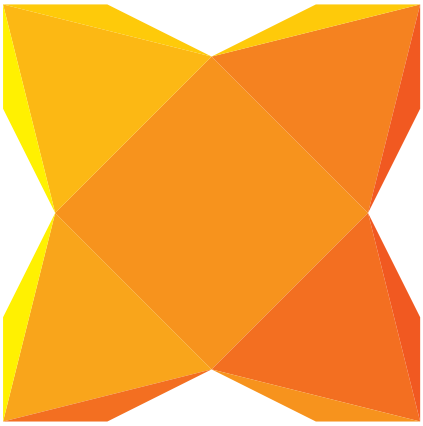

HAXE

Haxe3 マニュアル

Haxe Foundation
April 12, 2015
(訳: October 22, 2015)

Todo list

目次

章 1

導入

1.1 Haxe って何?

Could we have a big Haxe logo in the First Manual Page (Introduction) under the menu (a bit like a book cover ?) It looks a bit empty now and is a landing page for "Manual"

Haxe はオープンソースの高級プログラミング言語とコンパイラから成り、ECMAScript[1]を元にした構文のコードさまざまなターゲットの言語へとコンパイルすることを可能にします。適度な抽象化を行うため、1 つのコードベースから複数のターゲットへコンパイルすることも可能です。

Haxe は強く型付けされている一方で、必要に応じて型付けを弱めることも可能です。型情報を活用すれば、ターゲットの言語では実行時にしか発見できないようなエラーをコンパイル時に検出することができます。さらに型情報は、ターゲットへの変換時に最適化や堅牢なコードを生成するためにも使用されます。

現在、Haxe には 9 つのターゲット言語があり、さまざまな用途に利用できます。

名前	出力形式	主な用途
JavaScript	ソースコード	ブラウザ, デスクトップ, モバイル, サーバー
Neko	バイトコード	デスクトップ, サーバー
PHP	ソースコード	サーバー
Python	ソースコード	デスクトップ, サーバー
C++	ソースコード	デスクトップ, モバイル, サーバー
ActionScript 3	ソースコード	ブラウザ, デスクトップ, モバイル
Flash	バイトコード	ブラウザ, デスクトップ, モバイル
Java	ソースコード	デスクトップ, サーバー
C#	ソースコード	デスクトップ, モバイル, サーバー

この導入 (章 1) の残りでは、Haxe のプログラムがどのようなものなのか、Haxe はが 2005 年に生まれてからどのように進化してきたのか、を概要でお送りします。

型 (章 2) では、Haxe の 7 種類の異なる型についてとそれらがどう関わりあっているのかについて紹介します。型に関する話は、型システム (章 3) へと続き、単一化 (ユニファイ)、型パラメータ、型推論についての解説がされます。

クラスフィールド (章 4) では、Haxe のクラスの構造に関する全てをあつかいます。加えて、プロパティ、インラインフィールド、ジェネリック関数についてもあつかいます。

式 (章 5) では、式を使用して実際にいくつかの動作をさせる方法をお見せします。

言語機能 (章 6) では、パターンマッチング、文字列補間、デッドコード削除のような Haxe の詳細の機能について記述しています。ここで、Haxe の言語リファレンスは終わりです。

そして、Haxe のコンパイラリファレンスへと続きます。まずはコンパイラの使い方 (章 7) で基本的な内容を、そして、コンパイラの機能 (章 8) で高度な機能をあつかいます。最後にMacros (章 9) で、あり

[1] http://www.ecma-international.org/publications/standards/Ecma-327.htm

ふれたタスクを Haxe マクロがどのように単純かするのかを見ながら、刺激的なマクロの世界に挑んでいきます。

次のStandard Library (章 10) のでは、Haxe の標準ライブラリに含まれる主要な型や概念を一つ一つ見ていきます。そして、Haxelib (章 11) で Haxe のパッケージマネージャである Haxelib について学びます。

Haxe は様々なターゲット間の差を吸収してくれますが、場合によってはターゲットを直接的にあつかうことが重要になります。これが、Target Details (章 12) の話題です。

1.2 このドキュメントについて

このドキュメントは、Haxe3 の公式マニュアル（の日本語訳）です。そのため、初心者向けののチュートリアルではなく、プログラミングは教えません。しかし、項目は大まかに前から順番に読めるように並べてあり、前に出てきた項目と、次に出てくる項目との関連づけがされています。先の項目で後の項目でててくる情報に触れた方が説明しやすい場所では、先にその情報に触れています。そのような場面ではリンクがされています。リンク先は、ほとんどの場合で先に読むべき内容ではありません。

このドキュメントでは、理論的な要素を実物としてつなげるために、たくさんの Haxe のソースコードを使います。これらのコードのほとんどは main 関数を含む完全なコードでありそのままコンパイルが可能ですが、いくつかはそうではなくコードの重要な部分の抜き出しです。

ソースコードは以下のように示されます：

```
1 ここにHaxeのコード
```

時々、Haxe がどのようなコードを出力をするかを見せるため、ターゲットの JavaScript などのコードも示します。

さらに、このドキュメントではいくつかの単語の定義を行います。定義は主に、新しい型や Haxe 特有の単語を紹介するときに行われます。私たちが紹介するすべての新しい内容に対して定義をするわけではありません (例えば、クラスの定義など)。

定義は以下のように示されます。

> 定義: 定義の名前
> 定義の説明

また、いくつかの場所にはトリビア欄を用意してます。トリビア欄では、Haxe の開発過程でどうしてそのような決定がなされたのか、なぜその機能が過去の Haxe のバージョンから変更されたのかなど非公開の情報をお届けします。この情報は一般的には重要ではない、些細な内容なので読み飛ばしても構いません。

> トリビア: トリビアについて
> これはトリビアです

1.2.1 著者と貢献者

このドキュメントの大半の内容は、Haxe Foundation 所属の Simon Krajewski によって書かれました。そして、このドキュメントの貢献者である以下の方々に感謝の意を表します。

- Dan Korostelev: 追加の内容と編集
- Caleb Harper: 追加の内容と編集
- Josefiene Pertosa: 編集

- Miha Lunar: 編集
- Nicolas Cannasse: Haxe 創始者

1.2.2 ライセンス

Haxe Foundation制作の Haxe マニュアルは、クリエイティブコモンズ表示-4.0-国際ライセンスで提供されています。元データは、https://github.com/HaxeFoundation/HaxeManualで管理されています。

日本語訳のライセンス（訳注） 日本語訳も、クリエイティブコモンズ表示-4.0-国際ライセンスで提供しています。元データは、https://github.com/HaxeLocalizeJP/HaxeManualで管理されています。

1.3 Hello World

次のプログラムはコンパイルして実行をすると“Hello World”と表示します:

```
1 class Main {
2   static public function main():Void {
3     trace("Hello World");
4   }
5 }
```

上記のコードは、Main.hx という名前で保存して、haxe -main Main --interp というコマンドで Haxe コンパイラを呼び出すと実際に動作させることが可能です。これで Main.hx:3: Hello world という出力がされるはずです。このことから以下のいくつかのことを学ぶことができます。

This generates the following output: too many 'this'. You may like a passive sentence: the following output will be generated...though this is to be avoided, generally

- Haxe のコードは.hx という拡張子で保存する。
- Haxe のコンパイラはコマンドラインツールであり、-main Main や--interp のようなパラメータをつけて呼び出すことができる。
- Haxe のプログラムにはクラスがあり (Main、大文字から始まる)、クラスには関数がある (main、小文字からはじまる)。
- Haxe の main クラスをふくむファイルは、そのクラス名と同じ名前を使う (この場合だと、Main.hx)。

1.4 歴史

Haxe のプロジェクトは、2005 年 10 月 22 日にフランスの開発者の Nicolas Cannasse によって、オープンソースの ActionScript2 コンパイラである MTASC(Motion-Twin Action Script Compiler) と、Motion-Twin の社内言語であり、実験的に型推論をオブジェクト指向に取り入れた MTypes の後継として始まりました。Nicolas のプログラミング言語の設計に対する長年の情熱と、Motion-Twin でゲーム開発者として働くことで異なる技術が混ざり合う機会を得たことが、まったく新しい言語の作成に結び付いたのです。

そのころのつづりは haXe で、2006 年の 2 月に AVM のバイトコードと Nicolas 自身が作成した Neko バーチャルマシン[2]への出力をサポートするベータ版がリリースされました。

[2]http://nekovm.org

この日から Haxe プロジェクトのリーダーであり続ける Nicolas Cannasse は明確なビジョンをもって Haxe の設計を続け、そして 2006 年 5 月の Haxe1.0 のリリースに導きました。この最初のメジャーリリースから Javascript のコード生成をサポートの始まり、型推論や構造的部分型などの現在の Haxe の機能のいくつかはすでにこのころからありました。

Haxe1 では、2 年間いくつかのマイナーリリースを行い、2006 年 8 月に Flash AVM2 ターゲットと haxelib ツール、2007 年 3 月に ActionScript3 ターゲットを追加しました。この時期は安定性の改善に強く焦点が当てられ、その結果、数回のマイナーリリースが行われました。

Haxe2.0 は 2008 年 7 月にリリースされました。Franco Ponticelli の好意により、このリリースには PHP ターゲットが含まれました。同様に、Hugh Sanderson の貢献により、2009 年 7 月の Haxe2.04 リリースで C++ ターゲットが追加されました。

Haxe1 と同じように、以降の数か月で安定性のためのリリースを行いました。そして 2011 年 1 月、macros をサポートする Haxe2.07 がリリースされました。このころに、Bruno Garcia が JavaScript ターゲットのメンテナとしてチームに加わり、2.08 と 2.09 のリリースで劇的な改善が行われました。

2.09 のリリース後、Simon Krajewski がチームに加わり、Haxe3 の出発に向けて働き始めました。さらに Cauê Waneck の Java と C# のターゲットが Haxe のビルドに取り込まれました。そして Haxe2 は次で最後のリリースとなることが決まり、2012 年 1 月に Haxe2.10 がリリースされました。

2012 年の終盤、Haxe3 にスイッチを切り替えて、Haxe コンパイラチームは、新しく設立された Haxe Foundation[3]の援助を受けながら、次のメジャーバージョンに向かっていきました。そして、Haxe3 は 2013 年の 5 月にリリースされました。

[3]http://haxe-foundation.org

パートⅠ

言語リファレンス

章 2

型

Haxe コンパイラは豊かな型システムを持っており、これがコンパイル時に型エラーを検出することを手助けします。型エラーとは、文字列による割り算や、整数のフィールドへのアクセス、不十分な (あるいは多すぎる) 引数での関数呼び出し、といった型が不正である演算のことです。

いくつかの言語では、この安全性を得るためには各構文での明示的な型の宣言が強いられるので、コストがかかります。

```
1 var myButton:MySpecialButton = new MySpecialButton(); // AS3
2 MySpecialButton* myButton = new MySpecialButton(); // C++
```

一方、Haxe ではコンパイラが型を推論できるため、この明示的な型注釈は必要ではありません。

```
1 var myButton = new MySpecialButton(); // Haxe
```

型推論の詳細については型推論 (節 3.6) で説明します。今のところは、上のコードの変数 myButton は MySpecialButton のクラスインスタンスとわかると言っておけば十分でしょう。

Haxe の型システムは、以下の 7 つの型を認識します。

クラスインスタンス: クラスかインスタンスのオブジェクト

列挙型インスタンス: Haxe の列挙型 (enum) の値

構造体: 匿名の構造体。例えば、連想配列。

関数: 引数と戻り値 1 つの型の複合型。

ダイナミック: あらゆる型に一致する、なんでも型。

抽象 (abstract): 実行時には別の型となる、コンパイル時の型。

単相: 後で別の型が付けられる未知 (Unknown) の型。

ここからの節で、それぞれの型のグループとこれらがどうかかわっているのかについて解説していきます。

> 定義: 複合型
> 複合型というのは、従属する型を持つ型です。型パラメータ (3.2) を持つ型や、関数 (2.6) 型がこれに当たります。

2.1 基本型

基本型は `Bool` と `Float` と `Int` です。文法上、これらの値は以下のように簡単に識別可能です。

- `true` と `false` は `Bool`。
- `1`、`0`、`-1`、`0xFF0000` は `Int`。
- `1.0`、`0.0`、`-1.0`、`1e10` は `Float`。

Haxe では基本型はクラス (2.3) ではありません。これらは抽象型 (2.8) として実装されており、以降の項で解説するコンパイラ内部の演算処理に結び付けられています。

2.1.1 数値型

Type: Float
IEEE の 64bit 倍精度浮動小数点数を表します。

Type: Int
整数を表します。

`Int` は `Float` が期待されるすべての場所で使用することが可能です (Int は Float への代入が可能で、Float として表現可能です)。ですが、逆はできません。`Float` から `Int` への代入は精度を失ってしまう場合があり、信頼できません。

2.1.2 オーバーフロー

パフォーマンスのために Haxe コンパイラはオーバーフローに対する挙動を矯正しません。オーバーフローに対する挙動は、ターゲットのプラットフォームが責任を持ちます。各プラットフォームごとのオーバーフローの挙動を以下にまとめています。

C++, Java, C#, Neko, Flash: 一般的な挙動をもつ 32Bit 符号付き整数。

PHP, JS, Flash 8: ネイティブの Int 型を持たない。Float の上限 (2^{52}) を超えた場合に精度を失う。

代替手段として、プラットフォームごとの追加の計算を行う代わりに、正しいオーバーフローの挙動を持つ haxe.Int32 と haxe.Int64 クラスが用意されています。

2.1.3 数値の演算子

make sure the types are right for inc, dec, negate, and bitwise negate

While introducing the different operations, we should include that information as well, including how they differ with the ”C” standard, see http:// haxe.org/ manual/operators

以下は、Haxe の数値演算子です。優先度が降順になるようにグループ化して並べています。

算術演算				
演算子	演算	引数 1	引数 2	戻り値
++	1 増加	Int	なし	Int
		Float	なし	Float
--	1 減少	Int	なし	Int
		Float	なし	Float
+	加算	Float	Float	Float
		Float	Int	Float
		Int	Float	Float
		Int	Int	Int
-	減算	Float	Float	Float
		Float	Int	Float
		Int	Float	Float
		Int	Int	Int
*	乗算	Float	Float	Float
		Float	Int	Float
		Int	Float	Float
		Int	Int	Int
/	除算	Float	Float	Float
		Float	Int	Float
		Int	Float	Float
		Int	Int	Float
%	剰余	Float	Float	Float
		Float	Int	Float
		Int	Float	Float
		Int	Int	Int
比較演算				
演算子	演算	引数 1	引数 2	戻り値
==	等価	Float/Int	Float/Int	Bool
!=	不等価	Float/Int	Float/Int	Bool
<	より小さい	Float/Int	Float/Int	Bool
<=	より小さいか等しい	Float/Int	Float/Int	Bool
>	より大きい	Float/Int	Float/Int	Bool
>=	より大きいか等しい	Float/Int	Float/Int	Bool
ビット演算				
演算子	演算	引数 1	引数 2	戻り値
~	ビット単位の否定 (NOT)	Int	なし	Int
&	ビット単位の論理積 (AND)	Int	Int	Int
|	ビット単位の論理和 (OR)	Int	Int	Int
^	ビット単位の排他的論理和 (XOR)	Int	Int	Int
<<	左シフト	Int	Int	Int
>>	右シフト	Int	Int	Int
>>>	符号なしの右シフト	Int	Int	Int
Comparison				

等価性　enum:

パラメータなしの Enum 常に同じ値になるので、`MyEnum.A == MyEnum.A` で比較できる。

パラメータあり Enum `a.equals(b)` で比較できる。(これは `Type.enumEquals()` の短縮形である).

Dynamic: 1 つ以上の `Dynamic` な値に対する比較は、未定義であり、プラットフォーム依存です。

2.1.4 Bool(真偽値)

Type: Bool
真 (true) または、偽 (false) のどちらかになる値を表します。

Bool 型の値は、if (5.16) や while (5.14) のような条件文によく表れます。以下の演算子は、Bool 値を受け取って Bool 値を返します。

- && (and)
- || (or)
- ! (not)

Haxe は、Bool 値の 2 項演算は、実行時に左から右へ必要な分だけ評価することを保証します。例えば、A && B という式は、まず A を評価して A が true だった場合のみ B が評価されます。同じように、A || B では A が true だった場合は、B の値は意味を持たないので評価されません。
これは、以下のような場合に重要です。

```
1 if (object != null && object.field == 1) { }
```

object が null の場合に object.field にアクセスするとランタイムエラーになりますが、事前に object != null のチェックをすることでエラーから守ることができます。

2.1.5 Void

Type: Void
Void は型が存在しないことを表します。特定の場面 (主に関数) で値を持たないことを表現するのに使います。

Void は型システムにおける特殊な場合です。Void は実際には型ではありません。Void は特に関数の引数と戻り値で型が存在しないことを表現するのに使います。私たちはすでに最初の“Hello World”の例で Void を使用しています。

please review, doubled content

```
1 class Main {
2   static public function main():Void {
3     trace("Hello World");
4   }
5 }
```

関数型について詳しくは関数 (節 2.6) で解説しますが、ここで軽く予習をしておきましょう。上の例の main 関数は Void->Void 型です。これは“引数は無く、戻り値も無い”という意味です。
Haxe では、フィールドや変数に対して Void を指定することはできません。以下のように書こうとするとエラーが発生します。

review please, sounds weird

```
1 // 引数と変数の型にVoidは使えません。
2 var x:Void;
```

2.2 null 許容

定義: null 許容
Haxe では、ある型が値として null をとる場合に null 許容であるとみなす。

プログラミング言語は、null 許容についてなにか 1 つ明確な定義を持つのが一般的です。ですが、Haxe ではターゲットとなる言語の元々の挙動に従うことで妥協しています。ターゲット言語のうちのいくつかは全てがデフォルト値として null をとり、その他は特定の型では null を許容しません。つまり、以下の 2 種類の言語を区別しなくてはいけません。

定義: 静的ターゲット
静的ターゲットでは、その言語自体が基本型が null を許容しないような型システムを持っています。この性質は Flash、C++、Java、C# ターゲットに当てはまります。

定義: 動的ターゲット
動的ターゲットは型に関して寛容で、基本型が null を許容します。これは JavaScript と PHP、Neko、Flash 6-8 ターゲットが当てはまります。

for starters...please review

定義: デフォルト値
基本型は、静的ターゲットではデフォルト値は以下になります。

Int: 0。

Float: Flash では NaN。その他の静的ターゲットでは 0.0。

Bool: false。

その結果、Haxe コンパイラは静的ターゲットでは基本型に対する null を代入することはできません。null を代入するためには、以下のように基本型を Null<T> で囲う必要があります。

```
1 // 静的プラットフォームではエラー
2 var a:Int = null;
3 var b:Null<Int> = null; // こちらは問題ない
```

同じように、基本型は Null<T> で囲わなければ null と比較することはできません。

```
1 var a : Int = 0;
2 // 静的プラットフォームではエラー
3 if( a == null ) { ... }
4 var b : Null<Int> = 0;
5 if( b != null ) { ... } // 問題ない
```

この制限は unification (3.5) が動作するすべての状況でかかります。

Type: Null<T>
静的ターゲットでは、Null<Int>、Null<Float>、Null<Bool> の型で null を許容することが可能になります。動的ターゲットでは Null<T> に効果はありません。また、Null<T> はその型が null を持つことを表すドキュメントとしても使うことができます。

null の値は隠匿されます。つまり、Null<T> や Dynamic の null の値を基本型に代入した場合には、デフォルト値が使用されます。

```
var n : Null<Int> = null;
var a : Int = n;
trace(a); // 静的プラットフォームでは0
```

2.2.1 オプション引数と null 許容

null 許容について考える場合、オプション引数についても考慮しなくてはいけません。

特に、null 許容ではないネイティブのオプション引数と、それとは異なる、null 許容である Haxe 特有のオプション引数があることです。この違いは以下のように、オプション引数にクエスチョンマークを付けることで作ります。

```
// xはネイティブのInt(null許容ではない)
function foo(x : Int = 0) {}
// y is Null<Int> (null許容)
function bar( ?y : Int) {}
// z is also Null<Int>
function opt( ?z : Int = -1) {}
```

Is there a difference between ? y : Int and y : Null<Int> or can you even do the latter? Some more explanation and examples with native optional and Haxe optional arguments and how they relate to nullability would be nice.

トリビア: アーギュメント (Argument) とパラメータ (Parameter)
他のプログラミング言語では、よくアーギュメントとパラメータは同様の意味として使われます。Haxe では、関数に関連する場合にアーギュメントを、型パラメータ (節 3.2) と関連する場合にパラメータを使います。

2.3 クラスインスタンス

多くのオブジェクト指向言語と同じように、Haxe でも大抵のプログラムではクラスが最も重要なデータ構造です。Haxe のすべてのクラスは、明示された名前と、クラスの配置されたパスと、0 個以上のクラスフィールドを持ちます。ここではクラスの一般的な構造とその関わり合いについて焦点を当てていきます。クラスフィールドの詳細については後でクラスフィールド (章 4) の章で解説をします。

please review future tense

以下のサンプルコードが、この節で学ぶ基本になります。

```
class Point {
  var x : Int;
  var y : Int;
  public function new(x,y) {
    this.x = x;
    this.y = y;
  }
  public function toString() {
    return "Point("+x+","+y+")";
  }
}
```

意味的にはこれは不連続の 2 次元空間上の点を表現するものですが、このことはあまり重要ではありません。代わりにその構造に注目しましょう。

- class のキーワードは、クラスを宣言していることを示すものです。

- Point はクラス名です。型の識別子のルール (5) に従っているものが使用できます。
- クラスフィールドは {} で囲われます。
- Int 型の x と y の 2 つの変数フィールドと、
- クラスのコンストラクタとなる特殊な関数フィールド new と、
- 通常の関数 toString でクラスフィールドが構成されています。

また、Haxe にはすべてのクラスと一致する特殊な型があります。

Type: Class<T>
この型はすべてのクラスの型と一致します。つまり、すべてのクラス (インスタンスではなくクラス) をこれに代入することができます。コンパイル時に、Class<T> は全てのクラスの型の共通の親の型となります。しかし、この関係性は生成されたコードに影響を与えません。
　この型は、任意のクラスを要求するような API で役立ちます。例えば、Haxe リフレクション API (10.7) のいくつかのメソッドがこれに当てはまります。

2.3.1 クラスのコンストラクタ

クラスのインスタンスは、クラスのコンストラクタを呼び出すことで生成されます。この過程は一般的にインスタンス化と呼ばれます。クラスインスタンスは、別名としてオブジェクトと呼ぶこともあります。ですが、クラス/クラスインスタンスと、列挙型/列挙型インスタンスという似た概念を区別するために、クラスインスタンスと呼ぶことが好まれます。

```
1 var p = new Point(-1, 65);
```

　この例で、変数 p に代入されたのが Point クラスのインスタンスです。Point のコンストラクタは-1 と 65 の 2 つの引数を受け取り、これらをそれぞれインスタンスの x と y の変数に代入しています (クラスインスタンス (節 2.3) で、定義を確認してください)。new の正確な意味については、後の5.12の節で再習します。現時点では、new はクラスのコンストラクタを呼び、適切なオブジェクトを返すものと考えておきましょう。

2.3.2 継承

クラスは他のクラスから継承ができます。Haxe では、継承は extends キーワードを使って行います。

```
1 class Point3 extends Point {
2   var z : Int;
3   public function new(x,y,z) {
4     super(x,y);
5     this.z = z;
6   }
7 }
```

　この関係は、よく”B は A である (is-a)” の関係とよく言われます。つまり、すべての Point3 クラスのインスタンスは、同時に Point のインスタンスである、ということです。Point は Point3 の親クラスであると言い、Point3 は Point の子クラスであると言います。1 つのクラスはたくさんの子クラスを持つことができますが、親クラスは 1 つしか持つことができません。ただし、“クラス X の親クラス” というのは、直接の親クラスだけでなく、親クラスの親クラスや、そのまた親、また親のクラスなどを指すこともよくあります。
　上記のクラスは Point コンストラクタによく似ていますが、2 つの新しい構文が登場しています。

- `extends Point` は `Point` からの継承を意味します。
- `super(x, y)` は親クラスのコンストラクタを呼び出します。この場合は `Point.new` です。

上の例ではコンストラクタを定義していますが、子クラスで自分自身のコンストラクタを定義する必要はありません。ただし、コンストラクタを定義する場合 `super()` の呼び出しが必須になります。他のよくあるオブジェクト指向言語とは異なり、`super()` はコンストラクタの最初である必要はなく、どこで呼び出しても構いません。

また、クラスはその親クラスのメソッド (4.3) を `override` キーワードを明示して記述することで上書きすることができます。その効果と制限について詳しくはメソッドのオーバーライド (override) (節 4.3.1)であつかいます。

2.3.3 インターフェース

インターフェースはクラスのパブリックフィールドを記述するもので、クラスの署名ともいうべきものです。インターフェースは実装を持たず、構造に関する情報のみを与えます。

```
1 interface Printable {
2     public function toString():String;
3 }
```

この構文は以下の点をのぞいて、クラスによく似ています。

- `interface` キーワードを `class` キーワードの代わりに使う。
- 関数が式 (5) を持たない。
- すべてのフィールドが型を明示する必要がある。

インタフェースは、構造的部分型 (3.5.2) とは異なり、クラス間の静的な関係性について記述します。以下のように明示的に宣言した場合にのみ、クラスはインターフェースと一致します。

```
1 class Point implements Printable { }
```

`implements` キーワードの記述により、"`Point` は `Printable` である (is-a)" の関係性が生まれます。つまり、すべての `Point` のインスタンスは、`Printable` のインスタンスでもあります。クラスは親のクラスを 1 つしか持てませんが、以下のように複数の `implements` キーワードを使用することで複数のインターフェースを実装 (implements) することが可能です。

```
1 class Point implements Printable implements Serializable
```

コンパイラは実装が条件を満たしているかの確認を行います。つまり、クラスが実際にインターフェースで要求されるフィールドを実装しているかを確めます。フィールドの実装は、そのクラス自体と、その親となるいずれかのクラスの実装が考慮されます。

インターフェースのフィールドは、変数とプロパティのどちらであるかに対する制限は与えません:

```
1  interface Placeable {
2    public var x:Float;
3    public var y:Float;
4  }
5
6  class Main implements Placeable {
7    public var x:Float;
8    public var y:Float;
9    static public function main() { }
10 }
```

インターフェースは extends キーワードで複数のインタフェースを継承することができます。

> トリビア: Implements の構文
> Haxe の 3.0 よりも前のバージョンでは、implements キーワードはカンマで区切られていました。Java のデファクトスタンダードに合わせるため、私たちはカンマを取り除くことに決定しました。これが、Haxe2 と 3 の間の破壊的な変更の 1 つです。

2.4 列挙型インスタンス

Haxe には強力な列挙型 (enum) をもっています。この列挙型は実際には代数的データ型 (ADT)[1]に当たります。列挙型は式 (5) を持つことはできませんが、データ構造を表現するのに非常に役に立ちます。

```
1 enum Color {
2   Red;
3   Green;
4   Blue;
5   Rgb(r:Int, g:Int, b:Int);
6 }
```

このコードでは、enum は、赤、緑、青のいずれかか、または RGB 値で表現した色、を書き表しています。この文法の構造は以下の通りです。

- enum キーワードが、列挙型について定義することを宣言しています。
- Color が列挙型の名前です。型の識別子のルール (5) に従うすべてのものが使用できます。
- 中カッコ {} で囲んだ中に列挙型のコンストラクタを記述します。
- Red と Green と Blue には引数がありません。
- Rgb は、r、g、b の 3 つの Int 型の引数を持ちます。

Hax の型システムには、すべての列挙型を統合する型があります。

> Type: Enum<T>
> すべての列挙型と一致する型です。コンパイル時に、Enum<T> は全ての列挙型の共通の親の型となります。しかし、この関係性は生成されたコードに影響を与えません。

Same as in 2.2, what is Enum<T> syntax?

2.4.1 列挙型のコンストラクタ

クラスと同じように、列挙型もそのコンストラクタを使うことでインスタンス化を行います。しかし、クラスとは異なり列挙型は複数のコンストラクタを持ち、以下のようにコンストラクタの名前を使って呼び出します。

```
1 var a = Red;
2 var b = Green;
3 var c = Rgb(255, 255, 0);
```

list arguments

このコードでは変数 a、b、c の型は Color です。変数 c は Rgb コンストラクタと引数を使って初期化されています。

すべての列挙型のインスタンスは EnumValue という特別な型に対して代入が可能です。

> Type: EnumValue
> EnumValue はすべての列挙型のインスタンスと一致する特別な型です。この型は Haxe の標準ライブラリでは、すべての列挙型に対して可能な操作を提供するのに使われます。またユーザーのコードでは、特定の列挙型ではなく任意の列挙型のインスタンスを要求する API で利用できます。

以下の例からわかるように、列挙型とそのインスタンスを区別することは大切です。

```
1  enum Color {
2    Red;
3    Green;
4    Blue;
5    Rgb(r:Int, g:Int, b:Int);
6  }
7
8  class Main {
9    static public function main() {
10     var ec:EnumValue = Red; // 問題無い
11     var en:Enum<Color> = Color; // 問題無い
12     // エラー: Color should be Enum<Color>
13     // (Colorではなく Enum<Color>であるべきです)
14     //var x:Enum<Color> = Red;
15   }
16 }
```

もし、上でコメント化されている行のコメント化が解除された場合、このコードはコンパイルできなくなります。これは、列挙型のインスタンスである Red は、列挙型である Enum<Color> 型の変数には代入できないためです。

この関係性は、クラスとそのインスタンスの関係性に似ています。

> トリビア: Enum<T> の型パラメータを具体化する
> このマニュアルのレビューアの一人は上のサンプルコードの Color と Enum<Color> の違いについて困惑しました。実際、型パラメータの具体化は意味のないもので、デモンストレーションのためのものでしかありませんでした。私たちはよく型を書くのを省いて、型についてあつかうのを型推論 (3.6) にまかせてしまいます。
>
> しかし、型推論では Enum<Color> ではないものが推論されます。コンパイラは、列挙型のコンストラクタをフィールドとしてみなした、仮の型を推論します。現在の Haxe3.2.0 では、この仮の型について表現することは不可能であり、また表現する必要もありません。

2.4.2 列挙型を使う

列挙型は、有限の種類の値のセットが許されることを表現するだけでも有用です。それぞれのコンストラクタについて多様性が示されるので、コンパイラはありうる全ての値が考慮されていることをチェックすることが可能です。これは、例えば以下のような場合です。

```
1  enum Color {
```

[1] https://ja.wikipedia.org/wiki/%E4%BB%A3%E6%95%B0%E7%9A%84%E3%83%87%E3%83%BC%E3%82%BF%E5%9E%8B

```
2     Red;
3     Green;
4     Blue;
5     Rgb(r:Int, g:Int, b:Int);
6   }
7
8   class Main {
9     static function main() {
10      var color = getColor();
11      switch (color) {
12        case Red: trace("赤色");
13        case Green: trace("緑色");
14        case Blue: trace("青色");
15        case Rgb(r, g, b): trace("この色の赤成分は" +r);
16      }
17    }
18
19    static function getColor():Color {
20      return Rgb(255, 0, 255);
21    }
22  }
```

getColor() の戻り値を color に代入し、その値で switch 式 (5.17) の分岐を行います。

初めの Red、Green、Blue の 3 ケースについては些細な内容で、ただ Color の引数無しのコンストラクタとの一致するか調べています。最後の Rgb(r, g, b) のケースでは、コンストラクタの引数の値をどうやって利用するのかがわかります。引数の値はケースの式の中で出てきたローカル変数として、var の式 (5.10) を使った場合と同じように、利用可能です。

switch の使い方について、より高度な情報は後のパターンマッチング (6.4) の節でお話します。

2.5 匿名の構造体

匿名の構造体は、型を明示せずに利用できるデータの集まりです。以下の例では、x と name の 2 つのフィールドを持つ構造体を生成して、それぞれを 12 と"foo" の値で初期化しています。

```
1   class Main {
2     static public function main() {
3       var myStructure = { x: 12, name: "foo"};
4     }
5   }
```

構文のルールは以下の通りです:

1. 構造体は中カッコ {} で囲う。
2. カンマで区切られたキーと値のペアのリストを持つ。
3. 識別子 (5) の条件を満たすカギと、値がコロンで区切られる。
4. 値には、Haxe のあらゆる式が当てはまる。

please reformat

ルール4は複雑にネストした構造体を含みます。例えば、以下のような。

```
1   var user = {
2     name : "Nicolas",
```

```
  age : 32,
  pos : [
    { x : 0, y : 0 },
    { x : 1, y : -1 }
  ],
};
```

構造体のフィールドは、クラスと同じように、ドット (.) を使ってアクセスします。

```
// 名前を取得する。ここでは"Nicolas"。
user.name;
// ageを33に設定。
user.age = 33;
```

特筆すべきは、匿名の構造体の使用は型システムを崩壊させないことです。コンパイラは実際に利用可能なフィールドにしかアクセスを許しません。つまり、以下のようなコードはコンパイルできません。

```
class Test {
  static public function main() {
    var point = { x: 0.0, y: 12.0 };
    // { y : Float, x : Float } has no field z  (zフィールドが足りない)
    point.z;
  }
}
```

このエラーメッセージはコンパイラが point の型を知っていることを表します。この point の型は、x と y の Float 型のフィールドを持つ構造体であり、z というフィールドは持たないのでアクセスに失敗しました。この point の型は型推論 (3.6) により識別され、そのおかげでローカル変数では型を明示しなくて済みます。ただし、point が、クラスやインスタンスのフィールドだった場合、以下のように型の明示が必要になります。

```
class Path {
    var start : { x : Int, y : Int };
    var target : { x : Int, y : Int };
    var current : { x : Int, y : Int };
}
```

このような冗長な型の宣言をさけるため、特にもっと複雑な構造体の場合、以下のように typedef (3.1) を使うことをお勧めします。

```
typedef Point = { x : Int, y : Int }

class Path {
    var start : Point;
    var target : Point;
    var current : Point;
}
```

2.5.1 JSON で構造体を書く

以下のように、文字列の定数値をキーに使う JavaScript Object Notation(JSON) の構文を構造体に使うこともできます。

```
var point = { "x" : 1, "y" : -5 };
```

キーには文字列の定数値すべてが使えますが、フィールドが Haxe の識別子 (5) として有効である場合のみ型の一部として認識されます。そして、Haxe の構文では識別子として無効なフィールドにはアクセスできないため、リフレクション (10.7) の `Reflect.field` と `Reflect.setField` を使ってアクセスしなくてはいけません。

2.5.2 構造体の型のクラス記法

構造体の型を書く場合に、Haxe ではクラスフィールド (章 4) を書くときと同じ構文が使用できます。以下の typedef (3.1) では、`Int` 型の `x` の `y` 変数フィールドを持つ `Point` 型を定義しています。

```
1 typedef Point = {
2     var x : Int;
3     var y : Int;
4 }
```

2.5.3 オプションのフィールド

I don't really know how these work yet.

2.5.4 パフォーマンスへの影響

構造体をつかって、さらに構造的部分型付け (3.5.2) を使った場合、動的ターゲット (2.2) ではパフォーマンスに影響はありません。しかし、静的ターゲット (2.2) では、動的な検査が発生するので通常は静的なフィールドアクセスよりも遅くなります。

2.6 関数

It seems a bit convoluted explanations. Should we maybe start by "decoding" the meaning of Void -> Void, then Int -> Bool -> Float, then maybe have samples using $type

関数の型は、単相 (2.9) と共に、Haxe のユーザーからよく隠れている型の 1 つです。コンパイル時に式の型を出力させる `$type` という特殊な識別子を使えば、この型を以下のように浮かび上がらせることが可能です。

```
1  class Main {
2    static public function main() {
3      // i : Int -> s : String -> Bool
4      $type(test);
5      $type(test(1, "foo")); // Bool
6    }
7
8    static function test(i:Int, s:String):Bool {
9      return true;
10   }
11 }
```

初めの `$type` の出力は、test 関数の定義と強い類似性があります。では、その相違点を見てみます。

- 関数の引数は、カンマではなく-> で区切られる。
- 引数の戻り値の型は、もう一つ-> を付けた後に書かれる。

どちらの表記でも、test 関数が 1 つ目の引数として Int を受け取り、2 つ目の引数として String 型を受け取り、Bool 型の値を返すことはよくわかります。2 つ目の $type 式の test(1, "foo") のようにこの関数を呼び出すと、Haxe の型検査は 1 が Int に代入可能か、"foo" が String に代入可能かをチェックします。そして、その呼び出し後の型は、test の戻り値の型の Bool となります。

もし、ある関数の型が、別の関数の型を引数か戻り値に含む場合、丸かっこをグループ化に使うことができます。例えば、Int -> (Int -> Void) -> Void は初めの引数の型が Int、2 番目の引数が Int -> Void で、戻り値が Void の関数を表します。

2.6.1 オプション引数

オプション引数は、引数の識別子の直前にクエスチョンマーク (?) を付けることで表現できます。

```
1  class Main {
2    static public function main() {
3      // ?i : Int -> ?s : String -> String
4      $type(test);
5      trace(test()); // i: null, s: null
6      trace(test(1)); // i: 1, s: null
7      trace(test(1, "foo")); // i: 1, s: foo
8      trace(test("foo")); // i: null, s: foo
9    }
10
11   static function test(?i:Int, ?s:String) {
12     return "i: " +i + ", s: " +s;
13   }
14 }
```

test 関数は、2 つのオプション引数を持ちます。Int 型の i と String 型の s です。これは 3 行目の関数型の出力に直接反映されています。

この例では、関数を 4 回呼び出しその結果を出力しています。

1. 初めの呼び出しは引数無し。
2. 2 番目の呼び出しは 1 のみの引数。
3. 3 番目の呼び出しは 1 と"foo" の 2 つの引数。
4. 4 番目の呼び出しは"foo" のみの引数。

この出力を見ると、オプション引数が呼び出し時に省略されると null になることがわかります。つまり、これらの引数は null が入る型でなくてはいけないことになり、ここで null 許容 (2.2) に関する疑問が浮かび上がります。Haxe のコンパイラは静的ターゲット (2.2) に出力する場合に、オプションの基本型の引数の型を Null<T> であると推論することで、オプション引数の型が null 許容であることを保証してます。

初めの 3 つの呼び出しは直観的なものですが、4 つ目の呼び出しには驚くかもしれません。後の引数に代入可能な値が渡されたため、オプション引数はスキップされています。

2.6.2 デフォルト値

Haxe では、引数のデフォルト値として定数値を割り当てることが可能です。

```
1  class Main {
2    static public function main() {
3      // ?i : Int -> ?s : String -> String
```

```
    $type(test);
    trace(test()); // i: 12, s: bar
    trace(test(1)); // i: 1, s: bar
    trace(test(1, "foo")); // i: 1, s: foo
    trace(test("foo")); // i: 12, s: foo
  }

  static function test(?i = 12, s = "bar") {
    return "i: " +i + ", s: " +s;
  }
}
```

この例は、オプション引数 (節 2.6.1) のものとよく似ています。違いは、関数の引数の i と s それぞれに 12 と"bar" を代入していることだけです。これにより、引数が省略された場合に null ではなく、このデフォルト値が使われるようになります。

Haxe でのデフォルト値は、型の一部では無いので、出力時に呼び出し元で置き換えられるわけではありません (ただし、特有の動作を行うインライン (4.4.2) の関数を除く)。いくつかのターゲットでは、無視された引数に対してやはり null を渡して、以下の関数と同じようなコードを生成します。

```
    static function test(i = 12, s = "bar") {
        if (i == null) i = 12;
        if (s == null) s = "bar";
        return "i: " +i + ", s: " +s;
    }
```

つまり、パフォーマンスが要求されるコードでは、デフォルト値を使わない書き方をすることが重要だと考えてください。

2.7 ダイナミック

Haxe は静的な型システムを持っていますが、この型システムは Dynamic 型を使うことで事実上オフにすることが可能です。Dynamic な値は、あらゆるものに割り当て可能です。逆に、Dynamic に対してはあらゆる値を割り当て可能です。これにはいくつかの弱点があります。

- 代入、関数呼び出しなど、特定の型を要求される場面でコンパイラが型チェックをしなくなります。
- 特定の最適化が、特に静的ターゲットにコンパイルする場合に、効かなくなります。
- よくある間違い (フィールド名のタイポなど) がコンパイル時に検出できなくなって、実行時のエラーが起きやすくなります。
- デッドコード削除 (節 8.2) は、Dynamic を通じて使用しているフィールドを検出できません。

Dynamic が実行時に問題を起こすような例を考えるのはとても簡単です。以下の 2 行を静的ターゲットへコンパイルすることを考えてください。

```
var d:Dynamic = 1;
d.foo;
```

これをコンパイルしたプログラムを、Flash Player で実行した場合、**Number にプロパティ foo が見つからず、デフォルト値もありません。**というエラーが発生します。Dynamic を使わなければ、このエラーはコンパイル時に検出できます。

トリビア: Haxe3 より前の Dynamic の推論
Haxe3 のコンパイラは型を Dynamic として推論することはないので、Dynamic を使いたい場合はそのことを明示しなければ行きません。以前の Haxe のバージョンでは、混ざった型の Array を Array<Dynamic> として推論してました (例えば、[1, true, "foo"])。私たちはこの挙動はたくさんの型の問題を生み出すことに気づき、この仕様を Haxe3 で取り除きました。

実際のところ Dynamic は使ってしまいますが、多くの場面では他のもっと良い選択肢があるので Dynamic の使用は最低限にすべきです。例えば、Haxe の Reflection (節 10.7)API は、コンパイル時には構造のわからないカスタムのデータ構造をあつかう際に最も良い選択肢になりえます。

Dynamic は、単相 (monomorph) (2.9) を単一化 (3.5) する場合に、特殊な挙動をします。以下のような場合に、とんでもない結果を生んでしまうので、単相が Dynamic に拘束されることはありません。

```
class Main {
  static function main() {
    var jsonData = '[1, 2, 3]';
    var json = haxe.Json.parse(jsonData);
    $type(json); // Unknown<0>
    for (i in 0...json.length) {
      // Array access is not allowed on {+ length : Int }
      // ({+ length : Int }には配列アクセスは許可されていません)
      trace(json[0]);
    }
  }
}
```

Json.parse の戻り値は Dynamic ですが、ローカル変数の json の型は Dynamic に拘束されません。単相のままです。そして、json.length のフィールドにアクセスした時に匿名の構造体 (2.5) として推論されて、それにより json[0] の配列アクセスでエラーになっています。これは、json に対して、var json:Dynamic というように明示的に Dynamic の型付けをすることで避けることができます。

トリビア: 標準ライブラリでの Dynamic
Dynamic は Haxe3 より前の標準ライブラリではかなり頻繁に表れていましたが、Haxe3 までの継続的な型システムの改善によって Dynamic の出現頻度を減らすことができました。

2.7.1 型パラメータ付きのダイナミック

Dynamic は、型パラメータ (3.2) を付けても付けなくても良いという点でも特殊な型です。型パラメータを付けた場合、ダイナミック (節 2.7) のすべてのフィールドがパラメータの型であることが強制されます。

```
var att : Dynamic<String> = xml.attributes;
// 正当。値が文字列。
att.name = "Nicolas";
// 同上
att.age = "26";
// エラー。値が文字列ではない。
att.income = 0;
```

2.7.2 ダイナミックを実装 (implements) する

クラスは Dynamic と Dynamic<T> を実装 (2.3.3) することができます。これにより任意のフィールドへのアクセスが可能になります。Dynamic の場合、フィールドはあらゆる型になる可能性があり、

Dynamic<T> の場合、フィールドはパラメータの型と矛盾しない型のみに強制されます。

```
class ImplementsDynamic
  implements Dynamic<String> {
  public var present:Int;
  public function new() {}
}

class Main {
  static public function main() {
    var c = new ImplementsDynamic();
    // 問題無い。presentフィールドは存在する。
    c.present = 1;
    // 問題無い。Stringを代入している。
    c.stringField = "foo";
    // エラー。 Int should be String(Intではなく Stringであるべき)
    //c.intField = 1;
  }
}
```

Dynamic を実装しても、他のインターフェースが要求する実装を満たすことにはなりません。明示的な実装が必要です。

型パラメータなしの Dynamic を実装したクラスでは、特殊なメソッド resolve を利用することができます。読み込みアクセス (4.2) がありフィールドが存在しなかった場合、resolve メソッドが以下のように呼び出されます。

```
class Resolve implements Dynamic<String> {
  public var present:Int;
  public function new() {}
  function resolve(field:String) {
    return "Tried to resolve " +field;
  }
}

class Main {
  static public function main() {
    var c = new Resolve();
    c.present = 2;
    trace(c.present);
    trace(c.resolveMe);
  }
}
```

2.8 抽象型 (abstract)

抽象 (abstract) 型は、実行時には別の型になる型です。抽象型は挙動を編集したり強化したりするために、具体型 (= 抽象型でない型) を “おおう” 型を定義するコンパイル時の機能です。

```
abstract AbstractInt(Int) {
  inline public function new(i:Int) {
    this = i;
  }
```

```
}
```

上記のコードからは以下を学ぶことができます。

- abstract キーワードは、抽象型を定義することを宣言している。
- AbstractInt は抽象型の名前であり、型の識別子のルールを満たすものなら何でも使える。
- 丸かっこ () の中は、その基底型の Int である。
- 中カッコ {} の中はフィールドで、
- Int 型の i のみを引数とするコンストラクタの new 関数がある。

定義: 基底型
抽象型の基底型は、実行時にその抽象型を表すために使われる型です。基底型はたいていの場合は具体型ですが、別の抽象型である場合もあります。

構文はクラスを連想させるもので、意味合いもよく似ています。実際、抽象型のボディ部分 (中カッコの開始以降) は、クラスフィールドとして構文解析することが可能です。抽象型はメソッド (4.3) と、実体 (4.2.3) の無いプロパティ (4.2) フィールドを持つことが可能です。

さらに、抽象型は以下のように、クラスと同じようにインスタンス化して使用することができます

```
class Main {
  static public function main() {
    var a = new AbstractInt(12);
    trace(a); //12
  }
}
```

はじめに書いたとおり、抽象型はコンパイル時の機能ですから、見るべきは上記のコードの実際の出力です。この出力例としては、簡潔なコードが出力される JavaScript が良いでしょう。上記のコードを haxe -main MyAbstract -js myabstract.js でコンパイルすると以下のような JavaScript が出力されます。

```
var a = 12;
console.log(a);
```

抽象型の AbstractInt は出力から完全に消えてしまい、その基底型の Int の値のみが残っています。これは、AbstractInt のコンストラクタがインライン化されて、そのインラインの式が値を this に代入します (インライン化については後のinline(インライン化) (節 4.4.2) で学びます)。これは、クラスのように考えていた場合、驚くべきことかもしれません。しかし、これこそが抽象型を使って表現したいことそのものです。抽象型のすべてのインラインのメンバメソッドでは this への代入が可能で、これにより“内部の値”が編集できます。

“もしメンバ関数で inline が宣言されていなかった場合、何が起こるのか?”というのは良い疑問です。そのようなコードははっきりと成立します。その場合、Haxe は実装クラスと呼ばれる private のクラスを生成します。この実装クラスは抽象型のメンバ関数を、最初の引数としてその基底型の this を加えた静的な (static) 関数で持ちます。さらに実装の詳細の話をすると、この実装クラスは選択的関数 (2.8.4) でも使われます。

トリビア: 基本型と抽象型
抽象型が生まれる前には、基本型は extern クラスと列挙型で実装されていました。Int 型を Float 型の“子クラス”としてあつかうなどのいくつかの面では便利でしたが、一方で問題も引き起こしました。例えば、Float が extern クラスなので、実際のオブジェクトしか受け入れないはずの空の構造体の型 {} として単一化できました。

2.8.1 暗黙のキャスト

クラスとは異なり抽象型は暗黙のキャストを許します。抽象型には 2 種類の暗黙のキャストがあります。

直接: 他の型から抽象型への直接のキャストを許します。これは to と from のルールを抽象型に設定することでできます。これは、その抽象型の基底型に単一化可能な型のみで利用可能です。

クラスフィールド: 特殊なキャスト関数を呼び出すことによるキャストを許します。この関数は @:to と @:from のメタデータを使って定義されます。この種類のキャストは全ての型で利用可能です。

下のコードは、直接キャストの例です。

```
1  abstract MyAbstract(Int) from Int to Int {
2    inline function new(i:Int) {
3      this = i;
4    }
5  }
6
7  class Main {
8    static public function main() {
9      var a:MyAbstract = 12;
10     var b:Int = a;
11   }
12 }
```

from Int かつ to Int の MyAbstract を定義しました。これは Int を代入することが可能で、かつ Int に代入することが可能だという意味です。このことは、9、10 行目に表れています。まず、Int の 12 を MyAbstract 型の変数 a に代入しています (これは from Int の宣言により可能になります)。そして次に、Int 型の変数 b に、抽象型のインスタンスを代入しています (これは to Int の宣言により可能になります)。

クラスフィールドのキャストも同じ意味を持ちますが、定義の仕方はまったく異なります。

```
1  abstract MyAbstract(Int) {
2    inline function new(i:Int) {
3      this = i;
4    }
5
6    @:from
7    static public function fromString(s:String) {
8      return new MyAbstract(Std.parseInt(s));
9    }
10
11   @:to
12   public function toArray() {
13     return [this];
14   }
15 }
16
17 class Main {
18   static public function main() {
19     var a:MyAbstract = "3";
20     var b:Array<Int> = a;
21     trace(b); // [3]
22   }
23 }
```

静的な関数に @:from を付けることで、その引数の型からその抽象型への暗黙のキャストを行う関数として判断されます。この関数はその抽象型の値を返す必要があります。static を宣言する必要もあります。

同じように関数に @:to を付けることで、その抽象型からその戻り値の型への暗黙のキャストを行う関数として判断されます。この関数は普通はメンバ関数ですが、static でも構いません。そして、これは選択的関数 (2.8.4) として働きます。

上の例では、fromString メソッドが”3” の値を MyAbstract 型の変数 a への代入を可能にし、toArray メソッドがその抽象型インスタンスを Array<Int> 型の変数 b への代入を可能にします。

この種類のキャストを使った場合、必要な場所でキャスト関数の呼び出しが発生します。このことは JavaScript 出力を見ると明らかです。

```
var a = _ImplicitCastField.MyAbstract_Impl_.fromString(”3”);
var b = _ImplicitCastField.MyAbstract_Impl_.toArray(a);
```

これは 2 つのキャスト関数でインライン化 (4.4.2) を行うことでさらなる最適化を行うことができます。これにより出力は以下のように変わります。

```
var a = Std.parseInt(”3”);
var b = [a];
```

型 A から時の型 B への代入の時にどちらかまたは両方が抽象型である場合に使われるキャストの選択アルゴリズムは簡単です。

1. A が抽象型でない場合は 3 へ。
2. A が、B への変換を持っている場合、これを適用して 6 へ。
3. B が抽象型でない場合は 5 へ。
4. B が、A からの変換を持っている場合、これを適用して 6 へ。
5. 単一化失敗で、終了。
6. 単一化成功で、終了。

意図的に暗黙のキャストは連鎖的ではありません。これは以下の例でわかります。

```
abstract A(Int) {
  public function new() this = 0;
  @:to public function toB() return new B();
}

abstract B(Int) {
  public function new() this = 0;
  @:to public function toC() return new C();
}

abstract C(Int) {
  public function new() this = 0;
}

class Main {
  static public function main() {
    var a = new A();
    var b:B = a; // 問題無い。A.toBが使われる。
    var c:C = b; // 問題無い。B.toCが使われる。
```

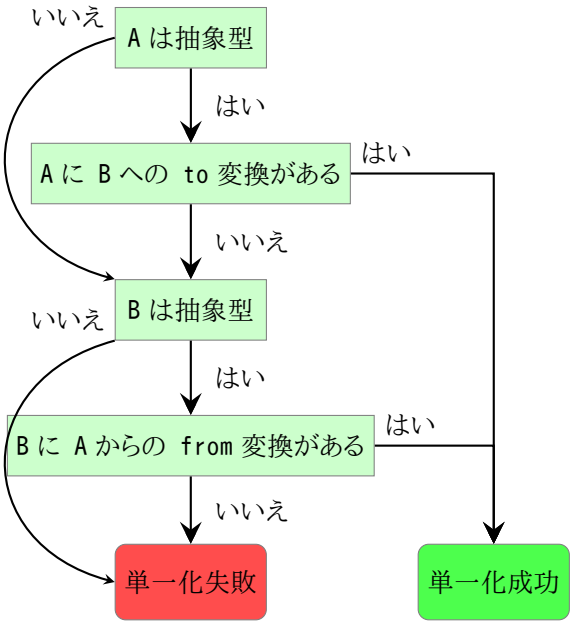

Figure 2.1: 選択アルゴリズムのフローチャート

```
20      var c:C = a; // エラー。A should be C(Aではなくであるべき)
21    }
22  }
```

A から B、B から C への個々のキャストは可能ですが、A から C への連鎖的なキャストはできません。これは、キャスト方法が複数生まれてしまうことは避けて、選択アルゴリズムの簡潔さを保つためです。

2.8.2 演算子オーバーロード

抽象型ではクラスフィールドに @:op メタデータを付けることで、単項演算子と 2 項演算子のオーバーロードが可能です。

```
1   abstract MyAbstract(String) {
2     public inline function new(s:String) {
3       this = s;
4     }
5
6     @:op(A * B)
7     public function repeat(rhs:Int):MyAbstract {
8       var s:StringBuf = new StringBuf();
9       for (i in 0...rhs)
10        s.add(this);
11      return new MyAbstract(s.toString());
12    }
13  }
14
15  class Main {
16    static public function main() {
17      var a = new MyAbstract("foo");
```

```
18      trace(a * 3); // foofoofoo
19    }
20  }
```

@:op(A * B) を宣言することで、repeat 関数は、左辺が MyAbstract で右辺が Int の場合の * 演算子による乗算の関数として利用されます。これは 18 行目で利用されています。この部分は JavaScript にコンパイルすると以下のようになります。

```
1  console.log(_AbstractOperatorOverload.
2    MyAbstract_Impl_.repeat(a,3));
```

クラスフィールドによる暗黙の型変換 (2.8.1) と同様に、オーバーロードメソッドも要求された場所で呼び出しが発生します。上記の例の repeat 関数は可換ではありません。MyAbstract * Int は動作しますが、Int * MyAbstract では動作しません。Int * MyAbstract でも動作させたい場合は @:commutative のメタデータが使えます。逆に、MyAbstract * Int ではなく Int * MyAbstract でのみ動作させてたい場合、1 つ目の引数で Int 型、2 つ目の引数で MyAbstract 型を受け取る静的な関数をオーバーロードメソッドにすることができます。

単項演算子の場合もこれによく似ています。

```
1  abstract MyAbstract(String) {
2    public inline function new(s:String) {
3      this = s;
4    }
5
6    @:op(++A) public function pre()
7      return "pre" + this;
8    @:op(A++) public function post()
9      return this + "post";
10 }
11
12 class Main {
13   static public function main() {
14     var a = new MyAbstract("foo");
15     trace(++a); // prefoo
16     trace(a++); // foopost
17   }
18 }
```

2 項演算子と単項演算子の両方とも、戻り値の型は何でも構いません。

基底型の演算を公開する　基底型が抽象型でそこで許容されている演算子でかつ戻り値を元の抽象型に代入可能なものについては、@:op 関数のボディを省略することが可能です。

```
1  abstract MyAbstractInt(Int) from Int to Int {
2    // 以下の行は基底のInt型の(A > B)の演算を行います。
3    // 関数が本体の式を持たないことに注目してください。
4    @:op(A > B) static function gt( a:MyAbstractInt, b:MyAbstractInt ) :
         Bool;
5  }
6
7  class Main {
8    static function main() {
9      var a:MyAbstractInt = 42;
10     if(a > 0) trace('正しく動作します。>の演算子は実装されています!');
```

```
11
12      // くの演算子は実装されていません。
13      // 'Cannot compare MyAbstractInt and Int'(MyAbstractIntとIntは比較
            できません)のエラーが起きます。
14      if(a < 100) { }
15    }
16  }
```

please review for correctness

2.8.3 配列アクセス

配列アクセスは、配列の特定の位置の値にアクセスするのに伝統的に使われている特殊な構文です。これは大抵の場合、Int のみを引数としますが、抽象型の場合はカスタムの配列アクセスを定義することが可能です。Haxe の標準ライブラリ (10) では、これを Map 型に使っており、これには以下の 2 つのメソッドがあります。

You have marked "Map" for some reason

```
1  @:arrayAccess
2  public inline function get(key:K) {
3    return this.get(key);
4  }
5  @:arrayAccess
6  public inline function arrayWrite(k:K, v:V):V {
7      this.set(k, v);
8      return v;
9  }
```

配列アクセスのメソッドは以下の 2 種類があります。

・@:arrayAccess メソッドが 1 つの引数を受け取る場合、それは読み取り用です。

・@:arrayAccess メソッドが 2 つの引数を受け取る場合、それは書き込み用です。

上記のコードの get メソッドと arrayWrite メソッドは、以下のように使われます。

```
1  class Main {
2    public static function main() {
3      var map = new Map();
4      map["foo"] = 1;
5      trace(map["foo"]);
6    }
7  }
```

ここでは以下のように出力に配列アクセスのフィールドの呼び出しが入ることになりますが、驚かないでください。

```
1  map.set("foo",1);
2  console.log(map.get("foo")); // 1
```

配列アクセスの解決順序　Haxe3.2 以前では、バグのため:arrayAccess のフィールドがチェックされる順序は定義されていませんでした。これは、Haxe3.2 では修正されて一貫して上から下へと確認が行われるようになりました。

```
abstract AString(String) {
  public function new(s) this = s;
  @:arrayAccess function getInt1(k:Int) {
    return this.charAt(k);
  }
  @:arrayAccess function getInt2(k:Int) {
    return this.charAt(k).toUpperCase();
  }
}

class Main {
  static function main() {
    var a = new AString("foo");
    trace(a[0]); // f
  }
}
```

a[0] の配列アクセスは、getInt1 のフィールドとして解決されて、小文字の f が返ります。バージョン 3.2 以前の Haxe では、結果が異なる場合があります。

暗黙のキャスト (2.8.1) が必要な場合であっても、先に定義されている方が優先されます。

2.8.4 選択的関数

コンパイラは抽象型のメンバ関数を静的な (static) 関数へと変化させるので、手で静的な関数を記述してそれを抽象型のインスタンスで使うことができます。この意味は、関数の最初の引数の型で、その関数が使えるようになる静的拡張 (6.3) に似ています。

```
abstract MyAbstract<T>(T) from T {
  public function new(t:T) this = t;

  function get() return this;

  @:impl
  static public function getString(v:MyAbstract<String>):String {
    return v.get();
  }
}

class Main {
  static public function main() {
    var a = new MyAbstract("foo");
    a.getString();
    var b = new MyAbstract(1);
    // Intではなく MyAbstract<String>であるべき
    b.getString();
  }
}
```

抽象型の MyAbstract の getString のメソッドは、最初の引数として MyAbstract<String> を受け取ります。これにより、14 行目の変数 a の関数呼び出しが可能になります (a の型が MyAbstract<String> なので)。しかし、MyAbstract<Int> の変数 b では使えません。

トリビア: 偶然の機能
実際のところ選択的関数は意図して作られたというよりも、発見された機能です。この機能について初めて言及されてから実際に動作せせるまでに必要だったのは軽微な修正のみでした。この発見が、Map のような複数の型の抽象型にもつながっています。

2.8.5 抽象型列挙体

【Haxe 3.1.0 から】

抽象型の宣言に `@:enum` のメタデータを追加することで、その値を有限の値のセットを定義して使うことができます。

```
@:enum
abstract HttpStatus(Int) {
  var NotFound = 404;
  var MethodNotAllowed = 405;
}

class Main {
  static public function main() {
    var status = HttpStatus.NotFound;
    var msg = printStatus(status);
  }

  static function printStatus(status:HttpStatus) {
    return switch(status) {
      case NotFound:
        "ページが見つかりません";
      case MethodNotAllowed:
        "無効なメソッドです";
    }
  }
}
```

以下の JavaScript への出力を見ても明らかなように、Haxe コンパイラは抽象型 `HttpStatus` の全てのフィールドへのアクセスをその値に変換します。

```
Main.main = function() {
    var status = 404;
    var msg = Main.printStatus(status);
};
Main.printStatus = function(status) {
    switch(status) {
    case 404:
        return "Not found";
    case 405:
        return "Method not allowed";
    }
};
```

これはインライン変数 (4.4.2) によく似ていますが、いくつかの利点があります。

- コンパイラがそのセットのすべての値が正しく型付けされていることを保証できます。
- パターンマッチで、抽象型列挙体へのマッチング (6.4) を行う場合に網羅性 (6.4.10) がチェックされます。
- 少ない構文でフィールドを定義できます。

2.8.6 抽象型フィールドの繰り上げ

【Haxe 3.1.0 から】

基底型をラップした場合、その機能性のを“保ちたい”場合があります。繰り上がりの関数を手で書くのは面倒なので、Haxe では `@:forward` メタデータを利用できるようにしています。

```
@:forward(push, pop)
abstract MyArray<S>(Array<S>) {
  public inline function new() {
    this = [];
  }
}

class Main {
  static public function main() {
    var myArray = new MyArray();
    myArray.push(12);
    myArray.pop();
    // MyArray<Int>にはlengthフィールドはありません。
    //myArray.length;
  }
}
```

この例では、抽象型の `MyArray` が `Array` をラップしています。この `@:forward` メタデータは、基底型から繰り上がらせるフィールド 2 つを引数として与えられています。上記の例の `main` 関数は、`MyArray` をインスタンス化して、その `push` と `pop` のメソッドにアクセスしています。コメント化されている行は、`length` フィールドは利用できないことを実演するものです。

ではどのようなコードが出力されるのか、いつものように JavaScript への出力を見てみましょう。

```
Main.main = function() {
    var myArray = [];
    myArray.push(12);
    myArray.pop();
};
```

全てのフィールドを繰り上げる場合は、引数なしの `@:forward` を利用できます。もちろんこの場合でも、Haxe コンパイラは基底型にそのフィールドが存在していることを保証します。

> トリビア: マクロとして実装
> `@:enum` と `@:forward` の両機能は、もともとはビルドマクロ (9.5) を利用して実装していました。この実装はマクロなしのコードから使う場合はうまく動作していましたが、マクロからこれらの機能を使った場合に問題を起こしました。このため、これらの機能はコンパイラへと移されました。

2.8.7　コアタイプの抽象型

Haxe の標準ライブラリは、基本型のセットをコアタイプの抽象型として定義しています。これらは `@:coreType` メタデータを付けることで識別されて、基底型の定義を欠きます。これらの抽象型もまた異なる型の表現として考えることができます。そして、その型は Haxe のターゲットのネイティブの型です。

カスタムのコアタイプの抽象型の導入は、Haxe のターゲットにその意味を理解させる必要があり、ほとんどのユーザーのコードで必要ないでしょう。ですが、マクロを使いたい人や、新しい Haxe のターゲットを作りたい人にとっては興味深い利用例があります

コアタイプの抽象型は、不透過の抽象型 (他の型をラップする抽象型のこと) とは異なる以下の性質をもちます。

- 基底型を持たない。
- `@:notNull` メタデータの注釈を付けない限り、null 許容としてあつかわれる。
- 式の無い配列アクセス (2.8.3) 関数を定義できる。
- Haxe の制限から離れた、式を持たない演算子オーバーロードのフィールド (2.8.2) が可能。

2.9　単相 (モノモーフ)

単相は、単一化 (3.5) の過程で、他の異なる型へと形を変える型です。これについて詳しくは型推論 (3.6) の節で話します。

章 3

型システム

私たちは型 (章 2) の章でさまざまな種類の型について学んできました。ここからはそれらがお互いにどう関連しあっているかを見ていきます。まず、複雑な型に対して別名を与える仕組みである Typedef (3.1) の紹介から簡単に始めます。typedef は特に、型パラメータ (3.2) を持つ型で役に立ちます。

任意の 2 つの型について、その上位の型のグループが矛盾しないかをチェックすることで多くの型安全性が得られます。これがコンパイラが試みる単一化であり、単一化 (ユニファイ) (節 3.5) の節で詳しく説明します。

すべての型はモジュールに所属し、パスを通して呼び出されます。モジュールとパス (節 3.7) では、これらに関連した仕組みについて詳しく説明します。

3.1 typedef

typedef は匿名構造体 (2.5) の節で、すでに登場しています。そこでは複雑な構造体の型について名前を与えて簡潔にあつかう方法を見ています。この利用法は typedef が一体なにに良いのかを的確に表しています。構造体の型 (2.5) に対して名前を与えるのは、typedef の主たる用途かもしれません。実際のところ、この用途が一般的すぎて多くの Haxe ユーザーが typdef を構造体のためのものだと思ってしまっています。

typedef は他のあらゆる型に対して名前を与えることが可能です。

```
1 typedef IA = Array<Int>;
```

これにより `Array<Int>` が使われる場所で、代わりに `IA` を使うことが可能になります。この場合はほんの数回のタイプ数しか減らせませんが、より複雑な複合型の場合は違います。これこそが、typedef と構造体が強く結びついて見える理由です。

```
1 typedef User = {
2   var age : Int;
3   var name : String;
4 }
```

typedef はテキスト上の置き換えではなく、実は本物の型です。Haxe 標準ライブラリの `Iterable` のように型パラメータ (3.2) を持つことができます。

```
1 typedef Iterable<T> = {
2   function iterator() : Iterator<T>;
3 }
```

オプションのフィールド　構造体のフィールドは、@:optional メタデータを使うことで、オプションのフィールドとして認識させることができます。

```
typedef User = {
  var age : Int;
  var name : String;
  @:optional var phoneNumber : String;
}
```

3.1.1　拡張

拡張は、構造体が与えられた型のフィールドすべてと、加えていくつかのフィールドを持っていることを表すために使われます。

```
typedef IterableWithLength<T> = {
  > Iterable<T>,
  // 読み込み専用プロパティ
  var length(default, null):Int;
}

class Main {
  static public function main() {
    var array = [1, 2, 3];
    var t:IterableWithLength<Int> = array;
  }
}
```

大なりの演算子を使うことで、追加のクラスフィールドを持つ Iterable<T> の拡張が作成されました。このケースでは、読み込み専用のプロパティ (4.2) である Int 型の length が要求されます。

IterableWithLength<T> に適合するためには、Iterable<T> にも適合してさらに読み込み専用の Int 型のプロパティ length が必要です。例では、Array が割り当てられており、これはこれらの条件をすべて満たしています。

【Haxe 3.1.0 から】
複数の構造体を拡張することもできます。

```
typedef WithLength = {
  var length(default, null):Int;
}

typedef IterableWithLengthAndPush<T> = {
  > Iterable<T>,
  > WithLength,
  function push(a:T):Int;
}

class Main {
  static public function main() {
    var array = [1, 2, 3];
    var t:IterableWithLengthAndPush<Int> = array;
  }
}
```

3.2 型パラメータ

クラスフィールド (4) や列挙型コンストラクタ (2.4.1) のように、Haxe ではいくつかの型についてパラメータ化を行うことができます。型パラメータは山カッコ <> 内にカンマ区切りで記述することで、定義することができます。シンプルな例は、Haxe 標準ライブラリの Array です。

```
1 class Array<T> {
2   function push(x : T) : Int;
3 }
```

Array のインスタンスが作られると、型パラメータ T は単相 (2.9) となります。つまり、1 度に 1 つの型であれば、あらゆる型を適用することができます。この適用は以下のどちらか方法で行います

明示的に、 new Array<String>() のように型を記述してコンストラクタを呼び出して適用する。

暗黙に 、型推論 (3.6) で適用する。例えば、arrayInstance.push("foo") を呼び出す。

型パラメータが付くクラスの定義の内部では、その型パラメータは不定の型です。制約 (3.2.1) が追加されない限り、コンパイラはその型パラメータはあらゆる型になりうるものと決めつけることになります。その結果、型パラメータの cast (5.23) を使わなければ、その型のフィールドにアクセスできなくなります。また、ジェネリック (3.3) にして適切な制約をつけない限り、その型パラメータの型のインスタンスを新しく生成することもできません。

以下は、型パラメータが使用できる場所についての表です。

パラメータが付く場所	型を適用する場所	備考
Class	インスタンス作成時	メンバフィールドにアクセスする際に型を適用することもできる
Enum	インスタンス作成時	
Enum コンストラクタ	インスタンス作成時	
関数	呼び出し時	メソッドと名前付きのローカル関数で利用可能
構造体	インスタンス作成時	

関数の型パラメータは呼び出し時に適用される、この型パラメータは (制約をつけない限り) あらゆる型を許容します。しかし、一回の呼び出しにつき適用は 1 つの型のみ可能です。このことは関数が複数の引数を持つ場合に役立ちます。

```
1  class Main {
2    static public function main() {
3      equals(1, 1);
4      // 実行時のメッセージ: bar should be foo
5      equals("foo", "bar");
6      // コンパイルエラー: String should be Int
7      equals(1, "foo");
8    }
9
10   static function equals<T>(expected:T, actual:T) {
11     if (actual != expected) {
12       trace('$actual should be $expected');
13     }
14   }
15 }
```

equals 関数の expected と actual の引数両方が、T 型になっています。これは equals の呼び出しで 2 つの引数の型が同じでなければならないことを表しています。コンパイラは最初 (両方の引数が Int 型) と 2 つめ (両方の引数が String 型) の呼び出しは認めていますが、3 つ目の呼び出しはコンパイルエラーにします。

トリビア: 式の構文内での型パラメータ
なぜ、method<String>(x) のようにメソッドに型パラメータをつけた呼び出しができないのか?という質問をよくいただきます。このときのエラーメッセージはあまり参考になりませんが、これには単純な理由があります。それは、このコードでは、くと > の両方が 2 項演算子として構文解析されて、(method < String) > (x) と見なされるからです。

3.2.1 制約

型パラメータは複数の型で制約を与えることができます。

```
1  typedef Measurable = {
2    public var length(default, null):Int;
3  }
4
5  class Main {
6    static public function main() {
7      trace(test([]));
8      trace(test(["bar", "foo"]));
9      // Stringではなく Iterable<String>であるべきです。
10     //test("foo");
11   }
12
13   static function test<T:(Iterable<String>, Measurable)>(a:T) {
14     if (a.length == 0) return "empty";
15     return a.iterator().next();
16   }
17 }
```

test メソッドの型パラメータ T は、Iterable<String> と Measurable の型に制約されます。Measurable の方は、便宜上 typedef (3.1) を使って、Int 型の読み込み専用プロパティ (4.2)length を要求しています。つまり、以下の条件を満たせば、これらの制約と矛盾しません。

- Iterable<String> である
- かつ、Int 型の length を持つ

7 行目では空の配列で、8 行目では Array<String> で test 関数を呼び出すことができることを確認しました。しかし、10 行目の String の引数では制約チェックで失敗しています。これは、String は Iterable<T> と矛盾するからです。

3.3 ジェネリック

大抵の場合、Haxe コンパイラは型パラメータが付けられていた場合でも、1 つのクラスや関数を生成します。これにより自然な抽象化が行われて、ターゲット言語のコードジェネレータは出力先の型パラメータはあらゆる型になりえると思い込むことになります。つまり、生成されたコードで型チェックが働き、動作が邪魔されることがあります。

クラスや関数は、:generic メタデータ (6.9) でジェネリック属性をつけることで一般化することができます。これにより、コンパイラは型パラメータの組み合わせごとのクラスまたは関数を修飾された名前で書き出します。このような設計により静的ターゲット (2.2) のパフォーマンスに直結するコード部位では、出力サイズの巨大化と引き換えに、速度を得られます。

```
@:generic
class MyValue<T> {
  public var value:T;
  public function new(value:T) {
    this.value = value;
  }
}

class Main {
  static public function main() {
    var a = new MyValue<String>("Hello");
    var b = new MyValue<Int>(42);
  }
}
```

めずらしく型の明示をしている MyValue<String> があり、それをいつもの型推論 (3.6) であつかっていますが、これが重要です。コンパイラはコンストラクタの呼び出し時にジェネリッククラスの正確な型な型を知っている必要があります。この JavaScript 出力は以下のような結果になります。

```
(function () { "use strict";
var Test = function() { };
Test.main = function() {
    var a = new MyValue_String("Hello");
    var b = new MyValue_Int(5);
};
var MyValue_Int = function(value) {
    this.value = value;
};
var MyValue_String = function(value) {
    this.value = value;
};
Test.main();
})();
```

MyValue<String> と MyValue<Int> は、それぞれ MyValue_String と MyValue_Int になっています。これはジェネリック関数でも同じです。

```
class Main {
  static public function main() {
    method("foo");
    method(1);
  }

  @:generic static function method<T>(t:T) { }
}
```

JavaScript 出力を見れば明白です。

```
(function () { "use strict";
var Main = function() { }
Main.method_Int = function(t) {
}
Main.method_String = function(t) {
}
```

```
Main.main = function() {
    Main.method_String("foo");
    Main.method_Int(1);
}
Main.main();
})();
```

3.3.1 ジェネリック型パラメータのコンストラクト

定義: ジェネリック型パラメータ
型パラメータを持っているクラスまたはメソッドがジェネリックであるとき、その型パラメータもジェネリックであるという。

普通の型パラメータでは、new T() のようにその型をコンストラクトすることはできません。これは、Haxe が 1 つの関数を生成するために、そのコンストラクトが意味をなさないからです。しかし、型パラメータがジェネリックの場合は違います。これは、コンパイラはすべての型パラメータの組み合わせに対して別々の関数を生成しています。このため new T() の T を実際の型に置き換えることができます。

```
typedef Constructible = {
  public function new(s:String):Void;
}

class Main {
  static public function main() {
    var s:String = make();
    var t:haxe.Template = make();
  }

  @:generic
  static function make<T:Constructible>():T {
    return new T("foo");
  }
}
```

ここでは、T の実際の型の決定は、トップダウンの推論 (3.6.1) で行われることに注意してください。この方法での型パラメータのコンストラクトを行うには 2 つの必須事項があります。

1. ジェネリックであること
2. 明示的に、コンストラクタ (2.3.1) を持つように制約 (3.2.1) されていること

先ほどの例は、1 つ目は make が @:generic メタデータを持っており、2 つ目 T が Constructible に制約されています。String と haxe.Template の両方とも 1 つ String の引数のコンストラクタを持つのでこの制約に当てはまります。確かに JavaScript 出力は予測通りのものになっています。

```
var Main = function() { }
Main.__name__ = true;
Main.make_haxe_Template = function() {
    return new haxe.Template("foo");
}
Main.make_String = function() {
```

```
    return new String("foo");
}
Main.main = function() {
    var s = Main.make_String();
    var t = Main.make_haxe_Template();
}
```

3.4 変性 (バリアンス)

変性とは他のものとの関連を表すもので、特に型パラメータに関するものが連想されます。そして、この文脈では驚くようなことがよく起こります。変性のエラーを起こすことはとても簡単です。

```
class Base {
  public function new() { }
}

class Child extends Base { }

class Main {
  public static function main () {
    var children = [new Child()];
    // Array<Child> should be Array<Base>
    // Type parameters are invariant
    // Child should be Base
    var bases:Array<Base> = children;
  }
}
```

見てわかるとおり、Child は Base に代入できるにもかかわらず、Array<Child> を Array<Base> に代入することはできません。この理由は少々予想外のものかもしれません。それはこの配列への書き込みが可能だからです。例えば、push() メソッドです。この変性のエラーを無視してしまうことは簡単です。

```
class Base {
  public function new() { }
}

class Child extends Base { }

class OtherChild extends Base { }

class Main {
  public static function main () {
    var children = [new Child()];
    // 型チェックを黙らせる
    var bases:Array<Base> = cast children;
    bases.push(new OtherChild());
    for(child in children) {
      trace(child);
    }
  }
}
```

cast (5.23) を使って型チェッカーを破壊して、12 行目の代入を可能にしてしまっています。bases は元々の配列への参照を持っており、Array<Base> の型付けをされています。このため、Base に適合する別の型の OtherChild を配列に追加できます。しかし、元々の children の参照は Array<Child> のままです。そのため良くないことに繰り返し処理の中で OtherChild のインスタンスに出くわします。

もし Array が push() メソッドを持っておらず、他の編集方法も無かったならば、適合しない型を追加することができなくなるのでこの代入は安全になります。Haxe では構造的部分型付け (3.5.2) を使って型を適切に制限することでこれを実現できます。

```
1  class Base {
2    public function new() { }
3  }
4
5  class Child extends Base { }
6
7  typedef MyArray<T> = {
8    public function pop():T;
9  }
10
11 class Main {
12   public static function main () {
13     var a = [new Child()];
14     var b:MyArray<Base> = a;
15   }
16 }
```

b は MyArray<Base> として型付けされており、MyArray は pop() メソッドしか持たないため、安全に代入することができます。MyArray には適合しない型を追加できるメソッドを持っておらず、このことは共変性と呼ばれます。

定義: 共変性
複合型 (2) がそれを構成する型よりも一般な型で構成される複合型に代入できる場合に、共変であるという。つまり、読み込みのみが許されて書き込みができない場合です。

定義: 反変性
複合型 (2) がそれを構成する型よりも特殊な型で構成される複合型に代入できる場合に、反変であるという。つまり、書き込みのみが許されて読み込みができない場合です。

3.5 単一化 (ユニファイ)

Mention toString()/ String conversion somewhere in this chapter.

単一化は型システムのかなめであり、Haxe の堅牢さに大きく貢献しています。この節ではある型が他の型と適合するかどうかをチェックする過程を説明していきます。

定義: 単一化
型 A の型 B での単一化というのは、A が B に代入可能かを調べる指向性を持つプロセスです。型が単相 (2.9) の場合または単相をふくむ場合は、それを変化させることができます。

単一化のエラーは簡単に起こすことができます。

```
1 class Main {
2   static public function main() {
3     // Int should be String
4     // (Intではなく Stringであるべき)
5     var s:String = 1;
6   }
7 }
```

Int 型の値を String 型の変数に代入しようとしたので、コンパイラは Int を String で単一化しようと試みます。これはもちろん許可されておらず、コンパイラは"Int should be String"というエラーを出力します。

このケースでは単一化は代入によって引き起こされており、この文脈は「代入可能」という定義に対して直感的です。ただ、これは単一化が働くケースのうちの 1 つでしかありません。

代入: a が b に代入された場合、a の型は b で単一化されます。

関数呼び出し: 関数 (2.6) の型の紹介ですでに触れています。一般的に言うと、コンパイラは渡された最初の引数の型を要求される最初の引数の型で単一化し、渡された二番目の引数の型を要求される二番目の引数の型で単一化するということを、すべての引数で行います。

関数の return: 関数が return e の式をもつ場合は常に e の型は関数の戻り値の型で単一化されます。もし関数の戻り値の型が明示されていなければ、e の型に型推論されて、それ以降の return 式は e の型に推論されます。

配列の宣言: コンパイラは、配列の宣言では与えられたすべての型に共通する最小の型を見つけようとします。詳細は共通の基底型 (節 3.5.5) を参照してください。

オブジェクトの宣言: オブジェクトを指定された型に対して宣言する場合、コンパイラは与えられたフィールドすべての型を要求されるフィールドの型で単一化します。

演算子の単一化: 特定の型を要求する特定の演算子は、与えられた型をその型で単一化します。例えば、a && b という式は a と b 両方を Bool で単一化します。式 a == b は a を b で単一化します。

3.5.1 クラスとインターフェースの単一化

クラスの間の単一化について定義を行う場合、単一化が指向性を持つことを心に留めておくべきです。より特殊なクラス (例えば、子クラス) を一般的なクラス (例えば、親クラス) に対して代入することはできますが、逆はできません。

以下のような、代入が許可されます。

- 子クラスの親クラスへの代入
- クラスの実装済みのインターフェースへの代入
- インターフェースの親インターフェースへの代入

これらのルールは連結可能です。つまり、子クラスをその親クラスの親クラスへ代入可能であり、さらに親クラスが実装しているインターフェースへ代入可能であり、そのインターフェースの親インターフェースへ代入可能であるということです。

"parent class" should probably be used here, but I have no idea what it means, so I will refrain from changing it myself.

3.5.2 構造的部分型付け

定義: 構造的部分型付け
構造的部分型付けは、同じ構造を持つ型の暗黙の関係を示します。

Haxe では構造的部分型付けは、以下の単一化するときに利用可能です。

- クラス (2.3) を構造体 (2.5) で単一化
- 構造体を別の構造体で単一化

以下のサンプルは、Haxe 標準ライブラリ (10) の Lambda のクラスの一部です。

```
1 public static function empty<T>(it : Iterable<T>):Bool {
2   return !it.iterator().hasNext();
3 }
```

empty メソッドは、Iterable が要素を持つかをチェックします。この目的では、引数の型について、それが列挙可能 (Iterable) であること以外に何も知る必要がありません。Haxe 標準ライブラリにはたくさんある Iterable<T> で単一化できる型すべてで、これを呼び出すことができるわけです。この種の型付けは非常に便利ですが、静的ターゲットでは大量に使うとパフォーマンスの低下を招くことがあります。詳しくはパフォーマンスへの影響 (節 2.5.4) に書かれています。

3.5.3 単相

単相 (2.9) である、あるいは単相をふくむ型についての単一化は型推論 (節 3.6) で詳しくあつかいます。

3.5.4 関数の戻り値

関数の戻り値の型の単一化では Void 型 (2.1.5) も関連しており、Void 型での単一化のはっきりとした定義が必要です。Void 型は型の不在を表し、あらゆる型が代入できません。Dynamic でさえも代入できません。つまり、関数が明示的に Dynamic を返すと定義されている場合、Void を返してはいけません。

その逆も同じです。関数の戻り値が Void であると宣言しているなら、Dynamic やその他すべての型は返すことができません。しかし、関数の型を代入する時のこの方向の単一化は許可されています。

```
1 var func:Void->Void = function() return "foo";
```

右辺の関数ははっきりと Void->String 型ですが、これを Void->Void 型の func 変数に代入することができます。これはコンパイラが戻り値は無関係だと安全に判断できるからで、その結果 Void ではないあらゆる型を代入できるようになります。

3.5.5 共通の基底型

複数の型の組み合わせが与えられたとき、そのすべての型が共通の基底型で単一化されます。

```
1 class Base {
2   public function new() { }
3 }
4
5 class Child1 extends Base { }
6 class Child2 extends Base { }
7
```

```
8  class Main {
9    static public function main() {
10     var a = [new Child1(), new Child2()];
11     $type(a); // Array<Base>
12   }
13 }
```

Base とは書かれていないにも関わらず、Haxe コンパイラは Child1 と Child2 の共通の型として Base を推論しています。Haxe コンパイラはこの方法の単一化を以下の場面で採用しています。

- 配列の宣言
- if/else
- switch のケース

3.6 型推論

型推論はこのドキュメントで何度も出てきており、これ以降でも重要です。型推論の動作の簡単なサンプルをお見せします。

```
1  class Main {
2    public static function main() {
3      var x = null;
4      $type(x); // Unknown<0>
5      x = "foo";
6      $type(x); // String
7    }
8  }
```

この特殊な構文 $type は、関数 (節 2.6) の型の説明をわかりやすくするためにも使っていました。それではここで公式な説明をしましょう。

> Construct: $type
> $type は関数のように呼び出せるコンパイル時の仕組みで、一つの引数を持ちます。コンパイラは引数の式を評価し、そしてその式の型を出力します。

上記の例では、最初の $type では Unknown<0> が表示されます。これは単相 (2.9) で、未知の型です。次の行の x = "foo" で定数値の String を x に代入しており、String の単相での単一化 (3.5) が起こります。そして、x がこのとき String に変わったことがわかります。

ダイナミック (節 2.7) 以外の型が、単相での単一化を行うと単相はその型になります (その型に変形 (morph) します)。このため、この型はもう別の型には変形できません。これが単相 (monomorph) の mono の部分です。

以下が単一化のルールです。型推論は複合型でも起こります。

```
1  class Main {
2    public static function main() {
3      var x = [];
4      $type(x); // Array<Unknown<0>>
5      x.push("foo");
6      $type(x); // Array<String>
7    }
8  }
```

変数 x は初め空の Array で初期化されています。この時点で x の型は配列であると言えますが、配列の要素の型については未知です。その結果 x の型は、Array<Unknown<0>> となります。この配列が Array<String> だとわかるには、String をこの配列にプッシュするだけで十分です。

3.6.1 トップダウンの推論

多くの場合、ある型はその型で要求される型を推論します。しかし一部では、要求される型で型を推論します。これをトップダウンの推論と呼びます。

> 定義: 要求される型
> 要求される型は、式の型が式が型付けされるより前にわかっている場合に現れます。例えば、式が関数の呼び出しの引数の場合です。この場合、トップダウンの推論 (3.6.1) と呼ばれる方法で、式に型が伝搬します。

良い例は型の混ざった配列です。ダイナミック (節 2.7) で書いた通り、[1, "foo"] は要素の型を決定できないので、コンパイラはこれを拒絶します。これはトップダウンの推論を使えば解決します。

```
1  class Main {
2    static public function main() {
3      var a:Array<Dynamic> = [1, "foo"];
4    }
5  }
```

ここでは、[1, "foo"] に型付けするとき、要求される型が Array<Dynamic> であり、その要素は Dynamic であるとわかります。コンパイラが共通の基底型 (3.5.5) を探す (そして失敗する) 通常の単一化の挙動の代わりに、個々の要素が Dynamic で単一化され、型付けされます。

ジェネリック型パラメータのコンスラクト (3.3.1) の紹介で、もう一つトップダウンの推論の面白い利用例を見ています。

```
1  typedef Constructible = {
2    public function new(s:String):Void;
3  }
4
5  class Main {
6    static public function main() {
7      var s:String = make();
8      var t:haxe.Template = make();
9    }
10
11   @:generic
12   static function make<T:Constructible>():T {
13     return new T("foo");
14   }
15 }
```

String と haxe.Template の明示された型が、make の戻り値の型の決定に使われています。これは、make() の戻り値が変数へ代入されるのがわかっているので動作します。この方法を使うと、未知の型 T をそれぞれ String と haxe.Template に紐づけることが可能です。

3.6.2 制限

ローカル変数をあつかう場合、型推論のおかげで多くの手動の型付けを省略できますが、型システムが助けを必要とする場面もあります。実際、変数フィールド (4.1) やプロパティ (4.2) では、直接の初期化

をしていない限りは型推論されません。

再帰的な関数呼び出しでも型推論が制限される場面があります。型がまだ (完全に) わかっていない場合、型推論が間違って特殊すぎる型を推論することがあります。

コードの可読性について、違った意味での制限もあります。型推論を乱用すれば、明示的な型が無くなってプログラムが理解しにくくなることもあります。特にメソッドを定義する場合です。型推論と明示的な型注釈のバランスはうまくとるようにしてください。

3.7 モジュールとパス

定義: モジュール
すべての Haxe のコードはモジュールに属しており、パスを使って指定されます。要するに、.hx ファイルそれぞれが一つのモジュールを表し、その中にいくつか型を置くことができます。型は `private` にすることが可能で、その場合はその型の属するモジュールからしかアクセスできません。

モジュールとそれにふくまれる型との区別は意図的に不明瞭です。実際、`haxe.ds.StringMap<Int>` の指定は、`haxe.ds.StringMap.StringMap<Int>` の省略形とも考えられます。後者は 4 つ部位で構成されています。

1. パッケージ `haxe.ds`
2. モジュール名 `StringMap`
3. 型名 `StringMap`
4. 型パラメータ `Int`

モジュールと型の名前が同じの場合、重複を取り除くことが可能で、これで `haxe.ds.StringMap<Int>` という省略形が使えます。しかし、長い記述について知っていれば、モジュールのサブタイプ (3.7.1) の指定の仕方の理解しやすくなります。

パスは、import (3.7.2) を使ってパッケージの部分を省略することで、さらに短くすることができます。import の利用は不適切な識別子を作ってしまう場合があるので、解決順序 (3.7.3) についての理解が必要です。

定義: 型のパス
(ドット区切りの) 型のパスはパッケージ、モジュール名、型名から成ります。この一般的な形は `pack1.pack2.packN.ModuleName.TypeName` です。

3.7.1 モジュールのサブタイプ (従属型)

モジュールサブタイプとは、その型を定義しているモジュールの名前と異なる名前の型です。これにより、一つの.hx ファイルに複数の型の定義が可能になり、これらの型はモジュール内では無条件でアクセス可能で、ほかのモジュールからも `package.Module.Type` の形式でアクセスできます。

```
1 var e:haxe.macro.Expr.ExprDef;
```

ここでは `haxe.macro.Expr` のサブタイプ `ExprDef` にアクセスしています。

この従属関係は、実行時には影響を与えません。このため、public なサブタイプはパッケージのメンバーになり、同じパッケージの別々のモジュールで同じサブタイプを定義した場合に衝突を起こします。当然、Haxe コンパイラはこれを検出して適切に報告します。上記の例では `ExprDef` は `haxe.macro.ExprDef` として出力されます。

サブタイプは以下のように `private` にすることが可能です。

```
private class C { ... }
private enum E { ... }
private typedef T { ... }
private abstract A { ... }
```

定義: private 型
型は private の修飾子を使って可視性を下げることが可能です。private 修飾子をつけると、その型を定義しているモジュール (3.7) 以外からは、直接アクセスできなくなります。
　private な型は public な型とは異なり、パッケージにはふくまれません。

型の可視性は、アクセスコントーロル (6.10) を使うことでより細かく制御することができます。

3.7.2　インポート (import)

型のパスが一つの.hx ファイルで複数回使われる場合、import を使ってそれを短縮するのが効果的でしょう。import は、以下のように型の使用時のパッケージの省略を可能にします。

```
import haxe.ds.StringMap;

class Main {
  static public function main() {
    // new haxe.ds.StringMap();の代わり。
    new StringMap();
  }
}
```

haxe.ds.StringMap を 1 行目でインポートをすることで、コンパイラは main 関数の StringMap を haxe.ds パッケージのものとして解決することができます。これを、StringMap が現在のファイルにインポートされていると言います。
　上記の例では、1 つのモジュールをインポートしていますが、インポートされる型は 1 つとは限りません。つまり、インポートしたモジュールにふくまれるすべての型が利用可能になります。

```
import haxe.macro.Expr;

class Main {
  static public function main() {
    var e:Binop = OpAdd;
  }
}
```

Binop 型は、haxe.macro.Expr モジュールで定義されている enum (2.4) で、このモジュールのインポートで利用可能になりました。もし、モジュール内の特定の型のみを指定してインポート (例えば、import haxe.macro.Expr.ExprDef) した場合、プログラムは Class not found : Binop でコンパイルが失敗します。
　インポートには、いくつかの知っておくべきポイントがあります。

- 　より後に書かれたインポートが優先されます。(詳しくは、解決順序 (節 3.7.3))
- 静的拡張 (6.3) のキーワードの using は import の効果をふくみます。
- enum がインポートされると、(直接か、モジュールの一部としてかを問わず)、その enum コンストラクタ (2.4.1) のすべてもインポートされます。(上述の例、OpAdd の利用例をみてください)

さらに、クラスの静的フィールド (4) をインポートして使うこともできます。

```
import Math.random;

class Main {
  static public function main() {
    random();
  }
}
```

フィールド名やローカル変数名と、パッケージ名は衝突するので、特別な気づかいが必要です。このとき、フィールド名やローカル変数は、パッケージ名よりも優先されます。haxe と命名された変数名は、haxe というパッケージの使用を妨害します。

ワイルドカードインポート　Haxe では.* を使用することで、パッケージにふくまれるすべてのモジュール、またはモジュールにふくまれるすべての型、あるいは型が持つすべてのフィールドをインポートすることができます。この種類のインポートは、以下のように一段階しか適用されないことに気をつけてください。

```
import haxe.macro.*;

class Main {
  static function main() {
    var expr:Expr = null;
    //var expr:ExprDef = null; // ExprDefクラスが見つかりません
  }
}
```

haxe.macro のワイルドカードインポートを使うことで、このパッケージにふくまれる Expr モジュールにアクセスできるようになりましたが、Expr モジュールのサブタイプの ExprDef にはアクセスできません。このルールは、モジュールをインポートしたときの静的フィールドに対しても同じです。

パッケージに対するワイルドカードインポートでは、コンパイラはそのパッケージにふくまれるモジュールを貪欲にコンパイルするわけではありません。つまり、これらのモジュールは明示的に使用されない限り、コンパイラが認識することはなく、生成された出力の中にはふくまれません。

別名 (エイリアス) を使ったインポート　型や静的フィールドをインポートしたモジュール内で大量につかう場合、短い別名をつけるのが有効かもしれません。別名は衝突した名前に対して、ユニークな名前をあたえて区別するのにも役立ちます。

```
import String.fromCharCode in f;

class Main {
  static function main() {
    var c1 = f(65);
    var c2 = f(66);
    trace(c1 + c2); // AB
  }
}
```

ここでは String.fromCharCode を f としてインポートしたので、f(65) や f(66) といった使い方ができます。同じことはローカル変数でもできますが、別名を使う方法はコンパイル時のみのものなので実行時のオーバーヘッドが発生しません。
【Haxe 3.2.0 から】

Haxe では、as の代わりにより自然な in キーワードを使うことも可能です。

3.7.3　解決順序

不適切な識別子が入り組んでいる場合には、解決順序があらわれます。式 (5) には、foo()、foo = 1、foo.field の形があり、とくに最後の形では、haxe が不適切な識別子な場合の haxe.ds.StringMap のようなモジュールのパスの可能性もふくんでいます。

これがその解決順序のアルゴリズムです。以下の各状態が影響しています。

- 宣言されているローカル変数 (5.10) (関数の引数もふくむ)
- インポート (3.7.2) されたモジュール、型、静的フィールド。
- 利用可能な静的拡張 (6.3)
- 現在のフィールドの種類 (静的フィールドなのか、メンバフィールドなのか)
- 現在のクラスと親クラスで定義されている、メンバフィールド
- 現在のクラスで定義されている、静的フィールド
- 期待される型 (3.6.1)
- untyped 中の式か、そうでないか

proper label and caption + code/identifier styling for diagram

i を例にすると、このアルゴリズムは以下のようなものです。

1. i が true、false、this、super、null のいずれかの場合、その定数として解決されて終了。
2. i というローカル変数があった場合、それに解決されて終了。
3. 現在いるフィールドが、静的フィールドであれば、6に進む。
4. 現在のクラスか、いずれかの親クラスで i のメンバフィールドが定義されている場合、それに解決されて終了。
5. 静的拡張の第 1 引数として現在のクラスのインスタンスが利用可能な場合、それに解決されて終了。
6. 現在のクラスが i という静的フィールドを持っている場合、それに解決されて終了。
7. インポート済みの enum に i というコンストラクタがあった場合、それに解決されて終了。
8. i という名前の静的フィールドが明示的にインポートされていた場合それに解決されて終了。
9. i が小文字から始まる場合、11に進む。
10. i という型が利用可能な場合、それに解決されて進む。
11. 式が untyped 中にいない場合、14に進む。
12. i が__this__の場合、this として解決されて終了。
13. ローカル変数の i を生成し、それに解決されて終了。
14. 失敗

10のステップについて、型の解決順序の定義も必要です。

1. i がインポートされている場合 (直接か、モジュールの一部としてか、にかかわらず)、それに解決されて終了。

2. 現在のパッケージが ｉという名前モジュールの ｉという型をふくんでいる場合、それに解決されて終了。

3. ｉがトップレベルで利用可能な場合、それに解決されて終了。

4. 失敗

このアルゴリズムの1のステップと、式の場合の5と7 のステップでは、以下のインポートの解決順序も重要です。

- インポートしたモジュールと静的拡張は、下から上へとチェックされて最初にマッチしたものが使われます。
- 一つのモジュールの中では、型は上から下へとチェックされていきます。
- インポートでは、名前が一致した場合ににマッチしたものとなります。
- 静的拡張 (6.3) では、名前が一致して、なおかつ最初の引数が単一化 (3.5) できると、マッチが成立します。静的拡張として使われる一つの型の中では、フィールドは上から下へとチェックされます

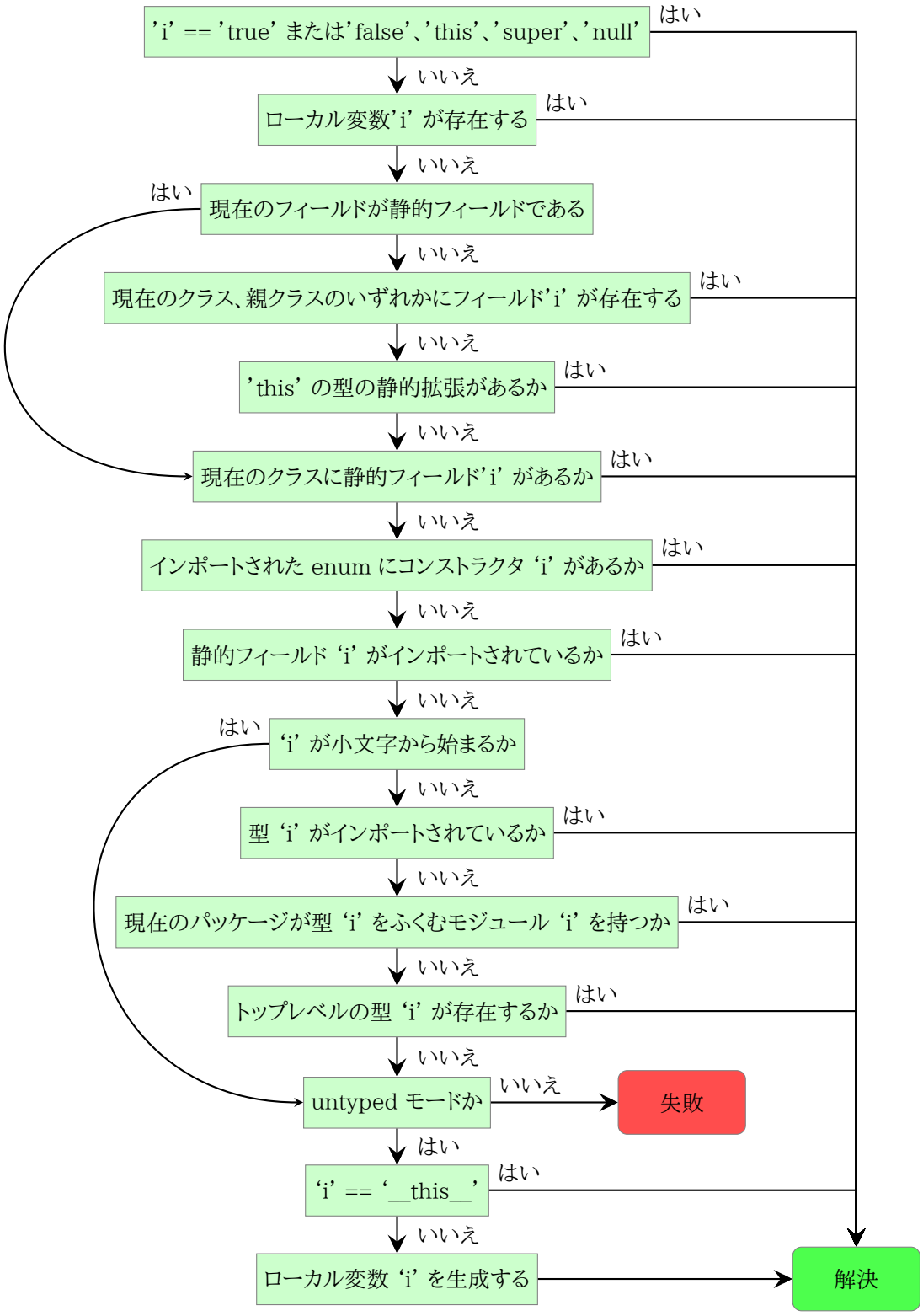

Figure 3.1: 識別子 'i' の解決順序

章 4

クラスフィールド

定義: クラスフィールド
クラスフィールドはクラスに属する変数、プロパティまたはメソッドです。これは静的、または非静的になることができます。静的メソッドとメンバ変数といった名前を使うのと同じように、非静的フィールドについてはメンバフィールドと呼びます。

ここまで、一般的な Haxe のプログラムと型がどのように構成されているのかを見てきました。このクラスフィールドに関する章では、構成に関する話題をまとめて、Haxe の動作に関する話題へのかけ橋とします。これはクラスフィールドが式 (5) を持つ場所だからです。

クラスフィールドには 3 種類あります。

変数: 変数 (4.1) クラスフィールドにはある型の値が入っていて、参照、または代入することがができます。

プロパティ: プロパティ (4.2) クラスフィールドはアクセスされた時のカスタムの動作を定義します。クラスの外からは変数フィールドのように見えます。

メソッド: メソッド (4.3) はコードを実行するために呼び出すことのできる関数です。

厳密に言うと、変数は特定のアクセス方法を持つプロパティであると見なせます。実際に、Haxe コンパイラは変数とプロパティを型付けの段階では区別していません。しかし、構文レベルでは区別されます。

用語について補足すると、メソッドは (静的また非静的の) クラスに属する関数であり、式の中で現れるローカル関数 (5.11) のようなその他の関数はメソッドではありません。

4.1 変数

変数フィールドについては、すでに前章でいくつかのサンプルコードで見てきました。変数フィールドは値を保持するもので、その性質はほとんどプロパティと共通しています (すべてでは無い)。

```
1 class Main {
2   static var member:String = "bar";
3
4   public static function main() {
5     trace(member);
6     member = "foo";
7     trace(member);
8   }
9 }
```

ここから変数が以下のようなものだとわかります。

1. 名前を持つ (ここでは `member`),
2. 型を持つ (ここでは `String`),
3. 一定の初期値を持つ場合がある (ここでは"bar") and
4. アクセス修飾子 (4.4) を持つ場合がある (ここでは `static`)

上の例は最初に `member` の初期値を出力した後、"foo" を割り当ててから新しい値を出力しています。アクセス修飾子の効果は 3 種類のクラスフィールドで共通しており、その内容については後の節で説明します。

変数フィールドが初期値をもつ場合には、型の明示は不要になります。この場合、コンパイラが推論 (3.6) を行います。

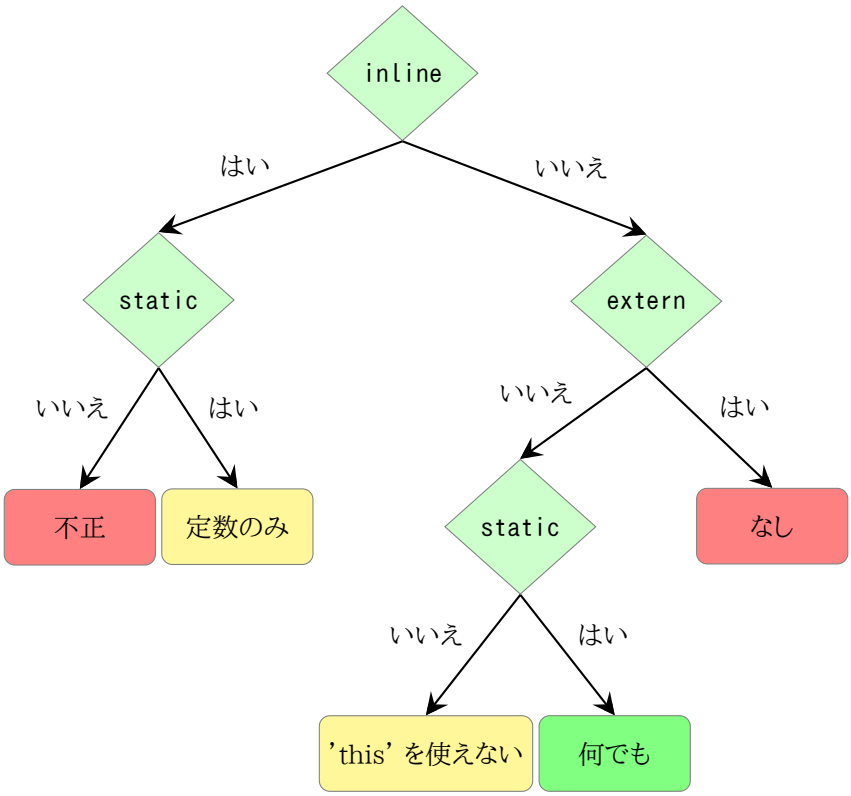

Figure 4.1: 変数フィールドの値の初期化

4.2 プロパティ

変数 (4.1) に続き、プロパティがクラスにデータ持つ 2 番目の方法になります。変数とは異なり、プロパティはどのようなアクセスが許可されるかと、どのように生成されるかのより細かい制御が要求されます。よくある使い方は、例えば以下のようなものです。

- どこからでも読み込み可能だが、書き込みは定義しているクラスからのみのフィールドを作る

・ 読み込みアクセスがされたときにゲッターメソッドが実行されるフィールドを作る。

・ 書き込みアクセスがされたときにセッターメソッドが実行されるフィールドを作る。

プロパティをあつかう場合、2 種類のアクセスについて理解することが重要です。

定義: 読み込みアクセス
読み込みアクセスは右辺側でフィールドアクセス式 (5.7) が使われると発生します。これには obj.field() の形の関数呼び出しもふくまれるため、この field も読み込みアクセスがされます。

定義: 書き込みアクセス
フィールドへの書き込みアクセスは、フィールドアクセス式 (5.7) に obj.field = value の形式で値の代入することで発生します。また、obj.field += value の式 += のような特殊な代入演算子を使うと、書き込みアクセスと読み込みアクセス (4.2) の両方が発生します。

読み込みアクセスと書き込みアクセスを以下の構文を使って直接指定します。

```
1 class Main {
2   public var x(default, null):Int;
3   static public function main() { }
4 }
```

大部分は変数の構文と同じで、同じルールが適用されます。プロパティは以下の点で異なります。

・ フィールド名の後から小かっこが始まります (()。

・ 次に、特殊なアクセス識別子が来ます (ここでは default)。

・ カンマ (,) で区切ります。

・ もう一つ特殊なアクセス識別子が続きます (ここでは null)。

・ 小かっこを閉じます ())。

1 つ目のアクセス識別子はフィールドの読み込み、2 つ目は書き込み時の挙動を決定します。アクセス識別子には以下の値が使用できます。

default: フィールドの可視性が public の場合、通常のフィールドと同じです。その他の場合、null アクセスと同じです。

null: 定義したクラスのみからアクセスできます。

get/set: アクセス時にアクセサメソッドを呼び出します。コンパイラが使用可能なアクセサの存在を確認します。

dynamic: get/set アクセスに似ていますが、アクセサフィールドの存在を確認しません。

never: いかなるアクセスも許可しません。

定義: アクセサメソッド
型が T でフィールド名が field のフィールドに対するアクセサメソッドは、Void->T 型のフィールド名 get_field のゲッターまたは T->T 型のフィールド名 set_field のセッターです。アクセサメソッドは略してアクセサとも呼びます。

トリビア: アクセサ名
Haxe2 では、アクセス識別子に自由な識別子を使うことが可能で、その場合はそれがカスタムのアクセサメソッド名となっていました。しかし、これにより実装は変則的なものになっていました。例えば、Reflect.getProperty() と Reflect.setProperty() はどのような名前が名前が使われていたとしても対応する必要がありました。そのため、ターゲット出力時に参照のためのメタ情報を生成する必要がありました。
この識別子の名前を get_、set_から始まるもののみに制限することで、実装を大きく簡略化することに成功しました。これが Haxe2 と 3 の間の破壊的な変更の 1 つです。

4.2.1 よくあるアクセス識別子の組み合わせ

次の例はプロパティのよくあるアクセス識別子の組み合わせです。

```
1  class Main {
2    // 外からの読み込みが可能で、Mainからのみ書き込みが可能。
3    public var ro(default, null):Int;
4
5    // 外からの書き込みが可能で、Mainからのみ読み込みが可能。
6    public var wo(null, default):Int;
7
8    // ゲッターのget_xと
9    // セッターのset_xを通してアクセスする
10   public var x(get, set):Int;
11
12   // ゲッターを通して読み込みアクセスし、
13   // 書き込みアクセスはできない。
14   public var y(get, never):Int;
15
16   // xフィールドに必要
17   function get_x() return 1;
18
19   // xフィールドに必要
20   function set_x(x) return x;
21
22   // yフィールドに必要
23   function get_y() return 1;
24
25   function new() {
26     var v = x;
27     x = 2;
28     x += 1;
29   }
30
31   static public function main() {
32     new Main();
33   }
34 }
```

main メソッドの JavaScript へのコンパイル結果は、フィールドアクセスがどのようなものなのか理解する助けになるでしょう。

```
var Main = function() {
    var v = this.get_x();
    this.set_x(2);
    var _g = this;
    _g.set_x(_g.get_x() + 1);
};
```

このとおり、読み込みアクセスは get_x() の呼び出しとなり、書き込みアクセスは x への 2 の代入が set_x(2) の呼び出しになりました。+= の場合の出力は最初は少し不思議に見えるかもしれませんが、次の例で簡単にわかるはずです。

```
class Main {
  public var x(get, set):Int;
  function get_x() return 1;
  function set_x(x) return x;

  public function new() { }

  static public function main() {
    new Main().x += 1;
  }
}
```

main メソッドの x のフィールドアクセスについて、ここで起きる事象は複雑です。まずこの場合は、Main のインスタンス化という副作用があります。そのため、コンパイラは new Main().x = new Main().x + 1 という出力を行わないように、複雑な式をローカル変数にキャッシュします。

```
Main.main = function() {
    var _g = new Main();
    _g.set_x(_g.get_x() + 1);
}
```

4.2.2 型システムへの影響

プロパティの存在は型システム対して、いくつかの重要な影響をもたらします。もっとも重要なのはプロパティはコンパイル時の機能であり、型がわかっている必要があるということです。クラスインスタンスを Dynamic に代入すると、フィールドアクセスはアクセサメソッドを参照しません。同じようにアクセス制限も働かなくなり、すべてのアクセスは public と同じになります。

get または set のアクセス識別子を使うと、コンパイラはゲッターとセッターが本当に存在するかを確認します。以下はコンパイルできません。

```
class Main {
  // xプロパティに必要とされるget_xメソッドが足りません。
  public var x(get, null):Int;
  static public function main() {}
}
```

get_x メソッドを忘れていますが、親クラスでそれが定義されていた場合は今のクラスでそれを定義する必要はなくなります。

```
class Base {
  public function get_x() return 1;
}

```

```
5  class Main extends Base {
6    // get_xが親クラスで宣言されているので問題ありません
7    public var x(get, null):Int;
8
9    static public function main() {}
10 }
```

dynamic アクセス識別子は get や set と同じように動作しますが、この存在チェックは行われません。

4.2.3 ゲッターとセッターのルール

アクセサメソッドの可視性は、プロパティの可視性に影響を与えません。つまり、プロパティが public であってもそのゲッターは private でも構わないということです。

ゲッターとセッターは、その物理的フィールドにアクセスしてデータを使用する場合があります。アクセサメソッド自身からそのフィールドへのアクセスが行われた場合、コンパイラはこれをアクセサメソッド経由しないアクセスと見なします。これにより無限ループが回避されます。

```
1  class Main {
2    public var x(default, set):Int;
3
4    function set_x(newX) {
5      return x = newX;
6    }
7
8    static public function main() {}
9  }
```

しかし、フィールドが少なくとも 1 つ、default または null のアクセス識別子を持つ時のみ、コンパイラはその物理的フィールドが存在していると考えます。

定義: 物理的フィールド
以下のいずれかの場合にフィールドが物理的であると考えられます

- 変数 (4.1)
- 読み込みアクセスか書き込みアクセスのアクセス識別子が default または null であるプロパティ (4.2)
- :isVar メタデータ (6.9) がつけられたプロパティ (4.2)

これらのケースに含まれない場合、アクセサメソッド内での自身のフィールドへのアクセスはコンパイルエラーを起こします。

```
1  class Main {
2    // 物理的フィールドを持たないので、フィールドにアクセス
3    public var x(get, set):Int;
4
5    function get_x() {
6      return x;
7    }
8
9    function set_x(x) {
```

```
10      return this.x = x;
11    }
12
13    static public function main() {}
14  }
```

物理的フィールドが必要であれば、:isVar メタデータ (6.9) をフィールドつけることでそれを強制できます。

```
1   class Main {
2     // @isVarが物理的フィールドを強制するのでコンパイル可能になります。
3     @:isVar public var x(get, set):Int;
4
5     function get_x() {
6       return x;
7     }
8
9     function set_x(x) {
10      return this.x = x;
11    }
12
13    static public function main() {}
14  }
```

トリビア: プロパティのセッターの型
新しい Haxe のユーザーにとって、セッターの型が T->Void ではなくて T->T でなくてはいけないというのはなじみがなく、驚かれるかもしれません。ではなぜ setter が値を返す必要があるのでしょうか?
それはセッターを使ったフィールドへの代入を右辺の式として使いたいからです。x = y = 1 のような連結された式は、x = (y = 1) として評価されます。x に y = 1 の結果を代入するためには、y = 1 が値を持たなければなりません。y のセッターの戻り値が Void であれば、それは不可能です。

4.3 メソッド

変数 (4.1) がデータを保持する一方で、メソッドは式 (5) をもってプログラムの動作を定義します。このマニュアルのさまざまなサンプルコード中で、メソッドフィールドを見てきました。最初の Hello World (1.3) の例ですら、main メソッドとして現れています。

```
1   class Main {
2     static public function main():Void {
3       trace("Hello World");
4     }
5   }
```

メソッドは function キーワードから始まることで識別されます。そして以下の要素を持ちます。

1. 名前を持つ (ここでは main)。
2. 引数のリストを持つ (ここでは空の ())。
3. 戻り値を持つ (ここでは Void)。

4. アクセス修飾子 (4.4) を持つ場合がある (ここでは static と public)

5. 式を持つ場合がある (ここでは {trace("Hello World");})。

引数と戻り値の型について学ぶために次の例を見てみましょう。

```
1  class Main {
2    static public function main() {
3      myFunc("foo", 1);
4    }
5
6    static function myFunc(f:String, i) {
7      return true;
8    }
9  }
```

引数はフィールド名の後に、小かっこ (() を続け、引数の詳細のリストをカンマ (,) 区切りで並べて、小かっこを閉じる ()) ことで記述します。引数の詳細についての情報は関数 (節 2.6) で説明されています。

この例からは型推論 (3.6) が引数と戻り値についてどのように動作するのかもわかります。myFunc は 2 つの引数を持ちますが、最初の引数の f のみで String の型が明示されていて、2 つ目の引数の i には型注釈がありません。コンパイラがこのメソッドの呼び出しから推論を行うように残してあります。同じように、メソッドの戻り値の型も return true から推論されて Bool になります。

4.3.1 メソッドのオーバーライド (override)

フィールドのオーバーライドは、クラスの階層構造を作る助けになります。オーバーライドはさまざまなデザインパターンで活用されますが、ここでは基本的な機能のみを説明します。クラスでオーバーライドを使うためには、親クラス (2.3.2) を持つ必要があります。次の例を見てみましょう。

```
1  class Base {
2    public function new() { }
3    public function myMethod() {
4      return "Base";
5    }
6  }
7
8  class Child extends Base {
9    public override function myMethod() {
10     return "Child";
11   }
12 }
13
14 class Main {
15   static public function main() {
16     var child:Base = new Child();
17     trace(child.myMethod()); // Child
18   }
19 }
```

ここで重要なのは以下の要素です。

- Base クラスは myMethod メソッドとコンストラクタを持つ。
- Child は Base を継承しており、override を宣言した myMethod を持つ。

・Main クラスは main メソッドで Child をインスタンス化して、Base 型を明示した child 変数に代入して、その myMethod() を呼び出している。

child 変数の Base 型を明示することで、コンパイル時には Base 型であっても、実行時には Child 型の myMethod メソッドが実行されるという重要なことを強調しました。これはフィールドのアクセスが実行時に動的に解決されるからです。

Child クラスでは super.methodName() を呼び出して、オーバーライドされたメソッドにアクセスすることができます。

```
1  class Base {
2    public function new() { }
3    public function myMethod() {
4      return "Base";
5    }
6  }
7
8  class Child extends Base {
9    public override function myMethod() {
10     return "Child";
11   }
12
13   public function callHome() {
14     return super.myMethod();
15   }
16
17 }
18
19 class Main {
20   static public function main() {
21     var child = new Child();
22     trace(child.callHome()); // Base
23   }
24 }
```

new コンストラクタ内での super() の使用については、継承 (節 2.3.2) の節で説明してあります。

4.3.2 変性とアクセス修飾子の影響

オーバーライドは変性 (3.4) のルールと深い関わりがあります。というのは、引数の型が反変性 (より一般的な型) を許容し、戻り値の型は共変性 (より具体的な型) を許容するからです。

```
1  class Base {
2    public function new() { }
3  }
4
5  class Child extends Base {
6    private function method(obj:Child):Child {
7      return obj;
8    }
9  }
10
11 class ChildChild extends Child {
12   public override function method(obj:Base):ChildChild {
```

```
13      return null;
14    }
15  }
16
17  class Main {
18    static public function main() { }
19  }
```

直観的には、この挙動は引数は関数へ「書き込み」であり戻り値は「読み込み」であるという事実から来ています。

この例から、可視性 (4.4.1) が変更できるということもわかります。オーバーライドされる側のフィールドが private の場合は、public のフィールドでオーバーライドすることができます。ただし、そのほかの場合は、可視性の変更はできません。

inline (4.4.2) の宣言をされたフィールドもオーバーライドできません。これはインライン化がコンパイル時に関数の中身で書き換えを行う一方で、オーバーライドのフィールドは実行時に解決される、という衝突を避けるためです。

4.4 アクセス修飾子

4.4.1 可視性

フィールドの可視性はデフォルトでは private です。つまり、そのクラス自身とその子クラスからしかアクセスできません。public アクセス修飾子を使うことでどこからでもアクセスができるようにフィールドを公開できます。

```
1   class MyClass {
2     static public function available() {
3       unavailable();
4     }
5     static private function unavailable() { }
6   }
7
8   class Main {
9     static public function main() {
10      MyClass.available();
11      // privateのフィールドunavailableにアクセスできません。
12      MyClass.unavailable();
13    }
14  }
```

Main から MyClass の available フィールドへのアクセスは、フィールドが public なので許可されます。しかし unavailable については、MyClass からのアクセスは許可されますが Main からは許可されません。これはフィールドが private だからです (ここでは無くてもいい明示的宣言を行っています)。

この例では static フィールドを使って可視性の実演をしていますが、メンバフィールドでもこのルールは同じです。次の例は継承 (2.3.2) がある場合の可視性について実演しています。

```
1   class Base {
2     public function new() { }
3     private function baseField() { }
4   }
5
6   class Child1 extends Base {
7     private function child1Field() { }
```

```
}

class Child2 extends Base {
  public function child2Field() {
    var child1 = new Child1();
    child1.baseField();
    // privateなフィールドchild1Fieldにアクセスできません
    child1.child1Field();
  }
}

class Main {
  static public function main() { }
}
```

Child2 からの、Child1 という異なる型の child1.baseField() へのアクセスが許可されていることがわかります。これはこのフィールドが共通の親クラスの Base で定義されているからです。反対に child1Field については、Child2 からはアクセスできません。

可視性の修飾子の省略はデフォルトでは private になることが多いですが、以下の場合は例外的に public になります。

1. クラスが extern として宣言されている。
2. インターフェース (2.3.3) で宣言しているフィールドである。
3. public フィールドをオーバーライド (4.3.1) している。

> トリビア: protected
> Haxe には Java や C++ やその他のオブジェクト指向言語で知られる protected キーワードはありません。しかし、private の挙動がこれらの言語の protected の挙動に当たります。つまり、Haxe にはこれらの言語の private に当たる挙動がありません。

4.4.2 inline(インライン化)

inline キーワードはその関数の式を関数を呼び出した位置に直接挿し込みできるようにします。これは強力な最適化手段ですが、すべての関数にインライン化の挙動を持つ資格があるわけでありません。基本的な使い方は以下の通りです。

```
class Main {
  static inline function mid(s1:Int, s2:Int) {
    return (s1 + s2) / 2;
  }

  static public function main() {
    var a = 1;
    var b = 2;
    var c = mid(a, b);
  }
}
```

JavaScript 出力を見るとインライン化の効果がわかります。

```
(function () { "use strict";
var Main = function() { }
Main.main = function() {
    var a = 1;
    var b = 2;
    var c = (a + b) / 2;
}
Main.main();
})();
```

見てのとおり mid フィールドの関数本体の (s1 + s2) / 2 が、mid(a, b) の位置で s1 を a に s2 を b に置き換えられて出力されています。ターゲットによっては消えない場合もありますが、関数呼び出しが消滅しており、これが大きなパフォーマンスの改善になりえます。

インライン化するべきかの判断は簡単ではありません。書き込み処理の無い短い関数 (= の代入のみといった) は、たいていインライン化すると良いですし、より複雑な関数であってもインライン化する候補となりえます。一方で、インライン化がパフォーマンスに悪影響を与える場合もあります (複雑な式では、コンパイラが一時変数を作るなどのため)。

inline キーワードは、インライン化されることを保証しません。コンパイラはさまざまな理由でインライン化をキャンセルします。例えば、コマンドラインの引数で--no-inline が与えられた場合です。例外としてクラスが extern (6.2) か、フィールドが:extern メタデータ (6.9) を付けられていた場合、インライン化が強制されます。インライン化ができない場合、コンパイラはエラーを出力します。

これはインライン化を使う上で重要なので覚えておきましょう。

```
class Main {
  public static function main () { }

  static function test() {
    if (Math.random() > 0.5) {
      return "ok";
    } else {
      error("random failed");
    }
  }

  @:extern static inline function error(s:String) {
    throw s;
  }
}
```

error の呼び出しがインライン化できれば、制御フローのチェッカーはインライン化された throw (5.22) に満足してプログラムは正しくコンパイルされます。インライン化されなければ、コンパイラは関数呼び出しのみを見て、A return is missing here(ここにリターンが足りません) というエラーを出力します。

4.4.3 dynamic

メソッドは dynamic キーワードをつけることで、束縛のしなおしをできるようにします。

```
class Main {
  static dynamic function test() {
    return "original";
  }

```

```
6     static public function main() {
7       trace(test()); // original
8       test = function() { return "new"; }
9       trace(test()); // new
10    }
11  }
```

最初の test() の呼び出しではもともとの関数を実行して"original" の文字列を返します。つぎの行で、test() に新しい関数が代入されます。これが dynamic が可能にする関数の再束縛です。その結果として、次の test() の呼び出しでは"new" の文字列が返っています。

dynamic フィールドは inline フィールドにできません。その理由は明らかです。インライン化はコンパイル時に行われますが、dynamic な関数は実行時に解決されます。

4.4.4 override

override アクセス修飾子は親クラス (2.3.2) ですでに存在するフィールドを定義するときに必要です。これはクラスの継承関係が大きい場合でも、書き手がオーバーライドに気づくようにするためです。同様に、実際には何もオーバーライドしないフィールドに override しようとするとエラーになります (例えば、スペルミスの場合)。

オーバーライドの効果については、メソッドのオーバーライド (override) (節 4.3.1) で詳しく説明しています。この修飾子はメソッド (4.3) フィールドのみに使用可能です。

章 5

式

Haxe の式はプログラムが何をするのかを決定します。ほとんどの式はメソッド (4.3) に書かれ、そのメソッドが何をすべきかをその式の合わせによって表現します。この章では、さまざまな種類の式を説明していきます。

ここでいくつかの定義を示しておきます。

定義: 名前
名前は次のいずれかに紐づきます。

- 型
- ローカル変数
- ローカル関数
- フィールド

定義: 識別子
Haxe の識別子はアンダースコア (_)、ドル ($)、小文字 (a-z)、大文字 (A-Z) のいずれかから始まり、任意の_、A-Z、a-z、0-9 のつなぎ合わせが続きます。

さらに使用する状況によって以下の制限が加わります。これらは型付けの時にチェックされます。

- 型の名前は大文字 (A-Z) か、アンダースコア (_) で始まる。
- 名前 (5) では、先頭にドル記号は使えません。(ドル記号はほとんどの場合、マクロの実体化 (9.3) に使われます)

5.1 ブロック

Haxe のブロックは中かっこ ({) から始まり、中かっこ閉 (}) で終わります。ブロックはいくつかの式をふくみ、各式はセミコロン (;) で終わります。一般の構文としては以下のとおりです。

```
1 {
2     式1;
3     式2;
4     ...
```

```
    式N;
}
```

ブロック式の値とその型はブロック式がふくむ最後の式の値と型と同じになります。

ブロック内では、var 式 (5.10) を使ったローカル変数の定義と function 式 (5.11) を使ったローカル関数の定義が可能です。これらのローカル変数とローカル関数はそのブロックとさらに入れ子のブロックの中では使用することができますが、ブロックの外では利用できません。また、定義よりも後でしか使えません。次の例では var を使っていますが、同じルールが function の場合でも使用されます。

```
{
    a; // エラー。aはまだ宣言されていない。
    var a = 1; // aを宣言。
    a; // 問題ない。aは宣言されている。
    {
        a; // 問題ない。子のブロックでもaは使用できる。
    }
    // 問題ない。子のブロックの後でもaは使用できる。
    a;
}
a; // エラー。ブロック外ではaは使用できない。
```

実行時には、ブロックは上から下へと評価されていきます。フロー制御 (例えば、例外 (5.18) や return 式 (5.19) など) によって、すべての式が評価される前に中断されることもあります。

5.2 定数値

Haxe の構文では以下の定数値をサポートしています。

Int: 0、1、97121、-12、0xFF0000 といった、整数 (2.1.1)

Float: 0.0、1.、.3、-93.2 といった浮動小数点数 (2.1.1)

String: ""、"foo"、'-'、'bar' といった文字列 (10.1)

true,false: 真偽値 (2.1.4)

null: null 値

また内部の構文木では、識別子 (5) は定数値としてあつかわれます。これはマクロ (9) を使っているときに関係してくる話題です。

5.3 2 項演算子

5.4 単項演算子

5.5 配列の宣言

配列はカンマ (,) で区切った値を、大かっこ ([]) で囲んで初期化します。空の [] は空の配列を表し、[1, 2, 3] は 1、2、3 の 3 つの要素を持つ配列を表します。

配列の初期化をサポートしていないプラットフォームでは、生成されるコードはあまり簡潔ではないかもしれません。以下のような意味合いのコードに見えるでしょう。

```
var a = new Array();
a.push(1);
a.push(2);
a.push(3);
```

つまり、関数をインライン化 (4.4.2) するかを決める場合には、この構文で見えているよりも多くのコードがインライン化されることがあることを考慮すべきです。

より高度な初期化方法については、配列内包表記 (節 6.6) で説明します。

5.6 オブジェクトの宣言

オブジェクトの宣言は中かっこ ({) で始まり、**キー: 値**のペアがカンマ (,) で区切られながら続いて、中かっこ閉 (}) で終わります。

```
{
    key1:value1,
    key2:value2,
    ...
    keyN:valueN
}
```

さらに詳しいオブジェクトの宣言については匿名構造体 (2.5) の節で書かれています。

5.7 フィールドへのアクセス

フィールドへのアクセスはドット (.) の後にフィールドの名前を続けることで表現します。

```
object.fieldName
```

この構文は `pack.Type` の形でパッケージ内の型にアクセスするのにも使われます。

型付け器はアクセスされたフィールドが本当に存在するかをチェックして、フィールドの種類に依存した変更を適用します。もしフィールドへのアクセスが複数の意味にとれる場合は、解決順序 (3.7.3) の理解が役に立つでしょう。

5.8 配列アクセス

配列アクセスは大かっこ ([) で始まり、インデックスを表す式が続き、大かっこ閉 (]) で閉じます。

```
expr[indexExpr]
```

`expr` と `indexExpr` について任意の式が許可されていますが、型付けの段階では以下の特定の組み合わせのみが許可されます。

- `expr` は `Array` か `Dynamic` であり、`indexExpr` が `Int` である。
- `expr` は抽象型 (2.8) であり、マッチする配列アクセス (2.8.3) が定義されている。

5.9 関数呼び出し

関数呼び出しは任意の式を対象として、小かっこ (() を続け、引数の式のリストを (,) で区切って並べて、小かっこ閉 ()) で閉じることで行います。

```
subject(); // call with no arguments
subject(e1); // call with one argument
subject(e1, e2); // call with two arguments
// call with multiple arguments
subject(e1, ..., eN);
```

5.10 var(変数宣言)

var キーワードはカンマ (,) で区切って、複数の変数を宣言することができます。すべての変数は正当な識別子 (5) を持ち、オプションとして = を続けて値の代入を行うこともできます。また変数に明示的な型注釈をあたえることもできます。

```
var a; // ローカル変数aを宣言。
var b:Int; // Int型のローカル変数bを宣言。
// cを宣言し、値を1で初期化。
var c = 1;
// 変数dと変数eを宣言。eを2で初期化。
var d,e = 2;
```

ローカル変数のスコープについての挙動はブロック (節 5.1) に書かれています。

5.11 ローカル関数

Haxe はファーストクラス関数をサポートしており、式の中でローカル関数を宣言することができます。この構文はクラスフィールドメソッド (4.3) にならいます。

```
class Main {
  static public function main() {
    var value = 1;
    function myLocalFunction(i) {
      return value + i;
    }
    trace(myLocalFunction(2)); // 3
  }
}
```

myLocalFunction を、main クラスフィールドのブロック式 (5.1) の中で宣言しました。このローカル関数は 1 つの引数 i を取り、それをスコープの外の value に足しています。

スコープについては変数の場合 (5.10) と同じで、多くの面で名前を持つローカル関数はローカル変数に対する匿名関数の代入と同じです。

```
var myLocalFunction = function(a) { }
```

しかしながら、関数の場所による型パラメータに関する違いがあります。これは定義時に何にも代入されていない「左辺値」の関数と、それ以外の「右辺値」の関数についての違いで、以下の通りです。

- 左辺値の関数は名前が必要で、型パラメータ (3.2) を持ちます。
- 右辺値の関数については名前はあってもなくてもかまいませんが、型パラメータを使うことができません。

5.12 new(インスタンス化)

new キーワードはクラス (2.3) と抽象型 (2.8) のインスタンス化を行います。new の後にはインスタンス化される型のパス (3.7) が続きます。場合によっては、<> で囲んでカンマ, で区切った、型パラメータ (3.2) の記述がされます。その後に、小かっこ ((), カンマ (,) 区切りのコンストラクタの引数が続き、小かっこ閉 ()) で閉じます。

```
1 class Main<T> {
2   static public function main() {
3     new Main<Int>(12, "foo");
4   }
5
6   function new(t:T, s:String) { }
7 }
```

main メソッドの中では型パラメータ Int の明示付き、引数が 12 と"foo" で、Main クラス自身のインスタンス化を行っています。私たちが知っているように、この構文は関数呼び出し (5.9) とよく似ており、「コンストラクタ呼び出し」と呼ぶことが多いです。

5.13 for ループ

Haxe は C 言語で知られる伝統的な for ループはサポートしていません。for キーワードの後には小かっこ ((), 変数の識別子、in キーワード、くり返しの処理を行うコレクションの任意の式が続き、小かっこ閉 ()) で閉じられて、最後にくり返しの本体の任意の式で終わります。

```
1 for (v in e1) e2;
```

型付け器は e1 の型がくり返し可能であるかを確認します。くり返し可能というのは iterator メソッドが Iterator<T> を返すか、Iterator<T> 自身である場合です。

変数 v はループ本体の e2 の中で使用可能で、コレクション e1 の個々の要素の値が保持されます。

Haxe にはある範囲のくり返しを表す特殊な範囲演算子があります。これは min...max といった 2 つの Int をとり、min(自身をふくむ) から max の一つ前までをくり返す IntIterator を返す 2 項演算子です。max が min より小さくしないように気をつけてください。

```
1 for (i in 0...10) trace(i); // 0 to 9
```

for 式の型は常に Void です。つまり、値は持たず、右辺の式としては使えません。

ループは break (5.20) と、continue (5.21) の式を使って、フロー制御が行えます。

5.14 while ループ

通常の while ループは while キーワードから始まり、小かっこ ((), 条件式が続き、小かっこ閉 ()) を閉じて、ループ本体の式で終わります。

```
1 while(condition) expression;
```

条件式は Bool 型でなくてはいけません。

各くり返しで条件式は評価されます。false と評価された場合ループは終了します。そうでない場合、ループ本体の式が評価されます。

```
1 class Main {
2   static public function main() {
3     var f = 0.0;
```

```
    while (f < 0.5) {
      trace(f);
      f = Math.random();
    }
  }
}
```

この種類の while ループはループ本体が一度も評価されないことがあります。条件式が最初から false だった場合です。この点が do-while ループ (5.15) との違いです。

5.15 do-while ループ

do-while ループは do キーワードから始まり、次にループ本体の式が来ます。その後に while、小かっこ (()、条件式、小かっこ閉 ()) となります。

```
do expression while(condition);
```

条件式は Bool 型でなくてはいけません。

この構文を見てわかるとおり、while (5.14) ループの場合とは違ってループ本体の式は少なくとも一度は評価をされます。

5.16 if

条件分岐式は if キーワードから始まり、小かっこ (()) で囲んだ条件式、条件が真だった場合に評価される式となります。

```
if (condition) expression;
```

条件式は Bool 型でなくてはいけません。

オプションとして、else キーワードを続けて、その後に、元の条件が偽だった場合に実行される式を記述することができます。

```
if (condition) expression1 else expression2;
```

expression2 は以下のように、また別の if 式を持つかもしれません。

```
if (condition1) expression1
else if(condition2) expression2
else expression3
```

if 式に値が要求される場合 (たとえば、var x = if(condition) expression1 else expression2 というふうに)、型付け器は expression1 と expression2 の型を単一化 (3.5) します。else 式がなかった場合、型は Void であると推論されます。

5.17 switch

基本的なスイッチ式は switch キーワードと、その分岐対象の式から始まり、中かっこ ({}) にはさまれてケース式が並びます。各ケース式は case キーワードからのパターン式か、default キーワードで始まります。どちらの場合も、コロンが続き、オプショナルなケース本体の式が来ます。

```
switch subject {
    case pattern1: case-body-expression-1;
    case pattern2: case-body-expression-2;
    default: default-expression;
}
```

ケース本体の式に、「フォールスルー」は起きません。このため、Haxe では break (5.20) キーワードは使用しません。

スイッチ式は値としてあつかうことができます。その場合、すべてのケース本体の式の型は単一化 (3.5) できなくてはいけません。

パターン式についてはパターンマッチング (節 6.4) で詳しく説明されています。

5.18 try/catch

Haxe では try/catch 構文を使うことで値を捕捉することができます。

```
try try-expr
catch(varName1:Type1) catch-expr-1
catch(varName2:Type2) catch-expr-2
```

実行時に、try-expression の評価が、throw (5.22) を引き起こすと、後に続く catch ブロックのいずれかに捕捉されます。これらのブロックは以下から構成されます

- throw された値を割り当てる変数の名前。
- 捕捉する値の型を決める、明示的な型注釈
- 捕捉したときに実行される式

Haxe では、あらゆる種類の値を throw して、catch することができます。その型は特定の例外やエラークラスに限定されません。catch ブロックは上から下へとチェックされていき、投げられた値と型が適合する最初のブロックが実行されます。

この過程はコンパイル時の単一化 (3.5) に似ています。しかし、この判定は実行時に行われるものでいくつかの制限があります。

- 型は実行時に存在するものでなければならない。クラスインスタンス (2.3)、列挙型インスタンス (2.4)、コアタイプ抽象型 (2.8.7)、Dynamic (2.7).
- 型パラメータは Dynamic (2.7) でなければならない。

5.19 return

return 式は値を取るものと、取らないものの両方があります。

```
return;
return expression;
```

return 式は最も内側に定義されている関数のフロー制御からぬけ出します。最も内側というのはローカル関数 (5.11) の場合での特徴です。

```
function f1() {
    function f2() {
        return;
    }
    f2();
    expression;
}
```

return により、ローカル関数 f2 からはぬけ出しますが、f1 からはぬけ出しません。つまり、expression は評価されます。

return が、値の式なしで使用された場合、型付け器はその関数の戻り値が Void 型であることを確認します。return が値の式を持つ場合、型付け器はその関数の戻り値の型と return している値の型を単一化 (3.5) します (明示的に与えられているか、前の return によって推論されている場合)。

5.20 break

break キーワードはそのキーワードをふくむ最も内側にあるループ (for でも、while でも) の制御フローからぬけ出して、くり返し処理を終了させます。

```
1 while(true) {
2     expression1;
3     if (condition) break;
4     expression2;
5 }
```

expression1 はすべてのくり返しで評価されますが、condition が偽になると expression2 は、実行されません。

型付け器は break がループの内部のみで使用されていることを確認します。switch のケース (5.17) に対する break は Haxe ではサポートしていません。

5.21 continue

continue キーワードはそのキーワードをふくむ最も内側にあるループ (for でも、while でも) の現在のくり返しを終了します。そして、次のくり返しのためのループ条件チェックが行われます。

```
1 while(true) {
2     expression1;
3     if(condition) continue;
4     expression2;
5 }
```

expression1 は各くり返しすべてで評価されますが、condition が偽の時はその回のくり返しについては評価がされません。break は異なりループ処理自体は続きます。

型付け器は continue がループの内部のみで使用されていることを確認します。

5.22 throw

Haxe では、以下の構文で値の throw をすることができます。

```
1 throw expr
```

throw された値は catch ブロック (5.18) で捕捉できます。捕捉されなかった場合の挙動はターゲット依存です。

5.23 cast

Haxe には以下の 2 種類のキャストがあります。

```
1 cast expr; // 非セーフキャスト
2 cast (expr, Type); // セーフキャスト
```

5.23.1　非セーフキャスト

非セーフキャストは型システムを無力化するのに役立ちます。コンパイラは expr を通常通りに型付けし、それを単相 (2.9) でつつみ込みます。これにより、その式をあらゆるものに割り当てすることが可能になります。

非セーフキャストは以下の例が示すように、Dynamic (2.7) への型変更ではありません。

```
1  class Main {
2    public static function main() {
3      var i = 1;
4      $type(i); // Int
5      var s = cast i;
6      $type(s); // Unknown<0>
7      Std.parseInt(s);
8      $type(s); // String
9    }
10 }
```

変数 i は Int と型付けされて、非セーフキャスト cast i を使って変数 s に代入しました。s は Unknown 型、つまり単相となりました。その後は、通常の単一化 (3.5) のルールに従って、あらゆる型へと結びつけることが可能です。例では、String 型となりました。

これらのキャストは「非セーフ」と呼ばれます。これは実行時の不正なキャストが定義されてないためです。ほとんどの動的ターゲット (2.2) では動作する可能性が高いですが、静的ターゲット (2.2) では未知のエラーの原因になりえます。

非セーフキャストは実行時のオーバーヘッドはほぼ、または全くありません。

5.23.2　セーフキャスト

非セーフキャスト (5.23.1) とは異なり、実行時のキャスト失敗の挙動を持つのがセーフキャストです。

```
1  class Base {
2    public function new() { }
3  }
4
5  class Child1 extends Base {}
6
7  class Child2 extends Base {}
8
9  class Main {
10   public static function main() {
11     var child1:Base = new Child1();
12     var child2:Base = new Child2();
13     cast(child1, Base);
14     // 例外: Class cast error
15     cast(child1, Child2);
16   }
17 }
```

この例では、最初に Child1 から Base へとキャストしています。これは Child1 が Base 型の子クラス (2.3.2) なので、成功しています。次に Child2 へキャストしていますが、Child1 のインスタンスは Child2 ではないので失敗しています。

Haxe コンパイラはこの場合 String 型の例外を投げます (5.22)。この例外は try/catch ブロック (5.18) を使って捕捉できます。

セーフキャストは実行時のオーバーヘッドがあります。重要なのはコンパイラがすでにチェックを行っているので、Std.is のようなチェックを自分で入れるのは、余分だということです。String 型の例外を捕捉する、try-catch を行うのがセーフキャストで意図された用途です。

5.24　型チェック

【Haxe 3.1.0 から】

以下の構文でコンパイルタイムの型チェックをつけることが可能です。

```
1 (expr : type)
```

小かっこは必須です。セーフキャスト (5.23.2) とは異なり、実行時に影響はありません。これはコンパイル時の以下の 2 つの挙動を持ちます。

1. トップダウンの型推論 (3.6.1) が expr に対して type の型で適用されます。
2. その結果、type の型との単一化 (3.5) がされます。

この 2 つの操作には、解決順序 (3.7.3) が発生している場合や、抽象型キャスト (2.8.1) で、期待する型へと変化させる、便利な効果があります。

章 6

言語機能

抽象型 (2.8): 抽象型は実行時には別の形として提供されるコンパイル時の構成要素です。これにより、すでに存在する型に別の意味をあたえることができます。

extern クラス (6.2): extern を使うことで、型安全のルールにしたがってターゲット固有の連携を記述することができます。

匿名構造体 (2.5): 匿名構造体を使うことでデータを簡単にまとめて、小さなデータクラスの必要性を減らすことができます。

```
1 var point = { x: 0, y: 10 };
2 point.x += 10;
```

配列内包表記 (6.6): ループと条件分岐を使って、素早く配列を生成して受け渡すことができます。

```
1 var evenNumbers = [ for (i in 0...100) if (i¥%2==0) i ];
```

クラス、インターフェース、継承 (2.3): Haxe はクラスを使ったコードの構造化ができる、オブジェクト指向言語です。継承やインターフェースといった Java でサポートされるようなオブジェクト指向言語の標準的な機能を備えています。

条件付きコンパイル (6.1): 条件付きコンパイルを使うことで、コンパイルのパラメータごとに固有のコードをコンパイルすることができます。これはターゲットごとの違いを抽象化する手助けになるだけでなく、詳細のデバッグ機能を提供するなどその他の用途にも使用できます。

```
1 #if js
2     js.Lib.alert("Hello");
3 #elseif sys
4     Sys.println("Hello");
5 #end
```

(一般化) 代数的データ型 (2.4): Haxe では enum として知られる、代数的データ型 (ADT) を使ってデータ構造を表現することができます。さらに、Haxe は一般化されたヴァリアント (GADT) もサポートしています。

```
enum Result {
    Success(data:Array<Int>);
    UserError(msg:String);
    SystemError(msg:String, position:PosInfos);
}
```

インライン呼び出し (4.4.2): 関数をインラインとして指定して、呼び出し場所にその関数のコードを挿入させることができます。これにより、手作業でのインライン化のようなコードの重複を発生させること無く、価値あるパフォーマンスの改善を得ることできます。

イテレータ (反復子) (6.7): Haxe はイテレータを適切にあつかっているので、値のセット (例えば、配列) の反復処理がとても簡単です。自前のクラスであっても、イテレータ機能の実装をすることで素早く反復可能にすることができます。

```
for (i in [1, 2, 3]) {
    trace(i);
}
```

ローカル関数とクロージャ (5.11): Haxe では関数はクラスフィールドに限定されず、式の中で定義することができます。その場合、強力なクロージャも使用可能です。

```
var buffer = "";
function append(s:String) {
    buffer += s;
}
append("foo");
append("bar");
trace(buffer); // foobar
```

メタデータ (6.9): フィールド、クラス、式に対してメタデータを追加できます。これにより、コンパイラ、マクロ、実行時のクラスに情報の受け渡しができます。

```
class MyClass {
    @range(1, 8) var value:Int;
}
trace(haxe.rtti.Meta.getFields(MyClass).value.range); // [1,8]
```

静的拡張 (6.3): 既に存在するクラスやその他の型に対して、静的拡張を使うことで追加の機能を足すことができます。

```
using StringTools;
"  Me & You    ".trim().htmlEscape();
```

文字列中の変数展開 (6.5): シングルクオテーションを使って宣言した文字列では現在の文脈中の変数へのアクセスができます。

```
trace('My name is $name and I work in ${job.industry}');
```

関数の部分適用 (6.8): すべての関数は部分適用を使って、いくつかの引数だけに値を適用して残りの引数を後で指定できるように残すことができます。

```
var map = new haxe.ds.IntMap();
var setToTwelve = map.set.bind(_, 12);
setToTwelve(1);
setToTwelve(2);
```

パターンマッチング (6.4): 複雑な構造体は enum や構造体から情報を抽出したり、特定の演算子で値の組み合わせを指定したりしながら、パターンを当てはめてマッチングすることができます。

```
var a = { foo: 12 };
switch (a) {
    case { foo: i }: trace(i);
    default:
}
```

プロパティ (4.2): 変数のクラスフィールドにはカスタムの読み込み書き込みアクセスを指定するプロパティが使えます。これにより、より良いアクセス制御が実現できます。

```
public var color(get,set);
function get_color() {
    return element.style.backgroundColor;
}
function set_color(c:String) {
    trace('Setting background of element to $c');
    return element.style.backgroundColor = c;
}
```

アクセス制御 (6.10): Haxe では、メタデータの構文を使って、クラスやフィールドに対してアクセスを許可したり強制したりといったアクセス制御をを行うことできます。

型パラメータ、共変性、反変性 (3.2): 型には型パラメータをつけて、型付きのコンテナなど複雑なデータ構造を表現できます。型パラメータは特定の型への制限が可能で、また、変性のルールに従います。

```
class Main<A> {
    static function main() {
        new Main<String>("foo");
        new Main(12); // 型推論を使う。
    }

    function new(a:A) { }
}
```

6.1 条件付きコンパイル

Haxe では、#if、#elseif、#else を使ってコンパイラフラグを確認することで条件付きコンパイルが可能です。

定義: コンパイラフラグ
コンパイラフラグはコンパイルの過程に影響をあたえる、設定可能な値です。このフラグは-D key=value あるいは単に-D key(この場合デフォルト値の"1" になる)の形式でコマンドラインから指定できます。そのほかにも、コンパイラはコンパイルの過程で別のステップへ情報伝達するために、内部的にいくつかのフラグを設定します。

以下は条件付きコンパイルの利用例のデモです。

```
1  class Main {
2    public static function main(){
3      #if !debug
4        trace("ok");
5      #elseif (debug_level > 3)
6        trace(3);
7      #else
8        trace("デバッグレベルが低すぎます。");
9      #end
10   }
11 }
```

これをフラグ無しでコンパイルした場合、main メソッドの trace("ok"); が実行されて終了します。他の分岐はファイルを構文解析する際に切り捨てられます。他の分岐についても、正しい Haxe の構文である必要がありますが、型チェックはされません。

#if と #elseif の直後の条件には以下の式が使えます。

- すべての識別子は同名のコンパイラフラグの値で置きかえられます。コマンドラインから-D some-flag を指定すると some-flag と some_flag のフラグが定義されることに気を付けてください。
- String、Int、Float の定数値は直接使用されます。
- Bool の演算 && (and)、|| (or)、! (not) は期待どおりに動作しますが、式全体を小かっこでかこむ必要があります。
- ==、!=、>、>=、<、<= の演算子が値の比較に使えます。
- 小かっこ () は通常通り、式をグループ化するのに使えます。

Haxe の構文解析器は some-flag を一つの句として認識しません、some - flag の 2 項演算として読み取ります。このような場合はアンダースコアを使う some_flag の版を使用する必要があります。

ビルトインのコンパイラフラグ　ビルトインのコンパイラフラグの完全なリストは Haxe コンパイラを--help-defines の引数をつけて呼び出すことで手に入れることができます。Haxe のコンパイラはコンパイルごとに複数の-D フラグを指定できます。

コンパイラフラグ一覧 (6.1.1) も確認してみてください。

6.1.1　グローバルコンパイラフラグ

Haxe 3.0 以降では haxe --help-defines を実行することで、サポートしているコンパイラフラグ (6.1) の一覧を取得することができます。

グローバ	
フラグ	説明
absolute-path	trace の出力を絶対パスで行います。
advanced-telemetry	SWF を Monocle のツールで測定できるようにします。
analyzer	静的解析器を使った最適化を行います (実験的)。
as3	flash9 の as3 のソースコードを出力する場合に定義されます。
check-xml-proxy	xml プロキシの使用済みフィールドを確認します。
core-api	core API の文脈で定義されています。
core-api-serialize	C# で、いくつかの core API クラスを Serializable 属性でマークします
cppia	実験的に C++ のインストラクションアセンブリを出力します
dce=<mode:std|full|no>	デッドコード削除 (8.2) のモードを設定します (デフォルトでは std)。
dce-debug	Show dead code elimination (8.2) log
debug	-debug をつけてコンパイルした場合に有効化されます。
display	補完中に有効化されます。
dll-export	実験的なリンクをつけて C++ 生成します。
dll-import	実験的なリンクをつけて C++ 生成します。
doc-gen	正しくドキュメントを生成するため、削除と変更をしないように振る舞いま
dump	dump サブディレクトリに、完全な型付け済みの抽象構文木を出力します
dump-dependencies	dump サブディレクトリに、クラスの依存関係を出力をします。
dump-ignore-var-ids	pretty ではない dump から、変数 ID を削除します。(diff を取るのに役
erase-generics	C# でジェネリッククラスを取り消します。
fdb	FDB の対話的なデバッグのために、flash のデバッグ情報をすべて有効
file-extension	C++ ソースコードで拡張子を出力します。
flash-strict	Flash 出力でより厳密な型付けを行います。
flash-use-stage	SWF ライブラリを初期の stage に保ちます。
force-lib-check	コンパイラが-net-lib と-java-lib の追加クラスを確認するように強制しま
force-native-property	3.1 の互換性のために、すべてプロパティに:nativeProperty のタグ付
format-warning	2.x の互換性のために、フォーマットされた文字列に対して警告を出しま
gencommon-debug	GenCommon の内部用
haxe-boot	flash の boot クラスに生成された名前の代わりに'haxe' という名前を
haxe-ver	現在の Haxe のバージョンの値です。
hxcpp-api-level	hxcpp のバージョン間の互換性を保ちます。
include-prefix	含有している出力ファイルにパスを付加します。
interp	--interp でコンパイルされて実行される場合に定義されます。
java-ver=[version:5-7]	ターゲットとする Java のバージョンを設定します。
js-classic	JavaScript 出力に function ラッパーと、strict モードを使いません。
js-es5	ES5 に準拠した実行環境のための JavaScript を出力します。
js-unflatten	package や型でネストしたオブジェクトを出力します。
keep-old-output	出力ディレクトリの古いコードのファイルを残します (C#/Java)。
loop-unroll-max-cost	ループ展開がキャンセルされる最大コスト (expressions * iterations、デ
macro	マクロの文脈 (9) でコンパイルされた場合に定義されます。
macro-times	--times と一緒に使用された場合にマクロごとの時間を表示します。
net-ver=<version:20-45>	ターゲットとする.NET のバージョンを設定します。
net-target=<name>	.NET のターゲット名を設定します。xbox、micro _(Micro Framework
neko-source	Neko のバイトコードではなくソースコードを出力します。
neko-v1	Neko の 1.x との互換性を保ちます。
network-sandbox	ローカルファイルアクセスの代わりに、ローカルネットワークサンドボックス
no-compilation	C++ の最終コンパイルを無効化します。
no-copt	コンパイル時の最適化を無効化します _(デバッグ用途)_
no-debug	C++ 出力からすべてのデバッグマクロを取り除きます。
no-deprecation-warnings	@:deprecated のフィールドが使われたことによる警告を無効化します。
no-flash-override	flash のみで、いくつかの基本クラスでの override を HX サフィックスの
no-opt	最適化を無効化します。
no-pattern-matching	パターンマッチングを無効化します。
no-inline	インライン化 (4.4.2) を無効化します。
no-root	GenCS の内部用
no-macro-cache	マクロの文脈でのキャッシュを無効化します。
no-simplify	簡易化のフィルタを無効化します。
no-swf-compress	SWF 出力の圧縮を無効化します。
no-traces	すべての trace 呼び出しを無効化します。

6.2 extern

extern はターゲット固有の連携を型安全のルールに従って記述するために使います。宣言は普通のクラスに似た形で、以下の要素が必要です。

- class キーワードの前に extern キーワードを置きます。
- メソッド (4.3) は式を持ちません。
- すべての引数と戻り値の型を明示します。

Haxe 標準ライブラリ (10) の Math クラスがちょうどいい例です。その一部を抜粋します。

```
1 extern class Math
2 {
3     static var PI(default,null) : Float;
4     static function floor(v:Float):Int;
5 }
```

extern が、メソッドと変数の両方を定義できることがわかります (実際のところ、PI は読み込み専用のプロパティ (4.2)) を定義しています。一度この情報がコンパイラで使用可能になると、型がわかり、フィールドへのアクセスが出来るようになります。

```
1 class Main {
2   static public function main() {
3     var pi = Math.floor(Math.PI);
4     $type(pi); // Int
5   }
6 }
```

floor メソッドの戻り値が Int して定義されているため、このように動作します。

Haxe 標準ライブラリは多くの extern を Flash、JavaScript ターゲット用にもっています。これにより、ネイティブの API に型安全のルールに従ってアクセス可能にし、より高いレベルの API 設計の助けになります。haxelib (11) でも、多くのネイティブのライブラリの extern を入手できます。

Flash、Java、C# ターゲットでは、コマンドライン (7) から直接ネイティブライブラリの取り込みを行うことができます。ターゲットごとの詳細はTarget Details (章 12) のそれぞれの節で説明されています。

Python や、JavaScript といったターゲットでは、extern クラスをネイティブのモジュールから読み込むために追加の「インポート」が必要になる場合があります。Haxe はそのような依存関係を宣言する仕組みを提供しているので、それらをTarget Details (章 12) のそれぞれの節で説明します。

可変長引数と、型の選択肢 【Haxe 3.2.0 から】

haxe.extern パッケージはネイティブの概念を Haxe に対応させるため、2 つの型を提供しています。

Rest<T>: この型は関数の最後の引数として使って、可変長の引数を追加で渡すことを可能にします。型パラメータは引数を特定の型に制限するのに使います。

EitherType<T1,T2>: この型はパラメータのどちらかの型を使うことができるようにする。つまり、型の選択肢を表現できます。3 つ以上の型を選ばせたい場合はネストさせて使います。

以下にデモを用意しました。

```haxe
import haxe.extern.Rest;
import haxe.extern.EitherType;

extern class MyExtern {
  static function f1(s:String, r:Rest<Int>):Void;
  static function f2(e:EitherType<Int, String>):Void;
}

class Main {
  static function main() {
    MyExtern.f1("foo", 1, 2, 3); // 1, 2, 3を可変長引数として渡す
    MyExtern.f1("foo"); // 可変長引数なし
    //MyExtern.f1("foo", "bar"); // Stringではなく Intであるべき

    MyExtern.f2("foo");
    MyExtern.f2(12);
    //MyExtern.f2(true); // Boolではなく EitherType<Int, String>であるべき
  }
}
```

6.3 静的拡張

定義: 静的拡張
静的拡張はすでに存在している型に対して、元のソースコードを変更することなく見せかけの拡張を行います。Haxe の静的拡張は最初の引数が拡張する対象の型である静的メソッドを宣言して、それ using を使って記述しているクラス内に持ちこむことで使用できます。

静的拡張は実際に型の変更を行うことなく型を強化する強力なツールです。以下の例で、その使い方を実演します。

```haxe
using Main.IntExtender;

class IntExtender {
  static public function triple(i:Int) {
    return i * 3;
  }
}

class Main {
  static public function main() {
    trace(12.triple());
  }
}
```

Int は元々 triple メソッドを持っていませんが、このプログラムは期待通り 36 を出力します。12.triple() の呼び出しが IntExtender.triple(12) に変形されるためです。これには必要な条件が 3 つあります。

1. 定数値の 12 と triple の最初の引数の型が、共に Int である

2. IntExtender クラスが using Main.IntExtender を使って現在の文脈に読み込まれている。

3. Int 自身は triple フィールドを持っていない (持っていた場合、静的拡張よりも高い優先度になる)。

静的拡張はシンタックスシュガーですが、コードの可読性に大きな影響を与えることには注目する価値があります。f1(f2(f3(f4(x)))) の形のネストされた呼び出しの代わりに、x.f4().f3().f2().f1() のチェーンの形での呼び出しが可能になります。

優先順位のルールは解決順序 (節 3.7.3) ですでに説明されているとおり、using 式が複数ある場合は下から上へと確認がされ、各モジュールでは各型のフィールドが上から下へと確認がされます。モジュールを静的拡張として using すると、そのすべての型が現在の文脈にインポートされます (モジュール内の特定の型の場合とは対照的です。詳しくはモジュールとパス (節 3.7) を見てください)。

6.3.1 Haxe 標準ライブラリについて

Haxe の標準ライブラリのいくつかのクラスは静的拡張の用途に合うように設計されています。次の例からは StringTools の使い方がわかります。

```
1 using StringTools;
2
3 class Main {
4   static public function main() {
5     "adc".replace("d", "b");
6   }
7 }
```

String 自身は replace を持っていませんが、using StringTools の静的拡張によって提供されます。いつものように、JavaScript への変換を見るとよくわかります。

```
1 Main.main = function() {
2     StringTools.replace("adc","d","b");
3 }
```

Haxe 標準ライブラリでは以下のクラスが静的拡張として使うように設計されています。

StringTools: 置換やトリミングといった、文字列に対する拡張を提供します。

Lambda: Iterable に対する関数型のメソッドを提供します。

haxe.EnumTools: 列挙型とそのインスタンスについての情報を得る機能を提供します。

haxe.macro.Tools: マクロをあつかう際のさまざまな拡張を提供します (詳しくはTools (節 9.4))。

トリビア: “using” using
using キーワードが追加されて以降、using を使う (using using) ときの問題や、その影響についての会話がよくされるようになりました。”using using” のせいでさまざまな場面でわかりにくい英語が生まれたため、このマニュアルの著者はこの機能をその実際の性質から静的拡張と呼ぶことに決めました。

6.4 パターンマッチング

6.4.1 導入

パターンマッチングは、与えられたパターンと値がマッチするかで分岐をする処理のことです。Haxe では、すべてのパターンマッチングは switch 式 (5.17) の個々の case 式が表すパターンに従って行われます。それでは以下のデータ構造を使って、さまざまなパターンの構文を見ていきましょう。

```
enum Tree<T> {
  Leaf(v:T);
  Node(l:Tree<T>, r:Tree<T>);
}
```

以下はパターンマッチングの基本事項です。

- パターンは上から下へとマッチングされます。
- 入力値にマッチする最上位のパターンの持っている式が実行されます。
- _はすべてにマッチします。このため、case _: は default: と同じです。

6.4.2 enum マッチング

enum のコンストラクタは直観的な方法でマッチングできます。

```
    var myTree = Node(Leaf("foo"), Node(Leaf("bar"), Leaf("foobar")));
    var match = switch(myTree) {
      // すべてのLeafにマッチする
      case Leaf(_): "0";
      // r = Leafであるすべての Nodeにマッチする
      case Node(_, Leaf(_)): "1";
      // r = Node で、
      // その中身が l = Leaf("bar")
      // であるすべての Nodeにマッチする
      case Node(_, Node(Leaf("bar"), _)): "2";
      // すべてにマッチする
      case _: "3";
    }
    trace(match); // 2
```

パターンマッチングでは、ケースを上から下へと確認していき、入力値とマッチする最初のものを見つけ出します。以下の各ケースを実行する流れの説明で、その過程を理解してください。

case Leaf(_): myTree は Node なので、マッチングに失敗します。

case Node(_, Leaf(_)): myTree の右側の子要素は Leaf ではなく、Node なので失敗します。

case Node(_, Node(Leaf("bar"), _)): マッチングに成功します。

case _: 前のケースでマッチングが成功したので確認が行われません。

6.4.3　変数の取り出し

パターンの一部のあらゆる値は識別子を使ってマッチングさせて、取り出すことができます。

```
var myTree = Node(Leaf("foo"), Node(Leaf("bar"), Leaf("foobar")));
var name = switch(myTree) {
  case Leaf(s): s;
  case Node(Leaf(s), _): s;
  case _: "none";
}
trace(name); // foo
```

これは以下の流れにしたがって return を行います。

- myTree が Leaf の場合、その名前が返る。
- myTree が Node でその左の子要素が Leaf の場合、その名前が返る (上の例の場合、これが適用されて"foo" が返る)。
- そのほかの場合、"none" が返る。

マッチされた値を取り出すのに = を使うこともできます。

```
var node = switch(myTree) {
  case Node(leafNode = Leaf("foo"), _): leafNode;
  case x: x;
}
trace(node); // Leaf(foo)
```

leafNode には Leaf("foo") が割り当てられているので、これにマッチします。そのほかのケースでは、myTree 自身が返ります。case x は case _と同じようにすべてにマッチしますが、x のような識別子が使われるとマッチした値がその変数に対して割り当てられます。

6.4.4　構造体マッチング

匿名構造体とインスタンスのフィールドに対してマッチさせることも可能です。

```
var myStructure = {
  name: "haxe",
  rating: "awesome"
};
var value = switch(myStructure) {
  case { name: "haxe", rating: "poor" }:
    throw false;
  case { rating: "awesome", name: n }:
    n;
  case _:
    "no awesome language found";
}
trace(value); // haxe
```

2 番目のケースでは、rating が”awesome” にマッチすると、name フィールドが識別子 n に割り当てられます。もちろん、この構造体を先の例の Tree に入れて、構造体と enum を合わせたマッチングを行うこともできます。

クラスインスタンスについては、その親クラスのフィールドについてはマッチングできないという制限があります。

6.4.5 配列マッチング

配列は固定長のマッチングを行うことができます。

```
var myArray = [1, 6];
var match = switch(myArray) {
  case [2, _]: ”0”;
  case [_, 6]: ”1”;
  case []: ”2”;
  case [_, _, _]: ”3”;
  case _: ”4”;
}
trace(match); // 1
```

この例では、array[1] が 6 にマッチし、array[0] は何でもよいので、1 が出力されます。

6.4.6 or パターン

| 演算子は複数のパターンが許容されることを示す用途で、パターン内のあらゆる箇所に使うことができます。

```
var match = switch(7) {
  case 4 | 1: ”0”;
  case 6 | 7: ”1”;
  case _: ”2”;
}
trace(match); // 1
```

or パターン内で変数の取得をしたい場合、その子要素両方で行わなくてはいけません。

6.4.7 ガード

case ... if(condition): の構文を使ってパターンをさらに限定することができます。

```
var myArray = [7, 6];
var s = switch(myArray) {
  case [a, b] if (b > a):
    b + ”>” +a;
  case [a, b]:
    b + ”<=” +a;
  case _: ”found something else”;
}
trace(s); // 6<=7
```

最初のケースは追加のガード条件 if (b > a) を持っています。このケースはこの条件が正だった場合のみ選択され、それ以外の場合は次のケースとのマッチングが続きます。

6.4.8　複数の値のマッチング

配列の構文は複数の値のマッチングにも使えます。

```
    var s = switch [1, false, "foo"] {
      case [1, false, "bar"]: "0";
      case [_, true, _]: "1";
      case [_, false, _]: "2";
    }
    trace(s); // 2
```

これは通常の配列のマッチングによく似ていますが、以下の点で違います。

- 要素数は固定です。このためパターンの配列の長さが違ってはいけません。
- switch している値を取得できません。例えば、case x は使えません (case _は使えます)。

6.4.9　抽出子 (エクストラクタ)

【Haxe 3.1.0 から】

抽出子 (エクストラクタ) はマッチした値に変更を適用することができます。マッチした値に小さな変更を適用して、さらにマッチングを行う場合に便利です。

```
enum Test {
  TString(s:String);
  TInt(i:Int);
}

class Main {
  static public function main() {
    var e = TString("fOo");
    switch(e) {
      case TString(temp):
        switch(temp.toLowerCase()) {
          case "foo": true;
          case _: false;
        }
      case _: false;
    }
  }
}
```

この場合、TString 列挙型コンストラクタの引数の値を、temp に割り当てて、さらにネストした temp.toLowerCase() に対する switch を行っています。見てのとおり、TString が"foo" の一部大文字のものを持っているので、このマッチングは成功します。これは抽出子を使うことで簡略化できます。

```
enum Test {
  TString(s:String);
  TInt(i:Int);
}

class Main {
  static public function main() {
```

```
    var e = TString("f0o");
    var success = switch(e) {
      case TString(_.toLowerCase() => "foo"):
        true;
      case _:
        false;
    }
  }
}
```

抽出子は extractorExpression => match の式によって認識されます。コンパイラはその前の例と同じようなコードを出力しますが、記述する構文はずいぶんと簡略化されました。抽出子は => で分断される以下の 2 つの部品からなります。

1. 左側はあらゆる式が可能で、アンダースコア (_) が出現する箇所すべてが、現在マッチする値で置き換えられます。
2. 右側は左側を評価した結果をマッチングするためのパターンです。

右側はパターンですから、さらに別の抽出子を使うことが可能です。以下の例では 2 つの抽出子をチェーンさせています。

```
class Main {
  static public function main() {
    switch(3) {
      case add(_, 1) => mul(_, 3) => a:
        trace(a);
    }
  }

  static function add(i1:Int, i2:Int) {
    return i1 + i2;
  }

  static function mul(i1:Int, i2:Int) {
    return i1 * i2;
  }
}
```

これは 3 がマッチして add(3, 1) を呼び出し、その結果の 4 がマッチして mul(4, 3) 呼び出された結果として、12 が出力されます。2 つ目の => の右側の a は変数取り出し (6.4.3) であることに注意してください。

現在は or パターン (6.4.6) 内で抽出子を使うことはできません。

```
class Main {
  static public function main() {
    switch("foo") {
      // orパターン内で抽出子は使えません。
      case (_.toLowerCase() => "foo") | "bar":
    }
  }
}
```

しかし、or パターンを抽出子の右側に使うことはできます。そのため、上の例は小かっこ無しの場合ではコンパイル可能です。

6.4.10 網羅性のチェック

コンパイラは起こりうるケースが忘れ去られてないかのチェックを行います。

```
1 switch(true) {
2     case false:
3 } // Unmatched patterns: true (trueにマッチするパターンが無い)
```

マッチング対象の Bool 型は true と false の 2 つの値を取り得ますが、false のみがチェックされています。

Figure out wtf our rules are now for when this is checked.

6.4.11 無意味なパターンのチェック

同じように、コンパイラはどのような入力値に対してもマッチしないパターンを禁止します。

```
1 switch(Leaf("foo")) {
2     case Leaf(_)
3         | Leaf("foo"): // This pattern is unused (このパターンは使用されない)
4     case Node(l,r):
5     case _: // This pattern is unused  (このパターンは使用されない)
6 }
```

6.5 文字列補間

Haxe3 では、文字列補間のおかげで、手動で文字列をつなげ合わせる必要がなくなりました。シングルクオート (') で囲まれた文字列の中で、ドル記号 ($) に続けて識別子を記述すると、その識別子を評価してつなげ合わせてくれます。

```
1 var x = 12;
2 // xの値は12
3 trace('xの値は$x');
```

さらに、$\${expr}$ を使うことで文字列内に式そのものを含めることが可能になります。この expr は Haxe の正当な式であれば、なんでもかまいません。

```
1 var x = 12;
2 // 12足す3は15
3 trace('$x足す3は${x + 3}');
```

文字列補間はコンパイル時の機能なので、実行時には影響を与えません。上の例は手動のつなげ合わせと同じです。コンパイラは以下と同様のコードを生成します。

```
1 trace(x + "足す3は" + (x + 3));
```

もちろん、一切の補間なしでシングルクオートで囲んだ文字列を使用することができますが、$ の文字が補間のトリガーとして予約されてしまっていることに気を付けてください。文字列内でドル記号そのものを使いたい場合は $$ を使います。

> トリビア: Haxe3 以前の文字列補間
> 文字列補間自体はバージョン 2.09 から Haxe の機能として存在しています。そのころは Std.format のマクロが使われいました。これは新しい文字列補間の構文よりも遅くてあまり快適でないものでした。

6.6　配列内包表記

Comprehensions are only listing Arrays, not Maps

Haxe の配列内包表記は既存の構文を配列の初期化をより簡単にするためにも使えるようにするものです。配列内包表記は for または while のキーワードによって識別されます。

```
class Main {
  static public function main() {
    var a = [for (i in 0...10) i];
    trace(a); // [0,1,2,3,4,5,6,7,8,9]

    var i = 0;
    var b = [while(i < 10) i++];
    trace(b); // [0,1,2,3,4,5,6,7,8,9]
  }
}
```

変数 a は 0 から 9 までの数値を要素として持つ配列として初期化されます。コンパイラはループを作ってその繰り返しの一つ一つで要素を追加するコードを出力します。つまり以下のコードと等価です。

```
var a = [];
for (i in 0...10) a.push(i);
```

変数 b も同じ値に初期化されますが、for ではなく while という異なる内包表記の形式を使っています。そして、これは以下のコードと等価です。

ループの式は条件分岐やループのネストを含めて、いかなる式でもかまいません。ですから、以下の式は期待通りに動作します。

```
class Main {
  static public function main() {
    var a = [
      for (a in 1...11)
        for(b in 2...4)
          if (a % b == 0)
            a+ "/" +b
    ];
    // [2/2,3/3,4/2,6/2,6/3,8/2,9/3,10/2]
    trace(a);
  }
}
```

6.7　イテレータ (反復子)

Haxe では、カスタムのイテレータや反復可能 (iterable) なデータ型を簡単に定義できます。これらの概念は Iterator<T> 型と Iterable<T> 型を使って以下のように表現されています。

```
typedef Iterator<T> = {
    function hasNext() : Bool;
    function next() : T;
}

typedef Iterable<T> = {
    function iterator() : Iterator<T>;
}
```

これらの型のいずれかで構造的に単一化できる (3.5.2) あらゆる class (2.3) は、for ループ (5.13) で反復処理を行うことができます。つまり、型が合うように hasNext と next メソッドを定義すればそのクラスはイテレータであるし、Iterator<T> を返す iterator メソッドを定義すれば反復可能な型です。

```
class MyStringIterator {
  var s:String;
  var i:Int;

  public function new(s:String) {
    this.s = s;
    i = 0;
  }

  public function hasNext() {
    return i < s.length;
  }

  public function next() {
    return s.charAt(i++);
  }
}

class Main {
  static public function main() {
    var myIt = new MyStringIterator("string");
    for (chr in myIt) {
      trace(chr);
    }
  }
}
```

この例での MyStringIterator は Bool 型を返す hasNext と String 型を返す next メソッドを定義しているので、イテレータであると見なされます。また next の戻り値の型から、これは Iterator<String> です。main メソッドでこれをインスタンス化して反復処理を行っています。

```
class MyArrayWrap<T> {
  var a:Array<T>;
  public function new(a:Array<T>) {
    this.a = a;
  }

  public function iterator() {
    return a.iterator();
  }
}

class Main {
  static public function main() {
    var myWrap = new MyArrayWrap([1, 2, 3]);
    for (elt in myWrap) {
      trace(elt);
    }
  }
```

```
19 }
```

こちらは 1 つ前の例とは異なり自前の Iterator を準備していませんが、代わりに MyArrayWrap<T> は Array<T> の iterator 関数を効果的に利用しています。

6.8 関数の束縛 (bind)

Haxe3 では、部分的に引数を適用して関数を束縛することが可能です。すべての関数型は bind フィールドを持っており、これを呼び出すことで引数の数を減らした新しい関数を作りだすことができます。その実例を示します。

```
1  class Main {
2    static public function main() {
3      var map = new haxe.ds.IntMap<String>();
4      var f = map.set.bind(_, "12");
5      $type(map.set); // Int -> String -> Void
6      $type(f); // Int -> Void
7      f(1);
8      f(2);
9      f(3);
10     trace(map); // {1 => 12, 2 => 12, 3 => 12}
11   }
12 }
```

行 4 では、map.set 関数に 2 番目の引数に 12 を適用し、f という変数に割り当てました。アンダースコア (_) はその引数を束縛しないことを表すのに使います。このことは map.set と、f の型の比較でもわかります。束縛された String 型の引数が取り除かれたので、Int->String->Void 型が Int->Void 型に変わっています。

f(1) を呼び出したことで実際には map.set(1, "12") が実行されます。f(2)、f(3) の呼び出しでも同じ関係性が成り立ちます。最後の行で、3 つのインデックスすべてに紐づく値が"12" になっていることが確認できます。

アンダースコア (_) は末尾の引数では省略することができます。つまり、map.set.bind(1) で最初の引数を束縛した場合、インデックス 1 について新しい値を設定する String->Void 関数が提供されます。

> トリビア: コールバック
> Haxe3 よりも前のバージョンでは、callback キーワードに 1 つの関数の引数と任意の個数の束縛する引数をつけて呼び出しをしていました。この束縛する機能に対してコールバック関数という名前が使われるようになっていました。
> callback は左から右への束縛のみでアンダースコア (_) はサポートしていませんでした。アンダースコアを使うという選択肢は論争を生み、そのほかの案もいくつか現れましたがこれより優れているものはありませんでした。少なくともアンダースコア (_) は「ここに値を入れて」と言っているように見えるので、この意味を書き表すのに適しているという結論にいたりました。

6.9 メタデータ

以下の要素はメタデータで属性をつけることができます。

- class、enum の定義
- クラスフィールド

・列挙型コンストラクタ

・式

これらのメタデータの情報は haxe.rtti.Meta の API を使って実行時に利用することが可能です。

```
import haxe.rtti.Meta;

@author("Nicolas")
@debug
class MyClass {
  @range(1, 8)
  var value:Int;

  @broken
  @:noCompletion
  static function method() { }
}

class Main {
  static public function main() {
    // { author : ["Nicolas"], debug : null }
    trace(Meta.getType(MyClass));
    // [1,8]
    trace(Meta.getFields(MyClass).value.range);
    // { broken: null }
    trace(Meta.getStatics(MyClass).method);
  }
}
```

メタデータは @ の文字で始まり、メタデータの名前が続き、その後にオプションでカンマで区切った定数値の引数が小かっこで囲まれている、ということで簡単に識別できます。

・MyClass クラスは"Nicolas" という文字列の引数 1 つを持つ author メタデータと、引数を持たない debug メタデータを持ちます。

・メンバ変数 value は 1 と 8 の 2 つの整数の引数を持つ range メタデータを持ちます。

・静的メソッド method は引数なしの broken メタデータと、引数なしの:noCompletion メタデータを持ちます。

main メソッドでは、API を通してこれらのメタデータへアクセスしています。この出力からは取得可能なデータの構造が分かります。

・各メタデータについてフィールドがあり、フィールドの名前はメタデータの名前です。

・フィールドの値はメタデータの引数に一致します。引数がない場合、フィールドの値は null です。その他の場合、フィールドの値は引数 1 つが要素 1 つになった配列です。

・: から始まるメタデータは省略されます。このメタデータはコンパイラメタデータとして知られます。

メタデータの引数の値は以下が使用できます。

・定数値 (5.2)

・配列の宣言 (5.5) (すべての要素がこのリストのいずれか)

・オブジェクトの宣言 (5.6) (すべての要素がこのリストのいずれか)

ビルトインのコンパイラメタデータ　コマンドラインから haxe --help-metas を実行することで、定義済みメタデータの完全なリストを得ることができます。
詳しくはコンパイラメタデータのリスト (8.1) を見てください。

6.10　アクセス制御

基本的な可視性 (4.4.1) のオプションで十分でない場合、アクセス制御が役に立ちます。アクセス制御はクラスレベルとフィールドレベル、そして以下の 2 方向の適用が可能です。

アクセス許可: :allow(target) メタデータ (6.9) を使うことで、対象を与えられたクラスやフィールドからのアクセスを許容するようにします。

アクセス強制: :access(target) メタデータ (6.9) を使うことで、対象からの与えられたクラスやフィールドへのアクセスを強制的に可能にします。

このとき、target には以下のドットパス (3.7) を使うことができます。

- クラスフィールド
- 　クラス、抽象型
- パッケージ

target はインポートを参照しません。つまり、完全なパスを正しく記述する必要があります。
クラスや抽象型の場合、アクセスの変更はその型のすべてのフィールドに反映されます。同じように、パッケージの場合、アクセスの変更はそのパッケージ内のすべての型のすべてのフィールドに反映されます。

```
1  @:allow(Main)
2  class MyClass {
3    static private var foo: Int;
4  }
5
6  class Main {
7    static public function main() {
8      MyClass.foo;
9    }
10 }
```

MyClass.foo は MyClass に @:allow(Main) を適用しているので、main メソッドからアクセスできます。このコードは @:allow(Main.main) でも動作しますし、以下のように MyClass クラスの foo フィールドにメタデータをつけても動作します。

```
1  class MyClass {
2    @:allow(Main.main)
3    static private var foo: Int;
4  }
5
6  class Main {
7    static public function main() {
8      MyClass.foo;
9    }
10 }
```

もし型にこのようなアクセスの変更ができない場合は、アクセス強制の方法が役立つかもしれません。

```
class MyClass {
  static private var foo: Int;
}

class Main {
  @:access(MyClass.foo)
  static public function main() {
    MyClass.foo;
  }
}
```

@:access(MyClass.foo) のメタデータは main メッソドからの foo の可視性を変更します。

> トリビア: メタデータという選択肢
> アクセス制御の言語機能には、新しい構文の導入ではなく、Haxe のメタデータの構文を使いました。これには以下のいくつかの理由があります。
>
> - 追加の構文は言語の構文解析を複雑にして、さらにはキーワードを増やしてしまします。
> - 追加の構文は言語のユーザーに追加の学習を要求します。メタデータであれば、それは既知のものです。
> - メタデータはこの拡張を行うのに十分な表現力を持っています。
> - メタデータは Haxe のマクロから、アクセスし、生成し、編集することが可能です。
>
> もちろん、メタデータ構文の主な不利益はメタデータの名前、クラスやパッケージ名についてスペルミス (例えば、@:acesss) をした場合に何のエラーも出ないことです。しかし、この機能では実際に private フィールドにアクセスしようとした場合にエラーがでるので、エラーが沈黙しているということにはなりえません。

【Haxe 3.1.0 から】

アクセスがインターフェース (2.3.3) に対して許可される場合、そのインターフェースを実装しているすべてのクラスに対してそれが引き継がれます。

```
class MyClass {
  @:allow(I)
  static private var foo: Int;
}

interface I { }

class Main implements I {
  static public function main() {
    MyClass.foo;
  }
}
```

これは親クラスの場合も同様です。その場合、子クラスに対して引き継ぎがされます。

トリビア: 壊れた機能
子クラスや実装クラスへのアクセスの継承は Haxe3.0 への導入を予定されており、そしてドキュメントまでも作られていました。しかし、このマニュアルを作る過程でこのアクセス制御の実装がぬけ落ちていることを発見しました。

6.11 インラインコンストラクタ

【Haxe 3.1.0 から】

コンストラクタに、inline (4.4.2) の宣言をつけると、コンパイラは特定の場合において最適化を試みます。この最適化が動作するためにはいくつかの必要事項があります。

- コンストラクタの呼び出しの結果はローカル変数への直接の代入でなければいけない。
- コンストラクタフィールドの式はそのフィールドへの代入のみでなければならない。

以下に、コンストラクタのインライン化の実例を挙げます。

```
1  class Point {
2    public var x:Float;
3    public var y:Float;
4
5    public inline function new(x:Float, y:Float) {
6      this.x = x;
7      this.y = y;
8    }
9  }
10
11 class Main {
12   static public function main() {
13     var pt = new Point(1.2, 9.3);
14   }
15 }
```

JavaScript 出力をみると、その効果がわかります。

```
1  Main.main = function() {
2      var pt_x = 1.2;
3      var pt_y = 9.3;
4  };
```

パート II

コンパイラリファレンス

章 7

コンパイラの使い方

基本的な使い方　Haxe コンパイラは基本的にはコマンドラインから以下の 2 つの質問に答える引数をつけて呼び出します。

- 何をコンパイルするのか?
- 何を出力するのか?

最初の質問に答えるためには、`-cp path` 引数でクラスパスを指定し、`-main dot_path` 引数でコンパイル対象のメインクラスを指定すれば十分です。これで Haxe コンパイラはメインクラスのファイルを解決しコンパイルを始めます。

2 つ目の質問に答えるためには、目的のターゲット特有の引数を指定します。Haxe ターゲットはそれぞれ専用のコマンドラインオプションを持っています。例えば、JavaScript は`-js file_name`、PHP は`-php directory` です。ターゲットによってファイル名を指定するもの (`-js`、`-swf`、`-neko`、`-python` が該当) と、ディレクトリを指定するものがあります。

よく使う引数　入力:

`-cp path` .hx のソースファイルまたはパッケージ (サブディレクトリ) が置かれているディレクトリのパスを追加します。

`-lib library_name` Haxelib (章 11) のライブラリを追加します。

`-main dot_path` メインクラスを設定します。

出力:

`-js file_name` 指定されたファイルに JavaScript (12.1) のソースコードを出力します。

`-as3 directory` 指定されたディレクトリに ActionScript3 のソースコードを出力します。

`-swf file_name` 指定されたファイルに Flash (12.2) の.swf を出力します。

`-neko file_name` 指定されたファイルに Neko (12.3) のバイナリを出力します。

`-php directory` 指定されたディレクトリに PHP (12.4) のソースコードを出力します。

`-cpp directory` 指定されたディレクトリに C++ (12.5) のソースコードを出力して、ネイティブの C++ コンパイラでコンパイルします。

`-cs directory` 指定されたディレクトリに C# (12.7) のソースコードを出力します。

`-java directory` 指定されたディレクトリに Java (12.6) のソースコードを出力して、Java コンパイラでコンパイルします。

`-python file_name` 指定されたファイルに Python (12.8) のソースコードを出力します。

章 8

コンパイラの機能

8.1　ビルトインのコンパイラメタデータ

Haxe 3.0 以降では、haxe --help-metas を実行することで定義済みのコンパイラメタデータのリストを得ることができます。

メタデータ	説明
@:abi	ABI 呼び出し規約を使う
@:abstract	基底クラスの実装を抽象型 (2.8) として使
@:access _(Target path)_	型またはフィールドへのプライベートなアク
@:allow _(Target path)_	型またはフィールドからのプライベートなア
@:analyzer	静的アナライザを設定する
@:annotation	Annotation (`@interface`) definitions o
@:arrayAccess	抽象型への配列アクセス (2.8.3) を許可す
@:autoBuild _(Build macro call)_	`@:build` メタデータをその型のすべての子
@:bind	SWF ライブラリで定義されているクラスを_
@:bitmap _(Bitmap file path)_	与えられたビットマップデータをクラスに埋
@:bridgeProperties	クラスのすべての Haxe プロパティに、ネイ
@:build _(Build macro call)_	マクロからクラスまたは列挙型を構築する。
@:buildXml	Build.xml に挿入する xml データを指定
@:callable	抽象型で基底型の関数呼び出しのアクセス
@:classCode	クラスにプラットフォームネイティブのコード
@:commutative	抽象型の 2 項演算子の各項を交換可能に
@:compilerGenerated	コンパイラから生成されたフィールドをマー
@:coreApi	このクラスをコア API のクラスとして認識す
@:coreType	抽象型をコアタイプ (2.8.7) として認識する
@:cppFileCode	C++ の出力ファイルに挿入するコード
@:cppInclude	C++ の出力ファイルに含めるファイル
@:cppNamespaceCode	
@:dce	`-dce full` が指定されていない場合でも、
@:debug	`-debug` が指定されていない場合でも、出力
@:decl	
@:defParam	
@:delegate	`-net-lib` のデリゲートで自動的に追加され
@:depend	
@:deprecated	`-java-lib` の `@Deprecated` で修飾されて
@:event	`-net-lib` のイベントに自動的に追加される
@:enum	抽象型の定義につけることで有限の値のセ
@:expose _(?Name=Class path)_	クラスを `window` オブジェクトか、node.js
@:extern	フィールドを `extern` としてマークする。結果
@:fakeEnum _(Type name)_	列挙型を指定した型の集合としてあつかう。
@:file(File path)	SWF ターゲットに指定したバイナリファイル
@:final	クラスが継承されるのを妨害する
@:font _(TTF path Range String)_	指定した TrueType フォントをクラスに埋め
@:forward _(List of field names)_	基底型からフィールドアクセスの繰り上げ (
@:from	抽象型のフィールドで、その型からのキャス
@:functionCode	
@:functionTailCode	
@:generic	クラスまたはクラスフィールドをジェネリック
@:genericBuild	型のインスタンスを指定したマクロを使って
@:getter _(Class field name)_	指定したフィールドにネイティブのゲッター
@:hack	`@:final` のマークがされているクラスの継承
@:headerClassCode	そのヘッダーで、生成されたクラスにコード
@:headerCode	生成されたヘッダファイルにコードを挿入す
@:headerNamespaceCode	
@:hxGen	Haxe によって生成された extern クラスに
@:ifFeature _(Feature name)_	指定された機能がコンパイルに含まれてい
@:include	
@:initPackage	
@:internal	クラスやフィールドに `internal` アクセスの
@:isVar	物理的フィールドが不要なプロパティに対し
@:javaCanonical _(Output type package,Output type name)_	Java ターゲットで型の正規パスを指定する
@:jsRequire	その `extern` に必要な JavaScript モジュ
@:keep	DCE (8.2) から、フィールドや型を保護する
@:keepInit	クラスからすべてのフィールドが削除された

8.2　デッドコード削除

デッドコード削除 (Dead Code Elimination、DCE) は、未使用のコードを出力から取り除くコンパイラ機能です。型付けの後に、DCE の始点（多くの場合は main メソッド）から再帰的にたどっていきどのフィールドと型が使用されているかを決定します。これにより使用済みのフィールドはマークされ、マークされていないフィールドはクラスから取り除かれます。

DCE には 3 つのモードがあり、コマンドラインからの呼び出し時に指定します。

-dce std: Haxe の標準ライブラリのクラスのみが DCE の影響を受けます。これがすべてのターゲットでのデフォルト値です。

-dce no: DCE されません。

-dce full: すべてのクラスが DCE の影響を受けます。

DCE のアルゴリズムは型付けされたコードではうまく働きますが、Dynamic (2.7) やリフレクション (10.7) を使っていると失敗する場合があります。そういった場合は、以下のメタデータを使ったクラスやフィールドの明示的な修飾が必要かもしれません。

`@:keep`: クラスに使用するとすべてのフィールドが DCE の対象から除外されます。フィールドに使用するとそのフィールドが DCE の対象になりません。

`@:keepSub`: クラスに使用すると、その子孫クラスすべてを `@:keep` で修飾したのと同様の動作をします。

`@:keepInit`: 通常、クラスはすべてのフィールドが DCE によって削除されると（あるいは最初から空だと）出力から削除されます。このメタデータを使うと、空のクラスが保持されます。

ソースコードを編集するのではなくコマンドラインからクラスを `@:keep` としてマークしたい場合、コンパイラマクロの`--macro keep('type dot path')` を使うことでそれが可能です。このマクロについて詳しくはhaxe.macro.Compiler.keep APIをご覧ください。パッケージをマークするとそのモジュールやサブタイプが DCE から保護されて、コンパイルに含まれます。

コンパイラは現在のモードに応じて、自動的に `dce` フラグの値を"std"、"no"、"full" のいずれかに設定します。このフラグは条件付きコンパイル (6.1) で使用できます。

> トリビア: DCE の書き直し
> DCE は元々 Haxe 2.07 で実装されましたが、その実装では関数は明示的に型付けされているときに使用しているという判定がされていました。このせいでいくつか機能で問題がありました。とくにインタフェースで深刻で、型安全性を確かめるためにはすべてのクラスフィールドを型付けする必要がありました。このせいで DCE は完全に破たんし、Haxe 2.10 での書き直しにつながりました。

> トリビア: DCE と try.haxe.org
> http://try.haxe.orgのサイトが公開されたとき、JavaScript ターゲットの DCE は大きく改善されました。JavaScript の出力コードに対する反応はさまざまでしたが、これにより削除されるコードの選択がより細かく行われるようになりました。

8.3　コンパイラサービス

8.3.1　概要

Haxe の豊富な型システム (3) は、IDE やエディタが正確な補完情報を提供することを難しくしています。型推論 (3.6) とマクロ (9) については、必要な挙動を再現するのに相当な量の仕事が必要です。こ

れが Haxe のコンパイラが、サードパーティーのソフトウェアが使うためのビルトインの補完モードを備えている理由です。

すべての補完は`--display file@position[@mode]` のコンパイラ引数を使うことで開始されます。この引数は以下を必要とします。

file: 補完のためチェックを行うファイルです。これは.hx ファイルの絶対パスまたは相対パスです。クラスパスやライブラリではありません。

position: 補完のためのチェックを行う与えられたファイルのバイト位置です（文字位置ではありません）。

mode: 使用する補完モードです（後述）。

補完モードの詳細については以下の通りです。

フィールドアクセス (8.3.2): その型のアクセス可能なフィールドのリストを提供します。

関数の引数 (8.3.3): 呼び出そうとしている関数の型を取得します。

型のパス (8.3.4): 子パッケージ、サブタイプ、静的フィールドをリストアップします。

使用状況 (8.3.5): コンパイルされるファイルのすべて中から指定された型またはフィールド、変数の出現位置をリストアップします。(”usage” モード)

定義位置 (8.3.6): 指定された型またはフィールド、変数の定義位置を取得します。(”position” モード)

トップレベル (8.3.7): 指定した位置で使用可能なすべての識別子（mode: ”toplevel”）

Haxe のコンパイラは非常に速いので、多くの場合は通常のコンパイラの呼び出しを使っても問題がありません。しかし、より大きな Haxe プロジェクトのためにサーバーモード (8.3.8) を用意してあります。これにより、ファイルに実際に変更があった場合や依存関係の更新がされたときだけ再コンパイルがされます。

インターフェースについての注意事項

- ファイルの目的の位置にパイプライン `|` の文字を置くことで、`position` 引数を 0 に設定することができます。これは、バイト数のカウント無しで IDE が何をすべきなのかをテストしたり実演したりするのに、便利です。この章のサンプルはこの機能を使っていきます。この機能は `|` が不正な構文になる位置に置かれている場合のみ動作します。例えば、ドットの直後(`.|`)や小かっこの直後(`(|`)です。
- 出力は HTML エスケープされます。つまり、&、く、> はそれぞれ `&`、`<`、`>` に変換されます。
- ドキュメントの出力は、ソースコード内のものと同じように改行とタブ文字を含みます。
- 補完モードの実行ではコンパイラはエラーを表示せずににエラーからの復帰を試みます。補完中に致命的なエラーが発生した場合、Haxe のコンパイラは補完結果の代わりにエラーメッセージを出力します。XML ではあらゆる出力は致命的なエラーメッセージであると判断できます。

8.3.2　フィールドアクセス補完

フィールドの補完はドット．文字の後から開始されて、その型で利用可能なフィールドをリストアップします。コンパイラは補完の位置までのすべての構文解析と型付けを行い、関連する情報を標準エラー出力に出力します。

```
1  class Main {
2    public static function main() {
3      trace("Hello".|
4    }
5  }
```

このファイルを Main.hx として保存すると、補完は haxe --display Main.hx@0 のコマンドを使って呼び出せます。その出力は以下のようなものでしょう（いくつかの情報を可読性のために削ったりフォーマットをかけたりしています）。

```
1  <list>
2  <i n="length">
3    <t>Int</t>
4    <d>
5      The number of characters in `this` String.
6    </d>
7  </i>
8  <i n="charAt">
9    <t>index : Int -&gt; String</t>
10   <d>
11     Returns the character at position `index` of `this` String.
12     If `index` is negative or exceeds `this.length`, the empty String
13     "" is returned.
14   </d>
15 </i>
16 <i n="charCodeAt">
17   <t>index : Int -&gt; Null&lt;Int&gt;</t>
18   <d>
19     Returns the character code at position `index` of `this` String.
20     If `index` is negative or exceeds `this.length`, null is returned.
21     To obtain the character code of a single character, "x".code can
22     be used instead to inline the character code at compile time.
23     Note that this only works on String literals of length 1.
24   </d>
25 </i>
26 </list>
```

この XML の構造は以下の通りです。

- ドキュメントの list ノードがいくつかの i ノードを含み、このそれぞれが 1 つのフィールドを表現しています。
- n 属性はフィールドの名前です。
- t ノードはフィールドの型です。
- d ノードはフィールドのドキュメントです。

【Haxe 3.2.0 から】

`-D display-details` をつけてコンパイルすると、各フィールドに `var` と `method` のいずれかの `k` 属性が付きます。これにより、関数型の変数フィールドとメソッドフィールドを区別できます。

8.3.3 呼び出し引数の補完

呼び出し引数の補完は小かっこ（の後から開始されて、呼び出しをしようとしている関数の型を返します。これはコンストラクタの呼び出しを含むすべての関数呼び出しで使用可能です。

```
1 class Main {
2   public static function main() {
3     trace("Hello".split(|
4   }
5 }
```

このファイルを Main.hx として保存すると、補完は `haxe --display Main.hx@0` のコマンドを使って呼び出せます。その出力は以下のようなものになります。

```
1 <type>
2 delimiter : String -&gt; Array&lt;String&gt;
3 </type>
```

IDE はここから、呼び出す関数が `delimiter` という `String` 型の引数が 1 つあって `Array<String>` を返すということを読み取れます。

> トリビア: 出力構造の問題
> 私たちは現在のフォーマットはほんの少しのうっとおしい自前の構文解析が必要になることを認めます。特に関数については、将来的にはより構造化された出力を提供するようになるかもしれません。

8.3.4 型のパスの補完

型のパスの補完は import 宣言 (3.7.2)、using 宣言 (6.3) あるいはあらゆる位置での型の記述で発生します。そしてこれは以降の 3 種類に分けることができます。

パッケージの補完　以下は haxe パッケージに属する子パッケージとモジュールのすべてをリストアップします。

```
1 import haxe.|
```

```
1  <list>
2  <i n="CallStack"><t></t><d></d></i>
3  <i n="Constraints"><t></t><d></d></i>
4  <i n="DynamicAccess"><t></t><d></d></i>
5  <i n="EnumFlags"><t></t><d></d></i>
6  <i n="EnumTools"><t></t><d></d></i>
7  <i n="Http"><t></t><d></d></i>
8  <i n="Int32"><t></t><d></d></i>
9  <i n="Int64"><t></t><d></d></i>
10 <i n="Json"><t></t><d></d></i>
11 <i n="Log"><t></t><d></d></i>
12 <i n="PosInfos"><t></t><d></d></i>
13 <i n="Resource"><t></t><d></d></i>
14 <i n="Serializer"><t></t><d></d></i>
```

```
<i n="Template"><t></t><d></d></i>
<i n="Timer"><t></t><d></d></i>
<i n="Ucs2"><t></t><d></d></i>
<i n="Unserializer"><t></t><d></d></i>
<i n="Utf8"><t></t><d></d></i>
<i n="crypto"><t></t><d></d></i>
<i n="ds"><t></t><d></d></i>
<i n="extern"><t></t><d></d></i>
<i n="format"><t></t><d></d></i>
<i n="io"><t></t><d></d></i>
<i n="macro"><t></t><d></d></i>
<i n="remoting"><t></t><d></d></i>
<i n="rtti"><t></t><d></d></i>
<i n="unit"><t></t><d></d></i>
<i n="web"><t></t><d></d></i>
<i n="xml"><t></t><d></d></i>
<i n="zip"><t></t><d></d></i>
</list>
```

モジュールのインポートの補完　以下は、haxe.Unserializer モジュールのサブタイプ (3.7.1) と、haxe.Unserializer の public static なフィールド(これらもインポート可能なので)のすべてをリストアップします。

```
import haxe.Unserializer.|
```

```
<list>
<i n="DEFAULT_RESOLVER">
  <t>haxe.TypeResolver</t>
  <d>
    This value can be set to use custom type resolvers.

    A type resolver finds a Class or Enum instance from a given String
       .
    By default, the haxe Type Api is used.

    A type resolver must provide two methods:

    1. resolveClass(name:String):Class&lt;Dynamic&gt; is called to
      determine a Class from a class name
    2. resolveEnum(name:String):Enum&lt;Dynamic&gt; is called to
      determine an Enum from an enum name

    This value is applied when a new Unserializer instance is created.
    Changing it afterwards has no effect on previously created
    instances.
  </d>
</i>
<i n="run">
  <t>v : String -&gt; Dynamic</t>
  <d>
    Unserializes `v` and returns the according value.
```

```
26
27      This is a convenience function for creating a new instance of
28      Unserializer with `v` as buffer and calling its unserialize()
29      method once.
30    </d>
31  </i>
32  <i n="TypeResolver"><t></t><d></d></i>
33  <i n="Unserializer"><t></t><d></d></i>
34  </list>
```

```
1  using haxe.Unserializer.|
```

その他のモジュールの補完　以下は、haxe.Unserializer のすべてのサブタイプ (3.7.1) をリストアップします。

```
1  using haxe.Unserializer.|
```

```
1  class Main {
2    static public function main() {
3      var x:haxe.Unserializer.|
4    }
5  }
```

```
1  <list>
2  <i n="TypeResolver"><t></t><d></d></i>
3  <i n="Unserializer"><t></t><d></d></i>
4  </list>
```

8.3.5　使用状況の補完

【Haxe 3.2.0 から】

使用状況の補完は"usage" モードの引数を使うことで使用可能です（詳しくは概要 (節 8.3.1)）。ローカル変数を使って実演してみますが、フィールドと型についても同じように動作することに気をつけてください。

```
1  class Main {
2    public static function main() {
3      var a = 1;
4      var b = a + 1;
5      trace(a);
6      a.|
7    }
8  }
```

このファイルを Main.hx として保存すると、補完は haxe --display Main.hx@0@usage のコマンドを使って呼び出せます。この出力は以下のようなものになります。

```
1  <list>
2  <pos>main.hx:4: characters 9-10</pos>
3  <pos>main.hx:5: characters 7-8</pos>
4  <pos>main.hx:6: characters 1-2</pos>
5  </list>
```

8.3.6　定義位置の補完

【Haxe 3.2.0 から】

定義位置の補完は”position” モードの引数を使うことで使用可能です（詳しくは概要 (節 8.3.1)）。フィールドを使って実演しますが、ローカル変数と型でも同じように動作することに気をつけてください。

```
1 class Main {
2   static public function main() {
3     "foo".split.|
4 }
```

このファイルを Main.hx として保存すると、補完は haxe --display Main.hx@0@position のコマンドを使って呼び出せます。この出力は以下のようなものになります。

```
1 <list>
2 <pos>std/string.hx:124: characters 1-54</pos>
3 </list>
```

トリビア: ターゲットの特定の省略による影響
このサンプルでは標準の String.hx が取得されましたが、実際の実装はありません。これはどのターゲットとも特定しなかったためであり、補完モードではそれでも構いません。例えば-neko neko.n のコマンドラインが含められた場合、結果として取得される位置は代わりに std/ neko/_std/ string.hx:84: lines 84-98. となるでしょう。

8.3.7　トップレベルの補完

【Haxe 3.2.0 から】

トップレベルの補完は、与えられた補完位置での使用可能な Haxe コンパイラが知るかぎりのすべての識別子を表示します。この補完機能だけはその効果を実演するために、実際の位置を引数であたえてやる必要があります。

```
1  class Main {
2    static public function main() {
3      var a = 1;
4    }
5  }
6
7  enum MyEnum {
8    MyConstructor1;
9    MyConstructor2(s:String);
10 }
```

このファイルを Main.hx として保存すると、補完は haxe --display Main.hx@63@toplevel のコマンドを使って呼び出せます。その出力は以下のようなものになります（簡潔さのためにいくつかの要素を削っています）。

```
1 <il>
2 <i k="local" t="Int">a</i>
3 <i k="static" t="Void -&gt; Unknown&lt;0&gt;">main</i>
4 <i k="enum" t="MyEnum">MyConstructor1</i>
5 <i k="enum" t="s : String -&gt; MyEnum">MyConstructor2</i>
```

```
<i k="package">sys</i>
<i k="package">haxe</i>
<i k="type" p="Int">Int</i>
<i k="type" p="Float">Float</i>
<i k="type" p="MyEnum">MyEnum</i>
<i k="type" p="Main">Main</i>
</il>
```

XML の構造は各要素の k 属性によります。すべての場合で i のノードはその値として名前を持ちます。

local, member, static, enum, global: t 属性にその変数やフィールドの型を持ちます。

global, type: p 属性にその型やフィールドが属するモジュールのパスを持ちます。

8.3.8 補完サーバー

コンパイルと補完を最速で行いたいのであれば、--wait のコマンドラインパラメータで Haxe の補完サーバーを立ち上げることができます。また、-v でサーバーがログを出力するようになります。以下が例です。

```
haxe -v --wait 6000
```

こうすることで、Haxe サーバーに接続して、コマンドラインパラメータを送って、NULL 文字を送ると、レスポンスの読み取りができます（補完が成功の場合も失敗の場合も）。

--connect のコマンドラインのパラメータを使うことで、Haxe はコンパイルコマンドを直接実行するのではなくサーバーに送るようになります。

```
haxe --connect 6000 myproject.hxml
```

はじめに--cwd のパラメータを使うことで、Haxe サーバーの現在の作業ディレクトリを変更することができることに気をつけてください。多くの場合クラスパスとそのほかのファイルはプロジェクトからの相対パスで指定されます。

動作の詳細　コンパイルサーバーは以下の内容をキャッシュします。

構文解析したファイル　ファイルは編集があせれたときが解析エラーになったときのみ再度構文解析が行われます。

Haxelib の呼び出し　前回の Haxelib の呼び出し結果は再度利用されます（補完時のみです。コンパイルには関係ありません）

型付けされたモジュール　モジュールのコンパイルはコンパイル成功した場合にキャッシュされて、その依存関係が更新されるまで補完とコンパイルに再利用されます。

コマンドラインで--times を追加することで、コンパイラが使用した正確な時間を取得して、コンパイルサーバーがどのような影響を与えたかを知ることができます。

プロトコル　次の Haxe/Neko の例からわかるとおり、サーバーのポートに単純につないで、1 行づつコマンドを送って、NULL 文字で終了します。その後に結果を読み取ります。

マクロやその他のコマンドはエラー以外のログを出力することができます。コマンドラインからの実行の場合は、標準出力と標準エラー出力にプリントされるものの違いがありますが、ソケットモードの場合は違います。この 2 つを区別するために、ログメッセージ（エラーではない）は
x01 の文字から始まり、そのメッセージのすべての改行文字は
x01 で置き換えられます。

警告やそのほかのメッセージはエラーと考えられますが、致命的なものではありません。致命的なエラーが起こると、
x02 から始まる 1 行のメッセージが送られます。
以下にサーバーへ接続して、このプロトコルに従った処理を行うコードがあります。

```
1  class Test {
2      static function main() {
3          var newline = "¥textbackslash¥ n";
4          var s = new neko.net.Socket();
5          s.connect(new neko.net.Host("127.0.0.1"),6000);
6          s.write("--cwd /my/project" + newline);
7          s.write("myproject.hxml" + newline);
8          s.write("¥textbackslash¥ 000");
9
10         var hasError = false;
11         for (line in s.read().split(newline))
12         {
13             switch (line.charCodeAt(0)) {
14                 case 0x01:
15                     neko.Lib.print(line.substr(1).split("¥
                           textbackslash¥ x01").join(newline));
16                 case 0x02:
17                     hasError = true;
18                 default:
19                     neko.io.File.stderr().writeString(line + newline);
20             }
21         }
22         if (hasError) neko.Sys.exit(1);
23     }
24 }
```

マクロの影響　コンパイルサーバーはマクロの実行 (9) に副作用を与えます。

8.4　リソース

Haxe は単純なリソース埋め込みのシステムを提供しています。これによりファイルをコンパイル後のアプリケーションに直接埋め込むことができます。
この方法は画像や音楽のような巨大なファイルの埋め込みには適していないかもしれませんが、設定や XML のようなより小さなデータを埋め込むのにはとても便利です。

8.4.1　リソースの埋め込み

以下のように、-resource のコンパイラ引数をつかって外部ファイルの埋め込みができます。

what to use for listing of non-haxe code like hxml? [1]

```
-resource hello_message.txt@welcome
```

@ マークの後の文字列はリソースの識別子です。コードからリソースを取得するのに使います。省略された場合（@ マークごと）、ファイル名がリソース識別子として使われます。

8.4.2　テキストリソースを取得する

埋め込んだリソースを取得するには、haxe.Resource の getString の静的メソッドにリソース識別子を渡して事項します。

```
1 class Main {
2   static function main() {
3     trace(haxe.Resource.getString("welcome"));
4   }
5 }
```

上記のコードは先ほどの welcome を識別子として使って hello_message.txt ファイルの内容を表示します。

8.4.3　バイナリリソースを取得する

巨大バイナリファイルをアプリケーションに埋め込むのは推奨されないものの、バイナリデータの埋め込みは便利です。埋め込んだリソースは haxe.Resource の getBytes の静的メソッドを使うことでバイナリとして取得できます。

```
1 class Main {
2   static function main() {
3     var bytes = haxe.Resource.getBytes("welcome");
4     trace(bytes.readString(0, bytes.length));
5   }
6 }
```

getBytes メソッドの戻り値の型は、データの各バイトにアクセスできる haxe.io.Bytes です。

8.4.4　実装の詳細

ターゲットのプラットフォームにリソースの埋め込み機能があればそれを使います。その他の場合、独自の実装を持ちます。

- Flash リソースは ByteArray として定義されて埋め込まれる。
- C# コンパイルされたアセンブリに含まれる。
- Java JAR ファイル内にパッケージされる。
- C++ グローバルなバイト列の定数として記録される。
- JavaScript Haxe シリアル化フォーマットに従ってシリアル化されて haxe.Resource の静的フィールドに記録される。
- Neko 文字列として haxe.Resource クラスの静的フィールドに記録される。

8.5　実行時型情報(RTTI)

Haxe コンパイラは:rtti メタデータで修飾されたクラス、あるいはその子孫クラスに対して実行時型情報(RTTI)を生成します。
【Haxe 3.2.0 から】

haxe.rtti.Rtti 型が RTTI についての処理を簡単にするために導入されました。現在では、情報の取得はとても簡単です。

```
@:rtti
class Main {
  var x:String;
  static function main() {
    var rtti = haxe.rtti.Rtti.getRtti(Main);
    trace(rtti);
  }
}
```

8.5.1 RTTI の構造

一般的な型情報

path: 型のパス (3.7)。

module: その型を含んでいるモジュール (3.7) のパス。

file: その型を含む.hx のファイルのフルパス。例えばマクロ (9) から定義された場合など、そのようなファイルがない場合 `null` になる場合がある。

params: その型が持つ型パラメータ (3.2) を表す文字列の配列。Haxe3.2.0 では、制約 (3.2.1) についての情報を持ちません。

doc: その型のドキュメント。この情報は`-D use_rtti_doc` のコンパイラフラグ (6.1) を付けた場合のみ使用できます。フラグがないか、ドキュメントがない場合は `null` です。

isPrivate: その型が private (3.7.1) かどうか。

platforms: その型が利用可能なターゲットのリスト。

meta: その型につけられているメタデータ。

クラスの型情報

isExtern: クラスが extern (6.2) かどうか。

isInterface: インターフェース (2.3.3) かどうか。

superClass: その親クラスの型のパスと型パラメータのリスト。

interfaces: そのクラスのインターフェースの型のパスと型パラメータのリストのリスト。

fields: クラスフィールド情報 (節 8.5.1) に記載されている、クラスフィールド (4) のリスト

statics: クラスフィールド情報 (節 8.5.1) に記載されている、静的クラスフィールドのリスト。

tdynamic: そのクラスに動的に実装 (2.7.2) された型、あるいは型が存在しない場合は `null`

列挙型の型情報

isExtern: その列挙型が extern (6.2) かどうか。

constructors: その列挙型コンストラクタのリスト。

抽象型の型情報

to: 定義されている暗黙の to キャスト (2.8.1) の配列。

from: 定義されている暗黙の to キャスト (2.8.1) の配列。

impl: 実装しているクラスのクラスの型情報 (8.5.1)。

athis: その抽象型の基底型 (2.8)。

クラスフィールド情報

name: フィールドの名前。

type: フィールドの型。

isPublic: フィールドが public (4.4.1) かどうか。

isOverride: フィールドが別のフィールドのオーバーライド (4.4.4) かどうか。

doc: フィールドのドキュメント。この情報は`-D use_rtti_doc` のコンパイラフラグ (6.1) を付けた場合のみ使用できます。フラグがないか、ドキュメントがない場合は `null` です。

get: フィールドの読み込みアクセスの挙動 (4.2)。

set: フィールドの書き込みアクセスの挙動 (4.2)。

params: そのフィールドが持つ型パラメータ (3.2) を表す文字列の配列。Haxe3.2.0 では、制約 (3.2.1) についての情報を持ちません。

platforms: フィールドが使用可能なターゲットのリスト。

meta: フィールドに付けられているメタデータ。

line: フィールドが定義されている行番号。この情報はフィールドが式を持つ場合のみ使用可能です。その他の場合は `null` です

overloads: このフィールドに対する利用可能なオーバーロードのリスト。存在しなければ `null` です。

列挙型コンストラクタ情報

name: コンストラクタ名。

args: 引数のリスト。存在しなければ `null` です。

doc: コンストラクタのドキュメント。この情報は`-D use_rtti_doc` のコンパイラフラグ (6.1) を付けた場合のみ使用できます。フラグがないか、ドキュメントがない場合は `null` です。

platforms: コンストラクタが使用可能なターゲットのリスト。

meta: コンストラクタに付けられているメタデータ。

章 9

Macros

Macros are without a doubt the most advanced feature in Haxe. They are often perceived as dark magic that only a select few are capable of mastering, yet there is nothing magical (and certainly nothing dark) about them.

定義: Abstract Syntax Tree (AST)
The AST is the result of parsing Haxe code into a typed structure. This structure is exposed to macros through the types defined in the file `haxe/macro/Expr.hx` of the Haxe Standard Library.

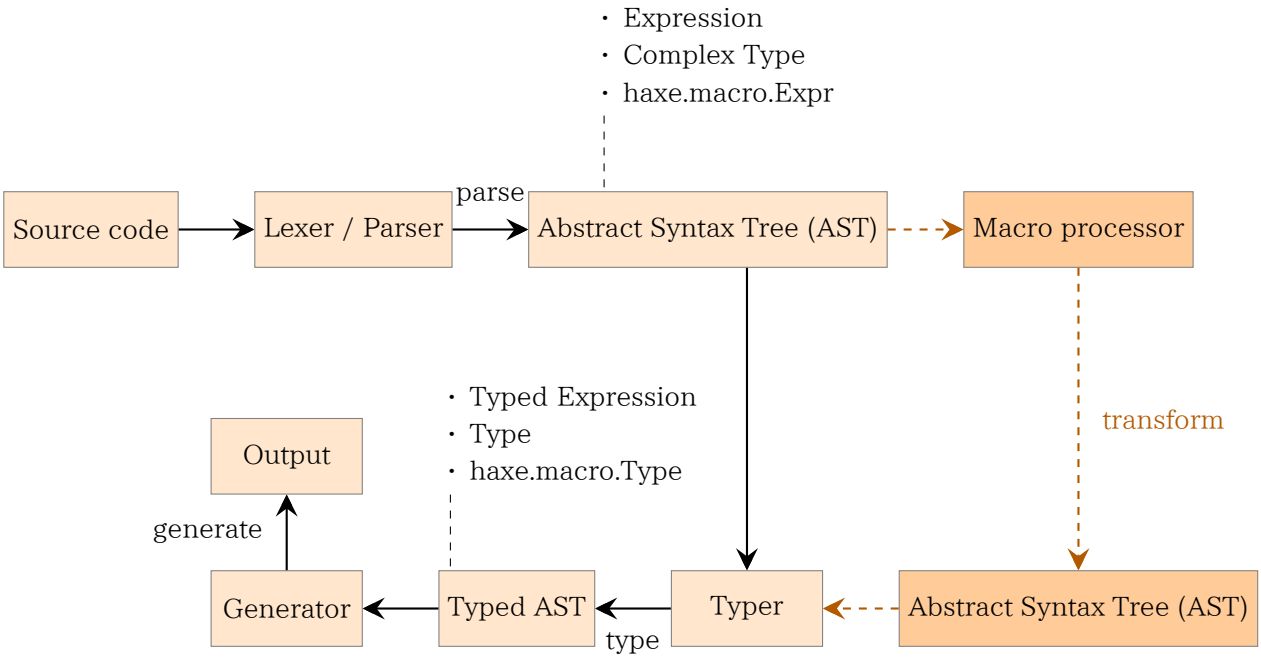

Figure 9.1: The role of macros during compilation.

A basic macro is a syntax-transformation. It receives zero or more expressions (5) and also returns an expression. If a macro is called, it effectively inserts code at the place it

was called from. In that respect, it could be compared to a preprocessor like `#define` in C++, but a Haxe macro is not a textual replacement tool.

We can identify different kinds of macros, which are run at specific compilation stages:

Initialization Macros: These are provided by command line using the `--macro` compiler parameter. They are executed after the compiler arguments were processed and the typer context has been created, but before any typing was done (see Initialization macros (節 9.7)).

Build Macros: These are defined for classes, enums and abstracts through the `@:build` or `@:autoBuild` metadata (6.9). They are executed per-type, after the type has been set up (including its relation to other types, such as inheritance for classes) but before its fields are typed (see Type Building (節 9.5)).

Expression Macros: These are normal functions which are executed as soon as they are typed.

9.1 Macro Context

定義: Macro Context
The macro context is the environment in which the macro is executed. Depending on the macro type, it can be considered to be a class being built or a function being typed. Contextual information can be obtained through the `haxe.macro.Context` API.

Haxe macros have access to different contextual information depending on the macro type. Other than querying such information, the context also allows some modifications such as defining a new type or registering certain callbacks. It is important to understand that not all information is available for all macro kinds, as the following examples demonstrate:

- Initialization macros will find that the `Context.getLocal*()` methods return `null`. There is no local type or method in the context of an initialization macro.
- Only build macros get a proper return value from `Context.getBuildFields()`. There are no fields being built for the other macro kinds.
- Build macros have a local type (if incomplete), but no local method, so `Context.getLocalMethod()` returns `null`.

The context API is complemented by the `haxe.macro.Compiler` API detailed in Initialization macros (節 9.7). While this API is available to all macro kinds, care has to be taken for any modification outside of initialization macros. This stems from the natural limitation of undefined build order (9.6.3), which could cause e.g. a flag definition through `Compiler.define()` to take effect before or after a conditional compilation (6.1) check against that flag.

9.2 Arguments

Most of the time, arguments to macros are expressions represented as an instance of enum `Expr`. As such, they are parsed but not typed, meaning they can be anything conforming to

Haxe's syntax rules. The macro can then inspect their structure, or (try to) get their type using `haxe.macro.Context.typeof()`.

It is important to understand that arguments to macros are not guaranteed to be evaluated, so any intended side-effect is not guaranteed to occur. On the other hand, it is also important to understand that an argument expression may be duplicated by a macro and used multiple times in the returned expression:

```
1  import haxe.macro.Expr;
2
3  class Main {
4    static public function main() {
5      var x = 0;
6      var b = add(x++);
7      trace(x); // 2
8    }
9
10   macro static function add(e:Expr) {
11     return macro $e + $e;
12   }
13 }
```

The macro `add` is called with `x++` as argument and thus returns `x+++x++` using expression reification (9.3.1), causing `x` to be incremented twice.

9.2.1 ExprOf

Since `Expr` is compatible with any possible input, Haxe provides the type `haxe.macro.ExprOf<T>`. For the most part, this type is identical to `Expr`, but it allows constraining the type of accepted expressions. This is useful when combining macros with static extensions (6.3):

```
1  import haxe.macro.Expr;
2  using Main;
3
4  class Main {
5    static public function main() {
6      identity("foo");
7      identity(1);
8      "foo".identity();
9      // Intにはidentityフィールドはありません。
10     //1.identity();
11   }
12
13   macro static function identity(e:ExprOf<String>) {
14     return e;
15   }
16 }
```

The two direct calls to `identity` are accepted, even though the argument is declared as `ExprOf<String>`. It might come as a surprise that the `Int 1` is accepted, but it is a logical consequence of what was explained about macro arguments (9.2): The argument expressions are never typed, so it is not possible for the compiler to check their compatibility by unifying (3.5).

This is different for the next two lines which are using static extensions (note the `using Main`): For these it is mandatory to type the left side (`"foo"` and `1`) first in order to make

sense of the `identity` field access. This makes it possible to check the types against the argument types, which causes `1.identity()` to not consider `Main.identity()` as a suitable field.

9.2.2 Constant Expressions

A macro can be declared to expect constant (5.2) arguments:

```
class Main {
  static public function main() {
    const("foo", 1, 1.5, true);
  }

  macro static function const(s:String, i:Int, f:Float, b:Bool) {
    trace(s);
    trace(i);
    trace(f);
    trace(b);
    return macro null;
  }
}
```

With these it is not necessary to detour over expressions as the compiler can use the provided constants directly.

9.2.3 Rest Argument

If the final argument of a macro is of type `Array<Expr>`, the macro accepts an arbitrary number of extra arguments which are available from that array:

```
import haxe.macro.Expr;

class Main {
  static public function main() {
    myMacro("foo", a, b, c);
  }

  macro static function myMacro(e1:Expr, extra:Array<Expr>) {
    for (e in extra) {
      trace(e);
    }
    return macro null;
  }
}
```

9.3 Reification

The Haxe Compiler allows reification of expressions, types and classes to simplify working with macros. The syntax for reification is `macro expr`, where `expr` is any valid Haxe expression.

9.3.1 Expression Reification

Expression reification is used to create instances of `haxe.macro.Expr` in a convenient way. The Haxe Compiler accepts the usual Haxe syntax and translates it to an expression object. It supports several escaping mechanisms, all of which are triggered by the `$` character:

`${}` and `$e{}`: `Expr -> Expr` This can be used to compose expressions. The expression within the delimiting `{ }` is executed, with its value being used in place.

`$a{}`: `Expr -> Array<Expr>` If used in a place where an `Array<Expr>` is expected (e.g. call arguments, block elements), `$a{}` treats its value as that array. Otherwise it generates an array declaration.

`$b{}`: `Array<Expr> -> Expr` Generates a block expression from the given expression array.

`$i{}`: `String -> Expr` Generates an identifier from the given string.

`$p{}`: `Array<String> -> Expr` Generates a field expression from the given string array.

`$v{}`: `Dynamic -> Expr` Generates an expression depending on the type of its argument. This is only guaranteed to work for basic types (2.1) and enum instances (2.4).

Additionally the metadata (6.9) `@:pos(p)` can be used to map the position of the annotated expression to `p` instead of the place it is reified at.

This kind of reification only works in places where the internal structure expects an expression. This disallows `object.${fieldName}`, but `object.$fieldName` works. This is true for all places where the internal structure expects a string:

- field access `object.$name`
- variable name `var $name = 1;`

【Haxe 3.1.0 から】

- field name `{ $name: 1}`
- function name `function $name() { }`
- catch variable name `try e() catch($name:Dynamic) {}`

9.3.2 Type Reification

Type reification is used to create instances of `haxe.macro.Expr.ComplexType` in a convenient way. It is identified by a `macro : Type`, where `Type` can be any valid type path expression. This is similar to explicit type hints in normal code, e.g. for variables in the form of `var x:Type`.

Each constructor of `ComplexType` has a distinct syntax:

```
TPath: macro : pack.Type

TFunction: macro : Arg1 -> Arg2 -> Return

TAnonymous: macro : { field: Type }

TParent: macro : (Type)

TExtend: macro : {> Type, field: Type }

TOptional: macro : ?Type
```

9.3.3 Class Reification

It is also possible to use reification to obtain an instance of `haxe.macro.Expr.TypeDefinition`. This is indicated by the `macro class` syntax as shown here:

```
class Main {
  macro static function generateClass(funcName:String) {
    var c = macro class MyClass {
      public function new() { }
      public function $funcName() {
        trace($v{funcName} + " was called");
      }
    }
    haxe.macro.Context.defineType(c);
    return macro new MyClass();
  }

  public static function main() {
    var c = generateClass("myFunc");
    c.myFunc();
  }
}
```

The generated `TypeDefinition` instance is typically passed to `haxe.macro.Context.defineType` in order to add a new type to the calling context (not the macro context itself).

This kind of reification can also be useful to obtain instances of `haxe.macro.Expr.Field`, which are available from the `fields` array of the generated `TypeDefinition`.

9.4 Tools

The Haxe Standard Library comes with a set of tool-classes to simplify working with macros. These classes work best as static extensions (6.3) and can be brought into context either individually or as a whole through `using haxe.macro.Tools`. These classes are:

`ComplexTypeTools`: Allows printing `ComplexType` instances in a human-readable way. Also allows determining the `Type` corresponding to a `ComplexType`.

`ExprTools`: Allows printing `Expr` instances in a human-readable way. Also allows iterating and mapping expressions.

`MacroStringTools`: Offers useful operations on strings and string expressions in macro context.

`TypeTools`: Allows printing `Type` instances in a human-readable way. Also offers several useful operations on types, such as unifying (3.5) them or getting their corresponding `ComplexType`.

Furthermore the `haxe.macro.Printer` class has public methods for printing various types as a human-readable format. This can be helpful when debugging macros.

トリビア: The tinkerbell library and why Tools.hx works
We learned about static extensions that using a module implies that all its types are brought into static extension context. As it turns out, such a type can also be a typedef (3.1) to another type. The compiler then considers this type part of the module, and extends static extension accordingly.
This "trick" was first used in Juraj Kirchheim's tinkerbell[1] library for exactly the same purpose. Tinkerbell provided many useful macro tools long before they made it into the Haxe Compiler and Haxe Standard Library. It remains the primary library for additional macro tools and offers other useful functionality as well.

9.5 Type Building

Type-building macros are different from expression macros in several ways:

- They do not return expressions, but an array of class fields. Their return type must be set explicitly to `Array<haxe.macro.Expr.Field>`.
- Their context (9.1) has no local method and no local variables.
- Their context does have build fields, available from `haxe.macro.Context.getBuildFields()`.
- They are not called directly, but are argument to a `@:build` or `@:autoBuild` metadata (6.9) on a class (2.3) or enum (2.4) declaration.

The following example demonstrates type building. Note that it is split up into two files for a reason: If a module contains a `macro` function, it has to be typed into macro context as well. This is often a problem for type-building macros because the type to be built could only be loaded in its incomplete state, before the building macro has run. We recommend to always define type-building macros in their own module.

```
import haxe.macro.Context;
import haxe.macro.Expr;

class TypeBuildingMacro {
  macro static public function build(fieldName:String):Array<Field> {
    var fields = Context.getBuildFields();
    var newField = {
      name: fieldName,
      doc: null,
      meta: [],
      access: [AStatic, APublic],
      kind: FVar(macro : String, macro "my default"),
      pos: Context.currentPos()
    };
    fields.push(newField);
    return fields;
  }
}
```

```
@:build(TypeBuildingMacro.build("myFunc"))
class Main {
```

```
3     static public function main() {
4       trace(Main.myFunc); // my default
5     }
6   }
```

The `build` method of `TypeBuildingMacro` performs three steps:

1. It obtains the build fields using `Context.getBuildFields()`.
2. It declares a new `haxe.macro.expr.Field` field using the `funcName` macro argument as field name. This field is a `String` variable (4.1) with a default value `"my default"` (from the `kind` field) and is public and static (from the `access` field).
3. It adds the new field to the build field array and returns it.

This macro is argument to the `@:build` metadata on the `Main` class. As soon as this type is required, the compiler does the following:

1. It parses the module file, including the class fields.
2. It sets up the type, including its relation to other types through inheritance (2.3.2) and interfaces (2.3.3).
3. It executes the type-building macro according to the `@:build` metadata.
4. It continues typing the class normally with the fields returned by the type-building macro.

This allows adding and modifying class fields at will in a type-building macro. In our example, the macro is called with a `"myFunc"` argument, making `Main.myFunc` a valid field access.

If a type-building macro should not modify anything, the macro can return `null`. This indicates to the compiler that no changes are intended and is preferable to returning `Context.getBuildFields()`.

9.5.1 Enum building

Building enums (2.4) is analogous to building classes with a simple mapping:

- Enum constructors without arguments are variable fields `FVar`.
- Enum constructors with arguments are method fields `FFun`.

Check if we can build GADTs this way.

```
1   import haxe.macro.Context;
2   import haxe.macro.Expr;
3
4   class EnumBuildingMacro {
5     macro static public function build():Array<Field> {
6       var noArgs = makeEnumField("A", FVar(null, null));
7       var eFunc = macro function(value:Int) { };
8       var fInt = switch (eFunc.expr) {
9         case EFunction(_,f): f;
10        case _: throw "false";
11      }
```

```
    var intArg = makeEnumField("B", FFun(fInt));
    return [noArgs, intArg];
  }

  static function makeEnumField(name, kind) {
    return {
      name: name,
      doc: null,
      meta: [],
      access: [],
      kind: kind,
      pos: Context.currentPos()
    }
  }
}
```

```
@:build(EnumBuildingMacro.build())
enum E { }

class Main {
  static public function main() {
    switch(E.A) {
      case A:
      case B(v):
    }
  }
}
```

Because enum `E` is annotated with a `:build` metadata, the called macro builds two constructors `A` and `B` "into" it. The former is added with the kind being `FVar(null, null)`, meaning it is a constructor without argument. For the latter, we use reification (9.3.1) to obtain an instance of `haxe.macro.Expr.Function` with a singular `Int` argument.

The `main` method proves the structure of our generated enum by matching (6.4) it. We can see that the generated type is equivalent to this:

```
enum E {
    A;
    B(value:Int);
}
```

9.5.2 @:autoBuild

If a class has the `:autoBuild` metadata, the compiler generates `:build` metadata on all extending classes. If an interface has the `:autoBuild` metadata, the compiler generates `:build` metadata on all implementing classes and all extending interfaces. Note that `:autoBuild` does not imply `:build` on the class/interface itself.

```
import haxe.macro.Context;
import haxe.macro.Expr;

class AutoBuildingMacro {
  macro static public
```

```
  function fromInterface():Array<Field> {
    trace("fromInterface: " + Context.getLocalType());
    return null;
  }

  macro static public
  function fromBaseClass():Array<Field> {
    trace("fromBaseClass: " + Context.getLocalType());
    return null;
  }
}
```

```
@:autoBuild(AutoBuildingMacro.fromInterface())
interface I { }

interface I2 extends I { }

@:autoBuild(AutoBuildingMacro.fromBaseClass())
class Base { }

class Main extends Base implements I2 {
  static public function main() { }
}
```

This outputs during compilation:

```
AutoBuildingMacro.hx:6:
  fromInterface: TInst(I2,[])
AutoBuildingMacro.hx:6:
  fromInterface: TInst(Main,[])
AutoBuildingMacro.hx:11:
  fromBaseClass: TInst(Main,[])
```

It is important to keep in mind that the order of these macro executions is undefined, which is detailed in Build Order (節 9.6.3).

9.5.3 @:genericBuild

【Haxe 3.1.0 から】

Normal build-macros (9.5) are run per-type and are already very powerful. In some cases it is useful to run a build macro per type usage instead, i.e. whenever it actually appears in the code. Among other things this allows accessing the concrete type parameters in the macro.

`@:genericBuild` is used just like `@:build` by adding it to a type with the argument being a macro call:

```
import haxe.macro.Expr;
import haxe.macro.Context;
import haxe.macro.Type;

class GenericBuildMacro1 {
  static public function build() {
```

```
    switch (Context.getLocalType()) {
      case TInst(_, [t1]):
        trace(t1);
      case t:
        Context.error("Classが要求されています", Context.currentPos())
            ;
    }
    return null;
  }
}
```

```
@:genericBuild(GenericBuildMacro1.build())
class MyType<T> { }

class Main {
  static function main() {
    var x:MyType<Int>;
    var x:MyType<String>;
  }
}
```

When running this example the compiler outputs `TAbstract(Int,[])` and `TInst(String,[])`, indicating that it is indeed aware of the concrete type parameters of `MyType`. The macro logic could use this information to generate a custom type (using `haxe.macro.Context.defineType`) or refer to an existing one. For brevity we return `null` here which asks the compiler to infer (3.6) the type.

In Haxe 3.1 the return type of a `@:genericBuild` macro has to be a `haxe.macro.Type`. Haxe 3.2 allows (and prefers) returning a `haxe.macro.ComplexType` instead, which is the syntactic representation of a type. This is easier to work with in many cases because types can simply be referenced by their paths.

Const type parameter Haxe allows passing constant expression (5.2) as a type parameter if the type parameter name is `Const`. This can be utilized in the context of `@:genericBuild` macros to pass information from the syntax directly to the macro:

```
import haxe.macro.Expr;
import haxe.macro.Context;
import haxe.macro.Type;

class GenericBuildMacro2 {
  static public function build() {
    switch (Context.getLocalType()) {
      case TInst(_,[TInst(_.get() => { kind: KExpr(macro $v{(s:String)
          }) },_)]):
        trace(s);
      case t:
        Context.error("Classが要求されています", Context.currentPos())
            ;
    }
    return null;
  }
}
```

```
@:genericBuild(GenericBuildMacro2.build())
class MyType<Const> { }

class Main {
  static function main() {
    var x:MyType<"myfile.txt">;
  }
}
```

Here the macro logic could load a file and use its contents to generate a custom type.

9.6 Limitations

9.6.1 Macro-in-Macro

9.6.2 Static extension

The concepts or static extensions (6.3) and macros are somewhat conflicting: While the former requires a known type in order to determine used functions, macros execute before typing on plain syntax. It is thus not surprising that combining these two features can lead to issues. Haxe 3.0 would try to convert the typed expression back to a syntax expression, which is not always possible and may lose important information. We recommend that it is used with caution.
【Haxe 3.1.0 から】

The combination of static extensions and macros was reworked for the 3.1.0 release. The Haxe Compiler does not even try to find the original expression for the macro argument and instead passes a special `@:this this` expression. While the structure of this expression conveys no information, the expression can still be typed correctly:

```
import haxe.macro.Context;
import haxe.macro.Expr;

using Main;
using haxe.macro.Tools;

class Main {
  static public function main() {
    #if !macro
    var a = "foo";
    a.test();
    #end
  }

  macro static function test(e:ExprOf<String>) {
    trace(e.toString()); // @:this this
    // TInst(String,[])
    trace(Context.typeof(e));
    return e;
  }
}
```

9.6.3 Build Order

The build order of types is unspecified and this extends to the execution order of build-macros (9.5). While certain rules can be determined, we strongly recommend to not rely on the execution order of build-macros. If type building requires multiple passes, this should be reflected directly in the macro code. In order to avoid multiple build-macro execution on the same type, state can be stored in static variables or added as metadata (6.9) to the type in question:

```
1  import haxe.macro.Context;
2  import haxe.macro.Expr;
3
4  #if !macro
5  @:autoBuild(MyMacro.build())
6  #end
7  interface I1 { }
8
9  #if !macro
10 @:autoBuild(MyMacro.build())
11 #end
12 interface I2 { }
13
14 class C implements I1 implements I2 { }
15
16 class MyMacro {
17   macro static public function build():Array<Field> {
18     var c = Context.getLocalClass().get();
19     if (c.meta.has(":processed")) return null;
20     c.meta.add(":processed",[],c.pos);
21     // process here
22     return null;
23   }
24 }
25
26 class Main {
27   static public function main() { }
28 }
```

With both interfaces `I1` and `I2` having `:autoBuild` metadata, the build macro is executed twice for class `C`. We guard against duplicate processing by adding a custom `:processed` metadata to the class, which can be checked during the second macro execution.

9.6.4 Type Parameters

9.7 Initialization macros

Initialization macros are invoked from command line by using the `--macro callExpr(args)` command. This registers a callback which the compiler invokes after creating its context, but before typing what was argument to `-main`. This then allows configuring the compiler in some ways.

If the argument to `--macro` is a call to a plain identifier, that identifier is looked up in the class `haxe.macro.Compiler` which is part of the Haxe Standard Library. It comes with

several useful initialization macros which are detailed in its API.

As an example, the `include` macro allows inclusion of an entire package for compilation, recursively if necessary. The command line argument for this would then be `--macro include('some.pack', true)`.

Of course it is also possible to define custom initialization macros to perform various tasks before the real compilation. A macro like this would then be invoked via `--macro some.Class.theMacro(args)`. For instance, as all macros share the same context (9.1), an initialization macro could set the value of a static field which other macros use as configuration.

パート III

標準ライブラリ

章 10

Standard Library

10.1 String

Type: String
A String is a sequence of characters.

Character code Use the `.code` property on a constant single-char string in order to compile its ASCII character code:

```
"#".code // will compile as 35
```

See the String API for details about its methods.

10.2 Data Structures

10.2.1 Array

An `Array` is a collection of elements. It has one type parameter (3.2) which corresponds to the type of these elements. Arrays can be created in three ways:

1. By using their constructor: `new Array()`
2. By using array declaration syntax (5.5): `[1, 2, 3]`
3. By using array comprehension (6.6): `[for (i in 0...10) if (i % 2 == 0) i]`

Arrays come with an API to cover most use-cases. Additionally they allow read and write array access (5.8):

```
class Main {
  static public function main() {
    var a = [1, 2, 3];
    trace(a[1]); // 2
    a[1] = 1;
    trace(a[1]); // 1
  }
}
```

Since array access in Haxe is unbounded, i.e. it is guaranteed to not throw an exception, this requires further discussion:

- If a read access is made on a non-existing index, a target-dependent value is returned.
- If a write access is made with a positive index which is out of bounds, `null` (or the default value (2.2) for basic types (2.1) on static targets (2.2)) is inserted at all positions between the last defined index and the newly written one.
- If a write access is made with a negative index, the result is unspecified.

Arrays define an iterator (6.7) over their elements. This iteration is typically optimized by the compiler to use a `while` loop (5.14) with array index:

```
class Main {
  static public function main() {
    var scores = [110, 170, 35];
    var sum = 0;
    for (score in scores) {
      sum += score;
    }
    trace(sum); // 315
  }
}
```

Haxe generates this optimized JavaScript output:

```
Main.main = function() {
    var scores = [110,170,35];
    var sum = 0;
    var _g = 0;
    while(_g < scores.length) {
        var score = scores[_g];
        ++_g;
        sum += score;
    }
    console.log(sum);
};
```

Haxe does not allow arrays of mixed types unless the parameter type is forced to `Dynamic` (2.7):

```
class Main {
  static public function main() {
    // コンパイルエラー:
    // Arrays of mixed types are only allowed if the type is  forced to Array<Dynamic>
    // (混ざった型の配列は、型がArray<Dynamic>に強制されている場合のみ許可されます。)
    //var myArray = [10, "Bob", false];

    // Array<Dynamic>で混ざった型の配列
    var myExplicitArray:Array<Dynamic> = [10, "Sally", true];
  }
}
```

トリビア: Dynamic Arrays
In Haxe 2, mixed type array declarations were allowed. In Haxe 3, arrays can have mixed types only if they are explicitly declared as `Array<Dynamic>`.

See the Array API for details about its methods.

10.2.2 Vector

A `Vector` is an optimized fixed-length collection of elements. Much like Array (10.2.1), it has one type parameter (3.2) and all elements of a vector must be of the specified type, it can be iterated over using a for loop (5.13) and accessed using array access syntax (2.8.3). However, unlike `Array` and `List`, vector length is specified on creation and cannot be changed later.

```
1  class Main {
2    static function main() {
3      var vec = new haxe.ds.Vector(10);
4
5      for (i in 0...vec.length) {
6        vec[i] = i;
7      }
8
9      trace(vec[0]); // 0
10     trace(vec[5]); // 5
11     trace(vec[9]); // 9
12   }
13 }
```

`haxe.ds.Vector` is implemented as an abstract type (2.8) over a native array implementation for given target and can be faster for fixed-size collections, because the memory for storing its elements is pre-allocated.

See the Vector API for details about the vector methods.

10.2.3 List

A `List` is a collection for storing elements. On the surface, a list is similar to an Array (節 10.2.1). However, the underlying implementation is very different. This results in several functional differences:

1. A list can not be indexed using square brackets, i.e. `[0]`.
2. A list can not be initialized.
3. There are no list comprehensions.
4. A list can freely modify/add/remove elements while iterating over them.

A simple example for working with lists:

```
1  class Main {
2    static public function main() {
3      var myList = new List<Int>();
4      for (ii in 0...5)
```

```
      myList.add(ii);
    trace(myList); //{0, 1, 2, 3, 4}
  }
}
```

See the List API for details about the list methods.

10.2.4 GenericStack

A `GenericStack`, like `Array` and `List` is a container for storing elements. It has one type parameter (3.2) and all elements of the stack must be of the specified type. Here is a small example program for initializing and working with a `GenericStack`.

```
import haxe.ds.GenericStack;

class Main {
  static public function main() {
    var myStack = new GenericStack<Int>();
    for (ii in 0...5)
      myStack.add(ii);
    trace(myStack); //{4, 3, 2, 1, 0}
    trace(myStack.pop()); //4
  }
}
```

> トリビア: FastList
> In Haxe 2, the GenericStack class was known as FastList. Since its behavior more closely resembled a typical stack, the name was changed for Haxe 3.

The Generic in `GenericStack` is literal. It is attributed with the `:generic` metadata. Depending on the target, this can lead to improved performance on static targets. See ジェネリック (節 3.3) for more details.

See the GenericStack API for details about its methods.

10.2.5 Map

A `Map` is a container composed of key, value pairs. A `Map` is also commonly referred to as an associative array, dictionary, or symbol table. The following code gives a short example of working with maps:

```
class Main {
  static public function main() {
    // Mapは配列と似た形で初期化するが、'=>'演算子を使う。
    // Mapでは、キーと値の型を明示的に宣言しても良い。
    var map1:Map<Int, String> =
      [1 => "one", 2=>"two"];

    // キーと値の型を推論させても良い。
    var map2 = [
      "one"=>1,
      "two"=>2,
```

```
12        "three"=>3
13      ];
14      $type(map2); // Map<String, Int>
15
16      // キーは重複してはいけない。
17      // エラー: Duplicate Key
18      //var map3 = [1=>"dog", 1=>"cat"];
19
20      // マップの値は配列アクセスの構文を使って取得できる。
21      var map4 = ["M"=>"Monday", "T"=>"Tuesday"];
22      trace(map4["M"]); //Monday
23
24      // Mapはデフォルトでその値を反復処理できる。
25      var valueSum;
26      for (value in map4) {
27        trace(value); // Monday ¥n Tuesday
28      }
29
30      // keys()メソッドを使ってそのキーを反復処理できる。
31      for (key in map4.keys()) {
32        trace(key); // M ¥n T
33      }
34
35      // 配列と同様に内包表記で初期化できる。
36      var map5 = [
37        for (key in map4.keys())
38          key => "FRIDAY!!"
39      ];
40      // {M => FRIDAY!!, T => FRIDAY!!}
41      trace(map5);
42    }
43  }
```

Under the hood, a `Map` is an abstract (2.8) type. At compile time, it gets converted to one of several specialized types depending on the key type:

- `String`: `haxe.ds.StringMap`
- `Int`: `haxe.ds.IntMap`
- `EnumValue`: `haxe.ds.EnumValueMap`
- `{}`: `haxe.ds.ObjectMap`

The `Map` type does not exist at runtime and has been replaced with one of the above objects.

Map defines array access (2.8.3) using its key type.

See the Map API for details of its methods.

10.2.6 Option

An Option is an enum (2.4) in the Haxe Standard Library which is defined like so:

```
enum Option<T> {
    Some(v:T);
    None;
}
```

It can be used in various situations, such as communicating whether or not a method had a valid return and if so, what value it returned:

```
import haxe.ds.Option;

class Main {
  static public function main() {
    var result = trySomething();
    switch (result) {
      case None:
        trace("Noneを取得");
      case Some(s):
        trace("値を取得: " +s);
    }
  }

  static function trySomething():Option<String> {
    if (Math.random() > 0.5) {
      return None;
    } else {
      return Some("Success");
    }
  }
}
```

10.3 Regular Expressions

Haxe has built-in support for regular expressions[1]. They can be used to verify the format of a string, transform a string or extract some regular data from a given text.

Haxe has special syntax for creating regular expressions. We can create a regular expression object by typing it between the ~/ combination and a single / character:

```
var r = ~/haxe/i;
```

Alternatively, we can create regular expression with regular syntax:

```
var r = new EReg("haxe", "i");
```

First argument is a string with regular expression pattern, second one is a string with flags (see below).

We can use standard regular expression patterns such as:

- . any character
- * repeat zero-or-more
- + repeat one-or-more

[1] http://en.wikipedia.org/wiki/Regular_expression

- `?` optional zero-or-one
- `[A-Z0-9]` character ranges
- `[^¥r¥n¥t]` character not-in-range
- `(...)` parenthesis to match groups of characters
- `^` beginning of the string (beginning of a line in multiline matching mode)
- `$` end of the string (end of a line in multiline matching mode)
- `|` "OR" statement.

For example, the following regular expression matches valid email addresses:

```
~/[A-Z0-9._¥%-]+@[A-Z0-9.-]+¥.[A-Z][A-Z][A-Z]?/i;
```

Please notice that the `i` at the end of the regular expression is a flag that enables case-insensitive matching.

The possible flags are the following:

- `i` case insensitive matching
- `g` global replace or split, see below
- `m` multiline matching, `^` and `$` represent the beginning and end of a line
- `s` the dot `.` will also match newlines (Neko, C++, PHP, Flash and Java targets only)
- `u` use UTF-8 matching (Neko and C++ targets only)

See the EReg API for details about its methods.

10.3.1 Matching

Probably one of the most common uses for regular expressions is checking whether a string matches the specific pattern. The `match` method of a regular expression object can be used to do that:

```
class Main {
  static function main() {
    var r = ~/world/;
    var str = "hello world";
    // true : 'world'が文字列に含まれています。
    trace(r.match(str));
    trace(r.match("hello !")); // false
  }
}
```

10.3.2 Groups

Specific information can be extracted from a matched string by using groups. If `match()` returns true, we can get groups using the `matched(X)` method, where X is the number of a group defined by regular expression pattern:

```
class Main {
  static function main() {
    var str = "Nicolas is 26 years old";
    var r =
      ~/([A-Za-z]+) is ([0-9]+) years old/;
    r.match(str);
    trace(r.matched(1)); // "Nicolas"
    trace(r.matched(2)); // "26"
  }
}
```

Note that group numbers start with 1 and `r.matched(0)` will always return the whole matched substring.

The `r.matchedPos()` will return the position of this substring in the original string:

```
class Main {
  static function main() {
    var str = "abcdeeeeefghi";
    var r = ~/e+/;
    r.match(str);
    trace(r.matched(0)); // "eeeee"
    // { pos : 4, len : 5 }
    trace(r.matchedPos());
  }
}
```

Additionally, `r.matchedLeft()` and `r.matchedRight()` can be used to get substrings to the left and to the right of the matched substring:

```
class Main {
  static function main() {
    var r = ~/b/;
    r.match("abc");
    trace(r.matchedLeft()); // a
    trace(r.matched(0)); // b
    trace(r.matchedRight()); // c
  }
}
```

10.3.3 Replace

A regular expression can also be used to replace a part of the string:

```
class Main {
  static function main() {
    var str = "aaabcbcbcbz";
    // g : インスタンスのすべてを置換する。
    var r = ~/b[^c]/g;
    // "aaabcbcbcxx"
    trace(r.replace(str,"xx"));
  }
}
```

We can use `$X` to reuse a matched group in the replacement:

```
class Main {
  static function main() {
    var str = "{hello} {0} {again}";
    var r = ~/{([a-z]+)}/g;
    // "*hello* {0} *again*"
    trace(r.replace(str,"*$1*"));
  }
}
```

10.3.4 Split

A regular expression can also be used to split a string into several substrings:

```
class Main {
  static function main() {
    var str = "XaaaYababZbbbW";
    var r = ~/[ab]+/g;
    // ["X","Y","Z","W"]
    trace(r.split(str));
  }
}
```

10.3.5 Map

The `map` method of a regular expression object can be used to replace matched substrings using a custom function. This function takes a regular expression object as its first argument so we may use it to get additional information about the match being done and do conditional replacement. For example:

```
class Main {
  static function main() {
    var r = ~/(dog|fox)/g;
    var s = "The quick brown fox jumped over the lazy dog.";
    var s2 = r.map(s, function(r) {
        var match = r.matched(0);
        switch (match) {
            case 'dog': return 'fox';
            case 'fox': return 'dog';
            default: throw '知らない動物です: $match';
        };
    });
    trace(s2); // The quick brown dog jumped over the lazy fox.
  }
}
```

10.3.6 Implementation Details

Regular Expressions are implemented:

- in JavaScript, the runtime is providing the implementation with the object RegExp.

- in Neko and C++, the PCRE library is used
- in Flash, PHP, C# and Java, native implementations are used
- in Flash 6/8, the implementation is not available

10.4 Math

Haxe includes a floating point math library for some common mathematical operations. Most of the functions operate on and return `floats`. However, an `Int` can be used where a `Float` is expected, and Haxe also converts `Int` to `Float` during most numeric operations (see 数値の演算子 (節 2.1.3) for more details).

Here are some example uses of the math library:

```
1  class Main {
2    static public function main() {
3      var x = 1/2;
4      var y = 20.2;
5      var z = -2;
6
7      trace(Math.abs(z)); //2
8      trace(Math.sin(x*Math.PI)); //1
9      trace(Math.ceil(y)); //21
10
11     // logは自然対数。
12     trace(Math.log(Math.exp(5))); //5
13
14     // nekoターゲットの場合の出力。
15     // 以下の出力はプラットフォームによって変わりえる。
16     trace(1/0); //inf
17     trace(-1/0); //-inf
18     trace(Math.sqrt(-1)); //nan
19   }
20 }
```

See the Math API for all available functions.

10.4.1 Special Numbers

The math library has definitions for several special numbers:

- NaN (Not a Number): returned when a mathmatically incorrect operation is executed, e.g. Math.sqrt(-1)
- POSITIVE_INFINITY: e.g. divide a positive number by zero
- NEGATIVE_INFINITY: e.g. divide a negative number by zero
- PI : 3.1415...

10.4.2 Mathematical Errors

Although neko can fluidly handle mathematical errors, like division by zero, this is not true for all targets. Depending on the target, mathematical errors may produce exceptions and ultimately errors.

10.4.3 Integer Math

If you are targeting a platform that can utilize integer operations, e.g. integer division, it should be wrapped in Std.int() for improved performance. The Haxe Compiler can then optimize for integer operations. An example:

```
1     var intDivision = Std.int(6.2/4.7);
```

I think C++ can use integer operatins, but I don't know about any other targets. Only saw this mentioned in an old discussion thread, still true?

10.4.4 Extensions

It is common to see 静的拡張 (節 6.3) used with the math library. This code shows a simple example:

```
1 class MathStaticExtension {
2   /* ラジアンから度数への角度の変換。 */
3   inline public static function toDegrees(radians:Float):Float {
4     return radians * 180 / Math.PI;
5   }
6 }
```

```
1 using MathStaticExtension;
2
3 class Main {
4   public static function main() {
5     var ang = 1/2*Math.PI;
6     trace(ang.toDegrees()); //90
7   }
8 }
```

10.5 Lambda

定義: Lambda
Lambda is a functional language concept within Haxe that allows you to apply a function to a list or iterators (6.7). The Lambda class is a collection of functional methods in order to use functional-style programming with Haxe.

It is ideally used with `using Lambda` (see Static Extension (6.3)) and then acts as an extension to `Iterable` types.

On static platforms, working with the `Iterable` structure might be slower than performing the operations directly on known types, such as `Array` and `List`.

Lambda Functions The Lambda class allows us to operate on an entire `Iterable` at once. This is often preferable to looping routines since it is less error prone and easier to read. For convenience, the `Array` and `List` class contains some of the frequently used methods from the Lambda class.

It is helpful to look at an example. The exists function is specified as:

```
1 static function exists<A>( it : Iterable<A>, f : A -> Bool ) : Bool
```

Most Lambda functions are called in similar ways. The first argument for all of the Lambda functions is the `Iterable` on which to operate. Many also take a function as an argument.

`Lambda.array`, `Lambda.list` Convert Iterable to `Array` or `List`. It always returns a new instance.

`Lambda.count` Count the number of elements. If the Iterable is a `Array` or `List` it is faster to use its length property.

`Lambda.empty` Determine if the Iterable is empty. For all Iterables it is best to use the this function; it's also faster than compare the length (or result of Lambda.count) to zero.

`Lambda.has` Determine if the specified element is in the Iterable.

`Lambda.exists` Determine if criteria is satisfied by an element.

`Lambda.indexOf` Find out the index of the specified element.

`Lambda.find` Find first element of given search function.

`Lambda.foreach` Determine if every element satisfies a criteria.

`Lambda.iter` Call a function for each element.

`Lambda.concat` Merge two Iterables, returning a new List.

`Lambda.filter` Find the elements that satisfy a criteria, returning a new List.

`Lambda.map`, `Lambda.mapi` Apply a conversion to each element, returning a new List.

`Lambda.fold` Functional fold, which is also known as reduce, accumulate, compress or inject.

This example demonstrates the Lambda filter and map on a set of strings:

```
1  using Lambda;
2  class Main {
3      static function main() {
4          var words = ['car', 'boat', 'cat', 'frog'];
5
6          var isThreeLetters = function(word) return word.length == 3;
7          var capitalize = function(word) return word.toUpperCase();
8
9          // Three letter words and capitalized.
10         trace(words.filter(isThreeLetters).map(capitalize)); // [CAR,
               CAT]
11     }
12 }
```

This example demonstrates the Lambda count, has, foreach and fold function on a set of ints.

```
1  using Lambda;
2  class Main {
3      static function main() {
4          var numbers = [1, 3, 5, 6, 7, 8];
5
```

```
        trace(numbers.count()); // 6
        trace(numbers.has(4)); // false

        // test if all numbers are greater/smaller than 20
        trace(numbers.foreach(function(v) return v < 20)); // true
        trace(numbers.foreach(function(v) return v > 20)); // false

        // sum all the numbers
        var sum = function(num, total) return total += num;
        trace(numbers.fold(sum, 0)); // 30
    }
}
```

See the Lambda API for all available functions.

10.6 Template

Haxe comes with a standard template system with an easy to use syntax which is interpreted by a lightweight class called haxe.Template.

A template is a string or a file that is used to produce any kind of string output depending on the input. Here is a small template example:

```
class Main {
  static function main() {
    var sample = "My name is <strong>::name::</strong>, <em>::age::</
        em> years old";
    var user = {name:"Mark", age:30};
    var template = new haxe.Template(sample);
    var output = template.execute(user);
    trace(output);
  }
}
```

The console will trace `My name is Mark, 30 years old`.

Expressions An expression can be put between the `::`, the syntax allows the current possibilities:

`::name::` the variable name

`::expr.field::` field access

`::(expr)::` the expression expr is evaluated

`::(e1 op e2)::` the operation op is applied to e1 and e2

`::(135)::` the integer 135. Float constants are not allowed

Conditions It is possible to test conditions using `::if flag1::`. Optionally, the condition may be followed by `::elseif flag2::` or `::else::`. Close the condition with `::end::`.

```
::if isValid:: valid ::else:: invalid ::end::
```

Operators can be used but they don't deal with operator precedence. Therefore it is required to enclose each operation in parentheses (). Currently, the following operators are allowed: +, -, *, /, >, <, >=, <=, ==, !=, && and ||.

For example `::((1 + 3) == (2 + 2))::` will display true.

```
::if (points == 10):: Great! ::end::
```

To compare to a string, use double quotes " in the template.

```
::if (name == "Mark"):: Hi Mark ::end::
```

Iterating Iterate on a structure by using `::foreach::`. End the loop with `::end::`.

```
<table>
    <tr>
        <th>Name</th>
        <th>Age</th>
    </tr>
    ::foreach users::
        <tr>
            <td>::name::</td>
            <td>::age::</td>
        </tr>
    ::end::
</table>
```

Sub-templates To include templates in other templates, pass the sub-template result string as a parameter.

```
var users = [{name:"Mark", age:30}, {name:"John", age:45}];

var userTemplate = new haxe.Template("::foreach users:: ::name::(::age
    ::) ::end::");
var userOutput = userTemplate.execute({users: users});

var template = new haxe.Template("The users are ::users::");
var output = template.execute({users: userOutput});
trace(output);
```

The console will trace `The users are Mark(30) John(45)`.

Template macros To call custom functions while parts of the template are being rendered, provide a `macros` object to the argument of Template.execute. The key will act as the template variable name, the value refers to a callback function that should return a `String`. The first argument of this macro function is always a `resolve()` method, followed by the given arguments. The resolve function can be called to retrieve values from the template context. If `macros` has no such field, the result is unspecified.

The following example passes itself as macro function context and executes `display` from the template.

```
class Main {
  static function main() {
    new Main();
```

```
  }

  public function new() {
    var user = {name:"Mark", distance:3500};
    var sample = "The results: $$display(::user::,::time::)";
    var template = new haxe.Template(sample);
    var output = template.execute({user:user, time: 15}, this);
    trace(output);
  }

  function display(resolve:String->Dynamic, user:User, time:Int) {
    return user.name + " ran " + (user.distance/1000) + " kilometers
        in " + time + " minutes";
  }
}
typedef User = {name:String, distance:Int}
```

The console will trace `The results: Mark ran 3.5 kilometers in 15 minutes`.

Globals Use the Template.globals object to store values that should be applied across all `haxe.Template` instances. This has lower priority than the context argument of `Template.execute`.

Using resources To separate the content from the code, consider using the resource embedding system (8.4). Place the template-content in a new file called `sample.mtt`, add `-resource sample.mtt@my_sample` to the compiler arguments and retrieve the content using `haxe.Resource.getString`.

```
class Main {
  static function main() {
    var sample = haxe.Resource.getString("my_sample");
    var user = {name:"Mark", age:30};
    var template = new haxe.Template(sample);
    var output = template.execute(user);
    trace(output);
  }
}
```

When running the template system on the server side, you can simply use `neko.Lib.print` or `php.Lib.print` instead of trace to display the HTML template to the user.

See the Template API for details about its methods.

10.7 Reflection

Haxe supports runtime reflection of types and fields. Special care has to be taken here because runtime representation generally varies between targets. In order to use reflection correctly it is necessary to understand what kind of operations are supported and what is not. Given the dynamic nature of reflection, this can not always be determined at compile-time.

The reflection API consists of two classes:

Reflect: A lightweight API which work best on anonymous structures (2.5), with limited support for classes (2.3).

Type: A more robust API for working with classes and enums (2.4).

The available methods are detailed in the API for Reflect and Type.

Reflection can be a powerful tool, but it is important to understand why it can also cause problems. As an example, several functions expect a String (10.1) argument and try to resolve it to a type or field. This is vulnerable to typing errors:

```
class Main {
  static function main() {
    trace(Type.resolveClass("Mian")); // null
  }
}
```

However, even if there are no typing errors it is easy to come across unexpected behavior:

```
class Main {
  static function main() {
    // null
    trace(Type.resolveClass("haxe.Template"));
  }
}
```

The problem here is that the compiler never actually "sees" the type `haxe.Template`, so it does not compile it into the output. Furthermore, even if it were to see the type there could be issues arising from dead code elimitation (8.2) eliminating types or fields which are only used via reflection.

Another set of problems comes from the fact that, by design, several reflection functions expect arguments of type Dynamic (2.7), meaning the compiler cannot check if the passed in arguments are correct. The following example demonstrates a common mistake when working with `callMethod`:

```
class Main {
  static function main() {
    // wrong
    //Reflect.callMethod(Main, "f", []);
    // right
    Reflect.callMethod(Main,
      Reflect.field(Main, "f"), []);
  }

  static function f() {
    trace('Called');
  }
}
```

The commented out call would be accepted by the compiler because it assigns the string `"f"` to the function argument `func` which is specified to be `Dynamic`.

A good advice when working with reflection is to wrap it in a few functions within an application or API which are called by otherwise type-safe code. An example could look like this:

```
typedef MyStructure = {
  name: String,
  score: Int
}

class Main {
  static function main() {
    var data = reflective();
    //この時点ではdataはちゃんとMyStructureとして型付けされている
  }

  static function reflective():MyStructure {
    // Work with reflection here to get some values we want to return.
    return {
      name: "Reflection",
      score: 0
    }
  }
}
```

While the method `reflective` could interally work with reflection (and `Dynamic` for that matter) a lot, its return value is a typed structure which the callers can use in a type-safe manner.

10.8 Serialization

Many runtime values can be serialized and deserialized using the haxe.Serializer and haxe.Unserializer classes. Both support two usages:

1. Create an instance and continuously call the `serialize`/`unserialize` method to handle multiple values.
2. Call their static `run` method to serialize/deserialize a single value.

The following example demonstrates the first usage:

```
import haxe.Serializer;
import haxe.Unserializer;

class Main {
  static function main() {
    var serializer = new Serializer();
    serializer.serialize("foo");
    serializer.serialize(12);
    var s = serializer.toString();
    trace(s); // y3:fooi12

    var unserializer = new Unserializer(s);
    trace(unserializer.unserialize()); // foo
    trace(unserializer.unserialize()); // 12
  }
}
```

The result of the serialization (here stored in local variable `s`) is a String (10.1) and can be passed around at will, even remotely. Its format is described in Serialization format (節 10.8.1).

Supported values

- `null`
- `Bool`, `Int` and `Float` (including infinities and `NaN`)
- `String`
- `Date`
- `haxe.io.Bytes` (encoded as base64)
- `Array` (10.2.1) and `List` (10.2.3)
- `haxe.ds.StringMap`, `haxe.ds.IntMap` and `haxe.ds.ObjectMap`
- anonymous structures (2.5)
- Haxe class instances (2.3) (not native ones)
- enum instances (2.4)

Serialization configuration Serialization can be configured in two ways. For both a static variable can be set to influence all haxe.Serializer instances, and a member variable can be set to only influence a specific instance:

`USE_CACHE`, `useCache`: If true, repeated structures or class/enum instances are serialized by reference. This can avoid infinite loops for recursive data at the expense of longer serialization time. By default, object caching is disabled; strings however are always cached.

`USE_ENUM_INDEX`, `useEnumIndex`: If true, enum constructors are serialized by their index instead of their name. This can make the resulting string shorter, but breaks if enum constructors are inserted into the type before deserialization. This behavior is disabled by default.

Deserialization behavior If the serialization result is stored and later used for deserialization, care has to be taken to maintain compatibility when working with class and enum instances. It is then important to understand exactly how unserialization is implemented.

- The type has to be available in the runtime where the deserialization is made. If dead code elimination (8.2) is active, a type which is used only through serialization might be removed.
- Each `Unserializer` has a member variable `resolver` which is used to resolve classes and enums by name. Upon creation of the `Unserializer` this is set to `Unserializer.DEFAULT_RESOLVER`. Both that and the instance member can be set to a custom resolver.
- Classes are resolved by name using `resolver.resolveClass(name)`. The instance is then created using `Type.createEmptyInstance`, which means that the class constructor is not called. Finally, the instance fields are set according to the serialized value.

- Enums are resolved by name using `resolver.resolveEnum(name)`. The enum instance is then created using `Type.createEnum`, using the serialized argument values if available. If the constructor arguments were changed since serialization, the result is unspecified.

Custom (de)serialization If a class defines the member method `hxSerialize`, that method is called by the serializer and allows custom serialization of the class. Likewise, if a class defines the member method `hxUnserialize` it is called by the deserializer:

```
1  import haxe.Serializer;
2  import haxe.Unserializer;
3
4  class Main {
5
6    var x:Int;
7    var y:Int;
8
9    static function main() {
10     var s = Serializer.run(new Main(1, 2));
11     var c:Main = Unserializer.run(s);
12     trace(c.x); // 1
13     trace(c.y); // -1
14   }
15
16   function new(x, y) {
17     this.x = x;
18     this.y = y;
19   }
20
21   @:keep
22   function hxSerialize(s:Serializer) {
23     s.serialize(x);
24   }
25
26   @:keep
27   function hxUnserialize(u:Unserializer) {
28     x = u.unserialize();
29     y = -1;
30   }
31 }
```

In this example we decide that we want to ignore the value of member variable `y` and do not serialize it. Instead we default it to `-1` in `hxUnserialize`. Both methods are annotated with the `@:keep` metadata to prevent dead code elimination (8.2) from removing them as they are never properly referenced in the code.

See Serializer and Unserializer API documentation for details.

10.8.1 Serialization format

Each supported value is translated to a distinct prefix character, followed by the necessary data.

`null: n`

Int: z for zero, or i followed by the integer display (e.g. i456)

Float:

NaN: k

negative infinity: m

positive infinity: p

finite floats: d followed by the float display (e.g. d1.45e-8)

Bool: t for true, f for false

String: y followed by the url encoded string length, then : and the url encoded string (e.g. y10:hi%20there for "hi there".

name-value pairs: a serialized string representing the name followed by the serialized value

structure: o followed by the list of name-value pairs and terminated by g (e.g. oy1:xi2y1:kng for {x:2, k:null})

List: l followed by the list of serialized items, followed by h (e.g. lnnh for a list of two null values)

Array: a followed by the list of serialized items, followed by h. For multiple consecutive null values, u followed by the number of null values is used (e.g. ai1i2u4i7ni9h for [1,2,null,null,null,null,7,null,9])

Date: v followed by the date itself (e.g. v2010-01-01 12:45:10)

haxe.ds.StringMap: b followed by the name-value pairs, followed by h (e.g. by1:xi2y1:knh for {"x" => 2, "k" => null})

haxe.ds.IntMap: q followed by the key-value pairs, followed by h. Each key is represented as :<int> (e.g. q:4n:5i45:6i7h for {4 => null, 5 => 45, 6 => 7})

haxe.ds.ObjectMap: M followed by serialized value pairs representing the key and value, followed by h

haxe.io.Bytes: s followed by the length of the base64 encoded bytes, then : and the byte representation using the codes A-Za-z0-9% (e.g. s3:AAA for 2 bytes equal to 0, and s10:SGVsbG8gIQ for haxe.io.Bytes.ofString("Hello !"))

exception: x followed by the exception value

class instance: c followed by the serialized class name, followed by the name-value pairs of the fields, followed by g (e.g. cy5:Pointy1:xzy1:yzg for new Point(0, 0) (having two integer fields x and y)

enum instance (by name): w followed by the serialized enum name, followed by the serialized constructor name, followed by :, followed by the number of arguments, followed by the argument values (e.g. wy3:Fooy1:A: 0 for Foo.A (with no arguments), wy3:Fooy1:B:2i4n for Foo.B(4,null))

enum instance (by index): j followed by the serialized enum name, followed by :, followed by the constructor index (starting from 0), followed by :, followed by the number of arguments, followed by the argument values (e.g. wy3:Foo: 0:0 for Foo.A (with no arguments), wy3:Foo:1:2i4n for Foo.B(4,null))

cache references:

`String`: `R` followed by the corresponding index in the string cache (e.g. `R456`)

class, enum or structure `r` followed by the corresponding index in the object cache (e.g. `r42`)

custom: `C` followed by the class name, followed by the custom serialized data, followed by `g`

Cached elements and enum constructors are indexed from zero.

10.9 Xml

Haxe provides built-in support for working with XML[2] data via the haxe.Xml class.

10.9.1 Getting started with Xml

Creating a root element A `Xml` root element can be created using the `Xml.createElement` method.

```
1 var root = Xml.createElement('root');
2 trace(root); // <root />
```

An root node element can also be created by parsing a `String` containing the XML data.

```
1 var root = Xml.parse('<root />').firstElement();
2 trace(root); // <root />
```

Creating child elements Adding child elements to the root can be done using the `addChild` method.

```
1 var child:Xml = Xml.createElement('child');
2 root.addChild(child);
3 trace(root); // <root><child/></root>
```

Adding attributes to an element can be done by using the `set()` method.

```
1 child.set('name', 'John');
2 trace(root); // <root><child name="John"/></root>
```

Accessing elements and values This code parses an XML string into an object structure `Xml` and then accesses properties of the object.

```
1 var xmlString = '<hello name="world!">Haxe is great!</hello>';
2 var xml:Xml = Xml.parse(xmlString).firstElement();
3
4 trace(xml.nodeName); // hello
5 trace(xml.get('name')); // world!
6 trace(xml.firstChild().nodeValue); // Haxe is great!
```

The difference between `firstChild` and `firstElement` is that the second function will return the first child with the type `Xml.Element`.

[2] http://en.wikipedia.org/wiki/XML

Iterate on Xml elements We can as well use other methods to iterate either over children or elements.

```
for (child in xml) {
    // iterate on all children.
}
for (elt in xml.elements()) {
    // iterate on all elements.
}
for (user in xml.elementsNamed("user")) {
    // iterate on all elements with a nodeName "user".
}
for (att in xml.attributes()) {
    // iterator on all attributes.
}
```

See Xml API documentation for details about its methods.

10.9.2 Parsing Xml

The static method Xml.parse can be used to parse XML data and obtain a Haxe value from it.

```
var xml = Xml.parse('<root>Haxe is great!</root>').firstElement();
trace(xml.firstChild().nodeValue);
```

10.9.3 Encoding Xml

The method xml.toString() can be used to obtain the `String` representation.

```
var xml = Xml.createElement('root');
xml.addChild(Xml.createElement('child1'));
xml.addChild(Xml.createElement('child2'));

trace(xml.toString()); // <root><child1/><child2/></root>
```

10.10 Json

Haxe provides built-in support for (de-)serializing JSON[3] data via the haxe.Json class.

10.10.1 Parsing JSON

Use the haxe.Json.parse static method to parse JSON data and obtain a Haxe value from it:

```
class Main {
  static function main() {
    var s = '{"rating": 5}';
    var o = haxe.Json.parse(s);
    trace(o); // { rating: 5 }
```

[3] http://en.wikipedia.org/wiki/JSON

```
  }
}
```

Note that the type of the object returned by `haxe.Json.parse` is `Dynamic`, so if the structure of our data is well-known, we may want to specify a type using anonymous structures (2.5). This way we provide compile-time checks for accessing our data and most likely more optimal code generation, because compiler knows about types in a structure:

```
typedef MyData = {
  var name:String;
  var tags:Array<String>;
}

class Main {
  static function main() {
    var s = '{
      "name": "Haxe",
      "tags": ["awesome"]
    }';
    var o:MyData = haxe.Json.parse(s);
    trace(o.name); // Haxe (文字列)
    // awesome (配列の中の文字列)
    trace(o.tags[0]);
  }
}
```

10.10.2 Encoding JSON

Use the haxe.Json.stringify static method to encode a Haxe value into a JSON string:

```
class Main {
  static function main() {
    var o = {rating: 5};
    var s = haxe.Json.stringify(o);
    trace(s); // {"rating":5}
  }
}
```

10.10.3 Implementation details

The haxe.Json API automatically uses native implementation on targets where it is available, i.e. JavaScript, Flash and PHP and provides its own implementation for other targets.

Usage of Haxe own implementation can be forced with `-D haxeJSON` compiler argument. This will also provide serialization of enums (2.4) by their index, maps (10.2.5) with string keys and class instances.

Older browsers (Internet Explorer 7, for instance) may not have native JSON implementation. In case it's required to support them, we can include one of the JSON implementations available on the internet in the HTML page. Alternatively, a `-D old_browser` compiler argument that will make `haxe.Json` try to use native JSON and, in case it's not available, fallback to its own implementation.

10.11 Input/Output

10.12 Sys/sys

10.13 Remoting

Haxe remoting is a way to communicate between different platforms. With Haxe remoting, applications can transmit data transparently, send data and call methods between server and client side.

See the remoting package on the API documentation for more details on its classes.

10.13.1 Remoting Connection

In order to use remoting, there must be a connection established. There are two kinds of Haxe Remoting connections:

haxe.remoting.Connection is used for synchronous connections, where the results can be directly obtained when calling a method.

haxe.remoting.AsyncConnection is used for asynchronous connections, where the results are events that will happen later in the execution process.

Start a connection There are some target-specific constructors with different purposes that can be used to set up a connection:

All targets: `HttpAsyncConnection.urlConnect(url:String)` Returns an asynchronous connection to the given URL which should link to a Haxe server application.

Flash: `ExternalConnection.jsConnect(name:String, ctx:Context)` Allows a connection to the local JavaScript Haxe code. The JS Haxe code must be compiled with the class ExternalConnection included. This only works with Flash Player 8 and higher.

`AMFConnection.urlConnect(url:String)` and `AMFConnection.connect( cnx : NetConnection )` Allows a connection to an AMF Remoting server such as Flash Media Server or AMFPHP.

`SocketConnection.create(sock:flash.XMLSocket)` Allows remoting communications over an `XMLSocket`

`LocalConnection.connect(name:String)` Allows remoting communications over a Flash LocalConnection

JavaScript: `ExternalConnection.flashConnect(name:String, obj:String, ctx:Context)` Allows a connection to a given Flash Object. The Haxe Flash content must be loaded and it must include the `haxe.remoting.Connection` class. This only works with Flash 8 and higher.

Neko: `HttpConnection.urlConnect(url:String)` Will work like the asynchronous version but in synchronous mode.

`SocketConnection.create(...)` Allows real-time communications with a Flash client which is using an `XMLSocket` to connect to the server.

Remoting context Before communicating between platforms, a remoting context has to be defined. This is a shared API that can be called on the connection at the client code.

This server code example creates and shares an API:

```
class Server {
    function new() { }
    function foo(x, y) { return x + y; }

    static function main() {
        var ctx = new haxe.remoting.Context();
        ctx.addObject("Server", new Server());

        if(haxe.remoting.HttpConnection.handleRequest(ctx))
        {
            return;
        }

        // handle normal request
        trace("This is a remoting server !");
    }
}
```

Using the connection Using a connection is pretty convenient. Once the connection is obtained, use classic dot-access to evaluate a path and then use `call()` to call the method in the remoting context and get the result. The asynchronous connection takes an additional function parameter that will be called when the result is available.

This client code example connects to the server remoting context and calls a function `foo()` on its API.

```
class Client {
  static function main() {
    var cnx = haxe.remoting.HttpAsyncConnection.urlConnect("http://
        localhost/");
    cnx.setErrorHandler( function(err) trace('Error: $err'); } );
    cnx.Server.foo.call([1,2], function(data) trace('Result: $data');)
        ;
  }
}
```

To make this work for the Neko target, setup a Neko Web Server, point the url in the Client to `"http:// localhost2000/ remoting.n"` and compile the Server using `-main Server -neko remoting.n`.

Error handling

- When an error occurs in a asynchronous call, the error handler is called as seen in the example above.
- When an error occurs in a synchronous call, an exception is raised on the caller-side as if we were calling a local method.

Data serialization Haxe Remoting can send a lot of different kinds of data. See Serialization (10.8).

See the remoting package on the API documentation for more details on its classes.

10.13.2 Implementation details

JavaScript security specifics The html-page wrapping the js client must be served from the same domain as the one where the server is running. The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. The same-origin policy is used as a means to prevent some of the cross-site request forgery attacks.

To use the remoting across domain boundaries, CORS (cross-origin resource sharing) needs to be enabled by defining the header `X-Haxe-Remoting` in the `.htaccess`:

```
# Enable CORS
Header set Access-Control-Allow-Origin "*"
Header set Access-Control-Allow-Methods: "GET,POST,OPTIONS,DELETE,PUT"
Header set Access-Control-Allow-Headers: X-Haxe-Remoting
```

See same-origin policy for more information on this topic.

Also note that this means that the page can't be served directly from the file system `"file:///C:/example/path/index.html"`.

Flash security specifics When Flash accesses a server from a different domain, set up a `crossdomain.xml` file on the server, enabling the `X-Haxe` headers.

```
<cross-domain-policy>
    <allow-access-from domain="*"/> <!-- or the appropriate domains
        -->
    <allow-http-request-headers-from domain="*" headers="X-Haxe*"/>
</cross-domain-policy>
```

Arguments types are not ensured There is no guarantee of any kind that the arguments types will be respected when a method is called using remoting. That means even if the arguments of function `foo` are typed to `Int`, the client will still be able to use strings while calling the method. This can lead to security issues in some cases. When in doubt, check the argument type when the function is called by using the `Std.is` method.

10.14 Unit testing

The Haxe Standard Library provides basic unit testing classes from the haxe.unit package.

Creating new test cases First, create a new class extending haxe.unit.TestCase and add own test methods. Every test method name must start with `"test"`.

```
class MyTestCase extends haxe.unit.TestCase {
  public function testBasic() {
    assertEquals("A", "A");
  }
}
```

Running unit tests　To run the test, an instance of haxe.unit.TestRunner has to be created. Add the TestCase using the `add` method and call `run` to start the test.

```
class Main {
  static function main() {
    var r = new haxe.unit.TestRunner();
    r.add(new MyTestCase());
    // ここに他のTestCaseを足す。

    // 最後にテストを走らせる。
    r.run();
  }
}
```

The result of the test looks like this:

```
Class: MyTestCase
.
OK 1 tests, 0 failed, 1 success
```

Test functions　The `haxe.unit.TestCase` class comes with three test functions.

`assertEquals(a, b)` Succeeds if `a` and `b` are equal, where `a` is value tested and `b` is the expected value.

`assertTrue(a)` Succeeds if `a` is `true`

`assertFalse(a)` Succeeds if `a` is `false`

Setup and tear down　To run code before or after the test, override the functions `setup` and `tearDown` in the `TestCase`.

`setup` is called before each test runs.

`tearDown` is called once after all tests are run.

```
class MyTestCase extends haxe.unit.TestCase {
  var value:String;

  override public function setup() {
    value = "foo";
  }

  public function testSetup() {
    assertEquals("foo", value);
  }
}
```

Comparing Complex Objects With complex objects it can be difficult to generate expected values to compare to the actual ones. It can also be a problem that `assertEquals` doesn't do a deep comparison. One way around these issues is to use a string as the expected value and compare it to the actual value converted to a string using `Std.string`. Below is a trivial example using an array.

```
1 public function testArray() {
2   var actual = [1,2,3];
3   assertEquals("[1, 2, 3]", Std.string(actual));
4 }
```

See the haxe.unit package on the API documentation for more details.

パート IV

Miscellaneous

章 11

Haxelib

Haxelib is the library manager that comes with any Haxe distribution. Connected to a central repository, it allows submitting and retrieving libraries and has multiple features beyond that. Available libraries can be found at http://lib.haxe.org.

A basic Haxe library is a collection of `.hx` files. That is, libraries are distributed by source code by default, making it easy to inspect and modify their behavior. Each library is identified by a unique name, which is utilized when telling the Haxe Compiler which libraries to use for a given compilation.

11.1 Using a Haxe library with the Haxe Compiler

Any installed Haxe library can be made available to the compiler through the `-lib <library-name>` argument. This is very similiar to the `-cp <path>` argument, but expects a library name instead of a directory path. These commands are explained thoroughly in コンパイラの使い方 (章 7).

For our exemplary usage we chose a very simple Haxe library called “random”. It provides a set of static convenience methods to achieve various random effects, such as picking a random element from an array.

```
1 class Main {
2   static public function main() {
3     var elt = Random.fromArray([1, 2, 3]);
4     trace(elt);
5   }
6 }
```

Compiling this without any `-lib` argument causes an error message along the lines of `Unknown identifier : Random`. This shows that installed Haxe libraries are not available to the compiler by default unless they are explicitly added. A working command line for above program is `haxe -lib random -main Main --interp`.

If the compiler emits an error `Error: Library random is not installed : run 'haxelib install random'` the library has to be installed via the `haxelib` command first. As the error message suggests, this is achieved through `haxelib install random`. We will learn more about the `haxelib` command in Using Haxelib (節 11.4).

11.2 haxelib.json

Each Haxe library requires a `haxelib.json` file in which the following attributes are defined:

name: The name of the library. It must contain at least 3 characters among the following:

$$A - Za - z0 - 9\_.$$

. In particular, no spaces are allowed.

url: The URL of the library, i.e. where more information can be found.

license: The license under which the library is released. Can be `GPL`, `LGPL`, `BSD`, `Public` (for Public Domain) or `MIT`.

tags: An array of tag-strings which are used on the repository website to sort libraries.

description: The description of what the library is doing.

version: The version string of the library. This is detailed in Versioning (節 11.2.1).

classPath: The path string to the source files.

releasenote: The release notes of the current version.

contributors: An array of user names which identify contributors to the library.

dependencies: An object describing the dependencies of the library. This is detailed in Dependencies (節 11.2.2).

The following JSON is a simple example of a haxelib.json:

```
{
  "name": "useless_lib",
  "url" : "https://github.com/jasononeil/useless/",
  "license": "MIT",
  "tags": ["cross", "useless"],
  "description": "This library is useless in the same way on every
     platform.",
  "version": "1.0.0",
  "releasenote": "Initial release, everything is working correctly.",
  "contributors": ["Juraj","Jason","Nicolas"],
  "dependencies": {
    "tink_macro": "",
    "nme": "3.5.5"
  }
}
```

11.2.1 Versioning

Haxelib uses a simplified version of SemVer. The basic format is this:

```
major.minor.patch
```

These are the basic rules:

- Major versions are incremented when you break backwards compatibility - so old code will not work with the new version of the library.
- Minor versions are incremented when new features are added.
- Patch versions are for small fixes that do not change the public API, so no existing code should break.
- When a minor version increments, the patch number is reset to 0. When a major version increments, both the minor and patch are reset to 0.

Examples:

"0.0.1": A first release. Anything with a "0" for the major version is subject to change in the next release - no promises about API stability!

"0.1.0": Added a new feature! Increment the minor version, reset the patch version

"0.1.1": Realised the new feature was broken. Fixed it now, so increment the patch version

"1.0.0": New major version, so increment the major version, reset the minor and patch versions. You promise your users not to break this API until you bump to 2.0.0

"1.0.1": A minor fix

"1.1.0": A new feature

"1.2.0": Another new feature

"2.0.0": A new version, which might break compatibility with 1.0. Users are to upgrade cautiously.

If this release is a preview (Alpha, Beta or Release Candidate), you can also include that, with an optional release number:

```
1 major.minor.patch-(alpha/beta/rc).release
```

Examples:

"1.0.0-alpha": The alpha of 1.0.0 - use with care, things are changing!

"1.0.0-alpha.2": The 2nd alpha

"1.0.0-beta": Beta - things are settling down, but still subject to change.

"1.0.0-rc.1": The 1st release candidate for 1.0.0 - you shouldn't be adding any more features now

"1.0.0-rc.2": The 2nd release candidate for 1.0.0

"1.0.0": The final release!

11.2.2 Dependencies

As of Haxe 3.1.0, haxelib supports only exact version matching for dependencies. Dependencies are defined as part of the haxelib.json (11.2), with the library name serving as key and the expected version (if required) as value in the format described in Versioning (節 11.2.1).

We have seen an example of this when introducing haxelib.json:

```
"dependencies": {
  "tink_macros": "",
  "nme": "3.5.5"
}
```

This adds two dependencies to the given Haxe library:

1. The library "tink_macros" can be used in any version. Haxelib will then always try to use the latest version.
2. The library "nme" is required in version "3.5.5". Haxelib will make sure that this exact version is used, avoiding potential breaking changes with future versions.

11.3 extraParams.hxml

If you add a file named `extraParams.hxml` to your library root (at the same level as `haxelib.json`), these parameters will be automatically added to the compilation parameters when someone use your library with `-lib`.

11.4 Using Haxelib

If the `haxelib` command is executed without any arguments, it prints an exhaustive list of all available arguments. Access the `http://lib.haxe.org` website to view all the libraries available.

The following commands are available:

Basic `haxelib install [project-name] [version]` installs the given project. You can optionally specify a specific version to be installed. By default, latest released version will be installed.

`haxelib update [project-name]` updates a single library to their latest version.

`haxelib upgrade` upgrades all the installed projects to their latest version. This command prompts a confirmation for each upgradeable project.

`haxelib remove project-name [version]` removes complete project or only a specified version if specified.

`haxelib list` lists all the installed projects and their versions. For each project, the version surrounded by brackets is the current one.

`haxelib set [project-name] [version]` changes the current version for a given project. The version must be already installed.

Information `haxelib search [word]` lists the projects which have either a name or description matching specified word.

`haxelib info [project-name]` gives you information about a given project.

`haxelib user [user-name]` lists information on a given Haxelib user.

`haxelib config` prints the Haxelib repository path. This is where Haxelib get installed by default.

`haxelib path [project-name]` prints paths to libraries and its dependencies (defined in `haxelib.xml`).

Development `haxelib submit [project.zip]` submits a package to Haxelib. If the user name is unknown, you'll be first asked to register an account. If the user already exists, you will be prompted for your password. If the project does not exist yet, it will be created, but no version will be added. You will have to submit it a second time to add the first released version. If you want to modify the project url or description, simply modify your `haxelib.xml` (keeping version information unchanged) and submit it again.

`haxelib register [project-name]` submits or update a library package.

`haxelib local [project-name]` tests the library package. Make sure everything (both installation and usage) is working correctly before submitting, since once submitted, a given version cannot be updated.

`haxelib dev [project-name] [directory]` sets a development directory for the given project. To set project directory back to global location, run command and omit directory.

`haxelib git [project-name] [git-clone-path] [branch] [subdirectory]` uses git repository as library. This is useful for using a more up-to-date development version, a fork of the original project, or for having a private library that you do not wish to post to Haxelib. When you use `haxelib upgrade` any libraries that are installed using GIT will automatically pull the latest version.

Miscellaneous `haxelib setup` sets the Haxelib repository path. To print current path use `haxelib config`.

`haxelib selfupdate` updates Haxelib itself. It will ask to run `haxe update.hxml` after this update.

`haxelib convertxml` converts `haxelib.xml` file to `haxelib.json`.

`haxelib run [project-name] [parameters]` runs the specified library with parameters. Requires a precompiled Haxe/ Neko `run.n` file in the library package. This is useful if you want users to be able to do some post-install script that will configure some additional things on the system. Be careful to trust the project you are running since the script can damage your system.

`haxelib proxy` setup the Http proxy.

章 12

Target Details

12.1 JavaScript

12.1.1 Getting started with Haxe/JavaScript

Haxe can be a powerful tool for developing JavaScript applications. Let's look at our first sample. This is a very simple example showing the toolchain.

Create a new folder and save this class as `Main.hx`.

```
import js.Lib;
import js.Browser;
class Main {
    static function main() {
        var button = Browser.document.createButtonElement();
        button.textContent = "Click me!";
        button.onclick = function(event) {
            Lib.alert("Haxe is great");
        }
        Browser.document.body.appendChild(button);
    }
}
```

To compile, either run the following from the command line:

```
haxe -js main-javascript.js -main Main -D js-flatten -dce full
```

Another possibility is to create and run (double-click) a file called `compile.hxml`. In this example the hxml-file should be in the same directory as the example class.

```
-js main-javascript.js
-main Main
-D js-flatten
-dce full
```

The output will be a main-javascript.js, which creates and adds a clickable button to the document body.

Run the JavaScript To display the output in a browser, create an HTML-document called `index.html` and open it.

```
<!DOCTYPE html>
<html>
    <body>
        <script src="main-javascript.js"></script>
    </body>
</html>
```

More information

- Haxe JavaScript API docs
- MDN JavaScript Reference

12.1.2 Using external JavaScript libraries

The externs mechanism (6.2) provides access to the native APIs in a type-safe manner. It assumes that the defined types exist at run-time but assumes nothing about how and where those types are defined.

An example of an extern class is the jQuery class of the Haxe Standard Library. To illustrate, here is a simplified version of this extern class:

```
package js.jquery;
@:native("$") extern class JQuery {
    /**
        Creates DOM elements on the fly from the provided string of raw HTML.
        OR
        Accepts a string containing a CSS selector which is then used to match a set of elements.
        OR
        Binds a function to be executed when the DOM has finished loading.
    **/
    @:selfCall
    @:overload(function(element:js.html.Element):Void { })
    @:overload(function(selection:js.jquery.JQuery):Void { })
    @:overload(function(callback:haxe.Constraints.Function):Void { })
    @:overload(function(selector:String, ?context:haxe.extern.EitherType<js.html.Element, js.jquery.JQuery>):Void { })
    public function new():Void;

    /**
        Adds the specified class(es) to each element in the set of matched elements.
    **/
    @:overload(function(_function:Int -> String -> String):js.jquery.JQuery { })
    public function addClass(className:String):js.jquery.JQuery;

    /**
        Get the HTML contents of the first element in the set of matched elements.
```

```
        OR
        Set the HTML contents of each element in the set of matched
            elements.
    **/
    @:overload(function(htmlString:String):js.jquery.JQuery { })
    @:overload(function(_function:Int -> String -> String):js.jquery.
        JQuery { })
    public function html():String;
}
```

Note that functions can be overloaded to accept different types of arguments and return values, using the `@:overload` metadata. Function overloading works only in externs.

Using this extern, we can use jQuery like this:

```
import js.jquery.*;
..
new JQuery("#my-div").addClass("brand-success").html("haxe is great!")
    ;
..
```

The package and class name of the extern class should be the same as defined in the external library. If that is not the case, rewrite the path of a class using `@:native`.

```
package my.application.media;

@:native('external.library.media.video')
extern class Video {
..
```

Some JavaScript libraries favor instantiating classes without using the `new` keyword. To prevent the Haxe compiler outputting the `new` keyword when using a class, we can attach a `@:selfCall` metadata to its constructor. For example, when we instantiate the jQuery extern class above, `new JQuery()` will be outputted as `$ ()` instead of `new $ ()`. The `@:selfCall` metadata can also be attached to a method. In this case, the method will be interpreted as a direct call to the object, illustrated as follows:

```
extern class Functor {
    public function new():Void;
    @:selfCall function call():Void;
}

class Test {
    static function main() {
        var f = new Functor();
        f.call(); // will be outputted as `f();`
    }
}
```

Beside externs, Typedefs (3.1) can be another great way to name (or alias) a JavaScript type. The major difference between typedefs and externs is that, typedefs are duck-typed but externs are not. Typedefs are suitable for common data structures, e.g. point (`{x:Float, y:Float}`). Use of a point structure typedef for function arguments allows external JavaScript functions to accept point class instances from Haxe or from another JavaScript library. It is also useful for typing JSON objects.

The Haxe Standard Library comes with externs of jQuery and SWFObject. Their version compatibility is summarized as follows:

Haxe version	Library	Externs location
3.3	jQuery 1.11.3 / 2.1.4	<code>js.jquery.*</code>
3.2-	jQuery 1.6.4	<code>js.JQuery</code>
3.3	SWFObject 2.3	<code>js.swfobject.*</code>
3.2-	SWFObject 1.5	<code>js.SWFObject</code>

There are many externs for other popular native libraries available on Haxelib library (11). To view a list of them, check out the extern tag.

12.1.3 Inject raw JavaScript

In Haxe, it is possible to call an exposed function thanks to the `untyped` keyword. This can be useful in some cases if we don't want to write externs. Anything untyped that is valid syntax will be generated as it is.

```
untyped window.trackEvent("page1");
```

12.1.4 JavaScript untyped functions

These functions allow to access specific JavaScript platform features. It works only when the Haxe compiler is targeting JavaScript and should always be prefixed with `untyped`.

Important note: Before using these functions, make sure there is no alternative available in the Haxe Standard Library. The resulting syntax can not be validated by the Haxe compiler, which may result in invalid or error-prone code in the output.

`untyped __js__(expr, params)` Injects raw JavaScript expressions (12.1.3). It's allowed to use `{0}`, `{1}`, `{2}` etc in the expression and use the rest arguments to feed Haxe fields. The Haxe compiler will take care of the surrounding quotes if needed. The function can also return values.

```
untyped __js__('alert("Haxe is great!")');
// output: alert("Haxe is great!");

var myMessage = "Haxe is great!";
untyped __js__('alert({0})', myMessage);
// output:
//  var myMessage = "Haxe is great!";
//  alert(myMessage);

var myVar:Bool = untyped __js__('confirm({0})', "Are you sure?");
// output: var myVar = confirm("Are you sure?");

var hexString:String = untyped __js__('({0}).toString({1})', 255, 16);
// output: var hexString = (255).toString(16);
```

`untyped __instanceof__(o,cl)` Same as `o instanceof cl` in JavaScript.

```
var myString = new String("Haxe is great");
var isString = untyped __instanceof__(myString, String);
output: var isString = (myString instanceof String);
```

`untyped __typeof__(o)` Same as `typeof o` in JavaScript.

```
var isNodeJS = untyped __typeof__(window) == null;
output: var isNodeJS = typeof(window) == null;
```

`untyped __strict_eq__(a,b)` Same as `a === b` in JavaScript, tests on strict equality (or "triple equals" or "identity").

```
var a = "0";
var b = 0;
var isEqual = untyped __strict_eq__(a, b);
output: var isEqual = ((a) === b);
```

`untyped __strict_neq__(a,b)` Same as `a !== b` in JavaScript, tests on negative strict equality.

```
var a = "0";
var b = 0;
var isntEqual = untyped __strict_neq__(a, b);
output: var isntEqual = ((a) !== b);
```

Expression injection In some cases it may be needed to inject raw JavaScript code into Haxe-generated code. With the __js__ function we can inject pure JavaScript code fragments into the output. This code is always untyped and can not be validated, so it accepts invalid code in the output, which is error-prone. This could, for example, write a JavaScript comment in the output.

```
untyped __js__('// haxe is great!');
```

A more useful demonstration would be to call a function and pass arguments using the __js__ function. This example illustrates how to call this function and how to pass parameters. Note that the code interpolation will wrap the quotes around strings in the generated output.

```
// Haxe code:
var myVar = untyped __js__('myObject.myJavaScriptFunction({0}, {1})',
   "Mark", 31);
```

This will generate the following JavaScript code:

```
// JavaScript Code
var myVar = myObject.myJavaScriptFunction("Mark", 31);
```

12.1.5 Debugging JavaScript

Haxe is able to generate source maps, allowing Javascript debuggers to map from generated JavaScript back to the original Haxe source. This makes reading error stack traces, debugging with breakpoints, and profiling much easier.

Compiling with the `-debug` flag will create a .map alongside the .js file. Enable it in Chrome by clicking on the cog settings button in the bottom right of the Developer Tools window, and checking "Enable source maps". The pause button on the bottom left can be toggled to pause on uncaught exceptions.

12.1.6 JavaScript target Metadata

This is the list of JavaScript specific metadata. For more information, see also the complete list of all Haxe built-in metadata (8.1).

JavaScript metadata	
Metadata	Description
@:expose _(?Name=Class path)_	Makes the class available on the `window` object or `exports` for node
@:jsRequire	Generate javascript module require expression for given extern
@:selfCall	Translates method calls into calling object directly

12.1.7 Exposing Haxe classes for JavaScript

It is possible to make Haxe classes or static fields available for usage in plain JavaScript. To expose, add the `@:expose` metadata to the desired class or static fields.

This example exposes the Haxe class `MyClass`.

```
@:expose
class MyClass {
  var name:String;
  function new(name:String) {
    this.name = name;
  }
  public function foo() {
    return 'Greetings from $name!';
  }
}
```

It generates the following JavaScript output:

```
(function ($hx_exports) { "use strict";
var MyClass = $hx_exports.MyClass = function(name) {
    this.name = name;
};
MyClass.prototype = {
    foo: function() {
        return "Greetings from " + this.name + "!";
    }
};
})(typeof window != "undefined" ? window : exports);
```

By passing globals (like `window` or `exports`) as parameters to our anonymous function in the JavaScript module, it becomes available which allows to expose the Haxe generated module.

In plain JavaScript it is now possible to create an instance of the class and call its public functions.

```
// JavaScript code
var instance = new MyClass('Mark');
console.log(instance.foo()); // logs a message in the console
```

The package path of the Haxe class will be completely exposed. To rename the class or define a different package for the exposed class, use `@:expose("my.package.MyExternalClass")`

Shallow expose　When the code generated by Haxe is part of a larger JavaScript project and wrapped in a large closure it is not always necessary to expose the Haxe types to global variables. Compiling the project using `-D shallow-expose` allows the types or static fields to be available for the surrounding scope of the generated closure only.

When the code is compiled using `-D shallow-expose`, the generated output will look like this:

```
var $hx_exports = $hx_exports || {};
(function () { "use strict";
var MyClass = $hx_exports.MyClass = function(name) {
    this.name = name;
};
MyClass.prototype = {
    foo: function() {
        return "Greetings from " + this.name + "!";
    }
};
})();
var MyClass = $hx_exports.MyClass;
```

In this pattern, a var statement is used to expose the module; it doesn't write to the `window` or `exports` object.

12.1.8　Loading extern classes using "require" function

【Haxe 3.2.0 から】

Modern JavaScript platforms, such as Node.js provide a way of loading objects from external modules using the "require" function. Haxe supports automatic generation of "require" statements for `extern` classes.

This feature can be enabled by specifying `@:jsRequire` metadata for the extern class. If our `extern` class represents a whole module, we pass a single argument to the `@:jsRequire` metadata specifying the name of the module to load:

```
@:jsRequire("fs")
extern class FS {
  static function readFileSync(path:String, encoding:String):String;
}
```

In case our `extern` class represents an object within a module, second `@:jsRequire` argument specifies an object to load from a module:

```
@:jsRequire("http", "Server")
extern class HTTPServer {
  function new();
}
```

The second argument is a dotted-path, so we can load sub-objects in any hierarchy.

If we need to load custom JavaScript objects in runtime, a `js.Lib.require` function can be used. It takes `String` as its only argument and returns `Dynamic`, so it is advised to be assigned to a strictly typed variable.

12.2 Flash

12.2.1 Getting started with Haxe/Flash

Developing Flash applications is really easy with Haxe. Let's look at our first code sample. This is a basic example showing most of the toolchain.

Create a new folder and save this class as `Main.hx`.

```
import flash.Lib;
import flash.display.Shape;
class Main {
    static function main() {
        var stage = Lib.current.stage;

        // create a center aligned rounded gray square
        var shape = new Shape();
        shape.graphics.beginFill(0x333333);
        shape.graphics.drawRoundRect(0, 0, 100, 100, 10);
        shape.x = (stage.stageWidth - 100) / 2;
        shape.y = (stage.stageHeight - 100) / 2;

        stage.addChild(shape);
    }
}
```

To compile this, either run the following from the command line:

```
haxe -swf main-flash.swf -main Main -swf-version 15 -swf-header
    960:640:60:f68712
```

Another possibility is to create and run (double-click) a file called `compile.hxml`. In this example the hxml-file should be in the same directory as the example class.

```
-swf main-flash.swf
-main Main
-swf-version 15
-swf-header 960:640:60:f68712
```

The output will be a main-flash.swf with size 960x640 pixels at 60 FPS with an orange background color and a gray square in the center.

Display the Flash Run the SWF standalone using the Standalone Debugger FlashPlayer.

To display the output in a browser using the Flash-plugin, create an HTML-document called `index.html` and open it.

```
<!DOCTYPE html>
<html>
    <body>
        <embed src="main-flash.swf" width="960" height="640">
    </body>
</html>
```

More information

- Haxe Flash API docs
- Adobe Livedocs

12.2.2 Embedding resources

Embedding resources is different in Haxe compared to ActionScript 3. Instead of using

embed

(AS3-metadata) use Flash specific compiler metadata (12.2.4) like `@:bitmap`, `@:font`, `@:sound` or `@:file`.

```
1  import flash.Lib;
2  import flash.display.BitmapData;
3  import flash.display.Bitmap;
4
5  class Main {
6    public static function main() {
7      var img = new Bitmap( new MyBitmapData(0, 0) );
8      Lib.current.addChild(img);
9    }
10 }
11
12 @:bitmap("relative/path/to/myfile.png")
13 class MyBitmapData extends BitmapData { }
```

12.2.3 Using external Flash libraries

To embed external `.swf` or `.swc` libraries, use the following compilation options:

`-swf-lib <file>` Embeds the SWF library in the compiled SWF.

`-swf-lib-extern <file>` Adds the SWF library for type checking but doesn't include it in the compiled SWF.

The standard compilation options also provide more Haxe sources to be added to the project:

- To add another class path use `-cp <directory>`.
- To add a Haxelib library (11) use `-lib <library-name>`.
- To force a whole package to be included in the project, use `--macro include('mypackage')` which will include all the classes declared in the given package and subpackages.

12.2.4 Flash target Metadata

This is the list of Flash specific metadata. For a complete list see Haxe built-in metadata (8.1).

Flash metadata	
Metadata	Description
@:bind	Override Swf class declaration
@:bitmap _(Bitmap file path)_	_Embeds given bitmap data into the class (must extend `flash.di`
@:debug	Forces debug information to be generated into the Swf even with
@:file(File path)	Includes a given binary file into the target Swf and associates it v
@:font _(TTF path Range String)_	Embeds the given TrueType font into the class (must extend `flas`
@:getter _(Class field name)_	Generates a native getter function on the given field
@:noDebug	Does not generate debug information into the Swf even if `-debug`
@:ns	Internally used by the Swf generator to handle namespaces
@:setter _(Class field name)_	Generates a native setter function on the given field
@:sound _(File path)_	Includes a given _.wav_ or _.mp3_ file into the target Swf and ass

12.3 Neko

12.4 PHP

12.4.1 Getting started with Haxe/PHP

To get started with Haxe/PHP, create a new folder and save this class as `Main.hx`.

```
1 import php.Lib;
2
3 class Main {
4   static function main() {
5     Lib.println('Haxe is great!');
6   }
7 }
```

To compile, either run the following from the command line:

```
1 haxe -php bin -main Main
```

Another possibility is to create and run (double-click) a file called `compile.hxml`. In this example the hxml-file should be in the same directory as the example class.

```
1 -php bin
2 -main Main
```

The compiler outputs in the given bin-folder, which contains the generated PHP classes that prints the traced message when you run it. The generated PHP-code runs for version 5.1.0 and later.

More information

- Haxe PHP API docs
- PHP.net Documentation
- PHP to Haxe tool

12.4.2 PHP untyped functions

These functions allow to access specific PHP platform features. It works only when the Haxe compiler is targeting PHP and should always be prefixed with `untyped`.

Important note: Before using these functions, make sure there is no alternative available in the Haxe Standard Library. The resulting syntax can not be validated by the Haxe compiler, which may result in invalid or error-prone code in the output.

`untyped __php__(expr)` Injects raw PHP code expressions. It's possible to pass fields from Haxe source code using String Interpolation (6.5).

```
var value:String = "test";
untyped __php__("echo '<pre>'; print_r($value); echo '</pre>';");
// output: echo '<pre>'; print_r('test'); echo '</pre>';
```

`untyped __call__(function, arg, arg, arg...)` Calls a PHP function with the desired number of arguments and returns what the PHP function returns.

```
var value = untyped __call__("array", 1,2,3);
// output returns a NativeArray with values [1,2,3]
```

`untyped __var__(global, paramName)` Get the values from global vars. Note that the dollar sign in the Haxe code is omitted.

```
var value : String = untyped __var__('_SERVER', 'REQUEST_METHOD')
// output: $value = $_SERVER['REQUEST_METHOD']
```

`untyped __physeq__(val1, val2)` Strict equals test between the two values. Returns a `Bool`.

```
var isFalse = untyped __physeq__(false, value);
// output: $isFalse = false === $value;
```

12.5 C++

12.5.1 Using C++ Defines

- ANDROID_HOST
- ANDROID_NDK_DIR
- ANDROID_NDK_ROOT
- BINDIR
- DEVELOPER_DIR
- HXCPP
- HXCPP_32
- HXCPP_COMPILE_CACHE
- HXCPP_COMPILE_THREADS

- HXCPP_CONFIG
- HXCPP_CYGWIN
- HXCPP_DEPENDS_OK
- HXCPP_EXIT_ON_ERROR
- HXCPP_FORCE_PDB_SERVER
- HXCPP_M32
- HXCPP_M64
- HXCPP_MINGW
- HXCPP_MSVC
- HXCPP_MSVC_CUSTOM
- HXCPP_MSVC_VER
- HXCPP_NO_COLOR
- HXCPP_NO_COLOUR
- HXCPP_VERBOSE
- HXCPP_WINXP_COMPAT
- IPHONE_VER
- LEGACY_MACOSX_SDK
- LEGACY_XCODE_LOCATION
- MACOSX_VER
- MSVC_VER
- NDKV
- NO_AUTO_MSVC
- PLATFORM
- QNX_HOST
- QNX_TARGET
- TOOLCHAIN_VERSION
- USE_GCC_FILETYPES
- USE_PRECOMPILED_HEADERS
- android
- apple
- blackberry

- cygwin
- dll_import
- dll_import_include
- dll_import_link
- emcc
- emscripten
- gph
- hardfp
- haxe_ver
- ios
- iphone
- iphoneos
- iphonesim
- linux
- linux_host
- mac_host
- macos
- mingw
- rpi
- simulator
- tizen
- toolchain
- webos
- windows
- windows_host
- winrt
- xcompile

12.5.2 Using C++ Pointers

12.6 Java

12.7 C#

12.8 Python