値リテラル上でゲッタ **runtimeType** を呼び出すと式 **String** が値である **Type** オブジェクトが返される。文字列リテラルの静的な型は **String** である。

16.5.1 文字列内挿入(String Interpolation)

それらの式が計算され、結果の値が文字列に変換され、それを包んでいる文字列に連結するように、式を文字列リテラルに埋め込むことが可能である。このプロセスは*文字列補完(または文字列インターポレーション)(string interpolation)*として知られるものである。

```
STRING_INTERPOLATION(文字列内挿入):
    '$' IDENTIFIER_NO_DOLLAR($を含まない識別子)
  | '$' '{' Expression(式) '}'
  ;
```

読者は内挿入内の式自身が文字列を含むことが出来、即ち再帰的に再度内挿入出来ることに気が付くであろう。

文字列内のエスケープしない$文字が文字列内挿入された式の始まりの印となる。$印のあとに以下のどれかが続く:

- $文字を含んではいけない識別子 *id*
- 中括弧で終端された式

$id の形式は**${id}**の形式と等価である。文字列内挿入された文字列's_1**${***e***}**$s_2$'は'$s_1$' + *e*. **toString**() + 's_2'と等価である。同様に+が文字列連結演算子だとすれば、文字列内挿入された文字列"s_1**${***e***}**$s_2$"は"$s_1$" + *e*. **toString**() + "s_2"と等価である。

16.6 シンボル(Symbols)

*シンボル・リテラル(symbol literal)*は Dart のプログラムのなかでのある宣言の名前を示す。

```
symbolLiteral(シンボル・リテラル):
    `#' (operator(演算子) | (identifier(識別子) ('.' identifier)*))
;
```

id がアンダスコア('_')では始まらないシンボル・リテラル**#id** は、式 **const** Symbol('id')と等価である。

シンボル・リテラル**#id** は **mirror.getPrivateSymbol**('id')呼び出しで返されるであろうオブジェクトを計算する。ここに mirror はライブラリ dart:mirrors のなかで定義されているクラス LibraryMirror のインスタンスであって、現在のライブラリを反射(reflect)している。

リテラルのシンボルを導入する動機は何かと思われるかもしれない。一部の言語に於いてはシンボルは正規化(canonicalize)されていて一方文字列はそうなっていない。しかしながら Dart ではリテラルの文字列は既に正規化されている。シンボルは文字列に比べて片づけが少しばかり簡単であり、それらの使用を不思議なくらい病みつきにさせるが、それはこの言語にリテラルの様式を付加することの正当化としては不十分である。一番の動機はリフレクションと縮小化(minification)として知られるウェブ固有の習慣に関係している。

縮小化はダウンロードのサイズを小さくする為にあるプログラムにわたって一貫したかたちで識別子たちを圧縮する。この慣習により、文字列を介してプログラムの宣言たちを参照する反射プログラム(reflective programs)を困難なものにしてしまう。文字列はそのソース・コードのなかのある識別子を参照するが、その識別子は縮小化されたコードのなかではもはや使われておらず、これらの変形を使った反射コードは動かなくなる。従って、Dart のリフレクションでは文字列ではなくて型 ***Symbol*** *のオブジェクトたちを使用している。* ***Symbol*** *のインスタンスたちは縮小化を繰り返しても安定であることが保証される。シンボルたちの為のリテラル形式を用意することで、反射型コードが読みやすくまた書きやすくなる。シンボルは片づけが容易であり、列挙たち(enums)の便利な置き換えとしても機能するという事実は第二の利点である。*

シンボル・リテラルの静的型は **Symbol** である。

16.7 リスト(Lists)

*リスト・リテラル(list literal)*は整数のインデックスが付けられたオブジェクトたちの集まりであるリストを意味する。

listLiteral(リスト・リテラル):
    const? typeArguments(型引数)? '[' (expressionList(式リスト) ','?)? ']'
    ;

リストはゼロまたはそれ以上のオブジェクトを持てる。あるリストの要素数はそのサイズ(size)である。リストはそれに結び付けられたインデックスたち(indices)のセットを持つ。空のリストは空のインデックスたちのセットを持つ。非空のリストはインデックスのセット{0 … n -1}をもち、ここに n はそのリストのサイズである。そのインデックスたちのセットのメンバでないインデックスを使ってあるリストをアクセスしようとするのは実行時エラーである。

あるリストが予約語である **const** で始まるときは、それは *定数リスト・リテラル(constant list literal)*であり、それはコンパイル時定数(16.1 節)であり、従ってそれはコンパイル時に計算される。そうでない

ときはそれは*実行時リスト・リテラル(runtime list literal)*で実行時に計算される。実行時リスト・リテラルのみが生成された後でも可変でありえる。定数リスト・リテラルを可変化しようとすれば動的エラーをもたらす。

ある定数リスト・リテラルがコンパイル時定数でないときはコンパイル時エラーである。ある定数リスト・リテラルの型引数が型変数を含んでいるときはコンパイル時エラーである。

型変数のバインドはコンパイル時には判っていない、従って我々はコンパイル時定数の内部に型変数を使用することはできない。

定数リスト・リテラル **const** <*E*>[e_1... e_n]の値は、組込みインターフェイスである *List*<*E*>を実装したオブジェクトである。i 番目の要素は v_{i+1} であり、ここに v_i はコンパイル時の式 e_i の値である。定数リスト・リテラル **const** [e_1... e_n]の値は、定数リスト・リテラル **const** <**dynamic**>[e_1... e_n]の値として定められる。

$list_1$ = **const** <*V*>[e_{11}... e_{1n}]と $list_2$ = **const** <*U*>[e_{21}... e_{2n}]が 2 つの定数リスト・リテラルで、$list_1$ と $list_2$ の要素が各々 o_{11}... o_{1n} 及び o_{21}... o_{2n} として計算されとする。もし 1 <= *i* <= *n* に対し **identical**(o_{1i}, o_{2i}) で *U*=*V* のときは、**identical**($list_1$, $list_2$)である。

言い換えると定数リスト・リテラルは正規化され(canonicalized)ている。

実行時リテラル<*E*>[e_1... e_n]は以下のように計算される:

- 最初に式たち e_1... e_n が左から右の順に計算され、結果としてオブジェクトたち o_1... o_n が得られる
- 組込みインターフェイス *List*<*E*>を実装したサイズ *n* の新規インスタンス(7.6.1 節) *a* が割り当てられる
- *i* を最初の引数に、o_{i+1}, 0 <= i <= n を第 2 の引数として *a* にたいし演算子[] = が呼ばれる
- その計算の結果は *a* である

本仕様ではどの要素がセットされるかの順番を規定していないことに注意のこと。これによりある実装がそれを望めばこのリストへの並列代入ができる。この順番はチェック・モードでのみ見られる：もし要素 *i* がこのリストの要素型の副型(subtype)でないときは、*a*[*i*]が o_{i-1} に割り当てられるとき動的型エラーが発生する。

実行時リスト・リテラル[e_1... e_n]は<**dynamic**>[e_1... e_n]として計算される。

リスト・リテラルのネストを排除する制約は無い。上記に従えば<*List*<*int*>>[[1, 2, 3] [4, 5, 6]]は型パラメタ・リスト *List*<*int*>をもったリストで、型パラメタ **dynamic** を持った 2 つのリストを含んでいる。

const <*E*>[e_1... e_n]の形式または<*E*>[e_1... e_n]の形式のリスト・リテラルの静的型は *List*<*E*>である。
const [e_1... e_n]の形式または[e_1... e_n]の形式のリスト・リテラルの静的型は *List*<**Dynamic**>である。

リスト・リテラルの型はその要素の型にもとづいて計算されるだろうと仮定しがちである。しかしなが

ら、可変リスト(mutable lists)ではこれは保証されない可能性がある。定数リストであってさえも、これの振る舞いは問題があることを我々は発見している。しばしばコンパイル時は実際は実行時なので、そのランタイムのシステムは妥当に正確な型を決めるのに複雑な最小上界(least upper bound)の計算が出来なければならなくなる。このタスクはIDE（統合開発環境）のなかのツールに任せたほうがベターである。型たちが指定されていないときはそれは未知の型 ***dynamic*** *であると考えるとこだわったほうがよりずっと一様化されている（そして従って予測可能で理解可能である）。*

16.8 マップ(Maps)

*マップ・リテラル(map literal)*はマップのオブジェクトを示す。

mapLiteral（マップ・リテラル）:
const? typeArguments（型引数）? '{' (mapLiteralEntry（マップ・リテラル・エントリ） (',' mapLiteralEntry（マップ・リテラル・エントリ）)* ','?)? '}'
;

mapLiteralEntry（マップ・リテラル・エントリ）:
expression（式） ':' expression（式）
;

マップ・リテラルはゼロまたはそれ以上のエントリ(entries)で構成される。各エントリはキー*(key)*と、*値(value)*を持つ。各キーと各値は式で示される。

マップ・リテラルが予約語 **const** で始まるときは、それは定数マップ・リテラル（*constant map literal*）で、それはコンパイル時定数（16.1 節）であり、従ってコンパイル時に計算される。そうでないときはそれは*実行時マップ・リテラル(run-time map literal)*で、それは実行時に計算される。実行時マップ・リテラルのみが生成された後でも可変でありえる。定数マップ・リテラルを可変化しようとすれば動的エラーをもたらす。

定数マップ・リテラルのなかのあるエントリのキーまたは値のどれかがコンパイル時定数でないときはコンパイル時エラーである。定数マップ・リテラルのなかのあるエントリのキーまたは値のどれかが、そのキーが文字列、整数、リテラル・シンボル、またはクラス Symbol の定数コンストラクタを呼び出した結果でない場合に限り、演算子==を実装したあるクラスのインスタンスであるときはコンパイル時エラーである。定数マップ・リテラルの型引数が型パラメタを含んでいるときはコンパイル時エラーである。

定数マップ・リテラル **const** <*K, V*>{k_1:e_1... k_n :e_n}の値は、組込みインターフェイス *Map*<*K, V*>を実装したオブジェクト *m* である。*m* のエントリたちは u_i:v_i, 1 <= *i* <= *n* で、ここに u_i はコンパイル時式 k_1 の値で、v_i はコンパイル時式 e_1 の値である。定数マップ・リテラル **const** {k_1:e_1... k_n :e_n}は定数マップ・リテラル **const** <**dynamic**, **dynamic**>{k_1:e_1... k_n :e_n}の値として定められる。

map_1 = **const** <*K, V*>{k_{11}:e_{11}... k_{1n} :e_{1n}}そして map_2 = **const** <*J, U*>{k_{21}:e_{21}... k_{2n} :$e2_n$}が 2 つの定数

マップ・リテラルだとする。map_1とmap_2のキーが各々 $s_{11}... s_{1n}$ と $s_{21}... s_{2n}$と計算されるとし、map_1とmap_2の要素たちが各々 $o_{11}... o_{1n}$と $o_{21}... o_{2n}$だと計算されるとする。もし 1 <= i <= n に対し **identical**(o_{1i}, o_{2i})そして **identical**(s_{1i} , s_{2i})でかつ $V = U$とすれば、**identical**(map_1 , map_2)である。

言い換えれば、定数マップ・リテラルは正規化され(canonicalized)ている。

実行時マップ・リテラル<*K, V*>{k_1:e_1... k_n :e_n}は次のように計算される:

- 最初に式たち $e_1... e_n$が左から右の順に計算され、結果としてオブジェクトたち $o_1... o_n$が得られる
- 組込みインターフェイス *Map*<*K, V*>を実装した新規インスタンス(10.6.1 節) m が割り当てられる
- 1 <= i <= n に対し　最初の引数 u_iと第 2 の引数 o_iで m 上で演算子[]=が呼び出される
- その計算の結果は m である

実行時マップ・リテラル{k_1:e_1... k_n :e_n}は<**dynamic**, **dynamic**>{k_1:e_1... k_n :e_n}として計算される。

あるマップ・リテラルのなかのすべてのキーがコンパイル時定数であるときに限り、あるマップ・リテラルのなかの任意の 2 つのキーの値が等しいときは静的警告となる。

マップ・リテラルは順序付けされている:キーたち間及び/あるいは値たち間の繰り返しは常にソース・コード内で出現するキーの順序で生じる。

無論あるキーが繰り返すときは、その順序は最初に起きたもので規定されるが、値は最後に起きたもので定められる。

const <*String, V*>{k_1:e_1... k_n :e_n}の形式のまたは<*K, V*>{k_1:e_1... k_n :e_n}の形式のマップ・リテラルの静的な型は *Map*<*K, V*>である。**const** {k_1:e_1... k_n :e_n}の形式のまたは{k_1:e_1... k_n :e_n}の形式のマップ・リテラルの静的な型は *Map*<**dynamic, dynamic**>である。

16.9 スロー(Throw)

*スロー式(throw expression)*は例外の生起のために使われる。

throwExpression(スロー式):
　throw expression?
　;
throwExpressionWithoutCascade(カスケードなしスロー式):
　throw expressionWithoutCascade?
　;

*現行例外(current exception)*とは最新のスローされた未処理例外(last unhandled exception thrown)

のことである。

throw *e* ; の形式のスロー式の計算は以下のように進行する:

その式が計算され値 v が得られる。

式 e が特別の例外またはエラーのオブジェクトの型として計算されるという要求はされていない。

もし *e* が **null**（16.2 節）と計算されれば、次に **NullThrownException** がスローされる。そうでないときは、現行例外に *v* がセットされ、制御は直近の包含する*例外ハンドラ(exception handler)*（17.12 節）に渡される。

式 e が特殊の例外または例外オブジェクトとして計算されるという要求は無い。

それらが現行関数を抜けるための相互に排他的なオプションを表現しているため、連行例外と現行戻り値は決して同時に定義されてはならない。

f をただちに包含している関数だとしよう。

もし *f* が同期関数（第 9 章）のときは、制御は直近の動的に包含している例外ハンドラに渡される。

もし *f* が sync*とマークされているときは、動的に包含している例外ハンドラは、該 throw 式の計算を開始させた moveNext()への呼び出しを包含する。

もし *f* が非同期関数のときは、現行の活性化で導入された動的に包含している例外ハンドラ *h*（17.11 節）が存在すれば、制御は *h* に移され、そうでないときは *f* は完了する。

スローされた例外がどこで処理されるかは必然的に同期の場合か非同期の場合かによって異なってくる。非同期関数はそれの外部で定義された例外ハンドラに制御を渡すことはできない。

非同期の発生器は例外をそれ自身のストリームにポストする。他の非同期関数はその future を介して例外を報告する。

もしスローされているオブジェクトがクラス Error のオブジェクトあるいはそれのサブクラスであるときは、その stackTrace ゲッタはそのオブジェクトが最初にスローされた点での現時点でのスタック・トレースを返す。

スロー式の型は bottom 型（訳者注: ⊥: 値を持たない空の型）である。

16.10 関数式(Function Expressions)

*関数リテラル(function literal)*はコードのある実行可能な単位をカプセル化するひとつのオブジェク

トである。

functionExpression（関数式）:
formalParameterList（仮パラメタ・リスト） functionExpressionBody（関数式ボディ）
;

functionExpressionBody（関数式ボディ）:
'=>' expression（式）
| block（ブロック）
;

関数リテラルは組込みインターフェイスである **Function** を実装している。

$(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1} = d_1, \ldots, T_{n+k}\ x_{n+k} = d_k])$ => e の形式の関数リテラルの静的な型は$(T_1, \ldots, Tn, [T_{n+1}\ x_{n+1}, .., T_{n+k}\ x_{n+k}]) \rightarrow T_0$であり、ここに T_0 は e の静的な型である。

$(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1} = d_1, \ldots, T_{n+k}\ x_{n+k} = d_k])$ **async** => e の形式の関数リテラルの静的な型は$(T_1, \ldots, Tn, [T_{n+1}\ x_{n+1}, .., T_{n+k}\ x_{n+k}]) \rightarrow Future<flatten(T_0)>$であり、ここに T_0 は e の静的な型であり、*flatten(T)*は以下のように定義される:

もし $T = Future < S >$ ならば $flatten(T) = flatten(S)$。
そうでないときは、もし $T <: Future$ ならば、S を $T << Future < S >$のような型だとし、そして総ての R に対しもし $T << Future < R >$なら、$S << R$。

これにより $Future < S >$が T のスーパー型である *Future* の最も特有なインスタンス化であることが確保される。

そうすると、$flatten(T) = S$。
それ以外の状況では、$flatten(T) = T$。

我々は future たちの複数の層をひとつに折りたたんでいる。もし e が future f と計算されれば、該 future は f が非 future の値で完了するまでその **then()** *コールバックを呼ばず、従ってある await の結果は決して future にならず、async 関数の結果は決して* **Future**< *X* >*の型を有さない。ここに X 自身は Future 呼び出しである。*

これに対する例外は、Future を実装したまたは継承した型 X となる。この場合は、唯一つのラップ解放が起きる。どうしてそうなるかと例として以下のものを考えよう:
class *C<T>* ***implements*** *Future<C<C<T>>>* . . .
ここでは、flatten のネーティブな定義が拡大されており、固定点でもない。

より洗練された flatten の定義も可能だが、既存のルールは有用である。

$(T_1\ a_1, \ldots, T_n\ a_n, \{T_{n+1}\ x_{n+1} : d_1, \ldots, T_{n+k} x_{n+k} : d_k\})$ => e の形式の関数リテラルの静的な型は$(T_1, \ldots, T_1 \ldots T_n, \{T_{n+1}\ x_{n+1}, .., T_{n+k}\ x_{n+k}\}) \rightarrow T_0$であり、ここに T_0 は e の静的な型である。

(T_1 a_1, …, T_n a_n, {T_{n+1} x_{n+1} : d_1, … ,T_{n+k}:x_{n+k} : d_k}) **async** > e の形式の関数リテラルの静的な型は(T_1, …, T_1...T_n, {T_{n+1} x_{n+1}, .., T_{n+k} x_{n+k}}) →Future<T_0>であり、ここに T_0 は e の静的な型である。

(T_1 a_1, …, T_n a_n, [T_{n+1} x_{n+1} = d_1, … ,T_{n+k} x_{n+k} = d_k]){s}の形式の関数リテラルの静的な型は(T_1, …, T_n, [T_{n+1} x_{n+1}, .., T_{n+k} x_{n+k}]) →**Dynamic** である。

(T_1 a_1, …, T_n a_n, [T_{n+1} x_{n+1} = d_1, … ,T_{n+k} x_{n+k} = d_k]) **async**{s}の形式の関数リテラルの静的な型は(T_1, …, T_n, [T_{n+1} x_{n+1}, .., T_{n+k} x_{n+k}]) →**Future** である。

(T_1 a_1, …, T_n a_n, [T_{n+1} x_{n+1} = d_1, … ,T_{n+k} x_{n+k} = d_k]) **async***{s}の形式の関数リテラルの静的な型は (T_1, …, T_n, [T_{n+1} x_{n+1}, .., T_{n+k} x_{n+k}]) →*Stream* である。

(T_1 a_1, …, T_n a_n, [T_{n+1} x_{n+1} = d_1, … ,T_{n+k} x_{n+k} = d_k]) **sync***{s}の形式の関数リテラルの静的な型は(T_1, …, T_n, [T_{n+1} x_{n+1}, .., T_{n+k} x_{n+k}]) →*Iterable* である。

(T_1 a_1, …, T_n a_n, {T_{n+1} x_{n+1} : d_1, … ,T_{n+k}:x_{n+k} : d_k}) **async** {s}の形式の関数リテラルの静的な型は(T_1, …, T_1...T_n, {T_{n+1} x_{n+1}, .., T_{n+k} x_{n+k}}) →*Future* である。

(T_1 a_1, …, T_n a_n, {T_{n+1} x_{n+1} : d_1, … ,T_{n+k}:x_{n+k} : d_k}) **async*** {s}の形式の関数リテラルの静的な型は (T_1, …, T_1...T_n, {T_{n+1} x_{n+1}, .., T_{n+k} x_{n+k}}) →*Stream* である。

(T_1 a_1, …, T_n a_n, {T_{n+1} x_{n+1} : d_1, … ,T_{n+k}:x_{n+k} : d_k}) **sync*** {s}の形式の関数リテラルの静的な型は(T_1, …, T_1...T_n, {T_{n+1} x_{n+1}, .., T_{n+k} x_{n+k}}) →*Iterable* である。

上記の総ての場合に於いて 1 <= i <= n である T_i が指定されていないどの場合おいても、それは **dynamic** として指定されているものと見做される。

16.11 This

予約語の **this** は現在のインスタンス・メンバ呼び出しのターゲットであることを意味する。

thisExpression（this 式）:
this
;

this の静的な型は直前に包含しているクラスのインターフェイスである。

我々は現時点では self 型に対応していない。

this がトップ・レベルの関数または変数のイニシャライザ内、ファクトリ・コンストラクタ内、または静的メソッドまたは変数イニシャライザ内、またはあるインスタンス変数のイニシャライザの中に出てくるときはコンパイル時エラーである。

16.12 インスタンス生成(Instance Creation)

インスタンス生成式たちはインスタンスを作る為のコンストラクタたちを呼び出す。

以下様式のいずれかのインスタンス生成式の中の型 *T* が奇形(malformed: 19.2 節)または奇形バインド(malbounded: 19.8 節)のときは静的型警告となる:

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

下記様式のいずれかのインスタンス生成式の中の型 *T* が列挙型(第 13 章)のときはコンパイル時エラーである:

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

16.12.1 New

*new 式(new expression)*はコンストラクタ(10.6 節)を呼び出す。

newExpression(new 式):
new type(型) ('.' identifier(識別子))? arguments(引数たち)
;

e が **new** $T.id(a_1, .., a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$または **new** $T(a_1, .., a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式の new 式だったとする。

もし T が現行スコープ内でアクセス可能なクラスまたはパラメタ化された型のときは次に:

- *e* が **new** $T.id(a_1, .., a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式のとき、*T.id* が型 *T* によって宣言されたコンストラクタの名前で無いときは静的警告となる。*e* が **new** $T(a_1, .., a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式のとき、型 *T* が *T* の宣言と同じ名前を持ったコンストラクタを宣言していないときは静的警告となる。

T がパラメタ化された型(parameterized type)(19.8 節) $S<U_1, ,.., U_m>$だとし、$R = S$ だとする。もし T がパラメタ化された型でないときは、$R = T$ としよう。さらに、もし *e* が *e* が **new** $T.id(a_1, .., a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式のときは、次に *q* はコンストラクタ *T.id* とし、そうでないときは *q* はコンストラクタ *T* としよう。

もし *R* が *l*=*m* 個の型パラメタたちをもった総称体のときは、次に。

- *T* はパラメタ化された型でない、次に 1 <= *i* <= *l* にたいし V_i = **dynamic** だとしよう。
- もし *T* がパラメタ化された型のときは、1 <= i <= *m* にたいし $V_i = U_i$ だとしよう。

もし R が *l* != *m* 個の型パラメタを持った総称体の時は、次に 1 <= *i* <= *l* にたいし V_i = **dynamic** だとしよう。どのケースにおいても、1 <= i <= *m* にたいし $V_i = U_i$ だとしよう。

e の計算は次のように進行する:

最初に、引数リスト(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})が計算される。

次に、もし *q* が抽象クラスの非ファクトリ・コンストラクタのときは、**AbstractClassInstantiationError** がスローされる。

もし *T* が奇形のときは、あるいはもし *T* が型変数の時は、動的エラーが発生する。チェック・モードに於いては、もし *T* またはそのスーパークラスのどれかが奇形バインドのときは、動的エラーが発生する。そうでないときは、もし *q* が定義されていないかアクセス出来ないときは、**NoSuchMethodError** がスローされる。

もし *q* が n 個の位置的パラメタより少ないあるいは n 個の要求されているパラメタよりも多いパラメタを持っているとき、あるいはもし q においてキーワード・パラメタたち $\{x_{n+1},...,x_{n+k}\}$ のどれかが欠けているときは **NoSuchMethodError** がスローされる。

そうでないとき、*q* が生成的コンストラクタ(10.6.1 節)の場合は次に:

この時点で我々は実際の型引数たちの数が仮型パラメタたちの数と合致していることが確認されることの注されたい。

クラス R の新規インスタンス(10.6.1 節)、*i* が割り当てられる。*i* の各インスタンス変数 *f* に対し、もし *f* の変数宣言がイニシャライザ式 e_f を持っているときは、実型引数 $V_1,...,V_l$ に対する *R* バウンドの型パラメタたちで(もしあれば)、が計算され、オブジェクト o_f が得られ、*f* はその o_f にバインドされる。それ以外のときは *f* は **null** にバインドされる。

this は e_f のスコープ内で無いことに注意。従ってこの初期化はインスタンス化されているオブジェクトの他の属性に依存できない。

次に、*i* に対する **this** バウンド、実際の型引数 V_1, ..., V_m にバインドされた *C* の型パラメタたち(もし有れば)、及び引数リストの計算から得られた仮パラメタのバインディングたちで、*q* が計算される。*e* の計算の結果は *i* である。*e* の計算の結果は *i* である。

そうでないとき、*q* はファクトリ・コンストラクタ(10.6.2 節)であり、そのときは:

Ti を *R* の型パラメタたち（もしあれば）だとし、*Bi* を *Ti, 1 <= i <= m* のバウンドたちだとしよう。チェック・モードにおいては、もし *Vi* が *[V1, ..., Vm/W1, ..., Wm]Bi, 1 <= i <= m* の副型でないときは動的型エラーである。

もし *q* が $T(p_1, ..., p_{n+k}) = c;$ または $T.id(p_1, ..., p_{n+k}) = c;$ の形式のリダイレクト・ファクトリ・コンストラクタであるなら、そのときは *e* の計算結果は式 $[V_1, ..., V_m/T_1, ..., T_m]$**new** $c(a_1, ..., a_n, x_{n+1}: a_{n+1}, ..., x_{n+k}: a_{n+k}))$の計算結果と等価である。

そうでないときは、引数リストの計算から得られたバインドたち、及び実際の型引数たち $V_1, ..., V_m$ にバインドされた *q* の型パラメタたち（もしあれば）に関し *q* の本体が計算され、オブジェクト *i* が結果として得られる。*e* の計算結果は *i* となる。

もし *q* がある抽象クラスのコンストラクタで *q* がファクトリ・コンストラクタで無いときは静的警告となる。

上記は抽象クラスのインスタンス化が警告をもたらすという考えに対する正確な意味づけを与えている。同じ条項は次の節の中の定数オブジェクト生成に適用される。

とりわけ、それは有効なインスタンスを作り出すかあるいはそれ自身の宣言の内部での警告となるかのいづれかなので、ファクトリ・コンストラクタは抽象クラス内で宣言可能であり安全に使われる得る。

new $T.id(a_1, .., a_n, x_{n+1} : a_{n+1}, ... x_{n+k}: a_{n+k})$または **new** $T(a_1, .., a_n, x_{n+1} : a_{n+1}, ... x_{n+k}: a_{n+k})$の形式のどれかの *new* 式の静的型は *T* である。$a_i, 1 <= i <= n+ k$ の静的型がコンストラクタ *T.id (respectively T)*の対応する仮パラメタの型に代入出来ないかもしれないときは静的警告となる。

16.12.2 Const

*定数オブジェクト式(constant object expression)*は定数コンストラクタ（10.6.3 項）を呼び出す。

> **constObjectExpression（定数オブジェクト式）:**
> **const** type（型） ('.' identifier（識別子）)? arguments（引数たち）
> ;

e が **const** $T.id(a_1, .., a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式または **const** $T(a_1, .., a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式の定数オブジェクト式だとする。もし *T* が現在のスコープ内でアクセス可能なクラスを示していないときはコンパイル時エラーである。

もし T が後回しの型（19.1 節）のときはコンパイル時エラーである。

特に、*T*は型変数であってはならない。

もし T がパラメタ化された型のときは、もし T がその型引数たちの中に型変数を含んでいるときはコンパイル時エラーである。

e が **const** *T.id*(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})の形式のとき、*T.id* が型 *T* で宣言された定数コンストラクタの名前でないときはコンパイル時エラーである。*e* が **const** *T*(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k}) の形式のとき、型 *T* が *T* の宣言と同じ名前を持った定数コンストラクタを宣言していないときはコンパイル時エラーである。

上記のケース総てで、a_i, 1 <= *i* <= *n*+ *k* がコンパイル時定数式でないときはコンパイル時エラーである。

e の計算は以下のように進む:

最初に、もし *e* が **const** *T.id*(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})の形式のときは、*i* が式 **new** *T.id*(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})の値だとする。そうでないときは *e* は **const** *T.id*(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})の形式でなければならず、この場合 *i* は **new** *T.id*(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})を計算した結果であるとする。そうすると:

- もしそのプログラムの実行中に、ある定数オブジェクト式が既に型引数たち V_i ,1 <= *i* <= *m* を持ったクラス *C* のインスタンス *j* として計算されていたとする、そうすると:
  - *i* の各インスタンス変数 *f* に対し、v_{if} が *i* のなかの *f* の値だとし、v_{jf} が *j* のなかのフィールド *f* の値だとする。もし *i* のなかの総てのフィールド *f* に対し **idencical**(v_{if} , v_{jf})なら、*e* の値は *j* であり、そうでなければ e の値は *i* である。
- そうでないときは e の値は *i* である。

言い換えると、定数オブジェクトは正規化され(canonicalized)ている。そのオブジェクトが実際に new かどうかを判断するには、それを計算しなければならない；次にそれをキャッシュされているどのインスタンスたちと比較できる。もし等しいオブジェクトがキャッシュの中に存在するなら、我々は新しく生成したオブジェクトを廃棄し、キャッシュされているものを使用する。それらが同一のフィールドと同一の型引数を持っていればオブジェクトたちは等しい。コンストラクタはどの副作用も誘発できないので、コンストラクタの実行は観測不能である。コンストラクタは呼び出しサイト(call site)あたり一回、コンパイル時に実行される必要がある。

const *T.id*(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})の形式または **const** *T*(a_1, .., a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})の形式のどれかの定数オブジェクト式の静的な型は *T* である。a_i, 1 <= *i* <= *n*+ *k* の静的型がコンストラクタ *T.id*（各 *T*）の対応する仮パラメタ型に割り当てられない可能性があるときは静的警告となる。

ある定数オブジェクトの計算によりキャッチされていない例外をスローされる結果になるときはコンパイル時エラーである。

そのような状況がどうして起きるかを調べるために、以下の例を考えよう:

```
class A {
  final var x;
  const A(var p): p = x * 10;
}
```

```
const A("x"); // コンパイル時エラー
const A(5); // 適法

class IntPair {
  const IntPair(this.x, this.y);
  final int x;
  final int y;
  operator *(v) => new IntPair(x*v, y*v);
}

const A(const IntPair(1, 2)); // コンパイル時エラー：微妙に違反
```

定数コンストラクタに関わる規則により、引数 "x"を持ったコンストラクタ A()あるいは const IntPair(1, 2)は例外を発生させ得、その為コンパイル時エラーが起きる。

const q(a1, .., an)の形式のインスタンス生成式に対しては、q がある抽象クラス(10.4 節)のコンストラクタであるが q はファクトリ・コンストラクタで無いときは静的警告となる。

16.13 アイソレートの産み付け(Spawning an Isolate)

アイソレートの産み付けは文法的には通常のライブラリ呼び出しであり、dart:isolate のなかで定義された spawnUri()または spawnFunction()関数のひとつを呼び出すことで達成される。しかしながら、そのような呼び出しはそれ自身のメモリと制御スレッドをもった新しいアイソレートを生成するという意味論的効果を持つ。

アイソレートのメモリは、そのスレッドの呼び出しスタックに使えるメモリであるので、有限である。アイソレートの実行でそのメモリまたはスタックを使い切り、有効にキャッチ出来ないかもしれない実行時エラーをもたらし、そのアイソレートが停まってしまうことはありえる。

第 7 章で論じたように、停止したアイソレートの取り扱いは組み込み者側の責任になる。

16.14 関数呼び出し(Function Invocation)

関数呼び出しは以下のケースで生じる:関数式(16.10 節)が呼び出された(16.14.4 節)、メソッドが呼び出された(16.17 節)、ゲッタ(16.16、16.18 節)またはセッタ(16.19 節)が呼び出された、またはコンストラクタが呼び出された(インスタンス生成(16.12 節)、コンストラクタ・リダイレクション(10.6.1 節)、または super 初期化のいずれかを介して)。いろんな種類の関数呼び出しが、どのようにその関数が呼び出されるか、*f* が決まるか、また **this**(16.11 節)がバインドされているかどうか、に関して異なってくる。*f* が一旦決まれば *f* の仮パラメタたちは対応した実際の引数たちにバインドされる。*f*

のボディが実行されるときは、前述のバインディングで実行される。

もし*f*が **async** (第 9 章)とマークされているときは、組み込みクラス **Future** を実装した新鮮インスタンス(10.6.1 節) *o* がその呼び出しに結び付けられ、直ちに呼び出し側に返される。次に*f*のボディはある将来時点で実行されるようスケジュールされる。future の *o* は*f*が終了したときに完了する。o を完了させるのに使われる値は、もし定まっている場合はときは現行の戻り値(17.12 節)であり、そうでないときは現行の例外(16.9 節)である。

もし*f*が **async*** (第 9 章)とマークされているときは、組み込みクラス **Stream** を実装した新鮮インスタンス(10.6.1 節) *s* がその呼び出しに結び付けられ、直ちに呼び出し側に返される。*s* がリスンされているときは、*f*のボディの実行が開始される。*f*が終了したときは:

- もし現行の戻り値が定まっておれば次に、もし *s* がキャンセルされてしまっておればそのキャンセルの future は **null** (16.2 節)で完了する。
- もし現行の例外 *x* が定まっておれば:
  - *x* が *s* に付加される
  - もし *s* がキャンセルされてしまっておれば、そのキャンセルの future がエラーとして *x* で完了する
- s が閉じる。

*非同期発生器のストリームがキャンセルされているときは、その発生器の中の **finally** 句(17.11 節)のなかでクリーンアップが生じる。我々はこの時点で生じるどの例外も無くなってしまうのではなくキャンセルの future に振り向けることを選択している。*

もし*f*が非同期の場合は次に、*f*が終了した時点で、*f*内で実行しているどの非同期の for ループ(17.6.3 節)または yield-each 文 (17.16.2 節)に結び付けられたどのオープンなストリーム加入も、それらのネストの順で(最も内側が最初で)、キャンセルされる。

例えばある例外がそれらの中でスローされたときにエスケープした for ループによってそのようなストリームはオープンのままになってしまい得る。

もし*f*が **sync*** (第 9 章)とマークされているときは、次に組み込みクラスの **Iterable** を実装した新鮮インスタンス *i* がその呼び出しに結び付けられ、直ちに返される。

Dart 実装は sync*メソッドで返される Iterable の特定の実装を提供する必要がある。典型的な戦略としては、dart:core で定義されている IterableBase クラスの副クラスのインスタンスを作ることになろう。その場合 Dart 実装で付加される必要がある唯一のメソッドはイタレータである。

この Iterable 実装は Iterable の規約に準拠しなければならず、その規約に対し例外的に有効だと特定されるステップは取ってはいけない。

該規約はある iterable たちが通常よりもより効果的であり得るような一連の状況を示している。例えば、それらの長さをあらかじめ計算することにより。

通常の iterable たちはそれらの長さを決めるのにそれらの要素たち上で繰り返しを行わねばならない。これは、各要素がある関数で計算されるような同期発生器の場合に確かにそうである。例えば、発生器の結果をあらかじめ計算しておきそれらをキャッシュするのは受け入れられないだろう。

その iterable 上での繰り返しが開始したときに、その iterable からイテレータ *j* を取得し、その上で **moveNext()** を呼ぶことで、*f* のボディの実行が開始される。*f* が終了したら、*j* は最後の要素の後の位置を指しているので、その現行値は **null** であり、*j* 上での **moveNext()** の呼び出しは総ての更なる呼び出し同様 **false** を返す。

各イテレータは別の計算を開始させる。もし **sync***関数が純粋でないときは各イテレータによってもたらされる値たちの順は異なりえる。

ある与えられた iterable から一つ以上のイタレータを導入できる。iterable 自身上での操作は別々のイテレータたちを生成し得ることに注意。一例として長さがそうであり得る。異なったイテレータたちが異なった長さのシーケンスをもたらし得ることは想定しうる。ある Iterator クラスを書く際と同じように **sync***関数を書く際にも同じ注意が必要である。特に、複数の同時のイテレータたちを丁寧にとり扱わねばならない。もし該イテレータが変化し得る外部状態に依存する際は各結果算出後にの該状態がまだ有効かをチェック（そしてそうでないときは ConcurrentModificationError をスローする）すべきである。

各イテレータはそれ自身の総てのローカル変数たちのシャロー・コピーで走る；特に各イテレータは、例えそれらのバインディングたちが該関数により変更されているとしても、同じ初期引数たちを有する。

あるイテレータの 2 つの実行は該関数の外部の状態を介してのみ関わりあう。

もし *f* が同期で、発生器(第 9 章)でないときは、次に f のボディの実行は即座に開始される。f が終了したときは呼び出し側に現行の戻り値が返される。

ボディの実行は以下のことが最初に起きた時点で終了する；

- キャッチされていない例外(uncaught exception)がスローされる
- f のボディのなかで直ちにネストした(immediately nested) return 行（17.12 節）が実行され、**finally** 句（17.11 節）の中では解釈されない。
- そのボディの最後の行が実行を完了させる

16.14.1 実引数リスト計算(Actual Argument List Evaluation)

関数呼び出しには、その関数への実際の引数たちのリストの計算と、その結果たちの関数の仮パラメタたちに対するバインディング、が関わる。

arguments（引数）:

'(' argumentList（引数リスト）? ')'
;

argumentList（引数リスト）:
namedArgument（名前付き引数） (',' namedArgument（名前付き引数）)*
| expressionList（式リスト） (',' namedArgument（名前付き引数）)*
;

namedArgument（名前付き引数）:
label（ラベル） expression（式）
;

(a_1 .. a_m, q_1: a_{m+1}, …, q_l: a_{m+l})の形式の実引数リストの計算は以下のように進行する:

引数たち a_1, …, a_{m+l}はそのプログラムで出現した順に計算され、オブジェクトたち o_1 .. o_{m+l}を得る。

簡潔に述べると、m 個の位置的引数と l 個の名前付き引数で構成される引数リストは左から右に計算される。

16.14.2 実引数たちの仮パラメタたちへのバインド(Binding Actuals to Formals)

その関数を*f*、*f*の位置的パラメタたちを p_1, …, p_n、また p_{n+1}, …, p_{n+k}を*f*で宣言された名前つきパラメタだとする。

(a_1 .. a_m, q_1: a_{m+1}, …, q_l: a_{m+l})の形式の実引数リストから得られた計算された実引数リスト(o_1, …, o_{m+l})が以下のように*f*の仮パラメタたちにバインドされる:

ここでも、我々は *m* 個の位置的引数たちと *l* 個の名前つきのパラメタたちを持つ。我々は *n* 個の必要なパラメタたちと *k* 個の名前つきパラメタたちの関数を持つ。位置的引数の数は少なくとも最大必要なパラメタたちの数で無ければならない。総ての名前つき引数たちは対応した名前つきパラメタを持たねばならない。位置的と名前つきの引数の双方と同じパラメタを用意しなくても良い。あるオプショナルなパラメタがそれに対応した引数をもっていないときは、それはデフォルトの値をとる。チェック・モードにおいては、総ての引数たちはそれらに対応した仮引数の型の副型に属さねばならない。

もし *l* > *0* なら、その場合は必然的に *n* = *h* のケースである。何故ならメソッドはオプショナルな位置的パラメタたちと名前付きパラメタたちの双方を持てないからである。

もし $m < h$ または $m > n$ なら、**NoSuchMethodError** がスローされる。更に、各 q_i, $1 <= i <= l$はセット{p_{m+1}, …, p_{m+k}}のなかに対応した名前つきパラメタを持っていなければならず、でないと **NoSuchMethodError** がスローされる。次に p_iが o_i, $1 <= i <= m$ の値にバインドされ、q_jが o_{m+j}, $1 <= j <= l$の値にバインドされる。総ての*f*の残りの仮パラメタたちはそれらのデフォルト値にバインド

される。

これらの残ったパラメタたちの総てが必然的にオプショナルであり、従ってデフォルト値をとる。

チェック・モードでは、もし o_i が null でなく p_i の実型（19.8.1 節）が o_i, 1 <= i <= m の型の副型でないときは動的型エラーとなる。もし、チェック・モードにおいて、o_{m+j} が null でなく q_j の実型（19.8.1 節）が o_{m+j}, 1 <= j <= l の型の副型でないときは動的型エラーとなる。

任意の i != j に対し $q_i = q_j$ のときはコンパイル時エラーである。

T_i を a_i の静的型だとし、S_i を p_i 1 <= i <= $n+k$ の型だとし、S_q を f の名前付きパラメタ q の型だとしよう。もし T_j が S_j, 1 <= j <= m に代入出来ない場合は静的な警告である。もし $m < n$ あるいは $m > n + k$ のときは静的な警告である。更に、各 q_i, 1 <= i <= l はセット{p_{m+1}, ..., p_{n+k}}のメンバでなければならず、そうでなければ静的警告が起きる。T_j が S_r、ここに $r = j-m$, $m+1$ <= j <= $m+l$、に代入出来ない場合は静的警告になる。

16.14.3 無修飾呼び出し（Unqualified Invocation）

無修飾関数呼び出し i は $id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式をとり、ここに id は識別子である。

id という名前が付けられた構文的に可視な宣言が存在したとして、f_{id} がそのような宣言の最も奥だとする。そうすると:

- もし f_{id} がローカル関数、ライブラリ関数、ライブラリまたは静的ゲッタまたは変数なら、i は関数式呼び出し（16.15.4 節）だと解釈される。
- そうでないとき、もし f_{id} がそれを包含しているクラス C の静的メソッドときは、i は静的メソッド呼び出し $C.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$と等しい。
- そうでないときは、f_{id} は通常のメソッド呼び出し **this**.$id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$と等しいと考えられる。

そうでないとき、もし i がトップ・レベルまたは静的関数（関数、メソッド、ゲッタ、あるいはセッタ）または変数イニシャライザの内部で生じるときは、i の計算は **NoSuchMethodError** のスローを起こさせる。

もし i がトップ・レベルまたは静的関数の内部で起きないときは、i は通常のメソッド呼び出し **this**.$id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$と等しい。

16.14.4 関数式呼び出し(Function Expression Invocation)

関数式呼び出し *i* は $e_f(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式をとり、ここに e_fは式である。e_fが識別子 *id* のときは、*id* は上記のとおり必然的にローカル関数、ライブラリ関数、ライブラリまたは静的ゲッタまたは変数を意味しなければならず、あるいは *i* は関数式呼び出しとは看做されない。もし e_f が属性アクセス式（16.14 節）のときは、*i* は通常のメソッド呼び出し（16.17.1 節）として扱われる。

a:b(x)は関数呼び出し(a:b)(x)が続いたゲッタ b の呼び出しとしてではなく、オブジェクト a 上でのメソッド b()の呼び出しとして解釈される。もしメソッドまたはゲッタ b が存在すればこの二つは等価である。しかしながら、もし b が a 上で定義されていないときは、結果としての noSuchMethod()呼び出しは異なってくる。noSuchMethod()にわたされたこの呼び出しは前者の場合は引数 x でメソッド b を呼び出すことになり、後者の場合は（引数なしで）ゲッタ b の呼び出しとなる。

そうでないときは:

関数式呼び出し $e_f(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の計算は通常のメソッド呼び出し e_f *call*$(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$と等価である。

この定義、及びメソッド call()がからむ他の定義たちの意味合いは、それらが call()メソッドを定義しているときに限りユーザ定義の型たちが関数値として使えるということである。メソッド call()はこの点に関し特別である。call()メソッドのシグネチュアが組み込み呼び出し文法を介してそのオブジェクトを使う際に適正なシグネチュアを決める。

e_fの静的な型がある関数型に割り当てられない可能性があるときは静的警告である。もし *F* が関数型でないときは、*i* の静的型は **dynamic** である。そうでないときは *i* の静的型は *F* の宣言された戻り型である。

16.15 検索(Lookop)

16.15.1 メソッド検索(Method Lookup)

ライブラリ *L* に於いてオブジェクト *o* のなかのメソッド *m* の検索の結果は、ライブラリ *L* におけるクラス *C* のなかのメソッド *m* の検索の結果である。ここに *C* は *o* のクラスである。

ライブラリ *L* に於いてクラス *C* のなかのメソッド *m* の検索の結果は、もし *C* が *L* でアクセス可能な *m* という名前の具体インスタンス・メソッドを宣言しているときは、そのメソッドが該検索の結果である。そうでないときは、もし *C* が *S* のスーパクラスであるなら、この検索の結果は *L* に関して *S* のなかの *m* の検索の結果である。そうでないときはわれわれはこのメソッド検索は失敗したという。

検索中に抽象メンバたちをスキップするということにした動機は、よりスムースなミクスイン構成を許すということに大きく依るものである。

16.15.2 ゲッタとセッタの検索 Getter and Setter Lookup()

ライブラリ *L* に於いてオブジェクト *o* のなかのゲッタ（またはセッタ）*m* の検索の結果は、ライブラリ *L* におけるクラス *C* のなかのゲッタ（またはセッタ）*m* の検索の結果である。ここに *C* は *o* のクラスである

ライブラリ *L* に於いてクラス *C* のなかのゲッタ（またはセッタ）*m* の検索の結果は、もし *C* が *L* でアクセス可能な *m* という名前の具体インスタンス・ゲッタ（またはセッタ）を宣言しているときは、そのゲッタ（またはセッタ）が該検索の結果である。そうでないときは、もし *C* が *S* のスーパクラスであるなら、この検索の結果は *L* に関して *S* のなかの *m* の検索の結果である。そうでないときはわれわれはこの検索は失敗したという。

検索中に抽象メンバたちをスキップするということにした動機は、よりスムースなミクスイン構成を許すということに大きく依るものである。

16.16 トップ・レベル・ゲッタ呼び出し(Top Level Getter Incocation)

m の形式のトップ・レベル・ゲッタ呼び出し *i* の計算は以下のように進行する:ここに *m* は識別子である。

ゲッタ関数 m が呼び出される。i の値はこのゲッタ関数の呼び出しで返された結果である。この呼び出しは常に定まっていることに注意。識別子参照の規則に従い、識別子は該ゲッタが定義されてい無い限りトップ・レベルのゲッタ呼びだしとしては扱われない。

i の静的型は m の宣言された戻り型である。

16.17 メソッド呼び出し(Method Invocation)

メソッド呼び出しは以下に規定されるように幾つかの形式をとり得る。

16.17.1 通常呼び出し(Ordinary Invocation)

通常メソッド呼び出し*条件つき(conditional)*と*無条件(unconditional)*であり得る。

o?.*m*(a_1, …, a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})の形式の*条件付き通常メソッド呼び出し(conditional ordinary method invorcation) e* の計算は、次の式の計算と等価である:
((*x*) => *x* == **null**?**null** : *x*.*m*(a_1, . . . , a_n, x_{n+1} : a_{n+1}, . . . , x_{n+k} : a_{n+k}))(*o*).

e の静的型は *o.m*(*a*1, . . . , *an*, *xn*+1 :*an*+1, . . . , *xn*+*k*: *an*+*k*)の静的型と同じである。*o.m*(*a*1, . . . , *an*, *xn*+1 : *an*+1, . . . , *xn*+*k* : *an*+*k*)で引き起こされ得るとまさしく同じ静的警告たちがまた *o*?.*m*(*a*1, . . . , *an*, *xn*+1 : *an*+1, . . . , *xn*+*k* : *an*+*k*)の場合にも発生させられる。

無条件通常メソッド呼びだし *i* は *o.m*(*a*1, . . . , *an*, *xn*+1 : *an*+1, . . . , *xn*+*k* : *an*+*k*)の形式となる。

o.m(*a*1, . . . , *an*, *xn*+1 : *an*+1, . . . , *xn*+*k*: *an*+*k*)の形式の無条件通常メソッド呼び出しの計算は以下のように進む:

最初に、式 *o* が値 v_o として計算される。次に、引数リスト(a_1, …, a_n, x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k})が計算され、実際のオブジェクト o_1, …, o_{n+k} が得られる。*f* が現在のライブラリ *L* に対する v_o のなかのメソッド *m* の検索(16.15．1 項)結果だとしよう。

p_1 ... p_h が *f* の要求パラメタたちだとし、p_1 ... p_m が *f* の位置的パラメタたちだとし、p_{h+1} ... p_{h+l} が *f* で宣言されているオプショナルなパラメタたちだとしよう。

n 個の位置的引数たちと k 個の名前付き引数たちを持つことになる。我々は h 個の要求パラメタたちと l 個のオプショナルなパラメタたちを持つことになる。位置的引数たちの数は少なくとも最大要求パラメタたちの数でなければならず、位置的パラメタたちの数よりも大きくなってはいけない。総ての名前付き引数たちはそれに対応した名前付きのパラメタを持っていなければならない。

もし $n < h$ または $n > m$ のときは、このメソッド検索は失敗する。更に、$n+1 <= i <= n+k$ の各 x_i はセット{p_{m+1} ... p_{h+l}}のなかの対応した名前付きパラメタを持っていなければならず、そうでないとこのメソッド検索は失敗する。そうでないときはメソッド検索は成功する。

もし v_0 が **Type** のインスタンスであるが *o* が定数型リテラルでないときは、もし *m* が static メソッドに転送する(9.1 節)メソッドの場合は、メソッド検索は失敗する。そうでないときはメソッド検索は成功したことになる。

もしそのメソッド検索が成功したら、*f* のボディが引数リストの計算から得られたバインディングたちに対し、そして v_o に対する **this** バウンドで、実行される。*i* の値は *f* が実行されて返される値である。

そのメソッド検索が失敗したら、そのときは *g* を *L* に対する v_o のなかのゲッタ(16.15.2 節) *m* の検索結果だとする。

もし v_0 が **Type** のインスタンスであるが *o* が定数型リテラルでないときは、もし *g* が static ゲッタに転送するゲッタの場合は、ゲッタ検索は失敗する。そうでないときはメソッド検索は成功したことになる。

もしそのゲッタ検索が成功すれば、v_g をそのゲッタ呼び出し *o.m* の値だとする。そうすると *i* の値は引数 *v.g*,[o_1,...,o_n],{x_{n+1}: a_{n+1}, …, x_{n+k}: a_{n+k}}で静的メソッド **Function.apply()**を呼び出した結果の値である。

もしこのゲッタ検索も失敗したら、あらかじめ定義されているクラスの Invocation の以下のような新し

いインスタンスの *im* が生成される:

- **im.isMethod** は **true** と計算される。
- **im.memberName** は‘m’と計算される。
- **im.positionalArguments** は[o_1, …, o_n]と値を含んだ不変リスト(immutable list)として計算さ
- **im.namedArguments** は{x_{n+1}:o_{n+1}, …, x_{n+k} : o_{n+k}}と同じキーと値を持った不変マップ(immutable map)として計算される。

次に *o* の中のメソッド **noSuchMethod**()が検索され引数 *im* で呼び出されこの呼び出しの結果が *i* の計算結果となる。

次に v_oのなかでメソッド **noSuchMethod()**が検索され、引数 *im* で呼びだされ、そしてこの呼び出しの結果は *i* の計算の結果である。

しかしながら、もし該実装が単一の位置的引数で呼び出せないということを見出した時は、クラス **Object** 内の **noSuchMethod()**実装が引数 *im'*で v_o上で呼び出される。ここに *im'*は以下のようである **Invocation** のインスタンスである:

- · im'.**isMethod** が **true** と計算される
- · im'.**memberName** が #**noSuchMethod**’と計算される.
- · im'.**positionalArguments** がその唯一の引数が *im* である不変リストとして計算され
- · im'.**namedArguments** が **const** {}の値として計算される

そして後者の呼び出しの結果は *i* の計算結果である。

noSuchMethod() *を正しくない引数の数でオーバライドすることでそのような状況を発生させることは可能である:*

```
class Perverse { noSuchMethod(x,y) => x + y; }
new Perverse.unknownMethod();
```

この記述は注意深くレシーバ o と引数 a_iの再計算を避けていることに注意。

T が *o* の静的型であるとしよう。もし *T* が以下のどれかでない限り *m* という名前のアクセス可能な(第6.2 節)インスタンス・メンバを持っていないときは静的型警告となる:

- *T* または *T* のスーパーインターフェイスが **dart:core** のなかで定義されている定数**@proxy** と等しい定数を示すアノテーションでアノテートされている。または
- *T* が **Type**、*e* が定数型リテラル、そして *e* に対応するクラスが *m* という名前の static ゲッタを有する。

もし *T.m* が存在すれば、もし *T.m* の型 *F* がある関数型に割り当てられない場合は静的型警告である。もし *T.m* が存在しなければ、あるいはもし *F* が函数型でない場合は、*i* の静的型は **dynamic** であり、そうでない場合は *i* の静的型は *F* の宣言された戻りの型である。

プレフィックス object（18.1 節）上で、あるいは直後にトークン‘.’がついた定数型リテラル上でクラス **Object** のメソッドを呼ぶのはコンパイル時エラーである。

16.17.2 カスケードされた呼び出し(Cascaded Invocations)

*カスケードされたメソッド呼び出し(cascaded method invocation)*は *e..suffix* の形式をとり、ここに *suffix* は演算子、メソッド、ゲッタ、あるいはセッタ呼び出しのシーケンスである。

cascadeSection（カスケード区間）:
'..' (cascadeSelector（カスケード・セレクタ） arguments（引数たち）*)
(assignableSelector（代入可能セレクタ） arguments*)* (assignmentOperator（代入可能演算子） expressionWithoutCascade（カスケードなし式）)?
;
cascadeSelector（カスケード・セレクタ）:
'[' expression '] '
| identifier
;

e..suffix の形式のカスケードされたメソッド呼び出し式は式(t){t.*suffix*; **return** t;}(*e*)と等価である。

*null 対応条件付き代入可能式(null-aware conditional assignable expressions)（16.32 節）の導入に伴い、null 対応条件の書式でカスケードを拡張することも同じく意味があろう。e?..suffix を(t){t?.suffix; **return** t;}(e)なる式と等価と定義できよう。*

現在の仕様書では、シンプルなこと及び急速な言語発展を考えてそのような規約は追加されていない。しかしながら Dart の実装物は第 2 章で記したように、そのような規約を実験しても構わない。

16.17.3 スーパー呼び出し(Super Invocation)

スーパー・メソッド呼び出し *i* は **super**.$m(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式をとる。**super**.$m(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$の形式のスーパー・メソッド呼び出し *i* の計算は以下のように進行する:

最初に、引数リスト$(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$が計算され、実引数オブジェクトたち $o_1, \ldots, o_{n+k}$を得る。*S* をただちに包含しているクラスのスーパークラスだとし、*f* を現在のライブラリ *L* に関する *S* の中の*検索*メソッド（16.17.1 節） *m* の結果だとする。

$p_1 \ldots p_h$ が *f* の要求パラメタたちだとし、$p_1 \ldots p_m$ が *f* の位置的パラメタたちだとし、$p_{h+1} \ldots p_{h+l}$ が *f* で宣言されているオプショナルなパラメタたちだとしよう。

もし $n < h$ または $n > m$ のときは、このメソッド検索は失敗する。更に、$n+1 <= i <= n+k$ の各 x_i はセット$\{p_{m+1} ... p_{h+l}\}$のなかの対応した名前付きパラメタを持っていなければならず、そうでないとこのメソッド検索は失敗する。そうでないときはメソッド検索は成功する。

もしこのメソッド検索が成功すれば、引数リストの計算で得られたバインディングたちに関して、また現在の **this** の値に対する **this** バウンドで、*f* のボディが実行される。*i* の値は *f* が実行されて返される値である。

そのメソッド検索が失敗したら、そのときは *g* を *L* に対する v_o のなかのゲッタ(16.15.2 節) *m* の検索結果だとする。もしそのゲッタ検索が成功すれば、v_g をそのゲッタ呼び出し **super**.*m* の値だとする。そうすると *i* の値は引数 $v.g,[o_1,...,o_n],\{x_{n+1}: a_{n+1}, ..., x_{n+k}: a_{n+k}\}$ で静的メソッド **Function.apply()**を呼び出した結果の値である。

もしこのゲッタ検索も失敗したら、あらかじめ定義されているクラスの Invocation の以下のような新しいインスタンスの *im* が生成される:

- **im.isMethod** は **true** と計算される。
- **im.memberName** はシンボル m と計算される。
- **im.positionalArguments** は$[o_1, \ldots, o_n]$と値を含んだ不変リスト(immutable list)として計算さ
- **im.namedArguments** は$\{x_{n+1}:o_{n+1}, \ldots, x_{n+k} : o_{n+k}\}$と同じキーと値を持った不変マップ(immutable map)として計算される。

次に *S* のなかでメソッド **noSuchMethod()**が検索され、引数 *im* で **this** 上で呼びだされ、そしてこの呼び出しの結果は *i* の計算の結果である。

しかしながら、もし該実装が単一の位置的引数で呼び出せないということを見出した時は、クラス **Object** 内の **noSuchMethod()**実装が引数 *im'* で **this** 上で呼び出される。ここに *im'* は以下のようである **Invocation** のインスタンスである:

- · im'.**isMethod** が **true** と計算される
- · im'.**memberName** が **#noSuchMethod**'と計算される.
- · im'.**positionalArguments** がその唯一の引数が *im* である不変リストとして計算され
- · im'.**namedArguments** が **const** {}の値として計算される

そして後者の呼び出しの結果は *i* の計算結果である。

もしこのスーパー・メソッド呼び出しがトップ・レベルの関数または変数イニシャライザ、クラス Object 内、ファクトリ・コンストラクタ内、インスタンス変数イニシャライザ内、コンストラクタ・イニシャライザ内、あるいは静的メソッドまたは変数イニシャライザ内で生じたときは、コンパイル時エラーである。

もし *S* が *m* という名前のアクセス可能(6.2 節)なインスタンス・メンバを持っていないときは、*S* または *S* のスーパーインターフェイスが **dart:core** のなかで定義されている定数**@proxy** とおなじ定数を示すアノテーションでアノテートされていない限り、静的型エラーである。もし *S.m* が存在するとき、

S.m の型 *F* が関数型に割り当てられない可能性があるときは静的警告となる。もし *S.m* が存在しないとき、あるいは *F* が関数型でないときは、*i* の静的型は **dynamic** である;そうでないときは *i* の静的型は宣言された *F* の戻り型である。

16.17.4 メッセージ送信(Sending Messages)

メッセージがアイソレートたち間の唯一の通信手段である。メッセージは Dart ライブラリのなかにある専用のメソッドたちを呼ぶことで送信される;メッセージ送信の為の固有の文法は無い。

言い換えれば、メッセージ送信をサポートするメソッドたちは通常のコードにアクセスできない Dart のプリミティブたちを具体化する。これはアイソレートたちの産み付けのメソッドたちと良く似ている。

16.18 属性の抽出(Poperty Extraction)

*属性抽出(Property extraction)*によりあるオブジェクトのメンバがそのオブジェクトから簡潔に抽出できる。属性抽出は以下のいずれかで起きえる:

1. *クロージャ化(closurization)*(16.18.2 節)で、あるメソッドまたはコンストラクタをクロージャに変換する。あるいは
2. あるゲッタ・メソッドを呼び出した結果を返す*ゲッタ呼び出し(getter invocation)*。

クロージャ化を介してメンバたちから導入されるクロージャたちは一般的にティアオフ(tear-offs)として知られる。

属性抽出は*条件付き(conditional)*または*無条件(unconditional)*のどちらかになり得る。

x#id 文法を使ったティアオフは現時点では条件付きとはなり得ない;this は一貫しておらず、近い将来多分 x?#id といった表記法を介して対処される可能性が高い。第 2 章で示したように、この領域での実験は許されている。

e_1*?.id* の形式の条件付き*属性抽出式(conditional property extraction expression)* *e* の計算は式((*x*) => *x* == **null**?**null** : *x.id*)(e_1)と等価である。*e* の静的型は e_1*.id* の静的型と同じである。 *T* を e_1 の静的型だとし、*y* を型 *T* の新鮮変数だとしよう。 *y.id* で引き起こされるとまさしく同じ静的警告は e_1*?.id* の場合でもまた発生される。

人は ***e*** **≠ null** に対し ***e*****?.*****v*** は常に ***e.v*** と等価だと結論したくなるかもしれない。しかしながらこれは必ずしもあたらない。もし ***e*** が静的メンバ ***v*** を持った型を表現する型リテラルだとすると、***e.v*** はそのメンバを参照するが、***e*****?.*****v*** はそうではない。

無条件属性抽出は幾つかの文法的書式をとる: *e.m* (16.18.1 節), **super**.*m* (16.18.2 節), *e#m* (16.18.3 節), **new** *T#m* (16.18.4 節), **new** *T#* (16.18.5 節) 及び **super#***m* (16.18.6)、ここに *e* は式であり、*m* は識別子でオプションとして等号符号が付き、*T* は型である。

16.18.1 ゲッタ・アクセスとメソッド抽出(Getter Access and Method Extraction)

e.m の形式の属性抽出 *i* の計算は以下のように進行する:

最初に、式 *e* が計算されてオブジェクト *o* を得る。*f*を現行のライブラリ *L* に対する *o* のなかのメソッド *m* の検索(16.15.1 節)の結果だとしよう。もし o が Type のインスタンスであるが定数型リテラルでないとすると、もし *m* が static メソッドに転送(10.1 節) するメソッドだとすると、メソッド検索は失敗する。もしメソッド検索が成功したら、i はオブジェクト *o* 上でのメソッド *f* のクロージャ化(16.18.7)として計算される。

メソッド検索は抽象メソッドをスキップするので、*f* は決して抽象メソッドではないことに注意。従って、もし m がある抽象メソッドを参照しているなら、我々は次のステップにすすむ。しかしながら、メソッドとゲッタは互いにオーバライドしないので、ゲッタ検索も同じく必然的に失敗し、noSuchMethod()が最終的に呼ばれよう。この残念な意味合いは、このエラーはある抽象メソッドのクロージャ化を試みるのではなく、存在しないゲッタを参照するということである。

そうでないときは、*i* はゲッタ呼び出しである。*f* を *L* に関して *o* のなかのゲッタ関数(10.2 節)*m* が検索(16.15.2 節)された結果だとしよう。もし *o* が **Type** のインスタンスであるが *e* が定数型リテラルでないときは、もし *f* が static ゲッタに転送するゲッタのときは、ゲッタ検索は失敗する。そうでないときは、*f* のボディが *o* への **this** バインドで実行される。*i* の値はこのゲッタ関数の呼び出しで返される結果である。

もしこのゲッタ検索が失敗したときは、あらかじめ定義されているクラス **Invocation** の以下のような新規のインスタンス *im* が生成される:

- · **im.isGetter** が **true** と計算される
- · **im.memberName** はシンボル m と計算される
- · **im.positionalArguments** は **const** []
- · **im.namedArguments** は **const** {}の値として計算される

o のなかでメソッド **noSuchMethod**()が検索され、引数 *im* で呼び出され、この呼び出しの結果が *i* の計算の結果となる。しかしながら、もし該実装が単一の位置的引数で呼び出せないということを見出した時は、クラス **Object** 内の **noSuchMethod()**実装が引数 *im()*で *o* 上で呼び出される。ここに *im()*は以下のようである **Invocation** のインスタンスである:

- · im'.**isMethod** が **true** と計算される

- · im'.**memberName** は#noSuchMethod と計算される
- · im'.**positionalArguments** はその唯一の要素が *im* である不変リストとして計算される
- · im'.**namedArguments** は const fg の値として計算される

そして終わりの呼び出しの結果が *i* の計算の結果である。

m がクラス **Object** のメンバで *e* がプレフィックス **object** (18.1)または定数型リテラルのいずれかのときはコンパイル時エラーである。

これは int.toString は除外されるが、(int).toString はそうではない。何故なら後者の場合は e は括弧でくくられた式であるからである。

T を *e* の静的型であるとしよう。もし *T* が以下のいずれかを除き *m* という名前のゲッタまたはメソッドを持っていないときは静的型警告である:

- *T* または *T* のスーパーインターフェイスが **dart:core** で定義されている定数**@proxy** と同一の定数を示すアノテーションでアノテートされている。または、
- *T* が **Type** で、*e* が定数型リテラルで、*e* に対応するクラスが *m* という名前の static メソッドまたはゲッタを有する

もし i がゲッタ呼び出しなら、i の静的型は:

- もし *T.m* が存在するなら、宣言された *T.m* の戻りの型である
- もし *T* が **Type**、*e* が定数型リテラルで、*e* に対応するクラスが *m* という名前の static メソッドまたはゲッタを持っているときは、*m* の宣言された戻りの型である
- もし *T* がアクセス可能な *m* という名前のインスタンスメソッドのときは関数 *T.m* の静的型である
- もし *T* が **Type** で、e が定数型リテラルで、e に対応したクラスがアクセス可能な *m* という名前の static メソッドを宣言しているときは関数 *m* の静的型である
- それ以外は **dynamic** 型である

16.18.2 スーパー・ゲッタ・アクセスとメソッドのクロージャ化(Super Getter Access and Method Closurization)

super.*m* の形式の属性抽出 *i* の計算は以下のように進行する:

S を直ちに包含しているクラスのスーパークラスだとしよう。*f* を現行ライブラリに関する *S* のなかのメソッド *m* の検索の結果だとしよう。もしメソッド検索が成功したら、*i* はスーパークラス S (16.18.1 節) に関するメソッド *f* のクロージャ化(closurization)として計算される。

そうでないときは、*i* はゲッタ呼び出しである。f を *L* に関して *S* のなかのゲッタ *m* の検索結果だとしよう。*f* のボディは **this** バインドで実行され **this** の現行の値となる。*i* の値はこのゲッタ関数の呼び出しで返される結果である。

もしこのゲッタ検索が失敗したときは、あらかじめ定義されているクラス **Invocation** の以下のような新規のインスタンス *im* が生成される:

- · **im.isGetter** が **true** と計算される
- · **im.memberName** は'm'と計算される
- · **im.positionalArguments** は **const** []の値として計算される
- · **im.namedArguments** は **const** {}の値として計算される

次に *S* のなかでメソッド **noSuchMethod**()が検索され、引数 *im* で呼び出され、この呼び出しの結果が *i* の計算の結果となる。しかしながら、もし該実装が単一の位置的引数で呼び出せないということを見出した時は、クラス **Object** 内の **noSuchMethod()**実装が引数 *im()* で *o* 上で呼び出される。ここに *im()*は以下のようである **Invocation** のインスタンスである:

- · im'.**isMethod** が **true** と計算される
- · im'.**memberName** は noSuchMethod と計算される
- · im'**.positionalArguments** はその唯一の要素が *im* である不変リストとして計算される
- · im'.**namedArguments** は **const** {}の値として計算される

そして終わりの呼び出しの結果が *i* の計算の結果である。

もし *S* がアクセス可能なインスタンス・メソッドまたは *m* という名前のゲッタを有していないときは静的警告である。

i の静的型は:

- もし *S* が *m* という名前のアクセス可能なインスタンス・ゲッタを有しているときは、ゲッタ *S.m* の宣言された戻り値
- もし *S* が *m* という名前のアクセス可能なインスタンス・メソッドを有しているときは、関数 *S.m* の宣言された戻り値
- それ以外は **dynamic**

16.18.3 一般的クロージャ化(General Closurization)

e#m の形式の属性抽出 *i* の計算は以下のように進行する:

最初に、式 *e* がオブジェクト *o* と計算される。次に:

もし *m* がセッタ名なら、を現行ライブラリ *L* に関して *o* のなかのセッタ *m* の検索の結果だとしよう。もし o が Type のインスタンスであるが *e* が定数型リテラルでないときは、次にもし *f* が static なセッタに転送するメソッドのときは、セッタ検索は失敗する。セッタ検索が成功すれば、次に *i* がオブジェクト *o* 上のセッタ *f* のクロージャ化(16.18.7)として計算される。もしセッタ検索が失敗すれば、NoSuchMethodError がスローされる。

これ及び以下に示す類似のケースに於いて noSuchMethod を呼び出すのに Dart の規則に準拠するのがより好ましいかのしれない。しかしながら現行の noSuchMethod の実装ではクロージャ化呼び出しと実際の呼び出しとの区別ができない。Dart の将来のバージョンでは、例えば isTearOff のようなゲッタによる手段で、クロージャ化に対応して noSuchMethod が呼び出されたかどうかを検出するメカニズムを備えることになろう。現段階においてより慎重でありまた失敗にこだわることで、我々はこの機能が導入されたときに機能しているコードが動かなくなることが起きないことを確保している。

もし *m* がセッタ名でないときは、*f* を現行ライブラリ *L* に関して *o* のなかのセッタ *m* の検索の結果だとしよう。もし *o* が Type のインスタンスであるが *e* が定数型リテラルでないときは、次にもし *f* が static なセッタに転送するメソッドのときは、セッタ検索は失敗する。セッタ検索が成功すれば、次に *i* はオブジェクト *o* 上のセッタ *f* のクロージャ化(16.18.7)として計算される。

もしメソッド検索が失敗すれば、f を現行ライブラリ *L* に関して *o* のなかのゲッタ *m* の検索の結果だとしよう。もし *o* が Type のインスタンスであるが *e* が定数型リテラルでないときは、次にもし *f* が static なゲッタに転送するメソッドのときは、ゲッタ検索は失敗する。もしゲッタ検索が成功すれば、*i* はオブジェクト *o* 上のゲッタ *f* のクロージャ化(16.18.7)と計算される。もしゲッタ検索が失敗すれば、NoSuchMethodError がスローされる。

もし *e* が前置オブジェクト(18.1)で *m* が型あるいは Object クラスのメンバを参照しているときはコンパイル時エラーである。

この制約はオブジェクトとしての前置詞使用に関する他の制約たちに沿ったものである。*p#m* の唯一の認められている使用はトップ・レベルのメソッドのクロージャ化と前置子 *p* を介してインポートされたゲッタたちである。トップ・レベルのメソッドたちはその修飾名 *p.m* により直接取得できる。しかしながら、ゲッタ及びセッタたいはそうではなく、それらのクロージャ化を許すのは文法 *p#m* の全論点である。

T を *e* の静的型だとしよう。もし *T* が以下のいずれかも場合を除いてアクセス可能なインスタンス・メソッドまたは *m* という名前のゲッタを有していないときは静的型傾向である:

- *T* または *T* のスーパー・インターフェイスが dart:core で定義された定数 **@proxy** と同一の定数を示すアノテーションでアノテートされている。または
- *T* が **Type**、*e* が定数型リテラルで *e* に対応したクラスが *m* という名前の static なメソッドまたはゲッタを宣言している。

i の静的型は:

- もし *T* が *m* という名前のアクセス可能なインスタンス・メンバを有しているならば関数 *T.m* の静的型。
- もし *T* が **Type** で、*e* が定数型リテラルで、*e* に対応したクラスが *m* という名前の static なメソッドまたはゲッタを宣言しているときは関数 *T.m* の静的型。
- それ以外は型 **dynamic**。

16.18.4 指名コンストラクタ抽出(Named Constructor Extraction)

new *T*#*m* の形式の属性抽出 *i* の計算は以下のように進行する:

もし T が奇形(19.1)のときは動的エラーが発生する。もし *T* が前置詞 *p* をもった後回しの型のときは、次にもし *p* が成功裏にロードされていないときは、動的エラーが発生する。もし *T* があるクラスを示していないときは、動的エラーが発生する。チェックド・モードに於いては、もし *T* またはそのスーパー・クラスたちのどれかが奇形のときは、動的エラーが発生する。そうでないときは、もし型 *T* が *m* という名前のアクセス可能な指名コンストラクタ *f* を宣言していないときは、NoSuchMethodError がスローされる。それ以外のときは *i* は型 *T* のコンストラクタ *f* のクロージャ化(16.18.8)として評価される。

もし *T* が奇形または奇形バウンドのときは、静的警告が常に生じることに注意。

もし *T* が包含スコープ内の *m* という名前のアクセス可能なコンストラクタ関数を持ったあるクラスを示しているときは、*i* の静的型は該コンストラクタ関数の型である。それ以外では *i* の静的型は **dynamic** である。

16.18.5 匿名コンストラクタ抽出(Anonymous Constructor Extraction)

new *T*#の形式の属性抽出 *i* の計算は以下のように進行する:

もし *T* が奇形(19.1)の時は、動的エラーが発生する。もし *T* が前置詞 *p* をもった後回しの型のときは、次にもし *p* が成功裏にロードされていないときは動的エラーが発生する。もし *T* があるクラスを示していないときは、動的エラーが発生する。チェックド・モードに於いては、もし *T* またはそのスーパー・クラスたちのどれかが奇バウンドのときは、動的エラーが発生する。そうでないときは、もし型 *T* が *m* という名前のアクセス可能な指名コンストラクタ *f* を宣言していないときは、NoSuchMethodError がスローされる。それ以外のときは *i* は型 *T* の匿名コンストラクタ *f* のクロージャ化(16.18.9)として評価される。

ここに於いても、もし *T* が奇形または奇形バウンドのときは、既存の規則により静的警告が確実に生じることに注意。このことはまた *x*#（ここに *x* が型でないとする）は常に静的警告が出されることを意味する。

もし *T* が包含スコープ内の匿名コンストラクタ関数 *T*()を持ったあるクラスを示しているときは、*i* の静的型は該コンストラクタ関数 *T*()の型である。それ以外では *i* の静的型は **dynamic** である。

16.18.6 一般的スーパー属性抽出(General Super Property Extraction)

super *T*#の形式の属性抽出 *i* の計算は以下のように進行する:

S を直ちに包含しているクラスのスーパークラスだとしよう。

もし *m* がセッタ名なら、*f* を現行ライブラリ *L* に関して *S* のなかでのセッタ *m* の検索結果だとしよう。もしこのセッタ検索が成功すれば、*i* はスーパークラス *S* に関してのセッタ *f* のクロージャ化(16.18.10)として評価される。もしセッタ検索が失敗すれば、**NoSuchMethodError** がスローされる。

もし m がセッタ名でないときは、f を現行ライブラリ L に関する S のなかのメソッド検索の結果だとしよう。もしメソッド検索が成功すれば i はスーパークラス S に関してのメソッド m のクロージャ化(16.18.10)だと評価される。

そうでないときは、*f* を現行ライブラリ *L* に関する S のなかのゲッタ *m* の検索の結果だとしよう。もしゲッタ検索が成功すれば、*i* はスーパークラス *S* に関してのゲッタ *f* のクロージャ化(16.18.10)だと評価される。もしゲッタ検索が失敗すれば、**NoSuchMethodError** がスローされる。

もし *S* が *m* という名前のアクセス可能なインスタンス・メンバを持っていないときはである静的型警告である。

S が *m* という名前のアクセス可能なインスタンス・メンバを持っているときは、*i* の静的型は関数 *S.m* の静的型である。それ以外は *i* の静的型は **dynamic** である。

16.18.7 通常のメンバー・クロージャ化 (Ordinary Member Closurization)

o をあるオブジェクトだとし、*u* を *o* にバインドされた新鮮な final 変数だとしよう。オブジェクト *o* 上のメソッド *f* のクロージャ化(*closurization of method f on object o*)は以下のものに等しいと定義される:

- もし *f* が *op* という名前で *op* が<, >, <=, >=, ==,-, +, /, ~/, *, %, |, ^, &, <<, >>(これには単項-は除外される)のどれかであるときは(){**return** *u* op *a*;}
- もし *f* が~ と名前がつけられている場合は(){**return** ~ *u*;}
- もし *f* が[] と名前がつけられている場合は(*a*){**return** *u*[*a*];}
- もし *f* が[] = と名前がつけられている場合は(*a, b*){**return** *u*[*a*] = *b*;}
- もし *f* が *m* と名前がつけられ要求パラメタたち $r_1, \ldots, r_n$ 及びデフォルトを $d1, \ldots, dk$ とする指名パラメタたち $p_1, \ldots, p_k$ を持っているときは
  $(r_1, \ldots, r_n, \{p_1 : d_1, \ldots, p_k : d_k\})$ {
  return $u.m(r_1, \ldots, r_n, p_1 : p_1, \ldots, p_k : p_k)$;
  }
- もし *f* が *m* と名前がつけられ要求パラメタたち $r_1, \ldots, r_n$ 及びデフォルトを $d1, \ldots, dk$ とするオプショナルな場所的パラメタたち $p_1, \ldots, p_k$ を持っているときは

($r_1, \ldots, r_n, \{p_1 = d_1, \ldots, p_k = d_k\}$) {
return *u*.*m*($r_1, \ldots, r_n, p_1, \ldots, p_k$);
}

もし **identical**(o_1, o_2)のときに限り o_1#*m* == o_2#*m*, o_1.*m* == o_2.*m*, o_1#*m*== o_2.*m* 及び o_1.*m* == o_2#*m* であることを除く。

オブジェクト o 上のゲッタ *f* のクロージャ化(*closurization of getter f on object o*)はもし f がmという名前であれば(){**return** u.m;}に等価であると定義される（identical(*o*1, *o*2)であるときに限り o_1#*m* == o_2#*m* である事を除き）。

オブジェクト *o* 上のセッタ *f* のクロージャ化(*closurization of setter f on object o*)はもし *f* が *m* =という名前であれば(a){**return** u.= a;}に等価であると定義される（identical(*o*1, *o*2)であるときに限り o_1#*m* = == o_2#*m* =である事を除き）。

identical(o_1.*m*, o_2.*m*)であることは保障されない Dart の実装はこれら及び他のクロージャたちを正規化することは要求されていない。

この場合の対等性に対する特別な取り扱いにより API たちのなかでの抽出された属性関数たち（イベント・リスナたちがしばしば登録され、追って登録から外されねばならないようなコールバック）の使用を促進させる。典型的な例はウェブ・ブラウザにおける DOM API である。

観察：
ドット・ベースの文法を介してコンストラクタ、ゲッタ、またはセッタをクロージャ化することはできない。#ベースの書式を使わねばならない。メソッドを介いしてかフィールド／ゲッタを介して属性を実装したかどうかを知らせることができる。このことはどのコンストラクタを使用するかに関して先行して計画し、その選択が該クラスのインターフェイスのなかに反映されていなければならないことを意味する。

16.18.8 指名コンストラクタのクロージャ化(Named Constructor Closurization)

*型 T のコンストラクタ f のクロージャ化(Closureization of constructor f of type T)*は以下に等価だと定義される:

- もし *f* が要求パラメタたち $r_1, \ldots, r_n$ とデフォルト $d_1, \ldots, d_k$ を持った指名パラメタたち $p_1, \ldots, p_k$ を持った名前 *m* を持った指名コンストラクタである場合は
  ($r_1, \ldots, r_n, \{p_1 : d_1, \ldots, p_k : d_k\}$) {
  return new *T*.*m*($r_1, \ldots, r_n, p_1 : p_1, \ldots, p_k : p_k$);
  }

- もし *f* が要求パラメタたち $r_1, \ldots, r_n$ とデフォルト $d_1, \ldots, d_k$ を持ったオプショナルな位置的パラメタたち $p_1, \ldots, p_k$ を持った名前 *m* を持った指名コンストラクタである場合は
  ($r_1, \ldots, r_n, [p_1 = d_1, \ldots, p_k = d_k]$) {

return new *T.m*(r_1, . . . , r_n, p_1, . . . , p_k);
}

identical(T_1, T_2)のときに限り **new** T_1#*m* == new T_2#*m*.であることを除く。

上記のことは非パラメタ化された型たちの場合「同じ」型上でのクロージャ化からもたらされるクロージャたちの対等性に依存できることを意味する。パラメタ化された型たちではこれらの正規化することは要求されていないので依存できない。

16.18.9 匿名コンストラクタのクロージャ化(Anonymous Constructor Closurization)

型 T の匿名コンストラクタ f のクロージャ化(*closurization of anonymous constructor f of type T*)は以下であると定義される:

- もし *f* が要求パラメタたち r_1, . . . , r_nとデフォルト d_1, . . . , d_k を持った指名パラメタたち p_1, . . . , p_k を持った匿名コンストラクタである場合は
  (r_1, . . . , r_n, {p_1 : d_1, . . . , p_k : d_k}) {
  return new *T*(r_1, . . . , r_n, p_1 : p_1, . . . , p_k : p_k);
  }

- もし *f* が要求パラメタたち r_1, . . . , r_nとデフォルト d_1, . . . , d_k を持ったオプショナルな位置的パラメタたち p_1, . . . , p_k を持った匿名コンストラクタである場合は
  (r_1, . . . , r_n, [p_1 = d_1, . . . , p_k = d_k]) {
  return new *T*(r_1, . . . , r_n, p_1, . . . , p_k);
  }

  identical(T_1, T_2)のときに限り **new** T_1#*m* == **new** T_2#*m* であることを除く。

16.18.10 Super のクロージャ化 (Super Closurization)

スーパークラス S に関するメソッド f のクロージャ化(*closurization of method f with respect to superclass S*)は以下に等価だと定義される:

- もし *f* が *op* という名前で *op* が<, >, <=, >=, ==, -, +, /, ~/, *, %, |, ^, &, <<, >>のひとつであるときは(*a*){**return super** *op a*;}
- もし *f* が~という名前なら(){**return** ~**super**;}
- もし f が[]という名前なら(*a*){**return super**[*a*];}
- もし f が[] =という名前なら(*a*, *b*){**return super**[*a*] = *b*;}

- もし *f* が *m* という名前を持ち、要求パラメタたち $r_1, \ldots, r_n$ とデフォルト $d_1, \ldots, d_k$ を持った指名パラメタたち $p_1, \ldots, p_k$ を持っている場合は
  $(r_1, \ldots, r_n, \{p_1 : d_1, \ldots, p_k : d_k\})$ {
  return super.m($r_1, \ldots, r_n, p_1 : p_1, \ldots, p_k : p_k$);

- もし *f* が *m* という名前を持ち、要求パラメタたち $r_1, \ldots, r_n$ とデフォルト $d_1, \ldots, d_k$ を持った位置的パラメタたち $p_1, \ldots, p_k$ を持っている場合は
  $(r_1, \ldots, r_n, \{p_1 = d_1, \ldots, p_k = d_k\})$ {
  return super.m($r_1, \ldots, r_n, p_1, \ldots, p_k$);

もし 2 つのクロージャ化が同一の this のバインドを持った同じクラス内で宣言されたコードで生成されているときに限り **super**$_1$*#m* == **super**$_2$*#m*, **super**$_1$*.m* == **super**$_2$*.m*, **super**$_1$*#m* == **super**$_2$*.m* 及び **super**$_1$*.m* == **super**$_2$*#m* であることを除く。

スーパークラス S に関するゲッタ f のクロージャ化(*closurization of getter f with respect to superclass S*)は(){**return super**.m;}と等価だと定義される。但し 2 つのクロージャ化が同一の **this** のバインドを持った同じクラス内で宣言されたコードで生成されているときに限り **super**$_1$*#m* == **super**$_2$*#m* であることを除く。

スーパークラス S に関するセッタ *f* のクロージャ化は(*a*){**return super**.m = *a*;}と等価だと定義される。但し 2 つのクロージャ化が同一の **this** のバインドを持った同じクラス内で宣言されたコードで生成されているときに限り **super**$_1$*#m* = == **super**$_2$*#m* =であることを除く。

16.19 代入(Assignment)

代入は可変変数(mutable variable)（訳者注：final でない変数）または属性(property)に結び付けられた値を変更する。

assignmentOperator（代入演算子）:
'='
| compoundAssignmentOperator（復号代入演算子）
;

v = *e* の形式の代入 *a* の計算は以下のように進行する：

d がその名前が *v* または *v*=である最も内側の宣言（もし存在すれば）だとしよう。

もし *d* があるローカル変数の宣言であるときは、式 *e* はあるオブジェクト *o* として計算される。次に *v* が **final** または **const** でない限りその変数 *v* は *o* にバインドされ、**final** または **const** であるときは動的エラーが生起される。エラーが発生しないときは、この代入式の値は *o* である。

もし *d* がライブラリ変数、トップ・レベル・ゲッタ、またはトップ・レベル・セッタの宣言のときは、式 *e* は

あるオブジェクト o として計算される。次に o にバインドされたその仮パラメタでセッタ v=が呼び出される。この代入式の値は o である。

そうでないときは、もし d がクラス C の中の static 変数、static ゲッタ、または static セッタの宣言のときは、この代入は $C.v = e$ なる代入と等しい。

そうでないときは、もし a がトップ・レベルまたは static 函数（関数、メソッド、ゲッタ、またはセッタ）、または変数イニシャライザの内部で生じるときは、a の計算は e の計算を引き起こし、その後で **NoSuchMethodError** がスローされる。

そうでないとき、その代入は代入 **this**.$v = e$ と等価である。

*チェック・モード*においては、もし o が **null** でなく、また o のクラスで導入されているインターフェイスが v の実型（19.8.1 節）の副型でないときは、動的型エラーである。

e の静的型が v の静的型に代入出来ない可能性があるときは静的型エラーである。式 v=e の静的型は e の静的型である。

$e_1?.v = e_2$ の形式の代入aの計算は式$((x) => x == \textbf{null}?\textbf{null} : x.v = e_2)(e_1)$の計算と等価である。a の静的型は e_2 の静的型である。T を e_1 の静的型だとし、y を型 T の新鮮変数だとしよう。まさしく $y.v = e_2$ で引き起こされると同じ静的警告が $e_1?.v = e_2$ においても起きる。

$e_1.v = e_2$ の形式の代入の計算は以下のように進行する：

最初にオブジェクト o_1 として式 e_1 が計算される。次にオブジェクト o_1 として式 e_2 が計算される。次に、現在のライブラリに関して o_1 のなかのセッタ v=が現行ライブラリに関して検索（16.15.2 節）される。もし o_1 が Type のインスタンスであるが e_1 は常数型リテラルでないときは、次にもし v = が static セッタに転送（9.1 節）するセッタの時は、検索は失敗する。そうでないときは、そのボディが o_1 に対する **this** バウンドと o_2 に対する仮パラメタ・バウンドで実行される。

もしこのセッタ検索が失敗したら、以下のようにあらかじめ定義されているクラスの Invocation のインスタンス *im* が生成される：

- im.isSetter が **true** と評価される。
- im.memberName が'v='と評価される。
- im.positionalArguments が[o_2]と同じ値を持った不変リストとして計算される。
- im.namedArguments が **const** {}という値として評価される。

次に o_1 のなかの noSuchMethod()が検索され引数 *im* で呼び出される。

しかしながら、もし該実装が単一の位置的引数で呼び出せないということを見出した時は、クラス **Object** 内の **noSuchMethod()**実装が引数 *im'* で o_1 上で呼び出される。ここに *im'*は以下のようである **Invocation** のインスタンスである：

- · im'.**isMethod** が **true** と計算される
- · im'.**memberName** は noSuchMethod と計算される
- · im'.**positionalArguments** はその唯一の要素が *im* である不変リストとして計算される
- · im'.**namedArguments** は const {}の値として計算される

この代入式の値はセッタ検索が成功しようと失敗しようと関係なく o_2 である。

チェック・モードにおいては、o_2 が **null** でなく o_2 のクラスで導入されたインターフェイスが $e_1.v$ の実型の副型でないときは動的型エラーである。

T を e_1 の静的型だとする。もし T が v=という名前のアクセス可能なインスタンス・セッタを持っていないときはきは次のいずれでもなけば静的型警告となる:

- T または T のスーパーインターフェイスが **dart:core** で定義されている定数**@proxy** と同一の定数を示すアノテーションでアノテートされている。または、
- T が **Type** で、e が定数型リテラルで、e に対応するクラスが m という名前の static メソッドまたはゲッタを有する

もし e_2 の静的型がセッタ $v=$ の仮パラメタの静的型に代入できない可能性があるときは静的警告である。式 $e_1.v = e_2$ の静的型は e_2 の静的型である。

super.$v = e$ の形式の代入の計算は以下のように進行する:

S を直ちに包含しているクラスのスーパークラスだとしよう。式 e はオブジェクト o として計算される。次に、セッタ $v=$ が現行ライブラリに対し S のなかで検索(16.15.2) される。$v=$ のボディは o にバインドされた仮パラメタたちと **this** にバインドされた **this** バインドで実行される。

もしこのセッタ検索が失敗すれば、次に以下のようなあらかじめ定義されているクラスの **Invocation** の新規のインスタンスである *im* が生成される:

- im.isSetter が **true** と計算される
- im.memberName はシンボル v=と計算される
- im.positionalArguments は [o]と同じ値たちを持った不変リストとして計算される
- im.namedArguments は **const** {} の値として計算される

次に S のなかの noSuchMethod()が検索され引数 *im* で呼び出される。

しかしながら、もし該実装が単一の位置的引数で呼び出せないということを見出した時は、クラス **Object** 内の **noSuchMethod()**実装が引数 im_o で **this** 上で呼び出される。ここに im_0 は以下のようである **Invocation** のインスタンスである:

- · im'.**isMethod** が **true** と計算される
- · im'.**memberName** は#noSuchMethod と計算される
- · im'.**positionalArguments** はその唯一の要素が *im* である不変リストとして計算される

- · im'.**namedArguments** は const {}の値として計算される

この代入式の値はセッタ検索が成功しようと失敗しようと無関係に *o* である。

チェック・モードにおいては、*o* が **null** でなく *o* のクラスで導入されたインターフェイスが S.*v* の実型の副型でないときは動的型エラーである。

もし *S* が *v*=という名前のアクセス可能なインスタンス・セッタを持っていないとき、あるいは *S* のスーパーインターフェイスが **dart:core** で定義されている定数**@proxy** と同一の定数を示すアノテーションでアノテートされているときは静的型警告となる。

もし e_2 の静的型がセッタ *v* = の仮パラメタの静的型に代入できない可能性があるときは静的警告である。式 **super**.*v* = *e* の静的型は *e* の静的型である。

$e_1[e_2] = e_3$ の形式の代入の計算は(*a, i, e*){*a*.[]=(*i, e*); **return** *e*; } (e_1, e_2, e_3)なる式の計算と等価である。式 $e_1[e_2] = e_3$ の静的型は e_3 の静的型である。

super[e_2] = e_3 の形式の代入の計算は super.[e1] = e2 なる式の計算と等価である。式 **super**[e_1] = e_2 の静的型は e_2 の静的型である。

もし *v* = *e* の形式の代入がトップ・レベルまたは static 関数（関数、メソッド、ゲッタ、またはセッタ）または変数初期化子の内部で発生しており、その代入を包含している構文スコープ内に名前が *v* のローカル変数宣言も *v* =という名前のセッタ宣言も存在していないときは、静的警告である。

直後にトーケン'.'が置かれた前置オブジェクト（18.1 節）上で、あるいは常数型リテラル上でクラス Object のセッタたちのどれかを呼び出すのはコンパイル時エラーである。

16.19.1 複合代入(Compound Assignment)

v ??= *e* の形式の複合代入の計算は式((*x*) => *x* == **null** ? *v* = *e* : *x*)(*v*)の計算と等価であり、ここに *x* は e のなかで使われていない新鮮変数である。 *C*.*v* ??= *e*（ここに *C* は型リテラル）の形式の複合代入の計算は式 ((*x*) => *x* == **null**? *C*.*v* = *e* : *x*)(*C*.*v*)と等価であり、ここに *x* は e のなかで使われていない新鮮変数である。

上記の２つのルールは変数 v または型 C がプレフィックスされている場合にも適用される。

e_1.*v* ??= e_2 の書式の複合代入の計算は式((*x*) => ((*y*) => *y* == **null** ? *x*.*v* = e_2 :*y*)(*x*.*v*))(e_1)の計算と等価である。ここに *x* と *y* は e_2 の中で使われていない別々の新鮮変数たちである。

$e_1[e_2]$??= e_3 の形式の複合代入の計算は式((*a, i*) => ((*x*) => *x* == **null** ? *a*[*i*] = e_3 : *x*)(*a*[*i*]))(e_1, e_2)の計算と等価である。ここに *x*、*a*、および *i* は e_3 のなかで使われていない別々の新鮮変数たちである。

super.*v* ??= *e* の形式の複合代入の計算は式((*x*) => *x* == **null** ? **super**.*v* = *e* : *x*)(**super**.*v*)の計算と

等価である。ここに *x* は *e* のなかで使われていない新鮮変数である。

e_1?.*v* ??= e_2 の形式の複合代入の計算は式((*x*) => *x* == **null** ? **null**: *x.v*?? = e_2)(e_1)の計算と等価である。ここに *x* は e_2 の中で使われていない変数である。

v ??= *e* の形式の復号代入の静的型は *v* の静的型及び *e* の静的型の最小上界である。*v* = *e* で引き起こされるであろうとまさしく同じ静的警告が *v* ??= *e* の場合においても生成される。

C.v ??= *e* の形式の復号代入の静的型は *C.v* の静的型及び *e* の静的型の最小上界である。*C.v* = *e* で引き起こされるであろうとまさしく同じ静的警告が C.*v* ??= *e* の場合においても生成される。

e_1?.*v* ??= e_2 の形式の複合代入の静的型は e_1.*v* の静的型及び e_2 の静的型の最小上界である。*T* を e_1 の静的型そして *z* を型 *T* の新鮮変数だとしよう。*z.v* = e_2 で引き起こされるであろうとまさしく同じ静的警告が e_1.*v* ??= e_2 の場合においても生成される。

e_1[e_2] ??= e_3 の形式の複合代入の静的型は e_1[e_2]の静的型及び e_3 の静的型の最小上界である。e_1[e_2] = e_3 で引き起こされるであろうとまさしく同じ静的警告が e_1[e_2] ??= e_3 の場合においても生成される。

super.*v* ??= *e* の形式の複合代入の静的型は **super**.*v* の静的型及び *e* の静的型の最小上界である **super**.*v* = *e* で引き起こされるであろうとまさしく同じ静的警告が **super**.*v* ??= *e* の場合においても生成される。

その他のどの任意の有効な演算子 *op* にたいし、*v op* =*e* の形式の復号代入は *v*=*v op e* と等価である。*C.v op*= *e* の形式の複合代入は *C.v* = *C.v op e* と等価である。e_1.*v op*= e_2 の形式の複合代入は((x) => *x.v* = *x.v op* e_2)(e_1)と等価であり、ここに *x* は e_2 で使われていない変数である。e_1[e_2] *op*=e_3 の形式の複合代入は((*a, i*) => *a[i]* = *a[i] op* e_3)(e_1, e_2)と等価で、ここに *a* と *i* は e_3 で使われていない変数たちである。

e_1?.*v op*= e_2 の形式の複合代入の計算は((x) => *x*?.*v* = *x.v op* e_2)(e_1)と等価であり、ここに *x* は e_2 のなかで使われていない変数である。e_1.*v op*= e_2 で引き起こされるであろうとまさしく同じ静的警告が e_1?.*v op*= e_2 の場合においても生成される。

compoundAssignmentOperator（複合代入演算子）:

‘*=’ |
‘/=’ |
‘~/=’ |
‘%=’ |
‘+=’ |
‘-=’ |
‘<<=’ |
‘>>=’ |

‘&=’ |
‘^=’ |
‘|=’ |
‘??=’ |
;

16.20 条件(Conditional)

*条件式(conditional expression)*はブール条件に基づき 2 つの式のひとつを計算する。

conditionalExpression（条件式）:
ifNullExpression（論理または式）('?' expressionWithoutCascade（カスケードなしの式）
':' expressionWithoutCascade（カスケードなしの式）)?
;

e_1 ? e_2 : e_3 の形式の条件式 c の計算は以下のように進行する:

最初に、オブジェクト o_1 として e_1 が計算される。チェック・モードでは、o_1 が型 **bool** でないときは動的型エラーである。そうでないときは、次に o_1 はブール変換（16.4.1 節）の対象であり、オブジェクト r をつくる。もし r が true なら、c の値は式 e_2 の計算結果である。そうでないときは、c の値は式 e_3 の計算結果である。

もし以下のすべてが成り立てば:

- e_1 が変数 v が型 T をもっていることを示している。
- v が e_2 のなかで潜在的に変えられない、またはあるクロージャ内にある。
- もし変数 v が e_2 のなかであるクロージャによって アクセスされているときは、その変数 v は v のスコープ内のどこかで潜在的に変えられていない。

v の型は e_2 のなかで T であることがわかる。

e_1 の型が **bool** に代入出来ない可能性があるときは静的型警告である。c の静的型は e_2 の静的型と e_3 の静的型の最小上界(least upper bound)（19.8.2 節）である

16.21 If-null 式(If-null Expressions)

*If-null 式(If-null Expressions)*は式を計算し、もしその結果が **null** のときは別の式を計算する。

ifNullExpression（ifNull 式）:
logicalOrExpression（論理 Or 式）(‘??’ logicalOrExpression（論理 Or 式）)*

e_1??e_2の形式の if-null 式の計算は((x) => x == **null**?e_2 : x)(e_1)という式の計算と等価である。

e の静的型は e_1 の静的型と e_2 の静的型の最小上階(19.8.2 節)である

16.22 論理ブール式(Logical Boolean Expressions)

論理ブール式はブール積と和の演算子を使ってブール値オブジェクトたちを組み合わせる。

logicalOrExpression（論理和式）:
　logicalAndExpression（論理積式） ('||' logicalAndExpression（論理積式）)*
;

logicalAndExpression（論理積式）:
　equalityExpression（対等式） ('&&' equalityExpression（対等式）)*
;

*論理ブール式(logical boolean expression)*はビット単位式（16.23 節）、または式 e_1 の引数 e_2 での論理ブール演算子の呼び出しのいずれかである。

e_1 || e_2 の形式の論理ブール式 b の計算は e_1 の計算をもたらす;もし e_1 が true と計算されるときは、b の計算は true であり、そうでないときは e_2 はあるオブジェクト o として計算され、それは次にその値が b であるオブジェクト r をつくるブール変換（16.4.1 節）の対象になる。

e_1 && e_2 の形式の論理ブール式 b の計算は e_1 の計算をもたらす;もし e_1 が true と計算されないときは、b の計算は false であり、そうでないときは e_2 はあるオブジェクト o として計算され、それは次にその値が b であるオブジェクト r をつくるブール変換の対象になる。

e_1 && e_2 の形式の論理ブール式 b は、もし以下のすべてが成立すれば変数 v は型 T を持っていることを示す:

- v が型 T を持っていることを e_1 が示している、または v が型 T を持っていることを e_2 が示しているかのどれかである。
- v はローカル変数か仮パラメタかである。
- v が e_1 、e_2、またはあるクロージャのなかで潜在的に変えられない、またはあるクロージャ内にある。

そのときは v の型は e_2 のなかで T である。

もし e_1 の静的型が **bool** に代入できない場合、あるいはもし e_2 の静的型が **bool** に代入できない場合は静的警告である。論理ブール式の静的な型は **bool** である。

16.23 等価性(Equality)

*等価式(equality expression)*はオブジェクトたちの等価性をテストする。

equalityExpression（等価式）:
relationalExpression（関係式）(equalityOperator（等価演算子）relationalExpression（関係式）)?
| **super** equalityOperator（等価演算子）relationalExpression（関係式）
;

equalityOperator（等価演算子）:
'=='
| '!='
;

等価式は関係式（16.24 節）、または **super** または式 e_1 に対し引数 e_2 での等価演算子の呼び出し、のいずれかである。

e_1 == e_2 の形式の等価式 *ee* は以下のように進行する:

- 式 e_1 が評価されオブジェクト o_1 となる。
- 式 e_2 が評価されオブジェクト o_2 となる。
- o_1 と o_2 のどちらかが **null** のときは、もし o_1 と o_2 の双方が **null** なら *ee* は **true** と計算され、それ以外のときは **false** と計算される。そうでないときは、
- *ee* はメソッド呼び出し o_1==(o_2)として計算される。

super == *e* の形式の等価式 ***ee*** の計算は以下のように進行する:

- 式 *e* が計算されオブジェクト *o* になる。
- もし **this** または *o* が **null** のときは、もし **this** と *o* の双方が **null** なら *ee* は **true** と計算され、それ以外のときは **false** と計算される。そうでないときは、
- *ee* はメソッド呼び出し **super**.==(*o*)と等価である。

上記の定義の結果、ユーザ定義の==メソッドはその引数が this でなくまた非 null であると仮定でき、次のような標準的おきまりのコーディングを回避する：

```
if identical(null, arg) return false;
```

更なる意味合いは、null に対するテストとして identical()を使う必要は決して無く、また null == e あるいは e == null を書くべきかどうかを心配することもないということである。

e_1 != e_2 の形式の等価式の計算は式 !(e_1 == e_2)と等価である。

super != *e* の形式の等価式は式!(**super** == *e*)と等価である。

等価式の静的な型は **bool** である。

16.24 関係式(Relational Expressions)

*関係式(relational expression)*はオブジェクトたちに対し関係演算子たちを呼び出す。

relationalExpression（関係式）:
shiftExpression（シフト式） (typeTest（型テスト） | typeCast（型キャスト） | relationalOperator（関係演算子） bitwiseOrExpression（ビット幅 OR 式）)?
| **super** relationalOperator（関係演算子） bitwiseOrExpression（ビット幅 OR 式）
;

relationalOperator（関係演算子）:
'>='
| '>'
| '<='
| '<'
;

*関係式*はビット単位式（16.25 節）、または **super** または式 e_1 に対し引数 e_2 での関係演算子の呼び出し、のいずれかである。

e_1 *op* e_2 の形式の関係式はメソッド呼び出し e_1.*op*(e_2)と等価である。**super** *op* e_2 の形式の関係式はメソッド呼び出し **super**.*op*(e_2)と等価である。

16.25 ビット単位式(Bitwise Expressions)

*ビット単位式(bitwise expression)*はオブジェクトたちに対しビット単位演算子を呼び出す。

bitwiseOrExpression（ビット単位 OR 式）:
bitwiseXorExpression（ビット単位 XOR 式） ('|' bitwiseXorExpression（ビット単位 XOR 式）)*
| **super** ('|' bitwiseXorExpression（ビット単位 XOR 式）)+
;

bitwiseXorExpression（ビット単位 XOR 式）:
bitwiseAndExpression（ビット単位 AND 式） ('^' bitwiseAndExpression（ビット単位 AND 式）)*

| **super** ('^' bitwiseAndExpression（ビット単位 AND 式）)+
;

bitwiseAndExpression（ビット単位 AND 式）:
shiftExpression（シフト式） ('&' shiftExpression（シフト式）)*
| **super** ('&' shiftExpression（等価式）)+

bitwiseOperator（ビット単位演算子）:
'&'
| '^'
| '|'
;

*ビット単位式*はシフト式（16.26 節）、または **super** または式 e_1 に対するビット単位演算子の引数 e_2 で呼び出し、のどれかである。

e_1 *op* e_2 の形式のビット単位式はメソッド呼び出し $e_1.op(e_2)$ と等価である。

super op e_2 の形式のビット単位式はメソッド呼び出し **super**.$op(e_2)$ と等価である。

これらの式たちの静的な型のルールは上記等価性で定義されていることは明白で、従って型 e_1 に対するメソッド呼び出し及び演算子たちのシグネチュアたちの型規則によって定まる。同じことは本仕様にわたって似たような状況に対し言える。

16.26 シフト(Shift)

*シフト式(shift expression)*はオブジェクトたちに対しシフト演算子たちを呼び出す。

shiftExpression（シフト式）:
additiveExpression（加減算式） (shiftOperator（シフト演算子） additiveExpression（加減算式）)*
| **super** (shiftOperator（シフト演算子） additiveExpression（加減算式）)+
;

shiftOperator（シフト演算子）:
'<<'
| '>>'
;

シフト式は加減算式（16.27 節）、または **super** または式 e_1 に対する引数 e_2 でのシフト演算子（16.25 節）の呼び出し、のいずれかである。

e_1 *op* e_2 の形式のシフト式はメソッド呼び出し $e_1.op(e_2)$ と等価である。**super** *op* e_2 の形式のシフト式は

メソッド呼び出し **super**.*op*(e_2)と等価である。

この定義はシフト式たちは暗示的に左から右への計算順序を意味していることに注意のこと：e_1 << e_2 << e_3は(e_1 << e_2).<< (e_3)として処理され、これはまた(e_1 << e_2) << e_3と等価である。

同じことは加減算及び乗除算式でもなりたつ。

16.27 加減算式(Additive Expressions)

*加減算式(additive expression)*はオブジェクトたちに対し加減算演算子を呼び出す。

additiveExpression（加減算式）:
multiplicativeExpression（乗除算式） (additiveOperator（加減算演算子）
multiplicativeExpression（乗除算式）)*
| **super** (additiveOperator（加減算演算子） multiplicativeExpression（乗除算式）)+
;

additiveOperator（加減算演算子）:
'+'
| '-'
;

*加減算式*は乗除算式（16.28 節）、または **super** または式 e_1 に対する引数 e_2 での加減算演算子の呼び出し、のいずれかである。

e_1 *op* e_2の形式の加減算式はメソッド呼び出し e_1.*op*(e_2)と等価である。**super** *op* e_2の形式の加減算式はメソッド呼び出し **super**.*op*(e_2)と等価である。

加減算式の静的型は通常使われている演算子の宣言の中で与えられているシグネチュアによって決まる。しかしながら、クラス **int** の演算子たち＋及び-の呼び出しは、型チェッカによって特別に取り扱われる。e_1 が静的型 **int** を持っている式 e_1+e_2 の静的型は、もし e_2 の静的型が **int** なら **int** であり、e_2 の静的型が **double** なら **double** である。e_1 が静的型 **int** を持っている式 e_1-e_2 の静的型は、もし e_2 の静的型が **int** なら **int** であり、e_2 の静的型が **double** なら **double** である。

16.28 乗除算式(Multiplicative Expressions)

*乗除算式(multiplicative expression)*はオブジェクトたちに対し乗除算演算子を呼び出す。

multiplicativeExpression（乗除算式）:

unaryExpression（単項式） (multiplicativeOperator（多項演算子） unaryExpression（単項式）)*
| **super** (multiplicativeOperator（多項演算子） unaryExpression（単項式）)+
;

multiplicativeOperator（多項演算子）:
'*'
| '/'
| '%'
| '~/'
;

乗除算式は単項式（16.28 節）、または **super** または式 e_1 に対する引数 e_2 での多項演算子の呼び出し、のいずれかである。

e_1 *op* e_2 の形式の乗除算式はメソッド呼び出し $e_1.op(e_2)$ と等価である。**super** *op* e_2 の形式の乗除算式はメソッド呼び出し **super**.*op*(e_2) と等価である。

乗除算式の静的型は通常使われている演算子の宣言の中で与えられているシグネチュアによって決まる。しかしながら、クラス **int** の演算子たち*, % 及び~/ の呼び出しは、型チェッカによって特別*に取り扱われる。e_1 が静的型 **int** を持っている式 e_1*e_2 の静的型は、もし e_2 の静的型が **int** なら **int** であり、e_2 の静的型が **double** なら **double** である。e_1 が静的型 **int** を持っている式 $e_1\%e_2$ の静的型は、もし e_2 の静的型が **int** なら **int** であり、e_2 の静的型が **double** なら **double** である。e_1 が静的型 **int** を持っている式 $e_1\sim/e_2$ の静的型は、もし e_2 の静的型が **int** なら **int** である。

16.29 単項式(Unary Expressions)

*単項式(unary expression)*はオブジェクトたちに対し単項演算子たちを呼び出す。

unaryExpression（単項式）:
prefixOperator（前置演算子） unaryExpression（単項式）
| awaitExpression（アウェイト式）
| (minusOperator（マイナス演算子） | tildeOperator（チルド演算子）) **super**
| incrementOperator（増分演算子） assignableExpression（代入可能式）
;
prefixOperator（前置演算子）:
minusOperator（マイナス演算子） |
negationOperator（否定演算子） |
tildeOperator（チルド演算子）
;
minusOperator（マイナス演算子）:

'-'
;
negationOperator（否定演算子）:
'!'
;
tildeOperator（チルド演算子）:
'~'
;

*単項式*は前置式、後置式（16.30 節）、await 式（16.29 節）、またはある式上のまたは **super** または式 *e* のどれかに対する単項演算子の呼び出し上での前置演算子の呼び出し、のいずれかである。

式!*e* は式 *e*? **false**: **true** と等価である。

++*e* の形式の式の計算は *e* += 1 と等価である。--*e* の形式の式の計算は *e* -= 1 と等価である。

op e 形式の単項式 *u* はメソッド呼び出し式 *e.op*()と等価である。

op **super** 形式の式はメソッド呼び出し（16.17.3 節）**super**.*op*()と等価である。

16.30 アウェイト式 (Await Expressions)

*アウェイト式 (Await Expressions)*により、ある非同期操作（第 9 章）が完了するまでコードが制御を譲ることができる。

awaitExpression（アウェイト式）:
await unaryExpression（単項式）

await *e* の形式のアウェイト式 *a* の計算は次のように進行する:

最初に式 *e* が計算される。次に:

もし *e* が例外 *x* を生起したら、次にクラス **Future** のインスタンス *f* が割り当てられ、あとで *x* で完了する。それ例外の場合は、もし *e* が **Future** のインスタンスではないあるオブジェクト *o* として計算されたときは、*f* を *o* をその引数とした **Future**.value()の呼び出しの結果だとしよう；それ以外のときは *f* は *e* の計算の結果だとしよう。

次に、*a* を直ちに包含している関数 *m* の実行が *f* が完了するまで保留にされる。もし最も内側に包含している for ループ(17.6.3 節)に結びつけられたストリームがあればそれは待機される。*f* が完了したあとのある時点で制御は現行の呼び出しに戻される。最も内側に包含している for ループ

(17.6.3 節)があれば、それが再開される。もし*f*が例外*x*で完了したときは、*a*は*x*を生起する。もし*f*が値*v*で完了すれば、*a*は*v*と計算される。

もし該関数が直ちに包含しているａが非同期と宣言されていないときはコンパイル時エラーである。しかしながら、通常の関数のコンテキストでは await は特別な意味を持っていないので、このエラーは単なる構文エラーである。

await 式は同期関数のなかでは意味を持たない。もしそのような関数がある将来の為に待つべきものなら、それはもはや動機ではなくなる。

もし e の型が Future の副型でないときは静的警告にはならない。ツールたちはそのような場合にヒントを与えることを選択しても良い。

a の静的型は*flatten*(*T*)である。ここに *T* は *e* の静的型である。*flatten* 関数は 16.10 節で定義されている。

16.31 後置式(Postfix Expressions)

*後置式(postfix expression)*はオブジェクトたちに対し後置演算子を呼び出す。

postfixExpression（後置式）:
　assignableExpression（代入可能式） postfixOperator（後置演算子）
| primary（プライマリ） selector（セレクタ）*
;

postfixOperator（後置演算子）:
　incrementOperator（増減分演算子）
;

selector（セレクタ）:
　assignableSelector（代入可能セレクタ）
| arguments（引数たち）
;

incrementOperator（増減分演算子）:
　'++'
| '--'
;

後置式はプライマリ式、関数、メソッドまたはゲッタ呼び出し、あるいは式 e に対する後置演算子の呼び出し、のいずれかである。

v ++の形式（ここに *v* は識別子）の後置式の計算は、(){**var** *r* = *v*; *v* = *r* + 1; **return** *r*}()と等価であ

る。

上記によりv がフィールドで、ゲッタがまさしく一度呼ばれることを確実にしている。同様に以下のケースたちでもそうである。

C.v ++の形式の後置式は、(){**var** *r* = *C.v*; *C.v* = *r* + 1; **return** *r*}()と等価である。

e_1*.v*++の形式の後置式は、(*x*){**var** *r* = *x.v*; *x.v* = *r* + 1; **return** *r*}(e_1)と等価である。

e_1[e_2]++の形式の後置式は(*a, i*){**var** r = *a[i]; a[i]* = *r* + 1; **return** *r*}(e_1, e_2)と等価である。

v--の形式の後置式(*v* は識別子)は(){**var** *r* = *v*; *v* = *r* - 1; **return** *r*}()と等価である。

C.v--の形式の後置式は(){**var** *r* = *C.v*; *C.v* = *r* - 1; **return** *r*}()と等価である。

e_1*.v*--の形式の後置式は(*x*){**var** *r* = *x.v*; *x.v* = *r* - 1; **return** *r*}(e_1)と等価である。

e_1[e_2]-- の形式の後置式は(*a, i)*{**var** *r* = *a*[*i*]; *a*[*i*] = *r* - 1; **return** *r*}(e_1, e_2)と等価である。

16.32 代入可能式(Assignable Expressions)

*代入可能式(assignable expression)*はある代入の左側に生じ得る式たちである。本節では式が完全な代入の左側を構成しないときにこれらの式をどう計算するかを記述している。

無論、もし代入可能式たちが常に左側に生じるのなら、それらの値の必要性はないだろうし、それらの計算の為の規則は不必要であろう。しかしながら、代入可能式は他の式の副式(subexpression)になりえ、従って計算しなければならない。

assignableExpression(代入可能式):
primary(プライマリ) (arguments(引数)* assignableSelector(代入可能セレクタ))+
| **super** assignableSelector(代入可能セレクタ)
| identifier(識別子)
;

assignableSelector(代入可能セレクタ):
'[' expression(式) ']'
| '.' identifier(識別子)
;

*代入可能式*は次のどれかである:

- 識別子

- ある式 *e* に対するメソッド、ゲッタ（10.2 節）、またはリスト・アクセス演算子の呼び出し
- **super** に対するゲッタまたはリスト・アクセス演算子の呼び出し

形式 *id* の代入可能式は識別子式（16.33 節）として計算される。

形式 *e.id* の代入可能式は属性抽出（16.18 節）として計算される。

形式 $e_1[e_2]$の代入可能式は引数 e_2での e_1 に対する演算子メソッド [] の呼び出しとして計算される。

形式 **super**.*id* の代入可能式は属性抽出として計算される。

形式 **super**[e_2]の代入可能式はメソッド呼び出し super.[e_2]と等価である。

16.33 識別子参照(Identifier Reference)

*識別子式(identifier expression)*は単一の識別子で構成される；これは無修飾名を介したオブジェクトへのアクセスを可能にする。

identifier（識別子）:
IDENTIFIER（識別子）
;

IDENTIFIER_NO_DOLLAR（$なし識別子）:
IDENTIFIER_START_NO_DOLLAR（非$で開始する識別子）
IDENTIFIER_PART_NO_DOLLAR（$を持たない識別子部）*
;

IDENTIFIER（識別子）:
IDENTIFIER_START（識別子_開始） IDENTIFIER_PART（識別子_部分）*
;

BUILT_IN_IDENTIFIER（組込み識別子）:
abstract
| as
| deffered
| dynamic
| export
| external
| factory
| get
| implements

| import
| library
| operator
| part
| set
| static
| typedef
;
;

IDENTIFIER_START（識別子_開始）:
IDENTIFIER_START_NO_DOLLAR（非$で開始する識別子）
| '$'
;

IDENTIFIER_START_NO_DOLLAR（非$で開始する識別子）:
LETTER（文字）
| '_'
;

IDENTIFIER_PART_NO_DOLLAR（$なし識別子部）:
IDENTIFIER_START_NO_DOLLAR（非$で開始する識別子）
| DIGIT（桁）
;

IDENTIFIER_PART（識別子部）:
IDENTIFIER_START（識別子_開始）
| DIGIT（桁）
;

qualified（修飾）:
identifier（識別子）('.' identifier（識別子）)?
;

*組込み識別子(reserved words)*というのはプロダクション BUILT_IN_IDENTIFIER により生産される識別子たちのひとつである。組込み識別子が前置詞、クラス、変数、または型エイリアスの宣言された名前として使われているときはコンパイル時エラーである。**dynamic** 以外の組込み識別子が型のテーションで使われているときはコンパイル時エラーである。

組込み識別子はDartにおけるキーワードとして使われる識別子たちであるが、Javascriptの予約語ではない。Javascriptコードを Dartにインポートする際の非互換性を最小化する為に、我々はこれらを予約語とはしていない。しかしながら、組込み識別子はクラスまたは型の名前としては使えない。言い換えると、これらは型として使われるときは予約語として取り扱われる。これにより、互換性の問題を引き起こすことなく多くの混乱する状況を無くしている。結局、型宣言または型アノテー

ションをもっていないのでJavascript プログラムはクラッシュが起き得ない。更に、型たちは大文字で始まらねばならず(添付参照)、兎に角 Dart どのユーザ・プログラムでもクラッシュは起きないはずである。

識別子たち **async**, **await** または **yield** のどれかが **async**, **async*** または **sync***のどれかでマークされたある関数ボディの識別子として使われているときはコンパイル時エラーである。

互換性の理由から、新規の構文は新しい予約語あるいはたとえ組み込み識別子も依存できない。しかしながら、上記の構文はこれらの構文と並行的に導入されていて古いコードはそれらを使わない特別のマーカを必要とするコンテキストの中では有用である。したがってこの制約はこれらの名前を限られたコンテキストの中で予約語として扱っている。

形式 *id* の*識別子式 e* の計算は以下のように進行する:

d をその名前が *id* である包含している構文スコープ内の最も内部の宣言だとする。もし *d* がクラス、インターフェイス、または型変数のときはコンパイル時エラーである。もしそのような宣言が構文スコープ内にないときは、*d* を *id* という名前の(もし存在すれば)継承したメンバの宣言だとする。

- もし *d* が前置子 *p* のときは、このトーケン *d* の直後に'.'がないかぎりコンパイル時エラーである。
- もし *d* がクラスまたは型エイリアス *T* のときは、*e* の値は *T* を具象化しているクラス **Type**(またはその副クラス)のインスタンスである。
- もし *d* が型パラメタ *T* のときは、*e* の値は **this** の現行バインディングを生成した生成的コンストラクタに渡された *T* に対応した実際の型引数の値である。我々は **this** は良く定義されていることが保証されている。しかしながら、もし *e* が static メンバたちの内部で生じているときはコンパイル時エラーが生じる。
- もし *d* が **const** *v* = *e*;または **const** *T v* = *e*;の形式のどれかだの定数変数とすると、その値 *id* はコンパイル時定数 *e* の値である。
- もし *d* がローカル変数または仮パラメタのときは、*e* は計算され *id* の現在のバインディングとなる。
- もし *d* が static メソッド、トップ・レベル関数、またはローカル関数のときは、次に *e* は *d* によって定義された関数として評価される。
- もし *d* が静的変数またはクラス *C* 内で宣言された静的ゲッタなら、次に *e* は属性抽出(16.18 節) *C.id* と等価である。
- もし *d* がライブラリ変数、トップ・レベルのゲッタ、またはトップ・レベルのセッタの宣言のときは、*e* はトップ・レベルのゲッタ呼び出し(16.16 節) *id* と等価である。

- そうでなければ、もし *e* がトップ・レベルまたは静的関数（関数、メソッド、ゲッタ、またはセッタ）、または変数イニシャライザのときは、*e* の計算により **NoSuchMethodError** がスローされる。

- そうでなければ *e* は属性取り出し（16.18 節）**this**.*id* と等価である

e の静的型は以下のように決まる:

- もし *d* がクラス、型エイリアス、または型パラメタのときは、*e* の静的型は **Type** である。

- もし *d* がローカル変数または仮パラメタのときは、*e* の静的型は変数 *id* の型である。但し *id* が何らかの型 *T* であることが分かっているときは除き、この場合は *T* が他の型 *S* よりもより特定的であるとすれば *v* が型 *S* を持っていることがわかる。

- もし *d* が静的メソッド、トップ・レベル関数、またはローカル関数のときは、*e* の静的型は *d* で定義された関数型である。

- もし *d* がクラス *C* のなかで宣言された静的変数または静的ゲッタまたは静的セッタのときは、*e* の静的型はゲッタ呼び出し（16.18 節）*C.id* の静的型である。

- もし *d* がライブラリ変数またはトップ・レベルのゲッタの宣言のときは、*e* の静的型はゲッタ呼び出し *id* の静的型である。

- そうでないときは、もし *e* がトップ・レベルまたは静的関数（関数、メソッド、ゲッタ、またはセッタ）、または変数イニシャライザの内部で生じているときは、*e* の静的型は **dynamic** である。

- そうでないときは、*e* の静的型は属性抽出（16.18 節）***this***.*id* の型である。

誰かがあるセッタを宣言するときは、我々はた例えそおれが存在しなくても対応するセッタにバインドすることに注意されたい。

これにより関係しないセッタとゲッタたちを使ってしまうという状況を防止される。その意図は周辺のスコープ内で偶発的にあるゲッタが使われているときのエラーを防止することにある。

形式 id の識別子式がトップ・レベルまたは静的関数（関数、メソッド、ゲッタ、またはセッタ）、または変数イニシャライザ内で生じており、その式を包含している構文スコープ内で id という名前を持った宣言 d が存在しないときは、静的警告となる。

16.34 型テスト(Type Test)

*is-式(is-expression)*はあるオブジェクトがある型のメンバであるかどうかをテストする。

typeTest（型テスト）:
  isOperator（is 演算子） type（型）
;

IsOperator（is 演算子）:
is '!'?
  ;

is-式 *e* **is** *T* の計算は以下のように進行する:

式 *e* は値 *v* として計算される。次に、もし T が奇形または後回しの型（19.1 節）のときは、動的エラーが発生する。そうでなければ、もし *v* のクラスのインターフェイスが *T* の副型のときは、その is-式は true と計算される。そうでないときは false として計算される。

これは Object が常に true だということに従っている。これは総てがオブジェクトだとする言語では道理にかなっている。

また *T* = Object または *T* = Null で無い限り null is *T* は false であることに注意のこと。クラス Null はコア・ライブラリからエクスポートされていないので、後者はユーザ・コードには出てこないだろう。前者は e is Object 形式のどれもがそうであるように意味のないものである。ユーザは型テストによらずに直接 null 値をテストすべきである。

is-式 *e* **is!** *T* は!(*e* **is** *T*)と等価である。

v がローカル変数または仮パラメタだとしよう。 *v* **is** *T* という形式の is 式は、*T* が式 *v* の型 *S* よりもより特定的で *T* != **dynamic** と *S* != **dynamic** の双方であるときに限り、*v* が型 *T* を有していることを示す。

「v が型 T を有することを示す」関係の動機は、紛らわしい警告を減らし、それによりより自然なコーディング・スタイルを可能とすることである。現行の仕様の規則は意図的にシンプルであることを維持している。これにより将来これらの規則を洗練化する際に上位互換性が得られよう;そのような洗練化は警告なしのコードを受け付けるが、現在警告にならないどのコードも排除しないようになろう。

ローカル解析ではアクセスできない副作用を持った関数またはメソッドによってフィールドが書き換えられる可能性があるので、この規則はローカルたちとパラメタたちにのみ適用される。

既に知られているより弱い型を減らすのは意味がない。更に、これは与えられた点で複数の型がある変数に結び付けられる状況をもたらし、これは状況を複雑化させる。従って T << S という要求となっている（副型は半順序(partial order)でないので我々は副型化ではなくて<<を使用している）。

*我々は dynamic 型の変数の型を洗練化したくない。なぜならこれは警告を減らすよりは増やしてしまいかねないからである。反対の要求である T != **dynamic** は S を bottom にするセーフガードである。*

is 式の静的型は **bool** である。

16.35 型キャスト(Type Cast)

キャスト式*(cast-expression)*はあるオブジェクトがある型のメンバであることを確保する。

typeCast（型キャスト）:
  asOperator type
;
asOperator（as 演算子）:
as
  ;

e **as** *T* という形式のキャスト式の計算は以下のように進行する:

式 *e* が計算され値 *v* が得られる。次に、もし *T* が奇形または後回しの型（19.1 節）の時は、動的エラーが発生する。そうでないときは、*v* のクラスのインターフェイスが *T* の副式であるならば、このキャスト式は *v* と計算される。そうでないときは、もし *v* が **null** なら、このキャスト式は *v* と計算される。それ以外の総てに対しては CastException がスローされる。

キャスト式 *e* **as** *T* の静的型は *T* である。

17. 文(Statements)

statements（文たち）:
　　statement（文）*
　;
statement（文）:
　　label（ラベル）* nonLabelledStatement（非ラベル付き文）
　;

nonLabelledStatement（非ラベル付き文）:
　　block（ブロック）
　| localVariableDeclaration（ローカル変数宣言） ';'
　| forStatement（for 文）
　| whileStatement（while 文）
　| doStatement（do 文）
　| switchStatement（switch 文）
　| ifStatement（if 文）
　| rethrowSatement（rethrow 文）
　| tryStatement（try 文）
　| breakStatement（break 文）
　| continueStatement（continue 文）
　| returnStatement（return 文）
　| yieldStatement（yield 文）
　| expressionStatement（式文）
　| assertStatement（assert 文）
　| localFunctionDeclaration（ローカル関数宣言）
　;

17.1 ブロック(Blocks)

*ブロック文(block statement)*はコードの順序化をサポートする。

ブロック行文{s_1 … s_n}の実行は以下のように進行する:

$i = 1 .. n$ に対し s_i が実行される。

ブロック文は新しいスコープをもたらし、そのスコープはそのブロック文が発生している構文的に包含しているスコープ内にネストされる。

17.2 式文(Expression Statements)

*式文(expression statement)*は明示的な型引数を持っていない非定数マップ・リテラル（16.8 節）以外の式で構成される。

このマップへの制限は文が{で始まる際の文法上の曖昧さを解決する為になされている。

expressionStatement（式文）:
expression（式）? ';' |
;

式文 *e*;の実行は *e* を計算することで進行する。

明示的な型引数たちを持たないある非 constant マップ・リテラルが、文があるべき場所に出現したときはコンパイル時エラーである。

17.3 ローカル変数宣言(Variable Declaration Statement)

*変数宣言文(variable declaration statement)*は新規ローカル変数を宣言する。

localVariableDeclaration（ローカル変数宣言）:
initializedVariableDeclaration（初期化された変数宣言） ';'

var *v* = *e*;, *T v* = *e*;, **const** *v* = *e*; , **const** *T v* = *e*;, **final** *v* = *e*;または **final** *T v* = *e*;の形式たちのひとつの変数宣言文の実行は次のように進行する:

式 *e* が計算されあるオブジェクト *o* が計算される。次にその変数 *v* が *o* にセットされる。

var *id;*の形式の変数宣言文は **var** *id* = **null**;と等価である。

T id;の形式の変数宣言文は *T id* = **null**;.と等価である。

このことは型 T に関わらずなりたつ。例えば、int *i*; は *i* をゼロに初期化することをしない。その代り、*i* は null に初期化され、これはあたかも var *i*; または Object *i*; または Collection<String> *i*;と書いた場合とおなじである。

ほかのやり方をすると Dart のオプショナルな型づけという性質を損ない、型アノテーションがプログラムの振る舞いを変えてしまう可能性がある。

17.4 ローカル関数宣言(Local Function Declaration)

関数宣言文(function declaration statement)は新しいローカル関数(9.1 節)を宣言する。

localFunctionDeclaration(ローカル関数宣言):
functionSignature(関数シグネチュア) functionBody(関数ボディ)
;

*id signature {statements}*または *T id signature {statements}*の形式たちのひとつの関数宣言文により、その関数宣言文の直後の場所での最も内側の包含するスコープに *id* という名前の新しい関数が生成される。

その宣言より前にあるローカル関数を参照するのはコンパイル時エラーである。

このことはローカル関数は直接に再帰出来るが相互に再帰はできないことを意味する。これらのサンプルを検討してみよう:

```
f(x) = x++; // トップ・レベル関数
top() { // 別のトップ・レベル関数
..f(3); // 違反
  f(x) => x > 0? x*f(x-1): 1; // 再帰は合法
  g1(x) => h(x, 1); // エラー:h は未だスコープ内にない
  h(x, n) => x > 1? h(x-1, n*x): n; // ここでも再帰は有効
  g2(x) => h(x, 1); // 合法
  p1(x) => q(x,x); // 違法
  q1(a, b) => a > 0 ? p1(a-1): b; // 有効
  q2(a, b) => a > 0 ? p2(a-1): b; // 違法
  p1(x) => q2(x,x); // 有効
}
```

相互に再帰するローカル関数のペアを書くことは、他方がスコープ内にある前にひとつが常にあらねばならないので、不可能である。これらのケースは滅多になく、変数たちのペアを最初に定義し、次にそれらをしかるべきクロージャを代入することで処理できる:

```
top2() { // トップ・レベル関数
  var p, q;
  p = (x) => q(x,x); // 合法
  q = (a, b) => a > 0 ? p(a-1): b; // 有効
}
```

ローカル関数のこの規則はローカル変数に対するそれと少し異なっており、関数はその宣言内でアクセス可能であるが、変数はその宣言の後でのみアクセス可能である。これは再帰的な関数は有用なものではあるが再帰的に定義された変数は殆ど常にエラーであるからである。従ってローカル変数の規則とではなくて関数の規則にローカル関数の規則を一般的に合わせることは意味があ

る。

17.5 If

*if 文(if statement)*により、文たちの条件つきでの実行が可能になる。

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

if(*b*) s_1 **else** s_2の形の if 文の実行は以下のように進行する:

最初に式 *b* が計算されオブジェクト *o* が得られる。次に、*o* は次にブール変換(16.4.1 節)の対象となりオブジェクト *r* が得られる。もし *r* が **true** なら、次に文{s_1}が実行され、そうでないときは文{s_2}が実行される。

if (*b*) s_1 ***else*** s_2の形式の if 文は **if**(*b*) {s_1} **else** {s_2}なる if 文と等価である。

この等価性の根拠は次のようなエラーを捕捉する為である:

```
void main() {
 if (somePredicate)
   var v = 2;
 print(v);
}
```

*然るべきスコープ規則のもとでは、このようなコードは問題がある。もし我々が v がメソッド main()のスコープ内で宣言されていると仮定すると、somePredicate が false のときは、v はアクセスされたときには初期化されなくなる。最もすっきりしたアプローチは勝手な文ではなくて、テストの後にブロックを必要とさせ、あるスコープを導入することであろう。しかしながらこれは積年の慣習に逆らい、Dart の馴染み易さという目標を阻害してしまう。我々はそうではなくてこの断定(predicate)のあとの文(そして同様に **else** とループに対し)の周りにブロックを挿入するという手段を選択している。これにより上記のケースでは警告と実行時エラーの双方を生じさせる。無論、囲んでいるスコープ内に v の宣言があれば、それでもプログラマたちは驚くかもしれない。我々はツールたちがそのような状況回避のために潜んでいるケースたちを浮かびださせることを期待している。*

式 b の型が **bool** に代入出来ない可能性があるときは静的型警告である。

もし:

- *b* はある変数 *v* が型 *T* を持っていることを示している。
- *v* が s_1 またはあるクロージャ内で潜在的に変化(potentially mutated)していない。
- もし変数 *v* が s_1 内であるクロージャによってアクセスされているときは、その変数 *v* は *v* のスコープのなかのどこにおいても潜在的に変化しない。

そうすると、*v* の型は s_1 内で *T* であることがわかる。

if (*b*) s_1 の形式の if 文は **if**(*b*) s_1 **else** {}という if 文と等価である。

17.6 For

*For 文(for statement)*は繰り返しをサポートする。

forStatement（for 文）:
for '(' forLoopParts（for ループ要素） ')' statement（文）
;

forLoopParts（for ループ要素）:
forInitializerStatement（for イニシャライザ文） expression（式）? ';' expressionList（式リスト）?
| declaredIdentifier（宣言された識別子） **in** expression（式）
| identifier（識別子） **in** expression（式）
;

forInitializerStatement（for イニシャライザ文）:
initializedVariableDeclaration（初期化された変数宣言） ';'
| expression（式）? ';'
;

for 文はふたつの形式、即ち従来の for ループとフォア・イン(for-in)文を持つ。

17.6.1 for ループ(For Loop)

for (**var** *v* = e_0 ; *c*; *e*) s の形式の for 文の実行は以下のように進行する:

もし *c* が空のときは *c′*を **true** とし、そうでなければ *c′*は *c* としよう。

最初に変数宣言文 **var** *v* = e_0 が実行される。次に;

1. もしこれがこの for ループの最初の繰り返しのときは、*v′*を *v* とし、そうでないときは *v′*はステップ 4 の以前の実行で作られた変数 *v″*としよう。

2. 式[*v'*/*v*]*c* が計算され、ブール変換（16.4 節）の対象とする。もしこの結果が **false** のときは、この for ループは終了する。そうでないときは、実行はステップ 3 で継続する。

3. 文[*v'*/*v*]{*s*}が実行される。

4. *v"*を新規の変数だとする。*v"*は*v'*にバインドされる。

5. 式[*v"*/*v*]*e* が計算され、このプロセスはステップ 1 で再帰呼び出しされる。

上記の定義は、ユーザがfor ループの中にクロージャを作ってしまうという一般的な間違い、即ちそのループ変数の現在のバインディングを閉じようとし、(通常はデバッグと学習という痛みのあるプロセスを経て)作られた総てのクロージャたちが同じ値(最後の繰り返しのなかでつくられた値)にされてしまう事態、を防止することを意図したものである。

そうではなくてこの定義では各繰り返しが別々の変数を持つ。最初の繰り返しは最初の宣言で作られた変数を使用する。各繰り返しの終わりに実行された式は新規な変数v"を使用し、現在の繰り返しの変数の値にバウンドされ、次に次の繰り返しで必要とされるようにv"を修正する。

17.6.2 For-in

for (*finalVarOrType* id **in** *e*) *s* の形式の for 文は以下のコードと等価である:

```
var n0 = e.iterator;
while (n0.moveNext()) {
  varOrType? id = n0.current;
  s
}
```

ここに n0 はこのプログラムのどこにも生じない識別子である。

n0.current は定数式ではないので、const 変数を使うことは実際コンパイル時エラーを起こすことになることに注意されたい。

17.6.3 非同期 For-in

for-in 文は非同期になり得る。非同期の形式はストリーム上での繰り返し操作の為に設計されている。非同期の for ループは for キーワードの直前に付された await というキーワードにいより区別される。

await for (*finalConstVarOrType*? id **in** e) *s* の形式の for-in 文の実行は以下のように進行する:

式 *e* が計算されオブジェクト *o* を得る。もし o が Stream を実装したクラスのインスタンスでないときは動的エラーである。そうでない場合は **await** v_f (16.29)が計算される。ここで v_fはその値が組み込みクラスの Future を実装した新鮮インスタンス *f* である新鮮変数(10.6.1)である。

ストリーム *o* はリスンされ、*o* のなかの各データ・イベントで、このストリームの現在の要素の値にバインドされた **id** で *s* が実行される。もし *s* が例外を生起すれば、あるいはもし *o* が例外を生起すれば、*f* はその例外で完了する。そうでない場合は、このストリーム *o* 内で総てのイベントたちが処理され、*f* は **null**(16.2)で完了する。

u を直ちに包含している非同期 for ループ、あるいは発生器関数(9)のどれかに結び付けられたストリームだとしよう。もし更なる *u* のイベント e_u が *s* の実行が完了するより前に発生したときは、e_u の取り扱いは *s* が完了するまで待たねばならない。

*future f とそれに対応する **await** 式は、ある非同期 for ループが開始しその for 文のあとでそれを再開することを確保する。これらはまた包含している非同期 for ループのストリームがこのループの時間は停止することを確保している。*

非同期 for-in 文が同期関数(9)の内側で出現したときはコンパイル時エラーである。従来の for ループ(17.6.1)に **await** キーワードが付されているときはコンパイル時エラーである。

await 式が同期関数内では意味がないと同じ意味合いで同期関数内での非同期ループは意味がない。

17.7 While

while 文は、その条件がそのループの前に計算された条件による繰り返しをサポートする。

whileStatement(**while 文**)**:**
    while '(' expression(式) ')' statement(文)
;

while (*e*) *s*;の形式の while 文の実行は以下のように進行する:

式 *e* がオブジェクト *o* として計算される。次に *o* はブール変換(16.4.1 節)の対象になり、あるオブジェクト *r* をつくる。もしその *r* が **true** なら、次に文{*s*}が実行され、そして次にこの while 文が繰り返し的に再実行される。*r* が **false** なら、その while 文の実行は終了する。

e の型が **bool** に代入出来ない可能性があるときは静的型警告となる。

17.8 Do

do 文は、その条件がそのループの後に計算された条件による繰り返しをサポートする。

doStatement(**do 文**)**:**
    do statement(文) **while** '(' expression(式) ')' ';'

;

do *s* **while** (*e*);の形式の do 文の実行は以下のとおり進行する:

文{s}が実行される。次に、式 *e* がオブジェクト *o* として計算される。次に *o* はブール変換（16.4.1 節）の対象になり、あるオブジェクト *r* をつくる。*r* が **true** なら、次にこの do 文が繰り返し的に再実行される。*r* が **false** なら、その while 文の実行は終了する。

e の型が **bool** に代入出来ない可能性があるときは静的型警告となる。

17.9 Switch

*スイッチ文(switch statement)*は多数のケースたち間への制御を振り分けをサポートする。

switchStatement（switch 文）:
switch '(' expression（式） ')' '{' switchCase（スイッチ・ケース）* defaultCase（デフォルト・ケース）? '}'
;

switchCase（スイッチ・ケース）:
label（ラベル）* (**case** expression（式） ':')+ statements（文たち）
;

defaultCase（デフォルト・ケース）:
label（ラベル）* **default** ':' statements（文たち）
;

switch (*e*) {
case $label_{11}$ ... $label_{1j_1}$ e_1: s_1
…
case $label_{n1}$ ..$label_{nj_n}$ e_n:s_n
default: s_{n+1}
}

または

switch (*e*) {
case $label_{11}$ ... $label_{1j_1}$ e_1: s_1
…
case $label_{n1}$ ..$label_{nj_n}$$e_n$: s_n
}

の形式の switch 文において、総ての 1 <= *k* <= *n* に対し式 e_k がコンパイル時定数でないときはコンパイル時エラーである。

式 e_k の値が以下のどれでもないときはコンパイル時エラーである:

- 総ての 1 <= *k* <= *n* に対し同じクラス *C* のインスタンスである、または
- 総ての 1 <= *k* <= *n* に対し int を実装したクラスのインスタンスである、または
- 総ての 1 <= *k* <= *n* に対し String を実装したクラスのインスタンスである

言い換えると、そのケースたちの式の総てが定数の整数として計算される、あるいはその総てが定数の文字列たちとして計算される。式たちの値たちがコンパイル時に判っており、どの静的型アノテーションたちとは独立していることに注意のこと。

その式の値が文字列、整数、リテラル・シンボル、またはクラス Symbol の定数コンストラクタを呼びいだした結果でない限り、**Object** から継承したもの以外は、クラス C が演算子==を実装しているときはコンパイル時エラーである。

ユーザ定義の対等性に対するこの禁止により我々はユーザ定義の型たちの為にこのスイッチを効率的に実装できるようになる。我々はその代り同じ効率性で対等性に関しマッチングを形成できよう。しかしながら、ある型が対等性演算子を定義していると、プログラマたちは対等なオブジェクトがマッチしないということで驚くことになろう。

switch 文は非常に限られた状況（例えばインタプリタまたはスキャナ）に限られるべきである。

switch (*e*) {
  case $label_{11}$ ... $label_{1j_1}$ e_1: s_1
  …
  case $label_{n1}$ ..$label_{nj_n}$ e_n:s_n
  default: s_{n+1}
}

または

switch (*e*) {
  case $label_{11}$ ... $label_{1j_1}$ e_1: s_1
  …
  case $label_{n1}$ ..$label_{nj_n}$ e_n: s_n
  }

の形式の switch 文の実行は次のように進行する:

文 **var** *id* = *e*; が計算される。ここに *id* はその名前が該プログラム内のどの変数とも区別される変数である。チェック・モードでは、もし *e* の値が定数たち e_1 ... e_n と同じ型のインスタンスで無いときは実行時エラーである。
もし case **句**(*n = 0*)**が存在しないときは** *e* **の型は問題でないことに注意。**

次に、case 句 **case** e_1: s_1 がもし存在すれば実行される。もし case 句 **case** e_1: s_1 が存在しなければ、次にもし **default** 句があればそれは s_{n+1} を実行することで実行される。

case 句は、それを包含する構文的スコープ内でネストした新たなスコープをもたらす。この case 句スコープはその case 句の文の直後で終了する。

Switch 文

switch (*e*) {
case *label11 ... label1j1 e1*: *s1*
…
case *labeln1 ..labelnjn en*:*sn*
default: *sn+1*
}

の case 句 **case** *ek*: *sk* の実行は以下のように進行する:

式 *ek* == ***id*** が計算されオブジェクト *o* が得られ、これが次にブール変換により値 ***v*** となる。

もし ***v*** が **true** でないときは、もし有ればそれに続く *case* である ***case*** *ek+1: sk+1* が実行される。もし ***case*** *ek+1: sk+1* が存在しなければ、次にその **default** 句が *sn+1* を実行することで実行される。

もし ***v*** が **true** のときは、*h* を *h* >= *k* であって s_h が非空であるような最小の整数だとする。もし *h* が存在しないときは、 *h* = *n* + *1* としよう。次に式のシーケンス s_h が実行される。もし実行が s_h の後の点に達したら、*h* = *n* + *1* で無い限り実行時エラーが発生する。

Switch 文

switch (*e*) {
case *label11 ... label1j1 e1*: *s1*
…
case *labeln1 ..labelnjn en*: *sn*
}

の case 句 **case** *ek*: *sk* の実行は以下のように進行する:

式 *ek* == *id* が計算されオブジェクト *o* が得られ、これが次にブール変換により値 ***v*** となる。

もし ***v*** が **true** でないときは、もし有ればそれに続く *case* である ***case*** *ek+1: sk+1* が実行される。

もし ***v*** が **true** のときは、*h* を *h* >= *k* であって s_h が非空であるような最小の整数だとする。もし存在すれば式のシーケンス s_h が実行される。もし実行が s_h の後の点に達したら、*h* = *n* で無い限り実行時エラーが発生する。

言い換えれば、case たち間の暗示的なフォール・スルー(fall-through)は存在しない。switch の最後の case（デフォルトまたはその他）はその分の最後に「フォール・スルー」でき得る。

e の型が e_k の型に代入出来ない可能性があるときは静的警告である。文のシーケンスの最後の文 *sk* が **break**、**continue**、**return** または **throw** でないときは静的警告である。

スイッチ・ケースのこの振る舞いはこれまでのC のものとは意図的に異なったものになっている。暗示的なフォール・スルーはプログラミング・エラーの要因として知られており、従って許されていない。どうして明示的なコードを要求するのではなくて各ケースの最後に暗示的にフローをブレークさせないのか？これは実際よりすっきりしている。またこれは各ケースが単一（複合も可）の文を持つようにさせたほうがよりすっきりするだろう。我々は他の言語からのswitch 文のポートを促進させるためにそうしないことを選択した。case の最後での暗示的に制御をブレークされることはポートされたフォール・スルーを使っているコードの意味を黙って変更させ、潜在的にそのプログラマに掴まえ難いバグに対処することを強いることになる。我々の設計ではその相違が即座にそのコードの作成者が確実に気が付くようにしている。そのプログラマは直線的な制御フローを終了させる文でケースを終了させることを忘れたらコンパイル時にそれが通知されることになる。我々はこの警告をコンパイル時エラーを出すことができるだろうが、そのプログラマに対しコードのポート中に直ちにこの問題に対処させることを強いたくない為、そうしなことにしている。もし開発者がその警告を無視し彼らのコードを実行させたら、実行時エラーがそのプログラムがデバッグが極めて難しいやり方の間違った振る舞いをするのを防ぐことになる（少なくともこの問題に関しては）。

フォール・スルー分析の洗練化はもうひとつの問題である。当面は我々は非常に率直な文法的要求をすることを選択している。それでもこれらのシンプルな規則たちを満たさないのにコードがフォール・スルーしない明らかな状況があり、例えば:

```
switch (x) {
 case 1: try { … return;} finally { … return;}
}
```

case 句内の非常に練ったコードはいずれにしても悪いスタイルであり、そのようなコードは常に手を加えられ得る。

以下の総ての条件が成り立つときは静的警告となる:

- その switch 文がデフォルト句を持っていない
- *e* の静的型が要素 $id_1, \ldots, id_n$を持った列挙型である
- セット$\{e_1, \ldots, e_k\}$とセット$\{id_1, \ldots, id_n\}$が同じでない

言い換えると、ある enum 上の switch 文が徹底していないときは警告が出される。

17.10 Rethrow

rethrow 文は例外を再生起するのに使われる。

rethrowStatement（rethrow 文）:
rethrow
;

rethrow 文の実行は以下のように進行する:

*f*を直ちに包含している関数だとし、**on** *T* **catch** (p_1, p_2)を直ちに包含している catch 句(17.11 節)だとしよう。

***rethrow** 文は常に **catch** 句のなかで出現し、どの **catch** 句も **on** T **catch** (p_1, p_2)の形式の何らかの **catch** 句と等価である。したがって我々は **rethrow** はその形式の **catch** 句の中に包含されていると仮定できる。*

現行の例外(16.9 節)が p_1 にセットされ、現行の戻り値(17.12 節)が未定となり、アクティブなスタック・トレース(17.11 節)が p_2 にセットされる。

もし*f*が **async** または async*とマークされており、現行の活性化によって導入された動的に包含している例外ハンドラ(17.11 節)が存在するときは、制御は *h* に渡され、そうでないときは*f*は終了する。

非同期関数の場合は、動的に包含している例外ハンドラはその関数の中でのみ意味を持つ。もし例外はこの関数のなかで捕捉されていないときは、例外ハンドラたちを介した伝搬ではなくて future または stream を介して例外値はキャンセルされる。

In the case of an asynchronous function, the dynamically enclosing exception handler is only relevant within the function. If an exception is not caught within the function, the exception value is channelled through a future or stream rather than propagating via exception handlers.

それ以外では制御は最も内側で包含している例外ハンドラに渡される。

制御に関するこの変更は、これらの関数が catch または finally 句（その双方が動的に包含する例外ハンドラを導入する）を介した例外を捕捉しないときは、複数の関数が終了してしまう結果をもたらす可能性がある。

もしある rethrow 文が on-catch 句のなかで包含されていないときはコンパイル時エラーである。

17.11 Try

*try 文(try statement)*は構造化されたやり方での例外処理コードの定義をサポートする。

TryStatement（try 文）:

try block（ブロック） (onPart（オン部）+ finallyPart（fainally 部）? | finallyPart（fainally 部）)
;

onPart（オン部）:

on type catchPart（キャッチ部）?block（ブロック）
;

catchPart（キャッチ部）:
catch ' (' identifier（識別子） (',' identifier)? ')'
;

finallyPart（fainally 部）:
finally block（ブロック）
;

try 文は少なくとも次のひとつが後に付いたブロック文で構成される:

1. **on-catch** 節たちのセットで、その各々が処理される例外オブジェクトの型、ひとつまたは 2 つの例外パラメタたち、及びひとつのブロック文を指定（明示的または暗示的に）している。

2. ブロック文で構成される **finally** 節

*この文法は既存の Javascript プログラムとの上位互換性の為に作られている。**on** 句はオミットでき、Javascript の catch 句のように見せることが出来る。*

on *T* **catch** (*p*1, *p*2) *s* の形式の on-catch 句は、*o* の型が *T* の副型ならオブジェクト *o* との一致を取る。もし *T* が奇形または後回し型（19.1 節）の時は、一致を取ろうとすると実行時エラーを起こす。

無論もし *T* が後回しの型または奇形型（19.1 節）のときは静的警告である。

on *T* **catch** (*p*1, *p*2) *s* の形式の on-catch 句は新しいスコープ *CS* をもたらし、その中で *p*1 および *p*2 で指定されたローカル変数たちが決まる。ステートメント *s* は *CS* の中に包含される。*p*1 の静的型は *T* で *p*2 の静的型は **StackTrace** である。

on *T* **catch** (*p*1) *s* の形式の on-catch 句は **on** *T* **catch** (*p*1, *p*2) *s* と等価である。ここに *p*2 はこのプログラムのどこにも他に生じない識別子である。

catch (*p*) *s* の形式の on-catch 句は **on dynamic catch** (*p*) *s* という on-catch 句と等価である。

catch (*p*1, *p*2) *s* の形式の on-catch 句は **on dynamic catch** (*p*1, *p*2) *s* という on-catch 句と等価である。

*アクティブ・スタック・トレース(active stack trace)*とは、現在の例外（16.9 節）がスローされた場所で実行がまだ終了していない現行アイソレート内でのまさしく関数の活性化たち(function activations)の記録である文字列を生み出す toString()メソッドを持たオブジェクトである。

このことはこのトレースには合成的な関数活性化は付加できないし、ソース・レベルでの活性化もオミットされないことを意味する。このことは、例えば、最適化のためとしての関数たちのインライン化はこのトレースからは見えないことを意味する。同じようにその

実装で使われている合成的ルーチンたちはこのトレース内に出現してはならない。

このトレース内でどのようにネーティブな呼び出しが表現されるかに関しては何も規定されていない。

我々はスタック・トレースの同一性に関しても、またスタック・トレースたちの対等性の概念に関してもなにものべていないことに注意されたい。

場所(position)という用語は行番号と解釈されるべきではなく、むしろ詳細な場所即ちその例外を生起させた」式の正確な文字の場所と解釈すべきである。

try 文 **try** s_1 on - $catch_1$ ... on - $catch_n$ **finally** s_f は、以下のように実行される例外ハンドラ *h* を定義する:

on-catch 節たちは順番に調べられ、$catch_1$から始まり、現在の例外（12.9 節）と一致する **catch** 節が見つかるまで、あるいは **on-catch** 節たちのリストを総て調べおわるまで調べる。もし **on-catch** 節 *on - $catch_k$* が見つかったら、次に現在の例外に p_{k1} がバインドされ、定義されていれば現在のスタック・トレースに p_{k2} がバインドされ、そして $catch_k$ が実行される。もし **on-catch** 節が見つからないときは、**finally** 節が実行される。次に、実行がその try 文の最後で再開される。

finally 節 **finally** *s* は以下のように実行する例外ハンドラ h を定義する:

r を現行の戻り値(17.12 節)だとしよう。次にこの現行の戻り値が未定(undefined)となる。h の動的スコープ内で実行している非同期 for ループ(17.6.3 節)と yield-each 文 (17.16.2 節)に結び付けられたオープンなストリームたちはキャンセルされる。

***for** ループにより何らかの理由でエスケープしたオープンのままのストリームたちは関数終了時にキャンセルされが、なるべく早く取り消すことが最善である。*

次に **finally** 句が実行される。*m* を即座に包含している関数だとしよう。もし *r* が定まっておればつぎに現行の戻り値が *r* にセットされ、次に:

- *m* のなかの **finally** 句で定義されている動的に包含しているエラー・ハンドラ *g* が存あ在すれば、制御は *g* に渡される。
- それ以外の場合は *m* は終了する。

それ以外の場合は、実行がその try 文の最後で再開される。

try 文 *t* の **on-catch** 節 **on** *T* **catch** (p_1, p_2) *s* の実行は次のように進行する:

t の **fainally** 節で定義された例外ハンドラの動的スコープ内で文 *s* が実行される。次に、現在の例外と現在のスタック・トレースの双方が未定(undefined)となる。

try 文の **finally** 節 **finally** *s* の実行は以下のように進行する:

x を現行の例外だとし、t がアクティブなスタック・トレースだとしよう。そうすると現行の例外とアクティブなスタック・トレースの双方が未定となる。文 s が実行される。つぎに、もし x が定義されておれば、あたかも **catch** (v_x, v_t)の形式の catch 句で包含されている rethrow 文(17.10)によるごとく再スローされる。ここに v_x と v_t は x（及び t）にバインドされた新鮮な変数である。

try 文 **try** s_1 on - $catch_1$ ... on - $catch_n$ **finally** s_f;の実行は以下のように進行する:

その try 文で定義された例外ハンドラの動的スコープ内で文 s_1 が実行される。次に、**finally** 節が実行される。

on-catch 節たちのどれかが実行されるかどうかは s_1 によって該当する例外が発生されたかどうか（throw 文の仕様を参照のこと）による。

もし s_1 がある例外を発生させていたら、それはその try 文のハンドラに制御を移し、そのハンドラが上記に規定したように on-catch 節たちを順番に一致するかを調べる。一致が見つからないときは、そのハンドラは finally 節を実行することになる。

もし一致する catch 節が見つかったら、それが最初に実行され、次に finally 節が実行される。

on-catch 節の実行中に例外が生起されたら、それは finally 節の為のハンドラに制御を移し、この場合は同様に finally 節が実行される。

もし例外が発生しなかったら、その finally 節も実行される。finally 節の実行はまた例外を発生し得、それが次の包含しているハンドラに制御を移させることになる。

try s_1 $on\text{-}catch_1$ ... $on\text{-}catch_n$の形式の try 文は **try** s_1 $on\text{-}catch_1$ ... $on\text{-}catch_n$ **finally** {}という文と等価である。

17.12 Return

*return 文(return statement)*は結果を同期関数の呼び出し側に返し、非同期関数に結び付けられた future を完了させ、あるいは発生器（第 9 章）に結び付けられた stream または iterable を完了させる。

returnStatement（return 文）:
return expression（式）? ';'
;

finally 句の為に、**return** の詳細な振る舞いは少し複雑になっている。ある **return** 文が返すはずの値が実際に返されるかどうかは、その **return** を実行する際に効果を与える **finally** 句の振る舞いに依存する。**finally** 句は別の値を返す、あるいは例外を生起する、あるいは制御フローを別の **return** あるいは **throw** にリダイレクトすることさえも選択できる。

return 文の総てが実際行うことは、その関数が終了したときに返すことが意図されている値をセットすることである。

現行の戻り値(current return value)は与えられた関数の活性化に固有なユニークな値である。本仕様書で明示的に設定されていない限りこれは未定である。

return 文 **return** *e*;の実行は次のように進行する:

最初に式 *e* が計算され、オブジェクト *o* が作られる。つぎに:

- 現行の戻り値が *o* にセットされ、現行の例外(16.9)とアクティブなスタック・トレース(17.11)が未定となる。
- c をもしあれば最も内側に包含されている **try-finally** 文(17.11)の **finally** 句だとしよう。もし *c* が定まっておれば *h* を *c* によって持ち込まれたハンドラだとする。もし *h* が定義されておれば制御は *h* に渡される。
- それ以外のときは現行のメソッドの実行が終了する。

最もシンプルなケースでは直ちに包含している関数は通常の、同期の非発生器であり、関数の終了に伴うもので、現行の戻り値は呼び出し側で与えられる。別の可能性は関数が async とマークされているもので、この場合は現行の戻り値はその関数呼び出しで結び付けられた future の完了に使われる。これらの双方のシナリオは 16.14 節で規定されている。包含する関数は発生器（言い換えれば async*または sync*）としてマークできない。何故なら発生器は以下に論ずるように return *e*; の形式の文を含むことが許されないからである。

T を *e* の静的型で、*f* を直ちに包含している関数だとしよう。

もし *f* のボディが **async** とマークされ、型 **Future<flatten(T)>** (16.29 節)が *f* の宣言された戻りの型に代入できない可能性があるときは静的型警告である。それ以外の時は、もし *T* が *f* の宣言された戻りの型に代入できない可能性があるときは静的型警告である。

S を *o* の実行時型だとしよう。チェック・モードにおいては:

- もし *f* のボディが **async** (第 9 章)とマークされているときは、もし *o* が **null** (16.2 節)でなくまた **Future**<*S*>が *f* の実戻り型(19.8.1 節) の副型でないときは動的型エラーである。
- そうでないときは、もし *o* が **null** でなくまた *o* の実行時の型が *f* の実戻り型の副型でないときは動的型エラーである。

もし **return** *e*;の形式の **return** 文が生成的コンストラクタ(10.6.1 節)のなかに出現したときはコンパイル時エラーである。

あるコンストラクタに factory プレフィックスを付加するのを忘れ偶発的にファクトリを生成的コンストラクタに変えてしまうことは容易に起きえることである。静的チェッカはこれらのケースの一部（全部

では無い)で型の不一致を検出できよう。上記規則はそのようなそうでなければ認識が非常に困難なエラーを捕捉しやすくしている。そうすることの実害は、生成的コンストラクタからの値を返すことは意味がないので、存在しない。

もし **return** *e*;の形式の **return** 文が発生器関数のなかに出現したときはコンパイル時エラーである。

発生器関数の場合はその関数によって返される値は iterable またはそれに結び付けられた stream であり、個々の要素は yield 文を使ってその iterable に付加され、従って値を返すことは意味がない。

f を **return** *e*;の形式の **return** 文を直ちに包含している関数だとしよう。もし *f* が発生器でも生成的コンストラクタでもなく、また以下のどちらかの場合は静的警告である。

- *f* が同期で *f* の戻りの型が **void** (19.7 節)に代入できない可能性がある、または、
- *f* が非同期で *f* の戻りの型が **Future<Null>**に代入できない可能性がある。

従って、戻りの型は dynamic になり、dynamic は void にあるいは Future<Null>に代入できるので、f が宣言された戻りの型を持っていなくても静的警告は出されない。しかしながら、戻りの型を宣言している同期の非発生器関数は明示的に式を返さねばならない。

従って、これはユーザが return 文のなかで値を返すのを忘れるような状況を捕捉するのに寄与する。

*非同期の非発生器は常に何らかの future を返す。もし式が与えられていないときはその future は **null** で完了し、このことが上記要求の要因になっている。*

async でマークされた関数の戻りの型を空白のままにすると、それは常に dynamic と解釈され、無論型エラーは起きない。Future あるいは Future<Object>を使いことも無論受け付けられるが、それ以外の型は null が副型を持たないので警告を引き起こす。

式を持たない **return**;文は以下のように実行される:

もし即座に包含している関数 *f* が発生器なら、次に:

- 現行の戻り値に **null** がセットされる。
- *c* を最も内側で包含する **try-finally** 文の **finally** 句だとしよう。もし *c* が定義されているなら、*h* を *c* によって導入されるハンドラだとしよう。もし *h* が定義されているなら、制御は *h* に渡される。
- それ例外では現行メソッドの実行は終了する。

それ以外では、もしこれがメソッド、ゲッタ、セッタ、あるいはファクトリの内部で起きているなら、この **return** 文は **return null**;文によって実行される。そうでないときはこの **return** 文は必然的に生成的

コンストラクタの内部で起きており、この場合は **return this;**を実行することで実行される。

return;があたかも return e;によって実行されるにもかかわらず、生成的コンストラクタ内で return;の形式の文を含めることは静的警告ではないことを理解することが重要である。これらの規則は特定の構文形式 return e; にのみ関連している。

*このように**return;**を構成するという動機は、総ての関数呼び出しが実際ある値を返すという基本的な要求から来ている。関数呼び出したちは式たちであり、我々は式の中に void 関数を使うことを常に禁止する為に義務的な型チェッカ(typechecker)に依存することはできない。従って return 文は例え式が指定されていなくてもある値を返さねばならない。*

*そうすると疑問が出てくる、return 式が与えられていないときは return 文はどんな値を返すべきであるか。生成的コンストラクタ内では、明らかにそれは生成中のオブジェクト**(this)** である。void 関数たちのなかでは我々は null を使う。void 関数はある式に参加することは予定されていない、だからそれは最初に **void** とマークされている。従って、この状況は間違いでありなるべく早く検出されねばならない。ここでは静的規則が寄与するが、もしそのコードが実行されていると、**null** の使用が早めの失敗をもたらし、このケースではそれが好ましい。同じことが全く return 文を含まない関数ボディたちにも適用される。*

もしある関数が **return;**の形式の return 文を一つ以上含んでおり、同時にまた **return e;**の形式の return 文を一つ以上含んでいるときは静的警告である。

17.13 ラベル(Labels)

*ラベル(label)*はその後にコロン(:)が付いた識別子である。*ラベルがついた文(labeled statement)*はラベルである *L* が頭に付いた文である。*ラベルが付いた case 節(labeled case clause)*は *L* が頭に付いた switch 文（17.9 節）内の case 節である。

ラベルの唯一の役割は break（17.14 節）及び continue（17.15 節）文の為のターゲットを提供することである。

label（ラベル）:
identifier（識別子） ':'
;

ラベルが付いた文 *L*: *s* の意味は、文 s のそれと同一である。ラベルたちの名前空間(namespace)は型、関数及び変数に使われるものとは全く別のものである。

文 *s* にラベルを付すラベルのスコープは *s* である。switch 文 *s* の case 節にラベルを付すラベルのスコープは *s* である。

プログラマたちは万難を排してラベルを回避すべきである。この言語にラベルを入れた動機は主と

してDart をコード生成のより良いターゲットにすることである。

17.14 Break

*break 文(break statement)*は予約語の **break** とオプショナルとしてのラベル（17.13 節）で構成される。

breakStatement（break 文）:
　break identifier（識別子）? ';'
;

s_bを **break** 文だとする。もし s_bが **break** *L*;の形式なら、次に s_Eがラベル *L* を持った s_bを包含する最も内側のラベル付き文だとする。もし s_bが **break**;の形式なら、s_Eを s_bを包含する最も内側に存在する **do**（17.8 節）、**for**（17.6 節）、**switch**（17.9 節）または **while**（17.7 節）文だとする。そのなかで s_bが起きる最も内側の関数のなかにそのような文または case 節 s_Eが存在しないときはコンパイル時エラーである。更に、s_1... s_nを s_bを包含する s_Eのなかに共に包含されたそれらの **try** 文たちで、**finally** 節を持っているとする。最後に、f_jを s_j, 1 <= *j* <= *n* の **finally** 節だとする。s_bの実行は最初に最も内側の節を最初にする順番(innermost-clause-first order)で f_1 ... f_nを実行し、次に s_Eを終了させる。

もし s_Eが非同期 for ループ(17.6.3 節)の時は、それに結び付けられたストリーム加入(stream subscription)は取り消される。更に a_k*(1 <= k <= m* 、ここに a_kは a_{k+1}に包含されている*)*を、s_E内に包含されている s_bを包含している非同期 for ループと yieldeach 文(17.16.2 節)のセットだとしよう。a_j*(1 <= j<= m)* に結び付けられているストリーム加入たちは、a_jが a_{j+1}よりも先に取り消されるよう、最も内側を最初にして取り消される。

17.15 Continue

*continue 文(continue statement)*は予約語の **continue** とオプショナルとしてのラベル（17.13 節）で構成される。

continueStatement（continue 文）:
　continue identifier（識別子）? ';'
;

s_cを **continue** 文だとする。もし s_cが **continue** *L*;の形式なら、次に s_Eが最も内側に存在する do（17.8 節）、for（17.6 節）または while（17.7 節）とラベルされた文、あるいは s_cを包含する最も内側のラベル付き case 文だとする。もし s_cが **continue**;の形式なら、s_Eを s_cを包含する最も内側に存在する do（17.8 節）、for（17.6 節）または **while**（17.7 節）文だとする。そのなかで s_bが起きる最も内側の関数のなかにそのような文または case 節 s_Eが存在しないときはコンパイル時エラーである。更に、s_1... s_nを s_bを包含する s_Eのなかに共に包含されたそれらの **try** 文たちで、**finally** 節を持っているとする。

最後に、f_jを s_j, 1 <= j <= n の **finally** 節だとする。s_cの実行は最初に最も内側の節を最初にする順番(innermost-clause-first order)でf_1 ... f_nを実行し、次にs_Eを終了させる;そうでないときはs_Eは必然的にループであり、実行はそのループ・ボディの最後の文の後で再開される。

while ループ内では、それはそのボディの前のブール式になる。do ループではそのボディの後のブール式になる。for ループでは、それは増減分句になる。言い換えれば、実行はそのループの次の繰り返しにむかって継続する。

もしs_Eが非同期 for ループ(17.6.3 節)の時は、a_k(1 <= k <= m)を、s_E内に包含されているs_bを包含している非同期 for ループと yieldeach 文(17.16.2 節)のセットだとしよう。a_j(1 <= j<= m) に結び付けられたストリーム加入(stream subscription)は、a_jが a_{j+1}よりも先に取り消されるよう、最も内側を最初にして、取り消される。

17.16 Yield と Yield-Each（Yield and Yield-Each）

17.16.1 Yield

*yield 文(yield statement)*は発生器関数（第 9 章）の結果にある要素を付加する。

yieldStatement（yield 文）:
    yield expression（式） ';'
;

yield *e*; の形式の文 *s* の実行は以下のとおりである:

最初に、式 *e* が計算されオブジェクト *o* が得られる。もし包含する関数 *m* が **async*** (第 9 章)とマークされ、*m* に結び付けられたストリーム *u* がポーズしていたとすると、m の実行は *u* が再開されるかキャンセルされるまで保留(suspended)にされる。

次に、その *o* は直ちに包含している関数に結び付けられた iterable または stream に付加される。

もし包含する関数 *m* が **async*** (第 9 章)とマークされ、*m* に結び付けられたストリーム *u* がキャンセルされているとし、次に *c* を最も内側に包含している try-finally 文の **finally** 句(17.11 節)だとする（もしあれば）。もし *c* が定義されていれば、*h* を *c* がもたらすハンドラだとする。もし *h* が定義されていないときは、直ちに包含している関数は終了する。

*非同期発生器に結び付けられたストリームはその発生器が非活性化された(passivated)どの場所においてもそのストリームを参照しているどのコードによっても取り消され得る。そのような取り消しはその発生器の繰り返しのエラー(irretrievable error)を構成する。この時点において、その発生器にとって唯一のポーズ可能なアクションはその **finally** 句を介してそれ自身の後でクリーンアップす*

ることである。

それ以外のとき、もし包含する関数 *m* が **async*** (第 9 章)とマークされているときは、包含している関数は保留できる。

*もしある **yield** が無限ループの内部で生じ、包含している関数が決して保留されないときは、該包含しているストリームの消費者(consumer)たちが走り該ストリーム内のデータにアクセスする機会がなくなり得る。該ストリームは際限のないない数の要素たちを蓄積してしまう。そのような状況は受け入れがたい。従って我々は、ある新しい値がその結びいつけられたストリームに付加されたときに包含する関数が保留となることを許している。しかしながら、各 yield 上の関数を保留とすることは不可欠ではない(そしてそれは実際コストがかかり得る)。実装に際しては包含する関数をどれだけ頻繁に保留するかを決めることは自由である。唯一の要求は、消費者たちは無限にブロックされないということである。*

もしこの包含する関数 m が **sync*** (第 9 章)とマークされていると:

- *s* を直ちに包含している関数 *m* の実行は、*m* の現行の呼び出しを開始させるのに使われた繰り返し子(iterator)によってメソッド **moveNext**()が呼ばれるまで保留にされる。
- **moveNext**()への現行の呼び出しが **true** を返す。

もし yield 文が発生器関数でない関数の中で出現したらコンパイル時エラーである。

T を *e* の静的型、*f* を直ちに包含している関数だとしよう。次のどれかの場合は静的型警告である:

- *f* のボディが **async*** とマークされ、型 **Stream**<*T*> が *f* の宣言された戻りの型に代入できない可能性があるとき
- *f* のボディが **sync*** とマークされ、型 **Iterable**<*T*> が *f* の宣言された戻りの型に代入できない可能性があるとき

17.16.2 Yield-Each

yield-each 文は発生器関数(第 9 章)の結果にある一連の値を付加する。

yieldEachStatement(yieldEach 文):
yield* expression(式) ';'
;

yield* *e*; の形式の文 *s* の実行は以下のとおりである:

最初に、式 *e* が計算されオブジェクト *o* が得られる。もし直ちに包含する関数 *m* が同期関数なら、次に:

1. *o* のクラスが ***Iterable*** を実装していないときは動的エラーである。
2. *o* に基づきメソッド **iterator** が呼ばれあるオブジェクト *i* を返す。
3. *i* のメソッドが引数なしで呼ばれる。もし **moveNext** が **false** を返せば *s* の実行は完了する。そうでないときは、
4. *i* 上でゲッタ **current** が呼ばれる。もしこの呼び出しが例外 *ex* を生起するときは、*s* の実行は *ex* をスローする。そうでないときは、このゲッタ呼びだしの結果である *r* が *m* に関連付けられた iterable に付加される。直ちに s を包含する関数 m の実行は、現行の m の呼び出しを開始するのに使われた iterator にもとづき引数なしのメソッド **moveNext()**が呼ばれるまで保留となり、その時点で実行はステップ 3 に続く。
5. **moveNext()**への現行の呼び出しが **true** を返す。

もし m が **async*** (第 9 章)とマークされておれば次に、

- *o* のクラスが **Stream** を実装していないときは動的エラーである。そうでないときは、
- *o* の各要素 *x* にたいし:
  - もし *m* に結び付けられたストリーム *u* がポーズした状態の時は、*m* の実行は *u* が再開または取り消されるまでは保留となる。
  - もし *m* に結び付けられたストリーム *u* がキャンセルされておれば、次に次に *c* を最も内側に包含している try-finally 文の **finally** 句(17.11 節)だとする(もしあれば)。もし *c* が定義されていれば、*h* を *c* がもたらすハンドラだとする。もし *h* が定義されていないときは、直ちに包含している関数は終了する。
  - そうでないときは、*m* で結び付けられた stream に *x* がそれが *o* のなかで出現した順で付加される。関数 *m* は保留し得る。

もし yield-each 文が発生器関数でない関数の中で出現したらコンパイル時エラーである。

T を *e* の静的型、*f* を直ちに包含している関数だとしよう。型 *T* が *f* の宣言された戻りの型に代入できない可能性があるときの場合は静的型警告である。もし *f* が同期なら、もし *T* が **Iterable** に代入できない可能性があるときは静的型警告である。もし *f* が同期なら、もし *T* が **Stream** に代入できない可能性があるときは静的型警告である。

17.17 Assert

*assert 文(assert statement)*は与えられたブール条件が成立しないときに通常の実行を中断する為

に使われる。

AssertStatement（assert 文）:
assert '(' conditionalExpression（条件式） ')' ';'
;

生産モードでは assert 文は効果を持たない。チェック・モードでは assert 文 **assert**(*e*);の実行は以下のように進行する:

条件式 *e* がオブジェクト *o* として計算される。もし *o* のクラスが **Function** の副型のときは、*r* を引数なしで *o* を呼び出した結果だとしよう。そうでないときは、*r* を *o* としよう。もし *o* が型 **bool** または型 **Function** でないときは動的エラーである。もし *r* が **false** なら、我々はその表明(assertion)が失敗したという。もし *r* が **true** なら、我々はその表明が成功したという。もしその表明が成功したら、その assert 文の実行は終了する。もしその表明が失敗したら、**AssertionError** がスローされる。

e の型が **bool** または() → **bool** のどれかとして代入出来ない可能性があるときは静的型警告である。

どうしてこれが組込み関数呼び出しでなくて文なのだろうか？何故ならこれがマジック的に処理されており、従って生産モードでは効果もオーバヘッド（処理負荷）ももたらさないからである。また final メソッドがないときは、それがオーバライドされる（それには実施の害は無いが）のを防止出来ない。全体として、多分これは関数として定義できようし、オーバヘッド問題は最適化としてみることが出来よう。

18.ライブラリとスクリプト(Libraries and Scripts)

Dart のプログラムはひとつあるいはそれ以上のライブラリたちで構成され、ひとつまたはそれ以上のコンパイル単位*(compilation units)*で組み立てられる。コンパイル単位はライブラリまたはパート（17.3 節）である。

ライブラリはインポートたちのセット（空のこともあり）、及びトップ・レベルの宣言たちのセットで構成される。トップ・レベルの宣言はクラス（第 10 章）、型エイリアス宣言（19.3.1 節）、関数（第 9 章）あるいは変数宣言（第 9 章）のどれかである。ライブラリ *L* のメンバ*(members)*たちは、*L* 内で与えられているこれらのトップ・レベルの宣言たちである。

topLevelDefinition（トップ・レベル定義）:
classDefinition（クラス定義）
typeAlias（型エイリアス）
| **external**? functionSignature（関数シグネチュア） ';'
| **external**? getterSignature（ゲッタ・シグネチュア） ';'
| **external**? setterSignature（セッタ・シグネチュア） ';'
functionSignature（関数シグネチュア） functionBody（関数ボディ）
| returnType（戻り型）? **get** identifier（識別子） functionBody（関数ボディ）
| returnType（戻り型）? **set** identifier（識別子） formalParameterList（仮パラメタリスト） functionBody（関数ボディ）
| (**final** | **const**) type（型）? staticFinalDeclarationList（静的 final 宣言リスト） ';'
| variableDeclaration（変数宣言） ';'
;

getOrSet（get または set）:
get
| **set**
;

libraryDefinition（ライブラリ定義）:
scriptTag（スクリプト・タグ）? LibraryName?（ライブラリ名） importOrExport* partDirective（part 指令）* topLevelDefinition（トップ・レベル定義）*
;

scriptTag（スクリプト・タグ）:
'#!' (~NEWLINE)* NEWLINE
;

libraryName（ライブラリ名）:
metadata（メタデータ） **library** identifier（識別子） ('.' identifier)* ';'
;

importOrExport（インポートまたはエクスポート）:
　libraryImport（ライブラリ・インポート）
| libraryExport（ライブラリ・エクスポート）
;

ライブラリは*明示的に名前がついている(explicitly named)*か、または*暗示的に名前が付いている(implicitly named)*かであっても良い。名前付きのライブラリは **library** という語（何らかの適用可能なメタデータ・アノテーションが先行する可能性あり）で始まり、そのライブラリの名前をあたえる修飾名が続く。

技術的には、各ドットと識別子は分離したトーケンであり従ってこれらの間のスペースは受け付けられる。しかしながら、実際のライブラリ名は単純な識別子たちとドットたちの連結であり、スペースを含まない。

暗示的に名前が付いているライブラリはその名前として空の文字列を持つ。

ライブラリの名前は、それをそのライブラリの別々にコンパイルされた部品たち（parts と呼ぶ）に結びつける為に使われ、また印刷の為に使われ、そしてより一般的にはリフレクションの為に使われる。その名前は更なる言語発展（例えばファースト・クラスのライブラリ）にも関連することになろう。

幅広く使用されることを意図したライブラリたちの名前は衝突を避けねばならない。Dart の Pub パッケージ管理システムはそうするためのメカニズムを持っている。各 pub パッケージ唯一無二であることが保証され、従ってグローバルな名前空間を施行している。

ライブラリはオプショナルとしてスクリプト・タグで始まってもよい。スクリプト・タグはスクリプト（18.4 節）つきで使うことを意図したものである。スクリプト・タグはそのスクリプトが組み込まれている計算環境にたいしそのスクリプトの解釈を特定するために使える。スクリプト・タグは 2 文字の#!で始まりその行の終わりで終了する。Dart の実装物たちはこのスクリプトの#!のあとの文字たちを無視する。

ライブラリはプライバシの単位である。ライブラリ *L* 内で宣言された private 宣言は L 内のコードによってのみアクセス可能である。プライベート・メンバ宣言を *L* の外部からアクセスしようとすると実行時エラーが発生する。

トップ・レベルのプライベートたちはインポートされていないので、これらを使用することはコンパイル時エラーであり、ここでは問題ではない。

ライブラリ L の*パブリック名前空間(public namespace)*は、L の各トップ・レベルのメンバのシンプルな名前 m を m にマップするマッピングである。

ライブラリ L のスコープは L 内で宣言された総てのトップ・レベル宣言たちが導入した名前たち、及び L のインポート（18.1 節）によって付加された名前たちで構成される。

18.1 インポート(Imports)

*インポート指令(import directive)*は別のライブラリのスコープ内で使うライブラリを指定する。

import（インポート）:
metadata **import** importSpecification（import 仕様）
;

importSpecification（import 仕様）:
import uri (**as** identifier)? combinator* ';' |
import uri **deferred as** identifier（識別子） combinator（組み合わせ子）* ';'
;

combinator（組合せ子）:
show *identifierList*（識別子リスト）
| **hide** *identifierList*（識別子リスト）
;

identifierList（識別子リスト）:
identifier（識別子） (, *identifier*)*
;

*インポート(import)*はそのインポートされたライブラリの宣言が見つかる URI x を指定する。

インポートは後回し(deferred)または即座(immediate)であり得る。後回しのインポートは URI の後に組み込み識別子の **deferred** が出現することで識別される。後回しでないインポートは即座である。

即座のインポートの指定された URI があるライブラリ宣言を参照していないときはで無いときはコンパイル時エラーである。URI の解釈に関しては以下の 18.5 節で記されている。

後回しのインポートの指定された URI があるライブラリ宣言を参照していないときはで無いときは静的警告である。

*現行ライブラリ(current library)*は現在コンパイル中のライブラリのことを言う。import は現行ライブラリの import 名前空間を、インポートされたライブラリによって決まるやりかたで、そしてその import で与えられるオプショナルな引数に基づいて、変更する。

即座インポート指令 *I* はオプショナルに *I* によってインポートされた前置子名に対し使われた **as Id** の形式の前置子句(prefix clause)を含んでよい。後回しのインポートでは前置子を含んでいなければならず、でなければコンパイル時エラーを生じる。もし後回しのインポートで使われた前置子が別のインポート句のなかで使われているときはコンパイル時エラーである。

インポート指令 *I* はオプショナルに *I* によってインポートされた名前のセットを制限するのに使われた名前空間組み合わせ句(namespace combinator clauses)を含んでよい。現在、**hide** 及び **show** と

いう二つの名前空間組み合わせ句が対応されている。

I を文字列 s_1 を介してある URI を参照しているインポート指令だとしよう。*I* の計算は以下のように進行する:

もし *I* が後回しのインポートの場合は、計算は起きない。その代り、後回しの前置子オブジェクト (deferred prefix object)に対する前置子の名前 *p* のマッピングが *L* のスコープに付加される。

後回しの前置子オブジェクトは以下のメソッドたちを有する:

- **loadLibrary.** このメソッドは future の f を返す。呼ばれたらこのメソッドは将来いつか実行されることになる *I'*の即座のインポートを起こす。ここに *I'*は *I* から deferred という語を省き **hide loadLibrary** 句を付加することで得られる。*I'*がエラーなしで実行すれば *f* は完了する。*I'*がエラーなしで実行すれば、我々は **loadLibrary** の呼び出しが成功したと言い、それ以外の時は失敗したという。

- *L* のなかのトップ・レベルの *id* という名前の関数 *f* に対し、*f* と同じシグネチュアをもつ *id* という名前の対応するメソッド。これらのメソッドたちの呼び出しは実行時エラーをもたらす。

- *L* のなかのトップ・レベルの *id* という名前のゲッタ *g* に対し、*g* と同じシグネチュアをもつ *id* という名前の対応するゲッタ。これらのメソッドたちの呼び出しは実行時エラーをもたらす。

- *L* のなかのトップ・レベルの *id* という名前のセッタ *s* に対し、*s* と同じシグネチュアをもつ *id* という名前の対応するセッタ。これらのメソッドたちの呼び出しは実行時エラーをもたらす。

- *L* のなかのトップ・レベルの *id* という名前の各型 *T* に対し、戻りの型 *T* を持った *id* という名前の対応するゲッタ。このメソッドたちの呼び出しは実行時エラーをもたらす。

呼び出しが成功すれば、以下に示す如く名前 *p* が非後回し前置子オブジェクトに対しマップされる。加えて、この前置子オブジェクトはまた **loadLibrary** メソッドに対応しており、従って **loadLibrary** メソッドを再度呼ぶことが可能である。もし呼び出しが失敗すれば何も起きず、**loadLibrary** を再度呼ぶオプションがある。**loadLibrary** の繰り返しの呼び出しが成功するかどうかは、以下に示すように異なる。

loadLibrary の繰り返しの呼び出しの効果は以下のとおりである:

- もし別の *p*.**loadLibrary** が既に成功しておれば、繰り返しの呼び出しも成功する。それ以外の時は、

- もし別の *p*.**loadLibrary** が既に失敗しておれば:
  - その失敗がコンパイル・エラーによるものなら、その繰り返しの呼び出しは同じ理由で失敗する。
  - その失敗が別の理由によるものなら、その繰り返しの呼び出しはあたかも以前の呼び出しがなかったかのごとくふるまう。。

言い換えると、ネットワークの障害あるいはファイルが存在しないあとでの繰り返しのロードを再試行できるが、一旦何らかのコンテントを見つけそれをロードするときはもはや再ロードできない。

我々は future が返したどの値を解決するかは規定しない。

もし I が即座のインポートの時は、最初に、

- s_i の値である URI のコンテンツが未だ現在のアイソレートのなかでインポートまたはエクスポート指令（18.2 節）によりアクセスされていないときは、次にその URI のコンテンツがコンパイルされ、ライブラリ *B* がもたらされる。
  ライブラリたちは相互に再帰的なインポート物を持つことが許されるので、無限再帰を回避する為の注意が必要である。

- そうでないときは、s_i で示される URI のコンテンツが、現行アイソレート内であるライブラリ *B* に既にコンパイルされている。

NS_0 を *B* のエクスポートされた名前空間（17.2 節）としよう。そうすると、各組合せ句(combinator clause)C_i, $1 <= i <= n$, *in I* において：

- *Ci* が **show** $id_1, ..., id_k$ の形式であるときは、NS_i = **show**$([id_1, ..., id_k], Ns_{i-1})$とする。ここに **show***(l,n)*は識別子たちのリスト *l* と名前空間 *n* を引数とし、*l* の中の各名前を *n* が行うと同じ要素へのマッピングしそうでなければ決まらない名前空間を作り出す。更に、*l* の中の各名前 *x* にたいし、もし *n* が名前 *x*= を定義しているときは、新しい名前空間は *x*= を *n* が行っていると同じ要素にマッピングする。そうでないときは、結果としてのマッピングは定まらない。
- もし *Ci* が **hide** $id_1, ..., id_k$ の形式であるときは、NS_i = **hide**$([id_1, ..., id_k], Ns_{i-1})$とする。ここに **hide***(l,n)*は識別子たちのリスト *l* と名前空間 *n* を引数とし、*l* の中の各名前を *k* に対しては決まらないことを除いて *n* と同じ名前空間を作り出す。

次に、もし *I* が ***as*** *p* の形式の前置句を含んでいるなら、*NS* = *NSn* U {*p*:*prefixObject*(NS_n)}（ここに *prefixObject*(NS_n)は以下のメンバたちを持つオブジェクトである名前空間 NS_n の前置子オブジェクト）としよう：

- *NSn* のなかの各 *id* という名前のトップレベルの関数 *f* に対し、*f* に転送（9.1 節）する *f* と同じ名前とシグネチュアを有する対応するメソッド。

- *NSn* のなかの各 *id* という名前のトップレベルのゲッタ *g* に対し、*g* に転送（9.1 節）する対応するゲッタ。

- *NSn* のなかの各 *id* という名前のトップレベルのセッタ *s* に対し、*s* に転送（9.1 節）する対応するゲッタ。

- *NSn* のなかの各 *id* という名前の型 *T* に対し、呼ばれたときに *T* の為の型オブジェクトを返

す戻りの型 **Type** をもった *id* という名前の対応するゲッタ。

そうでないときは、$NS = NS_n$ としよう。もし現行ライブラリが *p* という名前のトップ・レベル・メンバを宣言しているときはコンパイル時エラーである。

つぎに、*NS* のなかの宣言 *d* にたいする各エントリ・マッピング・キー *k* にたいし、以下のいずれでも無い限り *L* のトップ・レベルのスコープ内で *d* が使えるようになる:

- *L* のなかに名前 *k* を持ったトップ・レベル宣言が存在する、または
- ***as*** *k* の形式の前置子句が *L* のなかで使われている。

これにより、メンバたちが自分たちのインポータたちを壊すことなくライブラリたちに付加されるようになる。

システム・ライブラリ(*system library*)は Dart 実装の一部であるライブラリである。その他のライブラリは*非システム・ライブラリ*(*non-system library*)である。

もし名前 *N* がライブラリ *L* によって参照されており、*N* が 2 つのライブラリ L_1 および L_2 のインポートによって *L* のトップ・レベルのスコープに導入され、エクスポートされた L_1 の名前空間がシステム・ライブラリのなかで始まっている宣言に *N* をバインドしているなら:

- L1 のインポートは暗示的に ***hide*** *N* 句によって拡張される。
- 静的警告が出される。

*通常の矛盾は配備時に解決されるものの、**dart:** ライブラリたちの機能は実行時にアプリケーションに注入され、ブラウザが更新されるなかで時とともに変化してゆく可能性がある。従って、**dart:** ライブラリたちとの矛盾は実行時に生じ得、デベロッパのコントロール外である。このような配備されたアプリケーションたちを動かなくすることを避けるために、**dart:** ライブラリたちは特別に扱われている。*

*配備される Dart コードが、すべてのインポートたちがあるライブラリの名前空間への追加が配備されているコードに決して影響を与えないように **show** 句を使うような出力を生成するようなツールが推奨される。*

もし名前 *N* がライブラリ *L* によって参照されており、*N* が 1 つ以上のインポートによりトップ・レベルのスコープに導入されているときは:

- 静的警告がだされる。
- もし *N* が函数、ゲッタ、またはセッタとして参照されているときは NoSuchMethod が生起される。
- *N* が型として参照されているときは、これは奇形の型として扱われる。

我々は名前空間 *NS* が *L* にインポートされているという。

もしNが2つまたはそれ以上のインポートによって導入されているものの決して参照されていないときはエラーにも警告にもならない。

上記ポリシーにより、ライブラリたちは自分たちのインポートたちになされた追加にたいしより堅固なものとなる。

このアプローチと、おそらくはクラスたちまたはインターフェイスたちに関する同じようなポリシーたちとの間で明確な区別がなされる必要がある。クラスまたはインターフェイス及びそのメンバたちの使用はその宣言とは分離している。使用と宣言はそのコードの中の全く離れた場所のなかで生じ得、実際別の人たちまたは組織によって書かれ得る。違反した宣言に対しエラーを発生させ、そのエラー受けた側が意味ある形で対処できるようにすることが重要である。

これに比べライブラリはインポートたちとそれらの使用で構成されている;そのライブラリは単一のパーティの管理下にあり、インポートがもとでの何らかの問題はたとえそれがユーザ側で報告されたとしても解決できる。

2 つの異なったライブラリたちを同じ名前でインポートするのは静的警告である。

広く普及しているライブラリは他のそのようなライブラリとぶつからない名前が与えられねばならない。そのための好ましいメカニズムは Dart のパッケージ・マネージャである pub であり、これはライブラあｔいのグローバルな名前空間と、その名前区間を活用するしきたりが用意されている。

ある名前空間に入っていない名前を隠す(hide)または示しても(show)エラーも警告もされないことに注意されたい。

これによりあるライブラリからある名前を削除したときクライアントのライブラリを破壊してしまうような状況が防止される。

インポートするライブラリが明示的に dart:core をインポートしていない限り、Dart のコア・ライブラリは暗示的に次の import 句のかたちで各 dart ライブラリにインポートされている:

import “dart:core”;

例え **show**、**hide**、または **as** で制限されているとしても、dart:core のインポートはこの自動インポートより優越する。

dart:core に関してなにも特別なものが無ければ良い。しかしながらその使用は普及しており、その結果それを自動的にインポートするという決定させている。しかしながら、何らかのライブラリ L が dart:core で使われている名前を持ったなにかを定義したい場合(これはあるライブラリで宣言された名前が優先されるので容易にできる)があるかもしれない。他のライブラリは L を使い、プレフィックスを使わずともまた警告を受けないでコア・ライブラリとぶつかる L のメンバたちを使いたい場合があるかもしれない。上記のルールではそれが可能であり、別の特別なルールにより本質的に

dart:core の特別な取り扱いをキャンセルしている。

18.2 エクスポート(Exports)

あるライブラリ *L* がある名前空間（6.1 節）をエクスポートするということは、その名前空間内の宣言たちは、もし他のライブラリたちが *L* をインポートすることを選択したら、それらの他のライブラリたちにとって使用できるということを意味する（17.1 節）。*L* がエクスポートした名前空間はその*エクスポートした名前空間(exported namespace)*として知られる。

libraryExport（ライブラリ・エクスポート）:
metadata（メタデータ） **export** uri（URI） combinator（組合せ子）* “;”

エクスポートはエクスポートされたライブラリの宣言が見つかる URI *x* を指定する。指定した URI がライブラリ宣言を参照していないときはコンパイル時エラーである。

その名前がそのライブラリがエクスポートした名前空間内にあるなら、我々はその名前はそのライブラリによって*エクスポートされている*（あるいは等価的には、ライブラリが*名前をエクスポート*している）という。もしその宣言がそのライブラリがエクスポートした名前空間内にあるのなら、我々はその宣言はそのライブラリによって*エクスポートされている*（あるいは等価的には、ライブラリが*宣言をエクスポート*している）という。

ライブラリはそのパブリックな名前空間内の総ての名前と宣言を常にエクスポートする。加えて、ライブラリはそのインポートされたライブラリたちのどれかを再エクスポートすることを選択できる。

E を文字列 s_1 を介したある URI を参照するエクスポート指令だとする。*E* の計算は以下のように進行する:

最初に、

- もし s_1 の値であるその URI が現行アイソレートの中でインポートあるいはエクスポートにより未だアクセスされたことがないときは、その URI のコンテンツがコンパイルされ、ライブラリ *B* が得られる。
- そうでないときは、s_1 で示されたその URI は現行アイソレート内で既にライブラリ *B* にコンパイルされてしまっている。

NS_0 が B のエクスポートされた名前空間だとする。そうすると、各組合せ子句 C_i, $1 <= i <= n$, in *E* に対し:

- もし Ci が **show** $id_1, ..., id_k$ の形式のときは、NS_i = **show**([$id_1, ..., id_k$], NSi_{-1})としよう。
- もし Ci が **hide** $id_1, ..., id_k$ の形式のときは、NS_i = **hide**([$id_1, ..., id_k$], NSi_{-1})としよう。

NS_n のなかの宣言 d に対する各エントリ・マッピング・キーにたいし、k という名前を持ったトップ・レベル宣言が L の中に存在しない限り、L のエクスポートされた名前空間に d に対するエントリ・マッピング k が追加される。

もし名前 N がライブラリ L によって参照されており、また N がその **URI** が **dart:** で始まるライブラリからのエクスポートによって、そしてその **URI** が **dart:** で始まらないライブラリからのエクスポートによってエクスポートされた L の名前空間導入されることになるライブラリ N によって参照されているときは:

- L_1 のエクスポートは隠された **hide** N 句によって暗示的に拡張される
- 静的警告が出される。

この規則の背景となっている理由に関しては 18.1 節のインポートの議論を参照のこと。

我々は L *がライブラリ* B *を再エクスポート(re-exports)*すると言い、また L は*名前空間* NS_n *を再エクスポート*するという。混乱が生じない場合は、我々は単に L が B を再エクスポートする、あるいは L が NS_n を再エクスポートすると言っても良い。

もし名前 N がライブラリ L によって再エクスポートされていて N が 1 つ以上のエクスポートにより L のエクスポート名前空間に持ち込まれているときはコンパイル時エラーである。2 つの異なったライブラリを同じ名前でエクスポートするのは静的警告である。

18.3 パート(Parts)

ライブラリは各々が別の場所にストアされ得るパーツ*(parts)*に分割されても良い。ライブラリは **part** 指令を介してそれらをリストすることで、ライブラリはそのパーツを識別する。

*パート指令(part directive)*は現在のライブラリに組み入れるべき Dart のコンパイル単位が見つかるであろう URI を指定する。

```
partDirective(part 指令):
  metadata(メタデータ) part uri(URI) ";"
  ;

partHeader(part ヘッダ):
    metadata part of qualified(修飾された)
    ;

partDeclaration(part 宣言):
    partHeader topLevelDefinition(トップ・レベル定義) * EOF
    ;
```

*part ヘッダ(part header)*は **part of** で始まり、そのパートが属するライブラリの名前が続く。part 宣言は part ヘッダで始まり、トップ・レベルの宣言たちの並びが続く。

part *s;*の形式の part 指令をコンパイルすることで、Dart のシステムは *s* の値である URI の中身のコンパイルを開始する。次にその URI のトップ・レベル宣言たちが、現行ライブラリのスコープの中で Dart コンパイラによりコンパイルされる。その URI の中身が有効な part 宣言でないときはコンパイル時エラーである。参照された part 宣言 *p* がそこに *p* が属するライブラリとして現行ライブラリ以外のライブラリを指定しているときは静的警告となる。

18.4 スクリプト(Scripts)

*スクリプト(script)*は、そのエクスポートされた名前空間（18.2 節）がトップ・レベル関数の main を含むライブラリである。スクリプト *S* は以下のように実行されよう：

最初に、*S* は上記で規定されているようにライブラリとしてコンパイルされる。次に、エクスポートされた *S* の名前空間にあるトップ・レベルの関数の **main** が呼び出される。もし **main** が位置的パラメタを持っていないときは、それは引数なしで呼び出される。そうでない場合もし **main** がまさしく一つの位置的パラメタを有するときは、それはその実行時型が **List<String>**を実装した単一の実引数で呼び出される。それ以外の時は **main** は次の 2 つの実引数で呼び出される：

1. その実行時の型が **List<String>**を実装したひとつのオブジェクト。
2. *i* を産み付けた **Isolate.spawnUri** の呼び出しで決まった現行アイソレート *i* の初期メッセージ。

もし *S* がトップ・レベルの関数の **main** を宣言またはインポートしていないときは実行時エラーである。もし **main** が 2 つ以上の要求されたパラメタを持っているときは静的警告である。もし **main** が 2 つ以上の要求されたパラメタをしているときは、実行時エラーが生じることに注意。

スクリプトの名前はオプショナルであり、インタラクティブで非公式な使用の為使われる。しかしながら長期的に価値(long term value)を持つスクリプトには名前を与えるのが良い行為である。名前付きスクリプトは構成可能(composable)である。

Dart プログラムは一般的にあるスクリプトを実行することで走る。

18.5 URI

URI は文字列リテラルの手段で指定される：

uri:
  stringLiteral（文字列リテラル）
;

URI を記述した文字列リテラル *x* がコンパイル定数で無いとき、または *x* が文字列内挿入を含んでいるときはコンパイル時エラーである。

本仕様書は以下の例外たちを除き URI の解釈に関しては記述していない。

URI の解釈は殆ど周辺コンピューティング環境に任されている。例えば、もし Dart がウェブ・ブラウザの中で走っているときは、そのブラウザは一部の URI たちを解釈することになろう。例えば URI は IETF RFC 3986 のような標準に基づいて解釈されるといったように規定することが魅力的に見えるかもしれないが、実際にはこれは通常そのブラウザに依存し、頼りにはならない。

dart:*s* の形式の URI は Dart の実装の要素であるあるシステム・ライブラリ（18.1 節）*s* への参照と解釈される。

package:*s* の形式の URI は実装が指定した場所に対し相対的な **packages**/*s* の形式の URI として解釈される。

この場所はしばしば Dart コンパイラに提示されたルート・ライブラリの場所となる。しかしながら、実装はこの選択をオーバライドまたは置き換える手段を提供しても良い。

その意図は、開発中に於いて、Dart のプログラマたちは自分たちのプログラムの要素を見つけるのをパッケージ・マネージャに依存することができることにある。そのようなパッケージ・マネージャは彼らが必要な Dart コード（あるいはそこへのリンク）を置く場所であるディレクトリ **packages** *で始まるディレクトリ構造を提供することになろう。*

そうでないときは、どの相対 URI も現行ライブラリの場所に対し相対だと解釈される。URI の更なる総ての解釈は実装に依存する。

これはそれは組込み者に依存するということを意味する。

19.型(Types)

Dart はオプショナルなインターフェイス型に基づく型づけをサポートしている。

この型システムは総称型の共変性(covariance)の為にしっかりしたものではない。これは慎重に考えるべき(そして間違いなく意見が割れる)選択である。経験は総称型のためのしっかりした規則はプログラマたちの直観とは真っ向から対立することを示している。彼らが望むならツールたちがしっかりした型分析を提供するのは容易で、これはリファクタリング(プログラムの内部の改良)のようなタスクには有用かもしれない。

19.1 静的型(Static Types)

静的型アノテーションは変数宣言(第 8 章)(仮パラメタ(9.2 節)を含む)の中と関数(第 9 章)の戻りの型の中で使われる。静的型アノテーションは静的チェックの最中とチェック・モードでプログラムが走っているときに使われる。これらの静的型アノテーションたちは生産モードではどんな効果も持たない。

type(型):
　typeName(型名) typeArguments(型引数)?
;

typeName(型名):
… qualfied(修飾された)

typeArguments(型引数):
　'<' typeList(型リスト) '>'
;

typeList(型リスト):
　type(型) (',' type(型))*
;

Dart の実装はこの仕様が静的警告として特定しているまさしくこれらの状況を検出し報告する静的チェッカを用意しなければならない。しかしながら:

- プログラム *P* 上で静的チェッカを走らせるのは、*P* のコンパイルと実行させるときには要求されていない。
- プログラム *P* 上で静的チェッカを走らせるのは、どの静的警告が生じるかに関わらず、*P* の成功裏のコンパイルを阻止させてはならないし、*P* の実行も阻止してはならない。

代替的な静的分析（例えば非異型（訳者注：Java のように共変しない）総称型、または実

際の宣言からの宣言ベースの分散を推論する、のいずれかといったしっかりしたやり方で既存の型を翻訳する）を実装した追加のツールをなにも排除しない。しかしながら、これらのツールを使うことは成功裏のコンパイルと Dart コードの実行を排除しない。

もし次が成り立てば型 T は奇形（malformed）である：

- T が id の形式または refix.id の形式で、包含する構文スコープ内にあり、名前 id（あるいは refix.id）がある型を示していない。
- T が包含する構文スコープ内のある型変数を示しているが、そのシグネチュア内または static メンバのボディ内で生じている。
- T が複数の inport 句からインポートされている宣言たちを示している。
- T が G<S1, .., Sn>の形式のパラメタ化された型で G が奇形である。

奇形型を使用すると静的警告が起きる。奇形の型は明示的に指定されていない限り次に静的型チェッカによって及び実行時に **dynamic** として解釈される。

これにより開発者たちはその奇形の型が他の型たちと関わりあうことによる一連の連鎖した警告から解放される。

p.T の形式の時に限り型 *T* は後回しにされる。ここに *p* は後回しの前置子である。型アノテーション、型テスト、型キャスト、あるは型パラメタとして後回しの型を使うのは静的警告である。しかしながら、その他の総ての警告は、総ての後回しのライブラリたちが成功裏にロードされたとの仮定のもとに出されるべきである。

19.1.1 型プロモーション(Type Promotion)

静的型システムでは各式に対する静的型をアーカイブしている。いくつかの例ではローカル変数と仮パラメタたちの型は制御フローに基づきそれらの宣言された型たちからプロモート（訳者注：たとえば Java でいえば int から long、double あるいは float）されている。

我々は v の型の上位変換を許すときはいつでも「変数 *v* は型 *T* を持つことがわかっている」という。型プロモーションがいつ許されるかの正確な状態は本仕様の関連する節（16.22、16.20 および 17.5 節）で示されている。

そのようなプロモーションがあるブール式の分析に基づいて有効であると我々が演繹できるときに限り、ある変数 v のプロモーションが許される。そのようなケースでは、我々はその「ブール式 *b* は *v* が型 *T* を有することを示している」という。一般的に、すべての変数 v と型 T に対し、あるブール式が v は型 T を持つということを示さない。ある式がある変数はある型を持っていることを示しているような状況は、本仕様書の関連した節のなかで明示的に示されている（16.33 および 16.21 節）。

19.2 動的型システム(Dynamic Type System)

Dart 実装は*生産モード(production mode)*と*チェック・モード(checked mode)*の双方での実行に対応しなければならない。特にチェック・モードで生じると規定された動的チェックたちは、そのコードがチェック・モードで実行されるときに限り実行されねばならない。

例え後回しの型が既にロードされてしまっている前置子に依存している場合でもこれが該当することに注意。Dart のプログラマたちが多くの時間をチェック・モードに費やすので、後回しの型が関与する型アノテーションを使うことを強く抑制してしまうので、これは遺憾なことである。

実際問題として、後回しのロードが含まれる多くのシナリオは積極的にロードされたインターフェイスを実装したクラスの後回しのロードを含むので、この状況はしばしばそう思われるほど煩わしいものではない。現行の意味づけは実装のしやすさを念頭に置いて採用されている。

明らかに、もしある後回しの型が未だロードされていない場合は、それを含んだ副型の正しいテストを行うことは不可能であり、型テストと型キャストでの場合値同様、、人は動的なエラーを期待しよう。同様な理由で、一旦ある型がロードされたらチェック・モードはシームレスに動作することが期待されよう。我々は将来これらの意味づけを採択したいと期待している;そのような変更は上位互換性がある。

チェック・モードにおいて、*奇形(malformed)*または*奇形バインド(malbounded:* 19.8 節*)*の型が副型テストで使われているときは動的型エラーである。

次のプログラムを調べて見よう:

```
typedef F(bool x);
f(foo x) => x;
main() {
  if (f is F) {
    print("yoyoma");
  }
}
```

f の仮パラメタの型は *foo* であり、これは構文スコープ内では宣言されていない。これは静的型警告をもたらそう。生産モードでこのプログラムを走らせると *yoyoma* と印刷されよう。何故なら *foo* は dynamic として取り扱われるからである。

別の例として次のコードを見てみよう：

```
var i;
i  j; //  a variable j of type i (supposedly)
main() {
    j = 'I am not an I';
}
```

i は型ではないので、*j* の宣言のところで静的警告が出される。しかしながら、このプログラムは生産モードでは未宣言の型 i は **dynamic** として取り扱われるので、エラーなしで実行される。しかしながらチェック・モードでは、代入時における暗示的な副型テストが実行時にエラーを発生させよう。

奇形バインド型が関与する事例を以下に示す:

```
class I<T extends num> {}
class J{}
class A<T> implements J,  I<T> // 型警告: T は num の副型ではない
{ ...
}
```

上記のように宣言されていたとすると、次の

```
I x = new A<String>();
```

は、この代入は *A<String> <: I* で副型テストが必要な為、チェック・モードでは動的型エラーを発生させる。これが成り立つことを示す為に、我々は *A<String>* ≪ *I<String>* であることを示さねばならないが、*I<String>* は奇形バインド型であり、動的エラーを発生させる。生産モードではエラーはスローされない。*J x = new A<String>();* はこの場合 *I<String>* にたいしテストの必要がないので動的エラーを発生させないことに注意。

同じように、生産モードでは、

```
A x = new A<String>();
bool b = x is I;
```

b は true にバインドされているが、チェック・モードでは 2 行目で動的型エラーが発生する。

19.3 型宣言(Type Declarations)

19.3.1 Typedef

型エイリアス(type alias) はある型式の為の名前を宣言する。

typeAlias（型エイリアス）**:**
    metadata（メタデータ） **typedef** typeAliasBody（型エイリアス・ボディ）
;

typeAliasBody（型エイリアス・ボディ）**:**

functionTypeAlias（関数型エイリアス）
;

functionTypeAlias（関数型エイリアス）:
functionPrefix（関数プレフィックス） typeParameters（型パラメタたち）?
formalParameterList（仮パラメタ・リスト） ';'
;

functionPrefix（関数プレフィックス）:
returnType（戻りの型）? identifier（識別子）
;

ライブラリ L の中で宣言された **typedef** T *id* (T_1 p_1, .., T_n p_n, [T_{n+1} p_{n+1}, …, T_{n+k} p_{n+k}])の形式の型エイリアスの効果は、L のスコープ内に関数型(T_1 p_1, .., T_n p_n, [T_{n+1} p_{n+1}, …, T_{n+k} p_{n+k}]) → T.にバインドされた名前 *id* を持ちこむことである。ライブラリ L の中で宣言された **typedef** T *id* (T_1 p_1, .., T_n p_n, {T_{n+1} p_{n+1}, …, T_{n+k} p_{n+k}})の形式の型エイリアスの効果は、L のスコープ内に関数型(T_1 p_1, .., T_n p_n, {T_{n+1} p_{n+1}, …, T_{n+k} p_{n+k}}) → T.にバインドされた名前 *id* を持ちこむことである。どちらの場合も戻りの型が指定されていないときは、それは **dynamic** となるトークンである。同様に、仮パラメタ上で型アノテーションがオミットされているときは、それは **dynamic** となるトークンである。

もし関数型エイリアスのシグネチュアのなかに何らかのデフォルト値が指定されているときはコンパイル時エラーである。

直接的、あるいは別の **typedef** を介した再帰的な、ある **typedef** のなかでの自己参照はコンパイル時エラーである。

19.4 インターフェイス型(Interface Types)

クラス I の暗示的なインターフェイスは次の場合にのみクラス J の暗示的インターフェイスのスーパー型(supertype)である:

- I が **Object** で J が **extends** 節を持っていないとき
- I が J の **extends** 節にリストされているとき
- I が J の **implements** 節にリストされているとき
- I が J の with 節の中にリストされているとき
- J が I のミクスインのミクスイン・アプリケーション（12.1 節）であるとき

もし以下の条件のひとつが満たされれば、型 T は型 S よりもより特定(specific)で、$T \ll S$ と書かれ

る:

- 再帰性(reflexivity): *T* が *S*
- *T* が bottom 型(訳者注: ⊥: 値を持たない空の型)
- *S* が **dynamic**
- 直接のスーパー型: *S* は *T* の直接のスーパー型
- *T* が型パラメタで *S* は *T* の上界(upper bound)
- *T* が型パラメタで *S* は **Object**
- 共変性(covariance): *T* が $I<T_1, ..., T_n>$の形式で *S* が $I<S_1, ..., S_n>$ の形式で且つ $T_i \ll S_i$, $1 <= i <= n$
- *T* と *S* がともに関数型で、19.5 節での規則のもとで *T* << *S* である
- *T* が関数型で *S* が **Function** である
- 他動性(transitivity): $T \ll U$ で $U \ll S$

≪は型たちの半順序(partial order)である。[bottom(⊥)/**Dynamic**]$T \ll S$ のときのみ *T* は *S* の副型で *T* <: *S* と書かれる。

<:は型たちの半順序ではなく、型たちのバイナリな関連だけであることに注意のこと。これは<:は他動的でないからである。もしこれがそうだったら、この副型規則はサイクルを持つことになろう。例えば、List <: List<String>及びList<int> <: List、しかしList<int>はList<String>の副型ではない。<:は型たちの半順序ではないものの、これは半順序、即ち≪を含む。このことは、生の型たちを除いて、クラシックな副型規則に関する直感が適用されることを意味する。

T が *S* の副型である時に限り、*S* は *T* のスーパー型で *S* :> *T* と書かれる。

あるインターフェイスのスーパー型たちは、その直接のスーパー型たちとそれらのスーパー型たちである。

T <: *S* または *S* <: *T* であるときに限り、型 *T* は型 *S* に代入でき、*T* ⇔ *S* と書かれる。

この規則はこれまでの型チェックになじんでいる読者たちを驚かすかもしれない。⇔関係の意図はある代入を確実に正しいものにする事ではない。むしろ、間違っていないかも知れない代入を排除することなく、これは間違っていることが殆ど確実だという代入にフラグをたてることを意図している。

*例えば、静的型**Object** の値を静的型**String** を持ったある変数に代入することは、正しいということは保障されないものの、もし実行時の値がたまたま文字列であれば構わないかもしれない。*

19.5 関数型(Function Types)

*関数型(Function Types)*には 2 つのバリエーションがある:

1. 位置的なパラメタたちのみを持つ関数の型。これらの一般的な形式は(T_1, ..., T_n[T_{n+1}, ..., T_{n+k}]) → T である。

2. 名前付きパラメタたちを持つ関数の型。これらの一般的な形式は(T_1, ..., T_n、[$T_{x1}x1$, ..., $T_{xk}xk$]) → T である。

関数型(T_1, ..., T_k、{T_{k+1}, ..., T_{m+n}}) → T は,、以下の全部の条件を満たせば関数型(S_1, ..., S_{k+j}、{S_{k+j+1}, ..., S_n}) → T の副型である。

1. 以下のいずれか:
   - S が **void**、または
   - $T \Leftrightarrow S$
2. 総ての i , 1 <= i <= n に対し $T_i \Leftrightarrow S_i$

以下の総ての条件が満たされるときは、関数型(T_1, ..., T_n, [T_{n+1}, …, T_{n+k}]) → T は関数型(S_1, ..., S_n, [$S_{y1}y_1$, …, $S_{ym}y_m$]) → S の副型である:

1. 以下のいずれか:
   - S が **void**、または
   - $T \Leftrightarrow S$
2. k>=m 及び総ての i , 1 <= i <= n+m に対し $T_i \Leftrightarrow S_i$

以下の総ての条件が満たされるときは、関数型(T_1, ..., T_n, {T_{x1} x_1, …, T_{xk} x_k}) → T は関数型(S_1, ..., S_n, {S_{y1} y_1, …, S_{ym} y_m}) → S の副型である:

1. 以下のいずれか:
   - S が **void**、または
   - $T \Leftrightarrow S$
2. 総ての i , 1 <= i <= n に対し $T_i \Leftrightarrow S_i$
3. k >= m 及び x_i = y_i, 1 <= i <= m
4. {y_1, …, y_m}のなかの総ての y に対し、$S_y \Leftrightarrow T_y$

加えて、以下の副型規則が適用される:

(T_1, ..., T_n, []) → T <: (T_1, ..., T_n) → T.
(T_1, ..., T_n) → T <: (T_1, ..., T_n, {}) → T.
(T_1, ..., T_n, {}) → T <: (T_1, ..., T_n) → T.
(T_1, ..., T_n) → T <: (T_1, ..., T_n, []) → T.

空のオプショナルなパラメタ・リストを持った関数を宣言するのは無効なので、神経質な読者はこれらの規則は意味がないと結論するかもしれない。しかしながら、これはオプショナルなパラメタを宣言していない関数の型とそうでない関数の型の有用な関連をもたらす。

$T <: S$ であるときに限り T は関数型 S に代入でき、$T \Leftrightarrow S$ と書かれる。

関数は常にクラス **Function** を実装しその関数と同じシグネチュアを持った **call** メソッドを実装した何らかのクラスのインスタンスである。総ての関数型は Function の副型である。もしある型 I が **call** という名前のメソッドを含んでいて、**call** の型が関数型 F であったとすると、I は F の副型として考えられる。もしある具体クラスが **Function** を実装しているものの call()という名前の具体メソッドを持っていないときは、そのクラスが **noSuchMethod**()のそれ自身の実装を宣言していない限り、静的警告となる。

以下の総ての条件が満たされれば、関数型$(T_1, ..., T_k, [T_{k+1}, ..., T_{n+m}]) \rightarrow T$ は$(S_1, ..., S$k+j, $[S_{k+j+1}, \ldots, S_n]) \rightarrow S$ よりもより特定化(specific)されている:

1.
   - S が **void** または
   - T << S

2. 総ての i , $1 <= i <= n$ にたいし $T_i \ll S_i$

3. $k >= m$ で y_i の $\{x_1, ..., x_k\}$ にたいし $1 <= i <= m$

4. すべての $\{y_1, ..., y_m\}$ の y_i に対し $T_j \ll S_i$

更に、もし F が函数型のときは $F \ll$ **Function**

19.6 dynamic 型(Type dynamic)

組み込み識別子の **dynamic** は未知(unknown)の型を意味する。

静的型アノテーションが用意されていないときは、型システムはその宣言は未知の型を持っているとみなす。ある総称型が使われていてそれに対する型引数たちが提供されていないときは、そのぬけた型引数には未知の型がデフォルトとなる。

このことは総称型宣言 $G<T_1, \ldots, T_n>$が与えられたとすると、型 G は $G<$dynamic, …, dynamic>と等価であることを意味する。

型 **dynamic** は各あり得る識別子およびアリティ(訳者注:関数や演算(子)に対しそれらが取る引数またはその個数)に対するメソッドを、名前つきパラメタたちの各可能性ある組み合わせで、持つ。これらのメソッドたちの総てがそれらの戻りの型として **dynamic** をもち、またその仮パラメタたちのす

べてが型 **dynamic** を持つ。

型 **dynamic** は各あり得る識別子に対する属性たちを持つ。これらの属性たちの総てが型 **dynamic** を持つ。

使い勝手の観点からすると、我々は未知の型が使われている何処においてもチェッカが絶対エラーを出さないようにしたい。上記定義により、未知の型をアクセスしたときに確実に第 2 のエラー(訳者注:また同じエラーを出すこと)が報告されないようになる。

現在の規則では欠けた型引数たちはあたかもそれらが型 ***dynamic*** *だとして扱われるとしている。代替手段はこれらは* **Object** *を意味すると考えることである。これはチェック・モードでのより早い段階でのエラー検出が得られ、また静的型チェック中により積極的なエラー検出が得られる。例えば:*

```
(1) typedAPI(G<String> g){...}
(2) typedAPI(new G());
```

代替規則では、(2)はチェック・モードでエラーとなる。これはエラーのローカル化の観点では好ましい物に見える。しかしながら、(2)で Dynamic エラーが起きたとき、実行を継続する唯一の方法は(2)を次のように書き換えることである:

```
(3) typedAPI(new G<String>());
```

これはユーザたちに単に型付けられた API を呼んでいるからとして自分たちのクライアント・コードに型情報を書き込むことを強いることになる。我々はこれを Dart のプログラマたちに課したいとは思わない。彼らの一部は一般には至福の状態で型たちにそしてとりわけ総称的に気が付かないかもしれない。

静的チェックに関してはどうだろうか?確かにユーザが明示的に静的型チェックを要請したら我々は(2)のフラグを立てることになろう。しかし現実は Dart の静的チェッカはデフォルトではバックグラウンドで走っている可能性が高い。技術チームたちは一般的に警告がでない「クリーン・ビルド」を望んでおり、従ってこのチェッカは極めて慈善的であるよう設計されている。他のツールたちが型情報をより積極的に解釈し、これまでの(そしてしっかりした)静的型規律の違反に対し警告できる。

dynamic という名前は、例え **dynamic** がクラスでなくても、ある **Type** オブジェクトを意味する。

19.7 型 void (Type Void)

特別な型である **void** は関数の戻りの型としてのみ使われる:その他のコンテキストのなかで **void** を使うのはコンパイル時エラーである。

例えば型引数として、あるいは変数またはパラメタの型として。
void はインターフェイス型ではない。

従って **void** に適した唯一の副型関係は:

- **void** <: **void**（再帰性により）
- bottom <: **void**（bottom 型は総ての型の副型故）
- **void** <: **dynamic**（**dynamic** は総ての型のスーパー型故）

従って、静的チェッカは誰かが void メソッド呼び出しの結果のメンバにアクセスしようとしたら警告を出す（たとえ==のような null のメンバでも）。同様に、void メソッドの結果をある変数にパラメタとしてまたは代入の為に渡すと、その変数/仮パラメタが Dynamic 型でない限り、警告が出される。

一方、void メソッドの中から void メソッドの結果を返すことは可能である。また null あるいは型 dynamic の値を返すことも可能である。なにか別の結果を返すと型警告（あるいはチェック・モードでは動的型エラー）がだされる。チェック・モードでは、もし void メソッドから非 null オブジェクトが返されたら、動的型エラーが生起されよう（どのオブジェクトも実行時型として dynamic を持たないから）。

void という名前は Type オブジェクトを意味しない。

*ある式として **void** を使うのは文法的に違反であり、そうする意味もない。Type オブジェクトたちはインスタンスたちの実行時の型を具体化する。どのインスタンスも **void** を持つことはない。*

19.8 パラメタ化された型たち（Parameterized Types）

*パラメタ化された型(parameterized type)*は総称型宣言の呼び出しである。

T をパラメタ化された型 $G<A_1, \ldots, A_n>$ だとする。もし *G* が総称型で無いときは、型引数たち $S_1, \ldots, S_n$ は捨てられる。もし *G* が *m != n* 個の型パラメタたちを持っていれば、*T* はそのすべてが **dynamic** である m 個の引数たちを持ったパラメタ化された型だとして取り扱われる。

手短にいえば、アリティの不一致は総ての型引数たちの破棄をもたらし、総てが dynamic にセットされた正しい数の型引数たちに置き換えられる。無論、静的警告がだされる。この振る舞いは未だ実装されていない。

そうでないときは、T_i を *G* の型パラメタたちだとし、B_i が T_i, $1 <= i <= n$ にバンドされているとする。もし S_i が奇形バインドあるいは S_i が $[S_1, \ldots, S_n/T_1, \ldots, T_n]B_i$, $1 <= i <= n$ で無いときのいずれかのときに限り T は奇形バインドである。

チェック・モードにおいて、奇形バインド型が 19.2 節の型テストに使われているときは動的型エラーであることに注意。

奇形バインド型が使われているときは静的警告が生起される。

G が *n* 個の型パラメタたちでアクセス可能な総称型宣言でないときは静的型エラーである。もし A_i, $1 <= i <= n$ が包含している構文スコープ内の型を意味しないときは静的型警告である。

もし *S* が *G* のメンバ *m* の型だとすると、$G<A_1, \ldots, A_n>$のメンバ *m* の静的型は$[A_1, \ldots, A_n/T_1, \ldots, T_n]S$ で、ここに $T_1, \ldots, T_n$ は G の仮型パラメタたちである。B_i を T_i, $1 <= i <= n$ のバインドたちとしよう。もし A_i が$[A_1, \ldots, A_n/T_1, \ldots, T_n]B_i$, $1 <= i <= n$ の副型でないときは、静的型警告である。

19.8.1 宣言の実際の型 Actual Type of a Declaration()

以下のときに限り*型 T は型変数 U に依存する*:

- *T* は *U* である
- *T* はパラメタ化された型で、*T* の型引数たちのひとつが *U* に依存する

プログラム・ソースにでてくるように、宣言 *d* の宣言された型を *T* だとしよう。*d* の*実際の型(actual type)*は:

- もし *T* が型宣言 $U_1, \ldots, U_n$ に依存し、A_i が U_i, $1 <= i <= n$ の実型のときは、$[A_1, \ldots, A_n/U_1, \ldots, U_n]T$
- それ以外は *T*

19.8.2 最小上界(Least Upper Bounds)

2 つのインターフェイス *I* と *J* があったとし、S_I を *I* のスーパーインターフェイスたちのセット、S_J を *J* のスーパーインターフェイスたちのセットとし、$S = (I\ S_I)\ (J\ S_J)$とする。更に、我々は任意の有限の *n* に対する $S_n = \{T \mid T\ S\ \ depth(T) = n\}$と $k = max(depth(T_1), ..., depth(T_m))$、$T_i\ S$, $1 <= i <= m$ を定義する。ここに *depth*(*T*)は *T* から **Object** に至る最短継承パスのステップ数である。*q* を S_q が基数(cardinality)1 を持つような最小数としよう。*I* と *J* の最小上界は単一の要素 S_q である。

dynamic と型 *T* の最小上界は **dynamic** である。**void** と型 *T* != **dynamic** の最小上階は **void** である。*U* を上界 *B* を持った型変数だとしよう。*U* と型 *T* の最小上界は *B* と *T* の最小上階である

最小上界の関連は対象的(symmetric)であり再帰的(reflexive)である。

関数型とインターフェイス型 *T* の最小上界は **Function** と *T* の最小上界である。*F* と *G* を関数型だとしよう。もし *F* と *G* がその数と必要なパラメタで異なっていれば、*F* と *G* の最小上界は **Function** で

ある。そうでないときは:

- もし
  $F= (T_1 \ldots T_r, [T_{r+1}, \ldots, T_n]) \rightarrow T_0$ および
  $G= (S_1 \ldots S_r [S_{r+1}, \ldots, S_k]) \rightarrow S_0$
  ここに $k <= n$
  そうすると F と G の最小上界は
  $(L_1 \ldots L_r, [L_{r+1}, \ldots, L_k]) \rightarrow L_0$
  ここに L_i は T_i および S_i, $1 <= i <= k$ の最小上界である。

- もし
  $F= (T_1 \ldots T_r, [T_{r+1}, \ldots, T_n]) \rightarrow T_0$ および
  $G= (S_1 \ldots S_r \{\ldots\}) \rightarrow S_0$
  そうすると F と G の最小上界は
  $(L_1 \ldots L_r) \rightarrow L_0$
  ここに L_i は T_i および S_i, 0 <= i <= r の最小上界である。

  もし
  $F= (T_1 \ldots T_r, , \{T_{r+1}\ p_{r+1}, \ldots, T_f\ p_f\}) \rightarrow T_0$ および
  $G= (S_1 \ldots S_r, \{\ S_{r+1}\ \ q_{r+1}, \ldots, S_g\ q_g\}) \rightarrow S_0$
  次に $\{x_m, \ldots x_n\} = \{p_{r+1}, \ldots, p_f\}\ \ \{q_{r+1}, \ldots, q_g\}$ とし、また X_j が F と G m <= j <= n のなかの型たち x_j の最小上界だとしよう。
  そうすると F と G の最小上界は
  $(L_1 \ldots L_r, \{\ X_m\ x_m, \ldots, X_n\ x_n\}) \rightarrow L_0$
  ここに L_i は T_i および S_i 0 <= i <= r の最小上界である。

20.参照(Reference)

20.1 構文規則(Lexical Rules)

Dart のソース・テキストは Unicode コード・ポイントたちの並びで表現される。この並びは最初に本仕様書で与えられている構文規則に基づきトークンの並びに変換される。このトークン化のプロセスのどこにおいても、最長可能なトークンは認識される。

20.1.1 予約語(Reserved Words)

予約語は以下のとおりである（訳者注：15.32 節の組込み識別子も参考のこと）：

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

LETTER（文字）：
'a'..'z'
| 'A'..'Z'
;

DIGIT（数字の桁）：
'0'..'9'
;

WHITESPACE（ホワイトスペース）：
('\t' | ' ' | NEWLINE（改行）)+
;

20.1.2 コメント(Comments)

*コメント(comments)*はドキュメンテーションに使われるプログラムの区間たちである。

SINGLE_LINE_COMMENT（単行コメント）：
'//' ~(NEWLINE（改行）)* (NEWLINE（改行）)?
;

MULTI_LINE_COMMENT（複行コメント）：

'/*' (MULTI_LINE_COMMENT（複行コメント） | ~ '*/')* '*/'
;

Dart は単行と複行のコメントの双方に対応している。*単行コメント(single line comment)*はトーケン//で始まる。//とその行の終わりまでの総ては Dart のコンパイラによって無視されねばならない。

*複行コメント(multi-line comment)*はトーケン/*で始まりトーケン*/で終わる。/*と*/の間はそのコメントがドキュメンテーション・コメントでない限りなんでも Dart のコンパイラによって無視されねばならない。

コメントはネスト（入れ子に）できる。/**で始まる

*ドキュメンテーション・コメント (documentation comments)*はトーケン///または/**で始まる複行コメントである。ドキュメンテーション・コメントは人が読めるようなドキュメンテーションを作り出すツールによって処理されることを意図したものである。

ドキュメンテーション・コメントのスコープは常に包含しているライブラリのインポートされた名前空間を含めない。包含しているライブラリのなかで宣言されている名前たちのみがドキュメンテーション・コメントのスコープ内にあるとみなされる。

あるクラス *C* の宣言の直前にあるドキュメンテーション・コメントのスコープは、*C* のインスタンス・スコープであり、包含しているライブラリのインポートされた名前空間を介して導入されるどの名前たちも含まない。

関数 *f* の宣言の直前にあるドキュメンテーション・コメントのスコープは、*f* のボディ部の開始点で効力を持つスコープであり、包含しているライブラリのインポートされた名前空間を介して導入されるどの名前たちも含まない。

20.2 演算子の順位(Operator Precedence)

演算子の順位は本文法書により暗示的に与えられている。

以下の非基準の表が有用であろう:

記述(Description)	演算子(Operator)	結合則(Associativity)	優先度(Precedence)
単項後置 (Unary postfix)	. , ?id, e++, e--, e1[e2], e1(), ()		16
単項前置 (Unary prefix)	-e, !e, ~e, ++e, --e		15
乗除	*, /, ~/, %	左	14

(Multplicative)			
加減 (Additive)	+, -	左	13
シフト (Shift)	<<, >>	左	12
ビット AND (Bitwise AND)	&	左	11
ビット XOR (Bitwsie XOR)	^	左	10
ビット OR (Bitwise Or)	|	左	9
関係 (Relational)	<, >, <=, >=, **as**, **is, is!**	なし	8
等価 (Equality)	==, !=	なし	7
論理 AND (Logical And)	&&	左	6
論理 OR (Logical Or)	||	左	5
If-null	??	左	4
条件式 (Conditional)	e1? e2 : e3	なし	3
カスケード (Cascade)	..	左	2
代入 (Assignment)	=, *=, /+, +=, =+, -=, ~=, %=, <<=, >>>=, >>=, &=, ^= etc.	右	1

注意： 仕様書 0.70 版から Relational と Equality の順位が Bitwise OR の下に下がっている。

21.付録:名前付け規約(Naming Conventions)

Dart においては以下の付名規約が一般的である：

- コンパイル時定数変数の名前は小文字を使わない。もしそれらが複数の単語で構成されているときは、それらの単語をアンダスコアで分離させる。例： PI, I_AM_A_CONSTANT

- 関数（ゲッタ、セッタ、メソッド、及びローカル及びライブラリ関数）と非定数変数の名前は小文字で始まる。もしそれらが複数の単語で構成されているときは、各単語（最初を除く）は大文字で始まる。それ以外には大文字は使わない。例：camlCase, , dart4TheWorld

- 型（クラス、型変数、及び型エイリアス）の名前は大文字で始まる。もしそれらが複数の単語で構成されているときは、各単語は大文字で始まる。それ以外には大文字は使わない。例：CamlCase, Dart4TheWorld

- 型変数の名前は短くする（一文字が好ましい）。例：T, S, K, V , E

- ライブラリ、またはライブラリ・プレフィックスの名前は決して大文字を使わない。もしそれらが複数の単語で構成されているときは、それらの単語をアンダスコアで分離させる。例： my_favorite_library

22.参考（訳者追加）

22.1 和英対照表

本邦訳は基本的には下表にもとづいている。検索は各自のドキュメント閲覧ソフトウエアのツールを利用されたい。

日本語	英語	訳者注
運用モード	production mode	プログラムが安定化されたあとでの実際の運用に使われるモード。開発時のチェック・モード(checked mode)と対比される
演算子	operator	被演算子をひとつ（単項演算子）か二つ（二項演算子）持つ演算子
加算演算子	additive operator	加算および減算の演算子
複合代入演算子	compound assignment operator	
単項演算子	unary operator	
積算演算子	multiplicative operator	積算、除算、および剰余の演算子
否定演算子	negate operator	
開発モード	checked mode	プログラム開発時のモードで、型アノテーションに基づき型チェックが行われる。チェック・モードとも書く
型	type	
副型	subtype	サブタイプともいう
スーパー型	supertype	通常は super type と書く。スーパータイプとも訳す
型エイリアス	type alias	typedef を使って関数を変数型または戻り型として使うことができる
型テスト	type test	is 演算子を使ってオブジェクトの型があっているかを調べる
型プロモーション	type promotion	Java でいえば char->int byte->short->int->long->float->double といった上位の型への変換
実行時型	runtime type	実行時の型
可変	mutable	生成後値が変更可能なオブジェクト
仮パラメタ	formal parameter	
関数	function	Dart では関数はクラス内のメソッドたちに加えてライブラリ関数などトップ・レベルの関数を含めたものとして定義している

関数宣言	function declaration	
関数リテラル	function literal	
関数式	function expression	関数リテラルと同じ
境界	bound	バウンドとも訳す
上界	upper bound	
最小上界	least upper bound	
警告	warning	
静的警告	static warning	静的チェッカが出す警告
静的型警告	static type warning	静的チェッカが出す型に関する警告
計算する	evaluate	場合によっては評価するとも訳す。式の評価とはその式の値を計算する過程である
継承	inheritance	
構文	lexical	綴りまたは字句ともいう
式	expression	
常数式	constant expression	
代入可能式	assignable expression	
修飾または修飾された	qualified	ピリオド'.'でオブジェクトを指定すること
修飾名	qualified name	ある分類系中の位置が分るような仕方で名付けられたデータ名あるいは、複数の単純名をピリオドで連結した文字列
未修飾	unqualified	修飾されていない
定数	constant	
定数リスト・リテラル	constant list literal	
定数変数	Constant variable	const で宣言された変数
識別子	identifier	
初期化	initialization	
後回し初期化	lazy initialization	アプリケーションの起動を早くする為に、コンパイル時定数でない静的変数の初期化をコンパイル時ではなくて読み出し時におこなう方式
初期化子	initializer	イニシャライザとも呼ぶ
正規化	canonicalize	
静的	static	static とそのまま使うこともあり（予約語の場合など）
節	clause	
総称型	generics	汎用体ともいう
総称型	generic type	インスタンス化する際実際の型指定を行う汎用体
具象化された総称型	reified generics	Java と違い実際の型引数に関する情報が実行時に保持される

型引数	type argument	'<' typeList (型リスト) '>'で型を指定
属性	property	
代入	assignment	場合によっては割り当てと訳す。変数の現在の値を新しい値で置き換える演算
動的	dynamic	実行中に型の判断と処理がなされる方式
名前付きコンストラクタ	named constructor	var b = new Point.zero();のようなコンストラクタ。指名コンストラクタともいう
名前付きパラメタ	named parameter	指名パラメタともいう
名前付け規約	naming convention	付名規約とも訳す
名前空間	namespace	あるスコープ内で有効な名前
名前空間組合せ子	namespace combinators	#include 指令で使われる prefix と show
エクスポートされた名前空間	exported namespace	
インポート名前空間	import namespace	
パブリック名前空間	public namespace	
発生器	generator	そのボディが sync*または async*とマークされた関数等
引数	argument	関数やメソッドに渡す実際のパラメタ
名前付き引数	named argument	日本語では指名パラメタ、英語では keyword argument などともいう。関数呼び出し自身の中の各パラメタの名前を明確に示した関数呼び出し(例えば xPosition:20 など)をその言語が対応している場合に使う
不変	immutable	生成後に値が変わらないオブジェクト
文	statement	
文法構文	grammar production	
並行処理性	concurrency	並行性ともいう
変数	variable	
型変数	type variable	型にわたる変数
インスタンス変数	instance variable	クラス宣言のなかにすぐに含まれていて、**static** 宣言されていない変数
静的変数または static 変数	static variable	特定のインスタンスに結び付けられていない変数、またはクラス宣言のなかにすぐに含まれていて **static** 宣言されている変数(ふたつの意味を持つことに注意)。static 変数とも訳す。
可変変数	mutable Variable	値が変更できる変数
定数変数	constant variable	**const** を付して宣言された定数
文字列	string	

文字列内挿入	string interpolation	文字列補間、文字列内挿あるいは文字列インターポレーションとも訳す。文字列の中に式を入れ込みその式の計算結果をその文字列に連結する操作
戻り値	return value	関数が返す戻り値
現行戻り値	current return value	
例外	exception	
現行例外	current exception	
インポート	import	
インポート名前空間	import namespace	
イニシャライザ	initializer	初期化子とも訳されている
インスタンス	instance	具現化物
新規インスタンス	fresh instance	その識別がそのクラスにこれまでに割り当てられたインスタンスのどれとも区別されるインスタンス
インスタンス化	instantiate	具現化ともいう
インスタンス・メンバ	instance members	あるクラスがインスタンス化されたときのそのクラスの変数やメソッド等の要素
インスタンス・メソッド	instance method	クラスの中に含まれ **static** と指定されていないメソッド
インスタンス変数	instance variable	クラスの中に含まれ **static** と指定されていない変数
インターフェイス	interface	
暗示的インターフェイス	implicit interface	
スーパーインターフェイス	superinterface	通常は super interface と書く
エラー	error	
コンパイル時エラー	compile-time error	
実行時エラー	run time error	
動的型エラー	dynamic type error	
オーバライド	override	上書き再定義
オーバロード	overload	多重定義ともいう
クラス	class	
スーパークラス	superclass	通常は super class と書く
クラス変数	class variable	static 宣言されたクラス内変数
クロージャ	closures	ある関数がそれを包含する親の関数のスコープ内の変数を参照するような関数
ゲッタ	getter	クラスの属性（通常_を付けて private とする）のセッタやゲッタ（通常同じ属性の_を外す）を定義することで、Dart はそのオブジェクトの同じ名前の属性を参照するときにこのセッタまたはゲッタを呼び出す
コンストラクタ	constructor	

リダイレクト・コンストラクタ	redirect constructor	
名前付きコンストラクタ	named constructor	指名コンストラクタともいう。var b = new Point.zero(); のようなコンストラクタ
生成的コンストラクタ	generative constructor	
定数コンストラクタ	constant constructor	
潜在的常数式	potentially constant expression	
コード点	code point	code position ともいう。文字コード表の位置
シグネチュア	signature	メソッド等の定義に使われる狭い意味の構文
スコープ	scope	その変数等が有効で適用され得る範囲、可視域などともいう
構文スコープ	lexical scope	静的スコープともいう。構文構造からのみで決定できるスコープ
セッタ	setter	ゲッタを参照のこと
チェック・モード	checked mode	正確にはチェックド・モードとすべきだが、ここではチェック・モードとする。開発時に使われる実行モード
トップ・レベル	top level	クラスのメンバでない関数や変数など、つまりそのライブラリ内のどこからも可視な領域に対し使う
パラメタ	parameter	関数やメソッドの引数として必要なパラメタ。引数とは区別される
仮パラメタ	formal parameter	この仕様書では総ての仮パラメタをまとめて formals と表現されていることもある
仮型パラメタ	formal type parameter	型の仮パラメタ
名前つきパラメタ	named parameter	`['と`]'で挟まれたデフォルトの仮パラメタ。指名パラメタとも呼ぶ
位置的仮パラメタ	positional formal	名前付きと対比される。呼び出しパラメタの位置でその値が所定の変数に代入される
名前つきオプショナル仮パラメタ	named optional formal	`['と`]'で挟まれたデフォルト値を持つ仮パラメタ
バインディング	bind	オブジェクトと名前との関連付け、オブジェクトへの参照を取得すること。name binding ともいう
バウンド	bound	境界ともいう
ブロック	block	スコープを指定する
マップ	map	名前と値のペアの集まりからなるオブジェクト
マップ・リテラル	map literal	
定数マップ・リテラル	constant map literal	
実行時マップ・リテラル	run-time map literal	
ミクスイン	mixin	サブクラスによって継承されることにより機能を提供

		し、単体で動作することを意図しないクラス。メソッドを持ったインターフェイスともいえる。
ミクスイン構成	mixin compsition	複数のミクスインからなる構成
メソッド	method	Dart ではクラス内で定義された関数をメソッドと読んでいる
インスタンス・メソッド	instance method	クラスの中に含まれ **static** と指定されていないメソッド
抽象メソッド	abstract method	**abstract** と指定されたインスタンス・メソッドで実装が用意されていない
スタティック・メソッド	static method	クラスの中に含まれ **static** と指定されているメソッド。staatic メソッドとも訳す
ライブラリ	library	
ライブラリ関数	library function	トップ・レベルにある（つまりクラスのメンバでない）関数。トップ・レベル関数とも呼ぶ
ライブラリ変数	library variable	トップ・レベルにある（つまりクラスのメンバでない）変数。トップ・レベル変数とも呼ぶ
現行ライブラリ	current library	現在コンパイル中のライブラリ
リテラル	literal	字面どおりの値を表す
数値リテラル	numeric literal	数字を示すリテラル
文字列リテラル	string literal	文字列を示すリテラル
マップ・リテラル ()	map literal	'one': 1 のように識別子または文字列リテラルにコロンを付けたマップを示すリテラル
リスト・リテラル	list literal	[]で挟んだ式で構成される List 型を示すリテラル
リフレクション	reflection	プログラムの実行過程でプログラム自身の構造を読み取ったり書き換えたりする技術
bottom 型	type bottom	値を持たない型で、ゼロまたは空の型ともいう。論理学では記号としては⊥が使われる。総ての型の副型で、戻り値を持たない関数の戻りの型を示すときに使われる