



HAXE

Haxe 3 マニュアル

Haxe Foundation
April 12, 2015
(訳 : October 10, 2015)

Todo list

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"	8
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 . .	10
make sure the types are right for inc, dec, negate, and bitwise negate	14
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	14
please review, doubled content	16
review please, sounds weird	16
for starters...please review	17
Is there a difference between <code>? y : Int</code> and <code>y : Null<Int></code> 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.	18
please review future tense	18
Same as in 2.2, what is <code>Enum<T></code> syntax?	21
list arguments	21
please reformat	23
I don't really know how these work yet.	25
It seems a bit convoluted explanations. Should we maybe start by "decoding" the meaning of <code>Void -> Void</code> , then <code>Int -> Bool -> Float</code> , then maybe have samples using <code>\$type</code>	25
please review for correctness	34
You have marked "Map" for some reason	35
Mention <code>toString()/String</code> conversion somewhere in this chapter.	47
"parent class" should probably be used here, but I have no idea what it means, so I will refrain from changing it myself.	48
proper label and caption + code/identifier styling for diagram	55
Figure out wtf our rules are now for when this is checked.	93
Comprehensions are only listing Arrays, not Maps	94
what to use for listing of non-haxe code like <code>hxml</code> ?	116
Check if we can build GADTs this way.	127
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?	145

目次

Todo list	1
1 導入	8
1.1 Haxeって何?	8
1.2 このドキュメントについて	9
1.2.1 著者と貢献者	9
1.2.2 ライセンス	10
1.3 Hello World	10
1.4 歴史	10
I 言語リファレンス	12
2 型	13
2.1 基本型	14
2.1.1 数値型	14
2.1.2 オーバーフロー	14
2.1.3 数値の演算子	14
2.1.4 Bool(真偽値)	15
2.1.5 Void	16
2.2 null許容	16
2.2.1 オプション引数とnull許容	18
2.3 クラスインスタンス	18
2.3.1 クラスのコンストラクタ	19
2.3.2 継承	19
2.3.3 インターフェース	20
2.4 列挙型インスタンス	21
2.4.1 列挙型のコンストラクタ	21
2.4.2 列挙型を使う	22
2.5 匿名の構造体	23
2.5.1 JSONで構造体を書く	24
2.5.2 構造体の型のクラス記法	24
2.5.3 オプションのフィールド	25
2.5.4 パフォーマンスへの影響	25
2.6 関数	25
2.6.1 オプション引数	25
2.6.2 デフォルト値	26
2.7 ダイナミック	27
2.7.1 型パラメータ付きのダイナミック	28
2.7.2 ダイナミックを実装(implements)する	28

2.8	抽象型(abstract)	29
2.8.1	暗黙のキャスト	30
2.8.2	演算子オーバーロード	33
2.8.3	配列アクセス	34
2.8.4	選択的関数	36
2.8.5	抽象型列挙体	36
2.8.6	抽象型フィールドの繰り上げ	37
2.8.7	コアタイプの抽象型	38
2.9	単相(モノモーフ)	39
3	型システム	40
3.1	typedef	40
3.1.1	拡張	41
3.2	型パラメータ	42
3.2.1	制約	43
3.3	ジェネリック	43
3.3.1	ジェネリック型パラメータのコンストラクト	45
3.4	変性(バリエーション)	46
3.5	単一化(ユニファイ)	47
3.5.1	クラスとインターフェースの単一化	48
3.5.2	構造的部分型付け	48
3.5.3	単相	49
3.5.4	関数の戻り値	49
3.5.5	共通の基底型	49
3.6	型推論	50
3.6.1	トップダウンの推論	51
3.6.2	制限	51
3.7	モジュールとパス	52
3.7.1	モジュールのサブタイプ(従属型)	52
3.7.2	インポート(import)	53
3.7.3	解決順序	55
4	Class Fields	58
4.1	Variable	58
4.2	Property	59
4.2.1	Common accessor identifier combinations	61
4.2.2	Impact on the type system	62
4.2.3	Rules for getter and setter	63
4.3	Method	64
4.3.1	Overriding Methods	65
4.3.2	Effects of variance and access modifiers	66
4.4	Access Modifier	67
4.4.1	Visibility	67
4.4.2	Inline	68
4.4.3	Dynamic	70
4.4.4	Override	70

5	式	71
5.1	ブロック	71
5.2	定数値	72
5.3	2項演算子	72
5.4	単項演算子	72
5.5	配列の宣言	72
5.6	オブジェクトの宣言	73
5.7	フィールドへのアクセス	73
5.8	配列アクセス	73
5.9	関数呼び出し	73
5.10	var(変数宣言)	74
5.11	ローカル関数	74
5.12	new(インスタンス化)	75
5.13	forループ	75
5.14	whileループ	75
5.15	do-whileループ	76
5.16	if	76
5.17	switch	76
5.18	try/catch	77
5.19	return	77
5.20	break	78
5.21	continue	78
5.22	throw	78
5.23	cast	78
	5.23.1 非セーフキャスト	78
	5.23.2 セーフキャスト	79
5.24	型チェック	80
6	言語機能	81
6.1	条件付きコンパイル	83
	6.1.1 グローバルコンパイラフラグ	84
6.2	extern	86
6.3	静的拡張	87
	6.3.1 Haxe標準ライブラリについて	88
6.4	パターンマッチング	88
	6.4.1 導入	88
	6.4.2 enumマッチング	89
	6.4.3 変数の取り出し	89
	6.4.4 構造体マッチング	90
	6.4.5 配列マッチング	90
	6.4.6 orパターン	91
	6.4.7 ガード	91
	6.4.8 複数の値のマッチング	91
	6.4.9 抽出子(エクストラクタ)	92
	6.4.10 網羅性のチェック	93
	6.4.11 無意味なパターンのチェック	93
6.5	文字列補間	94
6.6	配列内包表記	94
6.7	イテレータ(反復子)	95
6.8	関数の束縛(bind)	96
6.9	メタデータ	97

6.10	アクセス制御	98
6.11	インラインコンストラクタ	100
II	Compiler Reference	102
7	Compiler Usage	103
8	Compiler Features	105
8.1	Built-in Compiler Metadata	105
8.2	Dead Code Elimination	107
8.3	Completion	107
8.3.1	Overview	107
8.3.2	Field access completion	109
8.3.3	Call argument completion	110
8.3.4	Type path completion	110
8.3.5	Usage completion	112
8.3.6	Position completion	113
8.3.7	Top-level completion	113
8.3.8	Completion server	114
8.4	Resources	115
8.4.1	Embedding resources	116
8.4.2	Retrieving text resources	116
8.4.3	Retrieving binary resources	116
8.4.4	Implementation details	116
8.5	Runtime Type Information	117
8.5.1	RTTI structure	117
9	Macros	120
9.1	Macro Context	121
9.2	Arguments	121
9.2.1	ExprOf	122
9.2.2	Constant Expressions	123
9.2.3	Rest Argument	123
9.3	Reification	123
9.3.1	Expression Reification	124
9.3.2	Type Reification	124
9.3.3	Class Reification	125
9.4	Tools	125
9.5	Type Building	126
9.5.1	Enum building	127
9.5.2	@:autoBuild	128
9.5.3	@:genericBuild	129
9.6	Limitations	131
9.6.1	Macro-in-Macro	131
9.6.2	Static extension	131
9.6.3	Build Order	132
9.6.4	Type Parameters	132
9.7	Initialization macros	132

III	Standard Library	134
10	Standard Library	135
10.1	String	135
10.2	Data Structures	135
10.2.1	Array	135
10.2.2	Vector	137
10.2.3	List	137
10.2.4	GenericStack	138
10.2.5	Map	138
10.2.6	Option	140
10.3	Regular Expressions	140
10.3.1	Matching	141
10.3.2	Groups	142
10.3.3	Replace	142
10.3.4	Split	143
10.3.5	Map	143
10.3.6	Implementation Details	144
10.4	Math	144
10.4.1	Special Numbers	144
10.4.2	Mathematical Errors	145
10.4.3	Integer Math	145
10.4.4	Extensions	145
10.5	Lambda	145
10.6	Template	147
10.7	Reflection	149
10.8	Serialization	151
10.8.1	Serialization format	153
10.9	Xml	155
10.9.1	Getting started with Xml	155
10.9.2	Parsing Xml	156
10.9.3	Encoding Xml	156
10.10	Json	156
10.10.1	Parsing JSON	156
10.10.2	Encoding JSON	157
10.10.3	Implementation details	157
10.11	Input/Output	158
10.12	Sys/sys	158
10.13	Remoting	158
10.13.1	Remoting Connection	158
10.13.2	Implementation details	160
10.14	Unit testing	160
IV	Miscellaneous	163
11	Haxelib	164
11.1	Using a Haxe library with the Haxe Compiler	164
11.2	haxelib.json	165
11.2.1	Versioning	165
11.2.2	Dependencies	167
11.3	extraParams.hxml	167

11.4 Using Haxelib	167
12 Target Details	169
12.1 JavaScript	169
12.1.1 Getting started with Haxe/JavaScript	169
12.1.2 Using external JavaScript libraries	170
12.1.3 Inject raw JavaScript	172
12.1.4 JavaScript untyped functions	172
12.1.5 Debugging JavaScript	173
12.1.6 JavaScript target Metadata	174
12.1.7 Exposing Haxe classes for JavaScript	174
12.1.8 Loading extern classes using "require" function	175
12.2 Flash	176
12.2.1 Getting started with Haxe/Flash	176
12.2.2 Embedding resources	177
12.2.3 Using external Flash libraries	177
12.2.4 Flash target Metadata	177
12.3 Neko	178
12.4 PHP	178
12.4.1 Getting started with Haxe/PHP	178
12.4.2 PHP untyped functions	179
12.5 C++	179
12.5.1 Using C++ Defines	179
12.5.2 Using C++ Pointers	181
12.6 Java	181
12.7 C#	181
12.8 Python	181

章 1

導入

1.1 Haxeって何?

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、ECMAScript¹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。

Class Fields (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Compiler Usage (章 7)で基本的な内容を、そして、Compiler Features (章 8)で高度な機能をあつかいます。最後にMacros (章 9)で、ありふれたタスクをHaxeマクロがどのように単純かするのを見ながら、刺激的なマクロの世界に挑んでいきます。

¹<http://www.ecma-international.org/publications/standards/Ecma-327.htm>

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"

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出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。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 Haxe code here

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](https://creativecommons.org/licenses/by/4.0/)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](https://creativecommons.org/licenses/by/4.0/)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が。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。

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。

²<http://nekovm.org>

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³JavaScript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 Foundation³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ます、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

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。

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>

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				

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。同様に、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。

```
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。

```
1 // Arguments and variables of type Void are not allowed  
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なりません。

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 // error on static platforms
2 var a:Int = null;
3 var b:Null<Int> = null; // allowed
```

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 // error on static platforms
3 if( a == null ) { ... }
4 var b : Null<Int> = 0;
5 if( b != null ) { ... } // allowed
```

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。

```
1 var n : Null<Int> = null;
2 var a : Int = n;
3 trace(a); // 0 on static platforms
```

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なりません。

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ションマークを付けることで作ります。

```
1 // x is a native Int (not nullable)
2 function foo(x : Int = 0) {}
3 // y is Null<Int> (nullable)
4 function bar( ?y : Int) {}
5 // z is also Null<Int>
6 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Class Fields \(章 4\)の章で解説をします。](#)

以下のサンプルコードが、この節で学ぶ基本になります。

```
1 class Point {
2     var x : Int;
3     var y : Int;
4     public function new(x, y) {
5         this.x = x;
6         this.y = y;
7     }
8     public function toString() {
9         return "Point("+x+", "+y+")";
10    }
11 }
```

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、

please review future tense

- ・ 通常、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Overriding Methods (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かめます。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。

```
1 interface Debuggable extends Printable extends Serializable
```

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。

Haxe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);
```

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。

list arguments

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。

¹http://en.wikipedia.org/wiki/Algebraic_data_type

```

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; // valid
11    var en:Enum<Color> = Color; // valid
12    // Error: Color should be Enum<Color>
13    //var x:Enum<Color> = Red;
14  }
15 }

```

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 {
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("Color was red");
13      case Green: trace("Color was green");
14      case Blue: trace("Color was blue");
15      case Rgb(r, g, b): trace("Color had a red value of " + 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",
3     age : 32,
4     pos : [
5         { x : 0, y : 0 },
6         { x : 1, y : -1 }
7     ],
8 };

```

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。

```

1 // get value of name, which is "Nicolas"
2 user.name;
3 // set value of age to 33
4 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。


```

1 class Test {
2     static public function main() {
3         var point = { x: 0.0, y: 12.0 };
4         // { y : Float, x : Float } has no field z
5         point.z;
6     }
7 }

```

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。

```

1 class Path {
2     var start : { x : Int, y : Int };
3     var target : { x : Int, y : Int };
4     var current : { x : Int, y : Int };
5 }

```

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。

```

1 typedef Point = { x : Int, y : Int }
2
3 class Path {
4     var start : Point;
5     var target : Point;
6     var current : Point;
7 }

```

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。

```

1 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なりません。

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Class Fields (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     }
8 }
```

```

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います。

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
4         $type(test);
5         trace(test()); // i: 12, s: bar
6         trace(test(1)); // i: 1, s: bar
7         trace(test(1, "foo")); // i: 1, s: foo
8         trace(test("foo")); // i: 12, s: foo
9     }
10
11     static function test(?i = 12, s = "bar") {
12         return "i: " + i + ", s: " + s;
13     }
14 }

```

この例は、[オプション引数](#) (節 2.6.1)のものによく似ています。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。

```

1 static function test(i = 12, s = "bar") {
2     if (i == null) i = 12;
3     if (s == null) s = "bar";
4     return "i: " + i + ", s: " + s;
5 }

```

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。
- ・ [Dead Code Elimination \(節 8.2\)](#)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。

```

1 var d:Dynamic = 1;
2 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>として推論していました(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は使ってしまいがちですが、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。

```

1 class Main {
2     static function main() {
3         var jsonData = '[1, 2, 3]';
4         var json = haxe.Json.parse(jsonData);
5         $type(json); // Unknown<0>
6         for (i in 0...json.length) {

```

```

7      // Array access is not allowed on
8      // {+ length : Int }
9      trace(json[0]);
10     }
11 }
12 }

```

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。

```

1 var att : Dynamic<String> = xml.attributes;
2 // valid, value is a String
3 att.name = "Nicolas";
4 // dito (this documentation is quite old)
5 att.age = "26";
6 // error, value is not a String
7 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。

```

1 class ImplementsDynamic
2     implements Dynamic<String> {
3     public var present:Int;
4     public function new() {}
5 }
6
7 class Main {
8     static public function main() {
9         var c = new ImplementsDynamic();
10        // valid, present is an existing field
11        c.present = 1;
12        // valid, assigned value is a String
13        c.stringField = "foo";
14        // error, Int should be String
15        //c.intField = 1;
16    }

```

17 }

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。

```
1 class Resolve implements Dynamic<String> {
2     public var present:Int;
3     public function new() {}
4     function resolve(field:String) {
5         return "Tried to resolve " +field;
6     }
7 }
8
9 class Main {
10     static public function main() {
11         var c = new Resolve();
12         c.present = 2;
13         trace(c.present);
14         trace(c.resolveMe);
15     }
16 }
```

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。

```
1 abstract AbstractInt(Int) {
2     inline public function new(i:Int) {
3         this = i;
4     }
5 }
```

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

```
1 class Main {
2   static public function main() {
3     var a = new AbstractInt(12);
4     trace(a); //12
5   }
6 }
```

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。

```
1 var a = 12;
2 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 (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、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。

```

1 var a = _ImplicitCastField.MyAbstract_Impl_. fromString("3");
2 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。

```
1 var a = Std.parseInt("3");
2 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。

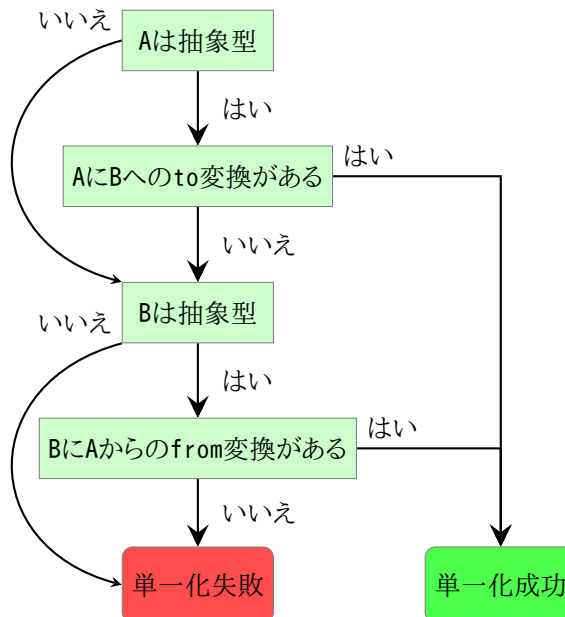


Figure 2.1: 選択アルゴリズムのフローチャート

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。

```
1 abstract A(Int) {
2   public function new() this = 0;
3   @:to public function toB() return new B();
4 }
5
6 abstract B(Int) {
7   public function new() this = 0;
8   @:to public function toC() return new C();
9 }
```

```

10
11 abstract C(Int) {
12     public function new() this = 0;
13 }
14
15 class Main {
16     static public function main() {
17         var a = new A();
18         var b:B = a; // valid, uses A.toB
19         var c:C = b; // valid, uses B.toC
20         var c:C = a; // error, A should be C
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++); // foo post
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   // The following line exposes the (A > B) operation from the
3   // underlying Int
4   // type. Note that no function body is used:
5   @:op(A > B) static function gt( a:MyAbstractInt, b:MyAbstractInt ) :
6     Bool;
7 }
8
9 class Main {
10   static function main() {
11     var a:MyAbstractInt = 42;
12     if(a > 0) trace('Works fine, > operation implemented!');
13
14     // The < operator is not implemented.
15     // This will cause an 'Cannot compare MyAbstractInt and Int' error
16     :
17     if(a < 100) { }
18   }
19 }
```

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

You have marked
"Map" for some reason

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。

```
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。

```
1 abstract AString(String) {
2     public function new(s) this = s;
3     @:arrayAccess function getInt1(k:Int) {
4         return this.charAt(k);
5     }
6     @:arrayAccess function getInt2(k:Int) {
7         return this.charAt(k).toUpperCase();
8     }
9 }
10
11 class Main {
12     static function main() {
13         var a = new AString("foo");
14         trace(a[0]); // f
15     }
16 }
```

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。

```
1 abstract MyAbstract<T>(T) from T {
2     public function new(t:T) this = t;
3
4     function get() return this;
5
6     @:impl
7     static public function getString(v:MyAbstract<String>):String {
8         return v.get();
9     }
10 }
11
12 class Main {
13     static public function main() {
14         var a = new MyAbstract("foo");
15         a.getString();
16         var b = new MyAbstract(1);
17         // Int should be MyAbstract<String>
18         b.getString();
19     }
20 }
```

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さ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。

```
1 @:enum
2 abstract HttpStatus(Int) {
3     var NotFound = 404;
4     var MethodNotAllowed = 405;
```

```

5 }
6
7 class Main {
8     static public function main() {
9         var status = HttpStatus.NotFound;
10        var msg = printStatus(status);
11    }
12
13    static function printStatus(status:HttpStatus) {
14        return switch(status) {
15            case NotFound:
16                "Not found";
17            case MethodNotAllowed:
18                "Method not allowed";
19        }
20    }
21 }

```

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。

```

1 Main.main = function() {
2     var status = 404;
3     var msg = Main.printStatus(status);
4 };
5 Main.printStatus = function(status) {
6     switch(status) {
7         case 404:
8             return "Not found";
9         case 405:
10            return "Method not allowed";
11     }
12 };

```

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“保ちたい”場合があります。繰り上がりの関数を手を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。

```

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

```

```

5   }
6   }
7
8   class Main {
9       static public function main() {
10          var myArray = new MyArray();
11          myArray.push(12);
12          myArray.pop();
13          // MyArray<Int> has no field length
14          //myArray.length;
15      }
16  }

```

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。

```

1 Main.main = function() {
2     var myArray = [];
3     myArray.push(12);
4     myArray.pop();
5 };

```

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ます、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 `typedef` 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。

```
1 typedef User = {
2   var age : Int;
3   var name : String;
4   @:optional var phoneNumber : String;
5 }
```

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。

```
1 typedef IterableWithLength<T> = {
2   > Iterable<T>,
3   // read only property
4   var length(default, null):Int;
5 }
6
7 class Main {
8   static public function main() {
9     var array = [1, 2, 3];
10    var t:IterableWithLength<Int> = array;
11  }
12 }
```

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。

```
1 typedef WithLength = {
2   var length(default, null):Int;
3 }
4
5 typedef IterableWithLengthAndPush<T> = {
6   > Iterable<T>,
7   > WithLength,
8   function push(a:T):Int;
9 }
10
11 class Main {
12   static public function main() {
13     var array = [1, 2, 3];
14     var t:IterableWithLengthAndPush<Int> = array;
15   }
16 }
```

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 メソッドと名前付きのローカル関数で利用可能
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	
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         // runtime message: bar should be foo  
5         equals("foo", "bar");  
6         // compiler error: 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、`<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 should be Iterable<String>
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 ジェネリックス

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つくことになります。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。

```

1 @:generic
2 class MyValue<T> {
3   public var value:T;
4   public function new(value:T) {
5     this.value = value;
6   }
7 }
8
9 class Main {
10  static public function main() {
11    var a = new MyValue<String>("Hello");
12    var b = new MyValue<Int>(42);
13  }
14 }

```

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。

```

1 (function () { "use strict";
2 var Test = function() { };
3 Test.main = function() {
4   var a = new MyValue_String("Hello");
5   var b = new MyValue_Int(5);
6 };
7 var MyValue_Int = function(value) {
8   this.value = value;
9 };
10 var MyValue_String = function(value) {
11   this.value = value;
12 };
13 Test.main();
14 })();

```

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。

```

1 class Main {
2   static public function main() {
3     method("foo");
4     method(1);
5   }
6
7   @:generic static function method<T>(t:T) { }
8 }

```

JavaScript出力を見れば明白です。

```

1 (function () { "use strict";
2 var Main = function() { }
3 Main.method_Int = function(t) {
4 }
5 Main.method_String = function(t) {
6 }

```

```

7 Main.main = function() {
8     Main.method_String("foo");
9     Main.method_Int(1);
10 }
11 Main.main();
12 })();

```

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。

```

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 }

```

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。

```

1 var Main = function() { }
2 Main.__name__ = true;
3 Main.make_haxe_Template = function() {
4     return new haxe.Template("foo");
5 }
6 Main.make_String = function() {

```

```

7     return new String("foo");
8 }
9 Main.main = function() {
10     var s = Main.make_String();
11     var t = Main.make_haxe_Template();
12 }

```

3.4 変性(バリエーション)

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。

```

1 class Base {
2     public function new() { }
3 }
4
5 class Child extends Base { }
6
7 class Main {
8     public static function main () {
9         var children = [new Child()];
10        // Array<Child> should be Array<Base>
11        // Type parameters are invariant
12        // Child should be Base
13        var bases:Array<Base> = children;
14    }
15 }

```

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が。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。

```

1 class Base {
2     public function new() { }
3 }
4
5 class Child extends Base { }
6
7 class OtherChild extends Base { }
8
9 class Main {
10    public static function main () {
11        var children = [new Child()];
12        // subvert type checker
13        var bases:Array<Base> = cast children;
14        bases.push(new OtherChild());
15        for(child in children) {
16            trace(child);
17        }
18    }
19 }

```

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的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         var s:String = 1;
5     }
6 }

```

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のを、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いければ、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。

3.5.2 構造的部分型付け

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

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、これ呼び出すことができます。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トラクト ([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。

```

1 private class C { ... }
2 private enum E { ... }
3 private typedef T { ... }
4 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、アクセスコントロール (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す。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。

```

1 import haxe.ds.StringMap;
2
3 class Main {
4     static public function main() {
5         // instead of: new haxe.ds.StringMap();
6         new StringMap();
7     }
8 }

```

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。

```

1 import haxe.macro.Expr;
2
3 class Main {
4     static public function main() {
5         var e:Binop = OpAdd;
6     }
7 }

```

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。

```
1 import Math.random;
2
3 class Main {
4     static public function main() {
5         random();
6     }
7 }
```

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。

```
1 import haxe.macro.*;
2
3 class Main {
4     static function main() {
5         var expr:Expr = null;
6         //var expr:ExprDef = null; // Class not found : ExprDef
7     }
8 }
```

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い、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。

```
1 import String.fromCharCode in f;
2
3 class Main {
4     static function main() {
5         var c1 = f(65);
6         var c2 = f(66);
7         trace(c1 + c2); // AB
8     }
9 }
```

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

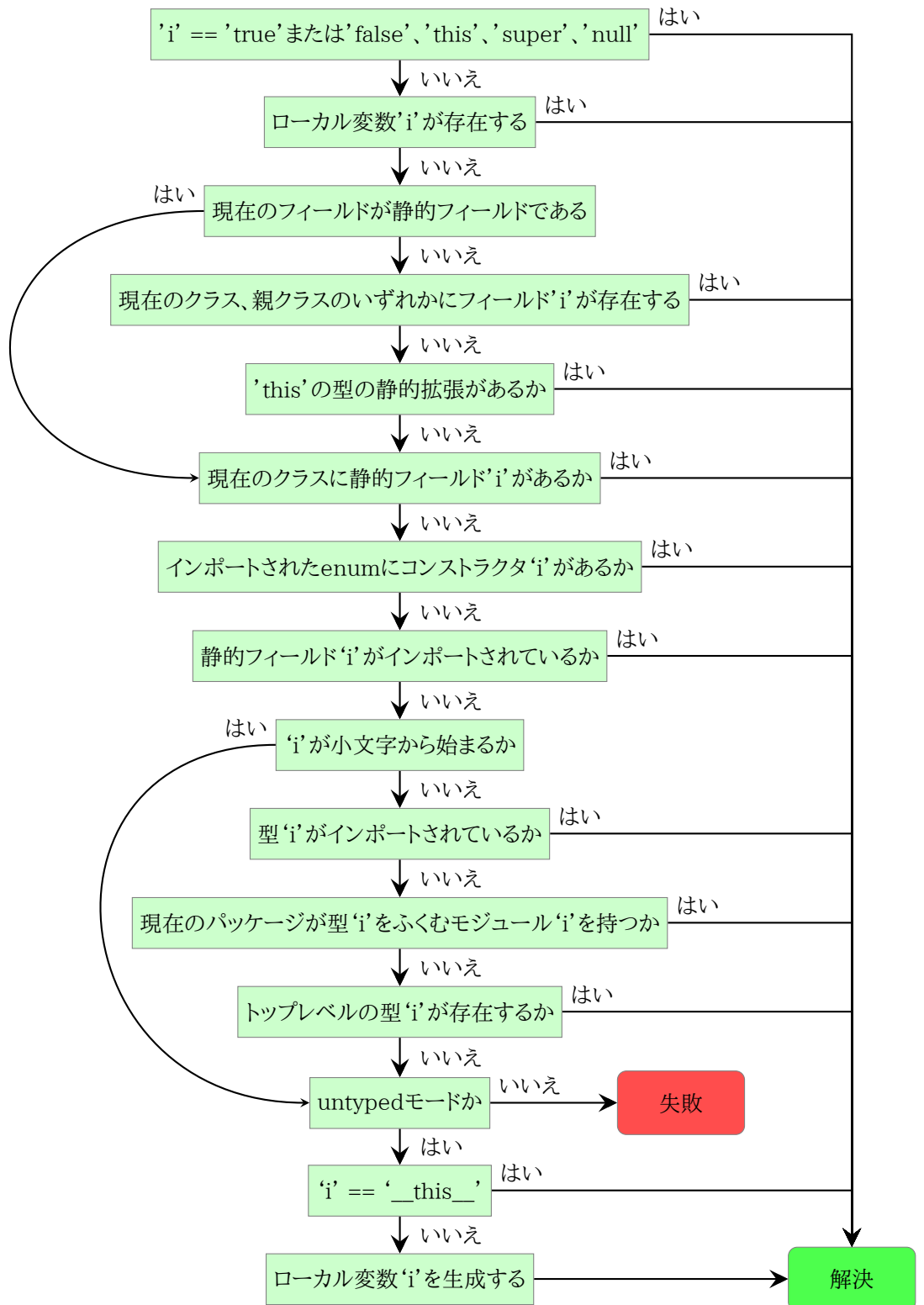


Figure 3.1: 識別子'i'の解決順序

章 4

Class Fields

定義: Class Field

A class field is a variable, property or method of a class which can either be static or non-static. Non-static fields are referred to as member fields, so we speak of e.g. a static method or a member variable.

So far we have seen how types and Haxe programs in general are structured. This section about class fields concludes the structural part and at the same time bridges to the behavioral part of Haxe. This is because class fields are the place where expressions (5) are at home.

There are three kinds of class fields:

Variable: A variable (4.1) class field holds a value of a certain type, which can be read or written.

Property: A property (4.2) class field defines a custom access behavior for something that, outside the class, looks like a variable field.

Method: A method (4.3) is a function which can be called to execute code.

Strictly speaking, a variable could be considered to be a property with certain access modifiers. Indeed, the Haxe Compiler does not distinguish variables and properties during its typing phase, but they remain separated at syntax level.

Regarding terminology, a method is a (static or non-static) function belonging to a class. Other functions, such as a local functions (5.11) in expressions, are not considered methods.

4.1 Variable

We have already seen variable fields in several code examples of previous sections. Variable fields hold values, a characteristic which they share with most (but not all) properties:

```
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 }

```

We can learn from this that a variable

1. has a name (here: `member`),
2. has a type (here: `String`),
3. may have a constant initialization (here: `"bar"`) and
4. may have access modifiers (4.4) (here: `static`)

The example first prints the initialization value of `member`, then sets it to `"foo"` before printing its new value. The effect of access modifiers is shared by all three class field kinds and explained in a separate section.

It should be noted that the explicit type is not required if there is an initialization value. The compiler will infer (3.6) it in this case.

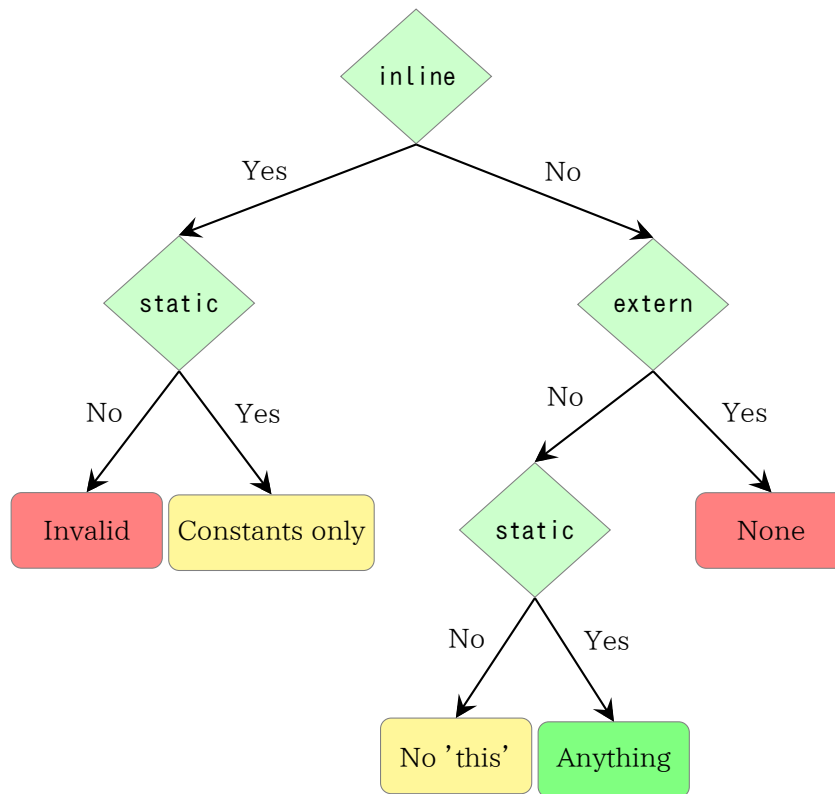


Figure 4.1: Initialization values of a variable field.

4.2 Property

Next to variables (4.1), properties are the second option for dealing with data on a class. Unlike variables however, they offer more control of which kind of field access should be

allowed and how it should be generated. Common use cases include:

- Have a field which can be read from anywhere, but only be written from within the defining class.
- Have a field which invokes a getter-method upon read-access.
- Have a field which invokes a setter-method upon write-access.

When dealing with properties, it is important to understand the two kinds of access:

定義: Read Access

A read access to a field occurs when a right-hand side field access expression (5.7) is used. This includes calls in the form of `obj.field()`, where `field` is accessed to be read.

定義: Write Access

A write access to a field occurs when a field access expression (5.7) is assigned a value in the form of `obj.field = value`. It may also occur in combination with read access (4.2) for special assignment operators such as `+=` in expressions like `obj.field += value`.

Read access and write access are directly reflected in the syntax, as the following example shows:

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

For the most part, the syntax is similar to variable syntax, and the same rules indeed apply. Properties are identified by

- the opening parenthesis (after the field name,
- followed by a special access identifier (here: `default`),
- with a comma , separating
- another special access identifier (here: `null`)
- before a closing parenthesis).

The access identifiers define the behavior when the field is read (first identifier) and written (second identifier). The accepted values are:

default: Allows normal field access if the field has public visibility, otherwise equal to `null` access.

null: Allows access only from within the defining class.

get/set: Access is generated as a call to an accessor method. The compiler ensures that the accessor is available.

dynamic: Like **get/set** access, but does not verify the existence of the accessor field.

never: Allows no access at all.

定義: Accessor method

An accessor method (or short accessor) for a field named `field` of type `T` is a getter named `get_field` of type `Void->T` or a setter named `set_field` of type `T->T`.

トリビア: Accessor names

In Haxe 2, arbitrary identifiers were allowed as access identifiers and would lead to custom accessor method names to be admitted. This made parts of the implementation quite tricky to deal with. In particular, `Reflect.getProperty()` and `Reflect.setProperty()` had to assume that any name could have been used, requiring the target generators to generate meta-information and perform lookups.

We disallowed these identifiers and went for the `get_` and `set_` naming convention which greatly simplified implementation. This was one of the breaking changes between Haxe 2 and 3.

4.2.1 Common accessor identifier combinations

The next example shows common access identifier combinations for properties:

```
1 class Main {
2     // read from outside, write only within Main
3     public var ro(default, null):Int;
4
5     // write from outside, read only within Main
6     public var wo(null, default):Int;
7
8     // access through getter get_x and setter
9     // set_x
10    public var x(get, set):Int;
11
12    // read access through getter, no write
13    // access
14    public var y(get, never):Int;
15
16    // required by field x
17    function get_x() return 1;
18
19    // required by field x
20    function set_x(x) return x;
21
22    // required by field y
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 }

```

The JavaScript output helps understand what the field access in the `main`-method is compiled to:

```

1 var Main = function() {
2     var v = this.get_x();
3     this.set_x(2);
4     var _g = this;
5     _g.set_x(_g.get_x() + 1);
6 };

```

As specified, the read access generates a call to `get_x()`, while the write access generates a call to `set_x(2)` where 2 is the value being assigned to `x`. The way the `+=` is being generated might look a little odd at first, but can easily be justified by the following example:

```

1 class Main {
2     public var x(get, set):Int;
3     function get_x() return 1;
4     function set_x(x) return x;
5
6     public function new() { }
7
8     static public function main() {
9         new Main().x += 1;
10    }
11 }

```

What happens here is that the expression part of the field access to `x` in the `main` method is complex: It has potential side-effects, such as the construction of `Main` in this case. Thus, the compiler cannot generate the `+=` operation as `new Main().x = new Main().x + 1` and has to cache the complex expression in a local variable:

```

1 Main.main = function() {
2     var _g = new Main();
3     _g.set_x(_g.get_x() + 1);
4 }

```

4.2.2 Impact on the type system

The presence of properties has several consequences on the type system. Most importantly, it is necessary to understand that properties are a compile-time feature and thus require the types to be known. If we were to assign a class with properties to `Dynamic`, field access would not respect accessor methods. Likewise, access restrictions no longer apply and all access is virtually public.

When using `get` or `set` access identifier, the compiler ensures that the getter and setter actually exists. The following problem does not compile:

```

1 class Main {
2     // Method get_x required by property x is missing
3     public var x(get, null):Int;
4     static public function main() {}
5 }

```

The method `get_x` is missing, but it need not be declared on the class defining the property itself as long as a parent class defines it:

```
1 class Base {
2     public function get_x() return 1;
3 }
4
5 class Main extends Base {
6     // ok, get_x is declared by parent class
7     public var x(get, null):Int;
8
9     static public function main() {}
10 }
```

The dynamic access modifier works exactly like `get` or `set`, but does not check for the existence

4.2.3 Rules for getter and setter

Visibility of accessor methods has no effect on the accessibility of its property. That is, if a property is `public` and defined to have a getter, that getter may be defined as `private` regardless.

Both getter and setter may access their physical field for data storage. The compiler ensures that this kind of field access does not go through the accessor method if made from within the accessor method itself, thus avoiding infinite recursion:

```
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 }
```

However, the compiler assumes that a physical field exists only if at least one of the access identifiers is `default` or `null`.

定義: Physical field

A field is considered to be physical if it is either

- a variable (4.1)
- a property (4.2) with the read-access or write-access identifier being `default` or `null`
- a property (4.2) with `:isVar` metadata (6.9)

If this is not the case, access to the field from within an accessor method causes a compilation error:

```
1 class Main {
```



```

2 // This field cannot be accessed because it is not a real variable
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 }

```

If a physical field is indeed intended, it can be forced by attributing the field in question with the `:isVar` metadata (6.9):

```

1 class Main {
2     // @isVar forces the field to be physical allowing the program to
3     // compile.
4     @:isVar public var x(get, set):Int;
5
6     function get_x() {
7         return x;
8     }
9
10    function set_x(x) {
11        return this.x = x;
12    }
13
14    static public function main() {}
15 }

```

トリビア: Property setter type

It is not uncommon for new Haxe users to be surprised by the type of a setter being required to be `T->T` instead of the seemingly more natural `T->Void`. After all, why would a setter have to return something?

The rationale is that we still want to be able to use field assignments using setters as right-side expressions. Given a chain like `x = y = 1`, it is evaluated as `x = (y = 1)`. In order to assign the result of `y = 1` to `x`, the former must have a value. If `y` had a setter returning `Void`, this would not be possible.

4.3 Method

While variables (4.1) hold data, methods are defining behavior of a program by hosting expressions (5). We have seen method fields in every code example of this document with even the initial Hello World (1.3) example containing a `main` method:

```

1 class Main {
2     static public function main():Void {

```

```

3     trace("Hello World");
4 }
5 }

```

Methods are identified by the `function` keyword. We can also learn that they

1. have a name (here: `main`),
2. have an argument list (here: empty `()`),
3. have a return type (here: `Void`),
4. may have access modifiers (4.4) (here: `static` and `public`) and
5. may have an expression (here: `{trace("Hello World");}`).

We can also look at the next example to learn more about arguments and return types:

```

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 }

```

Arguments are given by an opening parenthesis (after the field name, a comma , separated list of argument specifications and a closing parenthesis). Additional information on the argument specification is described in [関数 \(節 2.6\)](#).

The example demonstrates how type inference (3.6) can be used for both argument and return types. The method `myFunc` has two arguments but only explicitly gives the type of the first one, `f`, as `String`. The second one, `i`, is not type-hinted and it is left to the compiler to infer its type from calls made to it. Likewise, the return type of the method is inferred from the `return true` expression as `Bool`.

4.3.1 Overriding Methods

Overriding fields is instrumental for creating class hierarchies. Many design patterns utilize it, but here we will explore only the basic functionality. In order to use overrides in a class, it is required that this class has a parent class (2.3.2). Let us consider the following example:

```

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 }

```

The important components here are:

- the class `Base` which has a method `myMethod` and a constructor,
- the class `Child` which `extends Base` and also has a method `myMethod` being declared with `override`, and
- the `Main` class whose `main` method creates an instance of `Child`, assigns it to a variable `child` of explicit type `Base` and calls `myMethod()` on it.

The variable `child` is explicitly typed as `Base` to highlight an important difference: At compile-time the type is known to be `Base`, but the runtime still finds the correct method `myMethod` on class `Child`. This is because field access is resolved dynamically at runtime.

The `Child` class can access methods it has overridden by calling `super.methodName()`:

```

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 }

```

The section on [継承 \(節 2.3.2\)](#) explains the use of `super()` from within a `new` constructor.

4.3.2 Effects of variance and access modifiers

Overriding adheres to the rules of variance ([3.4](#)). That is, their argument types allow contravariance (less specific types) while their return type allows covariance (more specific types):

```

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 }

```

Intuitively, this follows from the fact that arguments are “written to” the function and the return value is “read from” it.

The example also demonstrates how visibility (4.4.1) may be changed: An overriding field may be **public** if the overridden field is **private**, but not the other way around.

It is not possible to override fields which are declared as **inline** (4.4.2). This is due to the conflicting concepts: While inlining is done at compile-time by replacing a call with the function body, overriding fields necessarily have to be resolved at runtime.

4.4 Access Modifier

4.4.1 Visibility

Fields are by default private, meaning that only the class and its sub-classes may access them. They can be made public by using the **public** access modifier, allowing access from anywhere.

```

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         // Cannot access private field unavailable
12         MyClass.unavailable();
13     }
14 }

```

Access to field `available` of class `MyClass` is allowed from within `Main` because it is denoted as being `public`. However, while access to field `unavailable` is allowed from within class `MyClass`, it is not allowed from within class `Main` because it is `private` (explicitly, although this identifier is redundant here).

The example demonstrates visibility through static fields, but the rules for member fields are equivalent. The following example demonstrates visibility behavior for when inheritance (2.3.2) is involved.

```
1 class Base {
2     public function new() { }
3     private function baseField() { }
4 }
5
6 class Child1 extends Base {
7     private function child1Field() { }
8 }
9
10 class Child2 extends Base {
11     public function child2Field() {
12         var child1 = new Child1();
13         child1.baseField();
14         // Cannot access private field child1Field
15         child1.child1Field();
16     }
17 }
18
19 class Main {
20     static public function main() { }
21 }
```

We can see that access to `child1.baseField()` is allowed from within `Child2` even though `child1` is of a different type, `Child1`. This is because the field is defined on their common ancestor class `Base`, contrary to field `child1Field` which can not be accessed from within `Child2`.

Omitting the visibility modifier usually defaults the visibility to `private`, but there are exceptions where it becomes `public` instead:

1. If the class is declared as `extern`.
2. If the field is declared on an interface (2.3.3).
3. If the field overrides (4.3.1) a public field.

トリビア: Protected

Haxe has no notion of a `protected` keyword known from Java, C++ and other object-oriented languages. However, its `private` behavior is equal to those language's protected behavior, so Haxe actually lacks their real private behavior.

4.4.2 Inline

The `inline` keyword allows function bodies to be directly inserted in place of calls to them. This can be a powerful optimization tool, but should be used judiciously as not all functions

are good candidates for inline behavior. The following example demonstrates the basic usage:

```
1 class Main {
2   static inline function mid(s1:Int, s2:Int) {
3     return (s1 + s2) / 2;
4   }
5
6   static public function main() {
7     var a = 1;
8     var b = 2;
9     var c = mid(a, b);
10  }
11 }
```

The generated JavaScript output reveals the effect of inline:

```
1 (function () { "use strict";
2 var Main = function() { }
3 Main.main = function() {
4   var a = 1;
5   var b = 2;
6   var c = (a + b) / 2;
7 }
8 Main.main();
9 })();
```

As evident, the function body $(s1 + s2) / 2$ of field `mid` was generated in place of the call to `mid(a, b)`, with `s1` being replaced by `a` and `s2` being replaced by `b`. This avoids a function call which, depending on the target and frequency of occurrences, may yield noticeable performance improvements.

It is not always easy to judge if a function qualifies for being inline. Short functions that have no writing expressions (such as `a = assignment`) are usually a good choice, but even more complex functions can be candidates. However, in some cases inlining can actually be detrimental to performance, e.g. because the compiler has to create temporary variables for complex expressions.

Inline is not guaranteed to be done. The compiler might cancel inlining for various reasons or a user could supply the `--no-inline` command line argument to disable inlining. The only exception is if the class is `extern` (6.2) or if the class field has the `:extern` metadata (6.9), in which case inline is forced. If it cannot be done, the compiler emits an error.

It is important to remember this when relying on inline:

```
1 class Main {
2   public static function main () { }
3
4   static function test() {
5     if (Math.random() > 0.5) {
6       return "ok";
7     } else {
8       error("random failed");
9     }
10  }
11
12  @:extern static inline function error(s:String) {
```

```

13     throw s;
14 }
15 }

```

If the call to `error` is inlined the program compiles correctly because the control flow checker is satisfied due to the inlined `throw` (5.22) expression. If `inline` is not done, the compiler only sees a function call to `error` and emits the error `A return is missing here`.

4.4.3 Dynamic

Methods can be denoted with the `dynamic` keyword to make them (re-)bindable:

```

1 class Main {
2     static dynamic function test() {
3         return "original";
4     }
5
6     static public function main() {
7         trace(test()); // original
8         test = function() { return "new"; }
9         trace(test()); // new
10    }
11 }

```

The first call to `test()` invokes the original function which returns the `String` "original". In the next line, `test` is assigned a new function. This is precisely what `dynamic` allows: Function fields can be assigned a new function. As a result, the next invocation of `test()` returns the `String` "new".

Dynamic fields cannot be `inline` for obvious reasons: While inlining is done at compile-time, dynamic functions necessarily have to be resolved at runtime.

4.4.4 Override

The access modifier `override` is required when a field is declared which also exists on a parent class (2.3.2). Its purpose is to ensure that the author of a class is aware of the override as this may not always be obvious in large class hierarchies. Likewise, having `override` on a field which does not actually override anything (e.g. due to a misspelled field name) triggers an error.

The effects of overriding fields are detailed in [Overriding Methods \(節 4.3.1\)](#). This modifier is only allowed on method (4.3) fields.

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のつながり合わせが続きます。

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icolon(;)で終わります。一般の構文としては以下のとおりです。

```
1 {式
2   1;式
3   2;
4   ... 式
```



```

5     N;
6 }

```

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。

```

1 {
2     a; // error, a is not declared yet
3     var a = 1; // declare a
4     a; // ok, a was declared
5     {
6         a; // ok, a is available in sub-blocks
7     }
8     // ok, a is still available after
9     // sub-blocks
10    a;
11 }
12 a; // error, a is not available outside

```

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。

```

1 var a = new Array();
2 a.push(1);
3 a.push(2);
4 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。

```

1 {
2     key1:value1,
3     key2:value2,
4     ...
5     keyN:valueN
6 }

```

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。

```

1 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。

```

1 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。

```

1 subject(); // call with no arguments
2 subject(e1); // call with one argument
3 subject(e1, e2); // call with two arguments
4 // call with multiple arguments
5 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。

```

1 var a; // declare local a
2 var b: Int; // declare variable b of type Int
3 // declare variable c, initialized to value 1
4 var c = 1;
5 // declare variable d and variable e
6 // initialized to value 2
7 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。

```

1 class Main {
2     static public function main() {
3         var value = 1;
4         function myLocalFunction(i) {
5             return value + i;
6         }
7         trace(myLocalFunction(2)); // 3
8     }
9 }

```

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。

```

1 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ことを確認します。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`型でなくてははいけません。

各くり返して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;
4         while (f < 0.5) {
```

```

5     trace(f);
6     f = Math.random();
7 }
8 }
9 }

```

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。

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

条件式はBool型でなくてはなりません。

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。

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

条件式はBool型でなくてはなりません。

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。

```
1 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。

```

1 if (condition1) expression1
2 else if(condition2) expression2
3 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、オプションなケース本体の式が来ます。

```

1 switch subject {
2     case pattern1: case-body-expression-1;
3     case pattern2: case-body-expression-2;
4     default: default-expression;
5 }

```

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)できなくてはなりません。

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。

```
1 try try-expr
2 catch(varName1:Type1) catch-expr-1
3 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。

```
1 return;
2 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。

```
1 function f1() {
2     function f2() {
3         return;
4     }
5     f2();
6     expression;
7 }
```

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; // unsafe cast  
2 cast (expr, Type); // safe cast
```

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         // Exception: 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エント(GADT)もサポートしています。

```

1 enum Result {
2     Success(data:Array<Int>);
3     UserError(msg:String);
4     SystemError(msg:String, position:PosInfos);
5 }

```

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なく、価値あるパフォーマンスの改善を得ることができます。

イテレータ(反復子) (6.7): Haxeはイテレータを適切にあつかつているので、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。

```

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

```

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。

```

1 var buffer = "";
2 function append(s:String) {
3     buffer += s;
4 }
5 append("foo");
6 append("bar");
7 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。

```

1 class MyClass {
2     @range(1, 8) var value:Int;
3 }
4 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。

```

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

```

文字列中の変数展開 (6.5): シングルクォ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。

```

1 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。

```
1 var map = new haxe.ds.IntMap();
2 var setToTwelve = map.set.bind(_, 12);
3 setToTwelve(1);
4 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。

```
1 var a = { foo: 12 };
2 switch (a) {
3     case { foo: i }: trace(i);
4     default:
5 }
```

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。

```
1 public var color(get, set);
2 function get_color() {
3     return element.style.backgroundColor;
4 }
5 function set_color(c:String) {
6     trace('Setting background of element to $c');
7     return element.style.backgroundColor = c;
8 }
```

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行うことができます。

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。

```
1 class Main<A> {
2     static function main() {
3         new Main<String>("foo");
4         new Main(12); // use type inference
5     }
6
7     function new(a:A) { }
8 }
```

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("debug level too low");
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#endif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。

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す。
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を取るのに役立ちます)
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化します。
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す。
force-native-property	3.1の互換性のために、すべてプロパティに:nativePropertyのタグ付けをします。
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す。
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'という名前を使用します。
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、デフォルトは1000)を設定します。
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を使用します。
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サフィックスのついたものに置き換えます。
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と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。

```
1 import haxe.extern.Rest;
2 import haxe.extern.EitherType;
3
```

```

4 extern class MyExtern {
5     static function f1(s:String, r:Rest<Int>):Void;
6     static function f2(e:EitherType<Int, String>):Void;
7 }
8
9 class Main {
10     static function main() {
11         MyExtern.f1("foo", 1, 2, 3); // use 1, 2, 3 as rest argument
12         MyExtern.f1("foo"); // no rest argument
13         //MyExtern.f1("foo", "bar"); // String should be Int
14
15         MyExtern.f2("foo");
16         MyExtern.f2(12);
17         //MyExtern.f2(true); // Bool should be EitherType<Int, String>
18     }
19 }

```

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。

```

1 using Main.IntExtender;
2
3 class IntExtender {
4     static public function triple(i:Int) {
5         return i * 3;
6     }
7 }
8
9 class Main {
10     static public function main() {
11         trace(12.triple());
12     }
13 }

```

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。

```

1 enum Tree<T> {
2     Leaf(v:T);
3     Node(l:Tree<T>, r:Tree<T>);
4 }

```

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。

```

1 var myTree = Node(Leaf("foo"), Node(Leaf("bar"), Leaf("foobar")));
2 var match = switch(myTree) {
3     // matches any Leaf
4     case Leaf(_): "0";
5     // matches any Node that has r = Leaf
6     case Node(_, Leaf(_)): "1";
7     // matches any Node that has has
8     // r = another Node, which has
9     // l = Leaf("bar")
10    case Node(_, Node(Leaf("bar"), _)): "2";
11    // matches anything
12    case _: "3";
13 }
14 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。

```

1 var myTree = Node(Leaf("foo"), Node(Leaf("bar"), Leaf("foobar")));
2 var name = switch(myTree) {
3     case Leaf(s): s;
4     case Node(Leaf(s), _): s;
5     case _: "none";
6 }
7 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。

```

1  var node = switch(myTree) {
2      case Node(leafNode = Leaf("foo"), _): leafNode;
3      case x: x;
4  }
5  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。その他のケースでは、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。

```

1  var myStructure = {
2      name: "haxe",
3      rating: "awesome"
4  };
5  var value = switch(myStructure) {
6      case { name: "haxe", rating: "poor" }:
7          throw false;
8      case { rating: "awesome", name: n }:
9          n;
10     case _:
11         "no awesome language found";
12 }
13 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。

```

1  var myArray = [1, 6];
2  var match = switch(myArray) {
3      case [2, _]: "0";
4      case [_ , 6]: "1";
5      case []: "2";
6      case [_ , _ , _]: "3";
7      case _: "4";
8  }
9  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。

```
1  var match = switch(7) {
2      case 4 | 1: "0";
3      case 6 | 7: "1";
4      case _: "2";
5  }
6  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。

```
1  var myArray = [7, 6];
2  var s = switch(myArray) {
3      case [a, b] if (b > a):
4          b + ">" + a;
5      case [a, b]:
6          b + "<=" + a;
7      case _: "found something else";
8  }
9  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。

```
1  var s = switch [1, false, "foo"] {
2      case [1, false, "bar"]: "0";
3      case [_, true, _]: "1";
4      case [_, false, _]: "2";
5  }
6  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。

```
1 enum Test {
2     TString(s:String);
3     TInt(i:Int);
4 }
5
6 class Main {
7     static public function main() {
8         var e = TString("f0o");
9         switch(e) {
10             case TString(temp):
11                 switch(temp.toLowerCase()) {
12                     case "foo": true;
13                     case _: false;
14                 }
15             case _: false;
16         }
17     }
18 }
```

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。見てのとおりに、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。

```
1 enum Test {
2     TString(s:String);
3     TInt(i:Int);
4 }
5
6 class Main {
7     static public function main() {
8         var e = TString("f0o");
9         var success = switch(e) {
10             case TString(_.toLowerCase() => "foo"):
11                 true;
12             case _:
13                 false;
14         }
15     }
16 }
```

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。

```
1 class Main {
2   static public function main() {
3     switch(3) {
4       case add(_, 1) => mul(_, 3) => a:
5         trace(a);
6     }
7   }
8
9   static function add(i1:Int, i2:Int) {
10    return i1 + i2;
11  }
12
13  static function mul(i1:Int, i2:Int) {
14    return i1 * i2;
15  }
16 }
```

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。

```
1 class Main {
2   static public function main() {
3     switch("foo") {
4       // Extractors in or patterns are not allowed
5       case (_.toLowerCase() => "foo") | "bar":
6     }
7   }
8 }
```

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、上の例は小さく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
```

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
4 case Node(l, r):
5 case _: // This pattern is unused
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てくれます。

```

1 var x = 12;
2 // The value of x is 12
3 trace('The value of x is $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 // The sum of 12 and 3 is 15
3 trace('The sum of $x and 3 is ${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("The sum of " + x +
2 " and 3 is " + (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`のマクロが使われていました。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。

```

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

```

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。

```
1 var a = [];  
2 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。

```
1 class Main {  
2   static public function main() {  
3     var a = [  
4       for (a in 1...11)  
5         for(b in 2...4)  
6           if (a % b == 0)  
7             a+ "/" +b  
8     ];  
9     // [2/2,3/3,4/2,6/2,6/3,8/2,9/3,10/2]  
10    trace(a);  
11  }  
12 }
```

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。

```
1 typedef Iterator<T> = {  
2   function hasNext() : Bool;  
3   function next() : T;  
4 }  
5  
6 typedef Iterable<T> = {  
7   function iterator() : Iterator<T>;  
8 }
```

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。

```
1 class MyStringIterator {  
2   var s:String;  
3   var i:Int;  
4  
5   public function new(s:String) {  
6     this.s = s;  
7     i = 0;  
8   }  
9  
10  public function hasNext() {  
11    return i < s.length;
```



```

12     }
13
14     public function next() {
15         return s.charAt(i++);
16     }
17 }
18
19 class Main {
20     static public function main() {
21         var myIt = new MyStringIterator("string");
22         for (chr in myIt) {
23             trace(chr);
24         }
25     }
26 }

```

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。

```

1 class MyArrayWrap<T> {
2     var a:Array<T>;
3     public function new(a:Array<T>) {
4         this.a = a;
5     }
6
7     public function iterator() {
8         return a.iterator();
9     }
10 }
11
12 class Main {
13     static public function main() {
14         var myWrap = new MyArrayWrap([1, 2, 3]);
15         for (elt in myWrap) {
16             trace(elt);
17         }
18     }
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出すことができます。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`型の引数を取り除かれたので、`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しました。

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。

```

1 import haxe.rtti.Meta;
2
3 @author("Nicolas")
4 @debug
5 class MyClass {
6     @range(1, 8)
7     var value: Int;
8
9     @broken
10    @:noCompletion

```

```

11  static function method() { }
12  }
13
14  class Main {
15  static public function main() {
16      // { author : ["Nicolas"], debug : null }
17      trace(Meta.getType(MyClass));
18      // [1,8]
19      trace(Meta.getFields(MyClass).value.range);
20      // { broken: null }
21      trace(Meta.getStatics(MyClass).method);
22  }
23  }

```

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カッコで囲まれている、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たつかもかもしれません。

```
1 class MyClass {
2     static private var foo: Int;
3 }
4
5 class Main {
6     @:access(MyClass.foo)
7     static public function main() {
8         MyClass.foo;
9     }
10 }
```

`@:access(MyClass.foo)`のメタデータは`main`メソッドからの`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用しました。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います。
- ・ 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。

```
1 class MyClass {
2     @:allow(I)
3     static private var foo: Int;
4 }
5
6 interface I { }
7
8 class Main implements I {
9     static public function main() {
10         MyClass.foo;
11     }
12 }
```

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た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

Compiler Reference

章 7

Compiler Usage

Basic Usage The Haxe Compiler is typically invoked from command line with several arguments which have to answer two questions:

- What should be compiled?
- What should the output be?

To answer the first question, it is usually sufficient to provide a class path via the `-cp path` argument, along with the main class to be compiled via the `-main dot_path` argument. The Haxe Compiler then resolves the main class file and begins compilation.

The second question usually comes down to providing an argument specifying the desired target. Each Haxe target has a dedicated command line switch, such as `-js file_name` for JavaScript and `-php directory` for PHP. Depending on the nature of the target, the argument value is either a file name (for `-js`, `-swf` and `neko`) or a directory path.

Common arguments Input:

`-cp path` Adds a class path where `.hx` source files or packages (sub-directories) can be found.

`-lib library_name` Adds a [Haxelib](#) (章 11) library.

`-main dot_path` Sets the main class.

Output:

`-js file_name` Generates JavaScript ([12.1](#)) source code in specified file.

`-as3 directory` Generates ActionScript 3 source code in specified directory.

`-swf file_name` Generates the specified file as Flash ([12.2](#)) `.swf`.

`-neko file_name` Generates Neko ([12.3](#)) binary as specified file.

`-php directory` Generates PHP ([12.4](#)) source code in specified directory.

`-cpp directory` Generates C++ ([12.5](#)) source code in specified directory and compiles it using native C++ compiler.

`-cs directory` Generates C# ([12.7](#)) source code in specified directory.

- `java directory` Generates Java ([12.6](#)) source code in specified directory and compiles it using the Java Compiler.
- `python file_name` Generates Python ([12.8](#)) source code in the specified file.

章 8

Compiler Features

8.1 Built-in Compiler Metadata

Starting from Haxe 3.0, you can get the list of defined compiler metadata by running `haxe --help-metas`

Glossary	
Metadata	Description
@:abi	Function ABI/calling convention
@:abstract	Sets the underlying class implementation
@:access _(Target path)_	Forces private access to package type
@:allow _(Target path)_	Allows private access from package type
@:analyzer	Used to configure the static analyzer
@:annotation	Annotation (@interface) definitions on
@:arrayAccess	Allows Array access (2.8.3) on an abs
@:autoBuild _(Build macro call)_	Extends @:build metadata to all exten
@:bind	Override Swf class declaration
@:bitmap _(Bitmap file path)_	_Embeds given bitmap data into the c
@:bridgeProperties	Creates native property bridges for al
@:build _(Build macro call)_	Builds a class or enum from a macro.
@:buildXml	Specify xml data to be injected into B
@:callable	Abstract forwards call to its underlyin
@:classCode	Used to inject platform-native code in
@:commutative	Declares an abstract operator as comm
@:compilerGenerated	Marks a field as generated by the com
@:coreApi	Identifies this class as a core api class
@:coreType	Identifies an abstract as core type (2.
@:cppFileCode	Code to be injected into generated cp
@:cppInclude	File to be included in generated cpp f
@:cppNamespaceCode	
@:dce	Forces Dead Code Elimination (8.2) e
@:debug	Forces debug information to be gener
@:decl	
@:defParam	
@:delegate	Automatically added by -net-lib on c
@:depend	
@:deprecated	Automatically added by -java-lib on
@:event	Automatically added by -net-lib on e
@:enum	Defines finite value sets to abstract d
@:expose _(?Name=Class path)_	Makes the class available on the wind
@:extern	Marks the field as extern so it is not g
@:fakeEnum _(Type name)_	Treat enum as collection of values of
@:file(File path)	Includes a given binary file into the ta
@:final	Prevents a class from being extended
@:font _(TTF path Range String)_	Embeds the given TrueType font into
@:forward _(List of field names)_	Forwards field access (2.8.6) to under
@:from	Specifies that the field of the abstract
@:functionCode	
@:functionTailCode	
@:generic	Marks a class or class field as generic
@:genericBuild	Builds instances of a type using the s
@:getter _(Class field name)_	Generates a native getter function on
@:hack	Allows extending classes marked as @
@:headerClassCode	Code to be injected into the generate
@:headerCode	Code to be injected into the generate
@:headerNamespaceCode	
@:hxGen	Annotates that an extern class was ge
@:ifFeature _(Feature name)_	Causes a field to be kept by DCE (8.2
@:include	
@:initPackage	
@:internal	Generates the annotated field/class w
@:isVar	Forces a physical field to be generate
@:javaCanonical _(Output type package,Output type name)_	Used by the Java target to annotate th
@:jsRequire	Generate javascript module require e
@:keep	Causes a field or type to be kept by D
@:keepInit	Causes a class to be kept by DCE (8.2
@:keepSub	Extends @:keep metadata to all impl

8.2 Dead Code Elimination

Dead Code Elimination or DCE is a compiler feature which removes unused code from the output. After typing, the compiler evaluates the DCE entry-points (usually the main-method) and recursively determines which fields and types are used. Used fields are marked accordingly and unmarked fields are then removed from their classes.

DCE has three modes which are set when invoking the command line:

- dce std: Only classes in the Haxe Standard Library are affected by DCE. This is the default setting on all targets.
- dce no: No DCE is performed.
- dce full: All classes are affected by DCE.

The DCE-algorithm works well with typed code, but may fail when dynamic (2.7) or reflection (10.7) is involved. This may require explicit marking of fields or classes as being used by attributing the following metadata:

@:keep: If used on a class, the class along with all fields is unaffected by DCE. If used on a field, that field is unaffected by DCE.

@:keepSub: If used on a class, it works like **@:keep** on the annotated class as well as all subclasses.

@:keepInit: Usually, a class which had all fields removed by DCE (or is empty to begin with) is removed from the output. By using this metadata, empty classes are kept.

If a class needs to be marked with **@:keep** from the command line instead of editing its source code, there is a compiler macro available for doing so: `--macro keep('type dot path')` See the [haxe.compiler.keep API](#) for details of this macro. It will mark package, module or sub-type to be kept by DCE and includes them for compilation.

The compiler automatically defines the flag `dce` with a value of either `"std"`, `"no"` or `"full"` depending on the active mode. This can be used in conditional compilation (6.1).

トリビア: DCE-rewrite

DCE was originally implemented in Haxe 2.07. This implementation considered a function to be used when it was explicitly typed. The problem with that was that several features, most importantly interfaces, would cause all class fields to be typed in order to verify type-safety. This effectively subverted DCE completely, prompting the rewrite for Haxe 2.10.

トリビア: DCE and try.haxe.org

DCE for the JavaScript target saw vast improvements when the website <http://try.haxe.org> was published. Initial reception of the generated JavaScript code was mixed, leading to a more fine-grained selection of which code to eliminate.

8.3 Completion

8.3.1 Overview

The rich type system (3) of the Haxe Compiler makes it difficult for IDEs and editors to provide accurate completion information. Between type inference (3.6) and macros (9), it

would require a substantial amount of work to replicate the required processing. This is why the Haxe Compiler comes with a built-in completion mode for third-party software to use.

All completion is triggered using the `--display file@position[@mode]` compiler argument. The required arguments are:

file: The file to check for completion. This must be an absolute or relative path to a .hx file. It does not respect any class paths or libraries.

position: The byte position (not character position) of where to check for completion in the given file.

mode: The completion mode to use (see below).

We will look into the following completion modes in detail:

Field access (8.3.2): Provides a list of fields that can be accessed on a given type.

Call argument (8.3.3): Reports the type of the function which is currently being called.

Type path (8.3.4): Lists sub-packages, sub-types and static fields.

Usage (8.3.5): Lists all occurrences of a given type, field or variable in all compiled files. (mode: "usage")

Position (8.3.6): Reports the position of where a given type, field or variable is defined. (mode: "position")

Top-level (8.3.7): Lists all identifiers which are available at a given position. (mode: "toplevel")

Due to Haxe being a very fast compiler, it is often sufficient to rely on the normal compiler invocation for completion. For bigger projects Haxe provides a server mode (8.3.8) which ensures that only those files are re-compiled that actually changed or had any of their dependencies changes.

General notes on the interface

- The position-argument can be set to 0 if the file in question contains a pipeline | character at the position of interest. This is very useful for demonstration and testing as it allows us to ignore the byte-counting process a real IDE would have to do. The examples in this section are making use of this feature. Note that this only works in places where | is not valid syntax otherwise, e.g. after dots (.) and opening parentheses (().
- The output is HTML-escaped so that &, < and > become &, < and > respectively.
- Otherwise any documentation output is preserved which means longer documentation might include new-line and tab-characters as it does in the source files.
- When run in completion mode, the compiler does not display errors but instead tries to ignore them or recover from them. If a critical error occurs while getting completion, the Haxe Compiler prints the error message instead of the completion output. Any non-XML output can be treated as a critical error message.

8.3.2 Field access completion

Field completion is triggered after a dot `.` character to list the fields that are available on the given type. The compiler parses and types everything up to the point of completion and then outputs the relevant information to stderr:

```
1 class Main {
2   public static function main() {
3     trace("Hello".|
4   }
5 }
```

If this file is saved to `Main.hx`, the completion can be invoked using the command `haxe --display Main.hx@0`. The output looks similar to this (we omit several fields for brevity and improve the formatting for readability):

```
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<Int>&Int</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>
```

The XML structure follows:

- The document node `list` encloses several nodes `i`, each representing a field.
- The `n` attribute contains the name of the field.
- The `t` node contains the type of the field.
- the `d` node contains the documentation of the field.

[Haxe 3.2.0から]

When compiling with `-D display-details`, each field additionally has a `k` attribute which can either be `var` or `method`. This allows distinguishing method fields from variable fields that have a function type.

8.3.3 Call argument completion

Call argument completion is triggered after an opening parenthesis character `(` to show the type of the function that is currently being called. It works for any function call as well as constructor calls:

```
1 class Main {
2   public static function main() {
3     trace("Hello".split(|
4   }
5 }
```

If this file is saved to `Main.hx`, the completion can be invoked using the command `haxe --display Main.hx@0`. The output looks like this:

```
1 <type>
2 delimiter : String -&gt; Array<String>;
3 </type>
```

IDEs can parse this to recognize that the called function requires one argument named `delimiter` of type `String` and returns an `Array<String>`.

トリビア: Problems with the output structure

We acknowledge that the current format requires a bit of manual parsing which can be annoying. In the future we might look into providing a more structured output, especially for functions.

8.3.4 Type path completion

Type path completion can occur in import declarations ([3.7.2](#)), using declarations ([6.3](#)) or any place a type is referenced. We can identify three different cases:

package completion This lists all sub-packages of the `haxe` package as well as all modules in that package:

```
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>
15 <i n="Template"><t></t><d></d></i>
16 <i n="Timer"><t></t><d></d></i>
17 <i n="Ucs2"><t></t><d></d></i>
18 <i n="Unserializer"><t></t><d></d></i>
19 <i n="Utf8"><t></t><d></d></i>
20 <i n="crypto"><t></t><d></d></i>
21 <i n="ds"><t></t><d></d></i>
22 <i n="extern"><t></t><d></d></i>
23 <i n="format"><t></t><d></d></i>
24 <i n="io"><t></t><d></d></i>
25 <i n="macro"><t></t><d></d></i>
26 <i n="remoting"><t></t><d></d></i>
27 <i n="rtti"><t></t><d></d></i>
28 <i n="unit"><t></t><d></d></i>
29 <i n="web"><t></t><d></d></i>
30 <i n="xml"><t></t><d></d></i>
31 <i n="zip"><t></t><d></d></i>
32 </list>

```

import module completion This lists all sub-types (3.7.1) of the module `haxe.Unserializer` as well as all its public static fields (because these can be imported too):

```

1 import haxe.Unserializer. |

1 <list>
2 <i n="DEFAULT_RESOLVER">
3   <t>haxe.TypeResolver</t>
4   <d>
5     This value can be set to use custom type resolvers.
6
7     A type resolver finds a Class or Enum instance from a given String
8     By default, the haxe Type Api is used.
9
10    A type resolver must provide two methods:
11
12    1. resolveClass(name:String):Class<Dynamic>; is called to
13       determine a Class from a class name
14    2. resolveEnum(name:String):Enum<Dynamic>; is called to
15       determine an Enum from an enum name
16
17    This value is applied when a new Unserializer instance is created.
18    Changing it afterwards has no effect on previously created
19    instances.
20   </d>
21 </i>
22 <i n="run">
23   <t>v : String -> Dynamic</t>

```



```

24 <d>
25     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. |
```

other module completion This lists all sub-types (3.7.1) of the module haxe.Unserializer:

```
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 Usage completion

【Haxe 3.2.0から】

Usage completion is enabled by using the "usage" mode argument (see [Overview \(節 8.3.1\)](#)). We demonstrate it here using a local variable. Note that it would work with fields and types the same way:

```

1 class Main {
2     public static function main() {
3         var a = 1;
4         var b = a + 1;
5         trace(a);
6         a. |
7     }
8 }

```

If this file is saved to Main.hx, the completion can be invoked using the command `haxe --display Main.hx@0@usage`. The output looks like this:

```

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 Position completion

【Haxe 3.2.0から】

Position completion is enabled by using the "position" mode argument (see [Overview \(節 8.3.1\)](#)). We demonstrate it using a field. Note that it would work with local variables and types the same way:

```

1 class Main {
2     static public function main() {
3         "foo".split.|
4     }

```

If this file is saved to Main.hx, the completion can be invoked using the command `haxe --display Main.hx@0@position`. The output looks like this:

```

1 <list>
2 <pos>std/string.hx:124: characters 1-54</pos>
3 </list>

```

トリビア: Effects of omitting a target specifier

In this example the compiler reports the standard String.hx definition which does not actually have an implementation. This happens because we did not specify any target, which is allowed in completion-mode. If the command line arguments included, say, `-neko neko.n`, the reported position would instead be `std/ neko/_std/ string.hx: 84: lines 84-98`.

8.3.7 Top-level completion

【Haxe 3.2.0から】

Top-level completion displays all identifiers which the Haxe Compiler knows about at a given compilation position. This is the only completion method for which we need a real position argument in order to demonstrate its effect:

```

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 }

```

If this file is saved to Main.hx, the completion can be invoked using the command `haxe --display Main.hx@63@toplevel`. The output looks similar to this (we omit several entries for brevity):

```

1 <i l>
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>
6 <i k="package">sys</i>
7 <i k="package">haxe</i>
8 <i k="type" p="Int">Int</i>
9 <i k="type" p="Float">Float</i>
10 <i k="type" p="MyEnum">MyEnum</i>
11 <i k="type" p="Main">Main</i>
12 </i l>

```

The structure of the XML depends on the `k` attribute of each entry. In all cases the node value of the `i` nodes contains the relevant name.

local, member, static, enum, global: The `t` attribute holds the type of the variable or field.

global, type: The `p` attribute holds the path of the module which contains the type or field.

8.3.8 Completion server

To get the best speed for both compilation and completion, you can use the `--wait` commandline parameter to start a Haxe compilation server. You can also use `-v` to have the server print the log. Here's an example:

```
1 haxe -v --wait 6000
```

You can then connect to the Haxe server, send commandline parameters followed by a 0 byte and, then, read the response (either completion result or errors).

Use the `--connect` commandline parameter to have Haxe send its compilation commands to the server instead of executing them directly :

```
1 haxe --connect 6000 myproject.hxml
```

Please note that you can use the parameter `--cwd` at the first sent command line to change the Haxe server's current working directory. Usually class paths and other files are relative to your project.

How it works The compilation server will cache the following things:

parsed files the files will only get parsed again if they are modified or if there was a parse error

haxelib calls the previous results of haxelib calls will be reused (only for completion : they are ignored when doing a compilation)

typed modules compilation modules will be cached after a successful compilation and can be reused in later compilation/completions if none of their dependencies have been modified

You can get precise reading of the times spent by the compiler and how using the compilation server affects them by adding `--times` to the command line.

Protocol As the following Haxe/ Neko example shows, you can simply connect on the server port and send all commands (one per line) ending with a 0 binary char. You can, then, read the results.

Macros and other commands can log events which are not errors. From the command line, we can see the difference between what is printed on `stdout` and what is print on `stderr`. This is not the case in socket mode. In order to differentiate between the two, log messages (not errors) are prefixed with a `x01` character and all newline-characters in the message are replaced by the same `x01` character.

Warnings and other messages can also be considered errors, but are not fatal ones. If a fatal error occurred, it will send a single `x02` message-line.

Here's some code that will treat connection to the server and handle the protocol details:

```
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.html" + 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("%
16                    textbackslash% x01").join(newline));
17                case 0x02:
18                    hasError = true;
19                default:
20                    neko.io.File.stderr().writeString(line + newline);
21            }
22        }
23        if (hasError) neko.Sys.exit(1);
24    }
25 }
```

Effect on macros The compilation server can have some side effects on macro execution (9).

8.4 Resources

Haxe provides a simple resource embedding system that can be used for embedding files directly into the compiled application.

While it may be not optimal to embed large assets such as images or music in the application file, it comes in very handy for embedding smaller resources like configuration or

XML data.

8.4.1 Embedding resources

External files are embedded using the `-resource` compiler argument:

```
-resource hello_message.txt@welcome
```

The string after the `@` symbol is the resource identifier which is used in the code for retrieving the resource. If it is omitted (together with the `@` symbol) then the file name will become the resource identifier.

8.4.2 Retrieving text resources

To retrieve the content of an embedded resource we use the static method `getString` of `haxe.Resource`, passing a resource identifier to it:

```
1 class Main {  
2     static function main() {  
3         trace(haxe.Resource.getString("welcome"));  
4     }  
5 }
```

The code above will display the content of the `hello_message.txt` file that we included earlier using `welcome` as the identifier.

8.4.3 Retrieving binary resources

While it's not recommended to embed large binary files in the application, it still may be useful to embed binary data. The binary representation of an embedded resource can be accessed using the static method `getBytes` of `haxe.Resource`:

```
1 class Main {  
2     static function main() {  
3         var bytes = haxe.Resource.getBytes("welcome");  
4         trace(bytes.readString(0, bytes.length));  
5     }  
6 }
```

The return type of `getBytes` method is `haxe.io.Bytes`, which is an object providing access to individual bytes of the data.

8.4.4 Implementation details

Haxe uses the target platform's native resource embedding if there is one, otherwise it provides its own implementation.

- Flash resources are embedded as `ByteArray` definitions
- C# resources are included in the compiled assembly
- Java resources are packed in the resulting JAR file
- C++ resources are stored in global byte array constants.

what to use for listing of non-haxe code like hxml?

- JavaScript resources are serialized in Haxe serialization format and stored in a static field of `haxe.Resource` class.
- Neko resources are stored as strings in a static field of `haxe.Resource` class.

8.5 Runtime Type Information

The Haxe compiler generates runtime type information (RTTI) for classes that are annotated or extend classes that are annotated with the `:rtti` metadata. This information is stored as a XML string in a static field `__rtti` and can be processed through `haxe.rtti.XmlParser`. The resulting structure is described in [RTTI structure](#) (節 8.5.1).
【Haxe 3.2.0から】

The type `haxe.rtti.Rtti` has been introduced in order to simplify working with RTTI. Retrieving this information is now very easy:

```

1 @:rtti
2 class Main {
3     var x:String;
4     static function main() {
5         var rtti = haxe.rtti.Rtti.getRtti(Main);
6         trace(rtti);
7     }
8 }
```

8.5.1 RTTI structure

General type information

path: The type path (3.7) of the type.

module: The type path of the module (3.7) containing the type.

file: The full slash path of the `.hx` file containing the type. This might be `null` in case there is no such file, e.g. if the type is defined through a macro (9).

params: An array of strings representing the names of the type parameters (3.2) the type has. As of Haxe 3.2.0, this does not include the constraints (3.2.1).

doc: The documentation of the type. This information is only available if the compiler flag (6.1) `-D use_rtti_doc` was in place. Otherwise, or if the type has no documentation, the value is `null`.

isPrivate: Whether or not the type is private (3.7.1).

platforms: A list of strings representing the targets where the type is available.

meta: The meta data the type was annotated with.

Class type information

isExtern: Whether or not the class is extern (6.2).

isInterface: Whether or not the class is actually an interface (2.3.3).

superClass: The class' parent class defined by its type path and list of type parameters.

interfaces: The list of interfaces defined by their type path and list of type parameters.

fields: The list of member class fields (4), described in [Class field information](#) (節 8.5.1).

statics: The list of static class fields, described in [Class field information](#) (節 8.5.1).

tdynamic: The type which is dynamically implemented (2.7.2) by the class or `null` if no such type exists.

Enum type information

isExtern: Whether or not the enum is extern (6.2).

constructors: The list of enum constructors.

Abstract type information

to: An array containing the defined implicit to casts (2.8.1).

from: An array containing the defined implicit from casts (2.8.1).

impl: The class type information (8.5.1) of the implementing class.

athis: The underlying type (2.8) of the abstract.

Class field information

name: The name of the field.

type: The type of the field.

isPublic: Whether or not the field is public (4.4.1).

isOverride: Whether or not the field overrides (4.4.4) another field.

doc: The documentation of the field. This information is only available if the compiler flag (6.1) `-D use_rtti_doc` was in place. Otherwise, or if the field has no documentation, the value is `null`.

get: The read access behavior (4.2) of the field.

set: The write access behavior (4.2) of the field.

params: An array of strings representing the names of the type parameters (3.2) the field has. As of Haxe 3.2.0, this does not include the constraints (3.2.1).

platforms: A list of strings representing the targets where the field is available.

meta: The meta data the field was annotated with.

line: The line number where the field is defined. This information is only available if the field has an expression. Otherwise the value is `null`.

overloads: The list of available overloads for the fields or `null` if no overloads exists.

Enum constructor information

name: The name of the constructor.

args: The list of arguments the constructor has or `null` if no arguments are available.

doc: The documentation of the constructor. This information is only available if the compiler flag (6.1) `-D use_rtti_doc` was in place. Otherwise, or if the constructor has no documentation, the value is `null`.

platforms: A list of strings representing the targets where the constructor is available.

meta: The meta data the constructor was annotated with.

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.

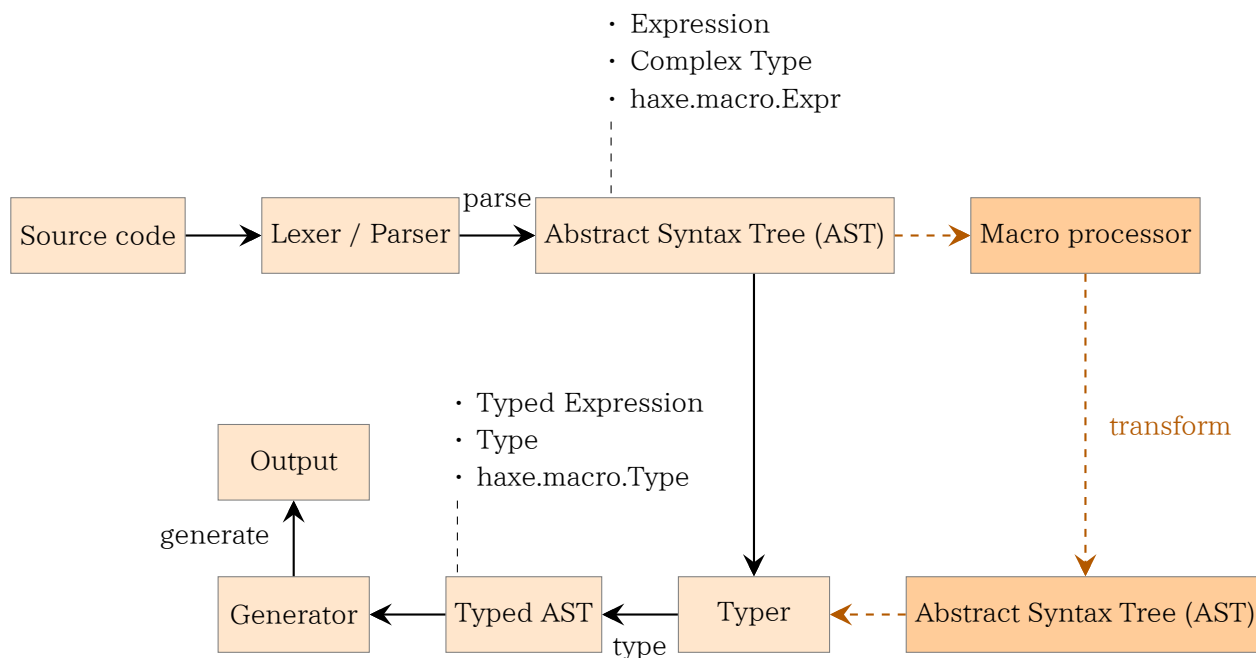


Figure 9.1: The role of macros during compilation.

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 has no field identity
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:

```
1 class Main {
2     static public function main() {
3         const("foo", 1, 1.5, true);
4     }
5
6     macro static function const(s:String, i:Int, f:Float, b:Bool) {
7         trace(s);
8         trace(i);
9         trace(f);
10        trace(b);
11        return macro null;
12    }
13 }
```

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:

```
1 import haxe.macro.Expr;
2
3 class Main {
4     static public function main() {
5         myMacro("foo", a, b, c);
6     }
7
8     macro static function myMacro(e1:Expr, extra:Array<Expr>) {
9         for (e in extra) {
10             trace(e);
11         }
12         return macro null;
13     }
14 }
```

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:

```
1 class Main {
2     macro static function generateClass(funcName:String) {
3         var c = macro class MyClass {
4             public function new() { }
5             public function $funcName() {
6                 trace($v{funcName} + " was called");
7             }
8         }
9         haxe.macro.Context.defineType(c);
10        return macro new MyClass();
11    }
12
13    public static function main() {
14        var c = generateClass("myFunc");
15        c.myFunc();
16    }
17 }
```

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 tinkerbell¹ 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.

```
1 import haxe.macro.Context;
2 import haxe.macro.Expr;
3
4 class TypeBuildingMacro {
5     macro static public function build(fieldName:String):Array<Field> {
6         var fields = Context.getBuildFields();
7         var newField = {
8             name: fieldName,
9             doc: null,
10            meta: [],
11            access: [AStatic, APublic],
12            kind: FVar(macro : String, macro "my default"),
13            pos: Context.currentPos()
14        };
15        fields.push(newField);
16        return fields;
17    }
18 }
```

```
1 @:build(TypeBuildingMacro.build("myFunc"))
2 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        }

```



```

12     var intArg = makeEnumField("B", FFun(fInt));
13     return [noArgs, intArg];
14 }
15
16 static function makeEnumField(name, kind) {
17     return {
18         name: name,
19         doc: null,
20         meta: [],
21         access: [],
22         kind: kind,
23         pos: Context.currentPos()
24     }
25 }
26 }

```

```

1 @:build(EnumBuildingMacro.build())
2 enum E { }
3
4 class Main {
5     static public function main() {
6         switch(E.A) {
7             case A:
8             case B(v):
9         }
10    }
11 }

```

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:

```

1 enum E {
2     A;
3     B(value: Int);
4 }

```

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.

```

1 import haxe.macro.Context;
2 import haxe.macro.Expr;
3
4 class AutoBuildingMacro {
5     macro static public

```

```

6     function fromInterface():Array<Field> {
7         trace("fromInterface: " + Context.getLocalType());
8         return null;
9     }
10
11     macro static public
12     function fromBaseClass():Array<Field> {
13         trace("fromBaseClass: " + Context.getLocalType());
14         return null;
15     }
16 }

```

```

1 @:autoBuild(AutoBuildingMacro.fromInterface())
2 interface I { }
3
4 interface I2 extends I { }
5
6 @:autoBuild(AutoBuildingMacro.fromBaseClass())
7 class Base { }
8
9 class Main extends Base implements I2 {
10     static public function main() { }
11 }

```

This outputs during compilation:

```

1 AutoBuildingMacro.hx:6:
2   fromInterface: TInst(I2,[])
3 AutoBuildingMacro.hx:6:
4   fromInterface: TInst(Main,[])
5 AutoBuildingMacro.hx:11:
6   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:

```

1 import haxe.macro.Expr;
2 import haxe.macro.Context;
3 import haxe.macro.Type;
4
5 class GenericBuildMacro1 {
6     static public function build() {

```

```

7     switch (Context.getLocalType()) {
8         case TInst(_, [t1]):
9             trace(t1);
10        case t:
11            Context.error("Class expected", Context.currentPos());
12        }
13        return null;
14    }
15 }

```

```

1 @:genericBuild(GenericBuildMacro1.build())
2 class MyType<T> { }
3
4 class Main {
5     static function main() {
6         var x:MyType<Int>;
7         var x:MyType<String>;
8     }
9 }

```

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:

```

1 import haxe.macro.Expr;
2 import haxe.macro.Context;
3 import haxe.macro.Type;
4
5 class GenericBuildMacro2 {
6     static public function build() {
7         switch (Context.getLocalType()) {
8             case TInst(_, [TInst(_.get() => { kind: KExpr(macro $v{(s:String)
9                 }) }, _)]):
10                trace(s);
11            case t:
12                Context.error("Class expected", Context.currentPos());
13        }
14        return null;
15    }
16 }

```

```

1 @:genericBuild(GenericBuildMacro2.build())

```

```

2 class MyType<Const> { }
3
4 class Main {
5     static function main() {
6         var x:MyType<"myfile.txt">;
7     }
8 }

```

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 of 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:

```

1 import haxe.macro.Context;
2 import haxe.macro.Expr;
3
4 using Main;
5 using haxe.macro.Tools;
6
7 class Main {
8     static public function main() {
9         #if !macro
10         var a = "foo";
11         a.test();
12         #end
13     }
14
15     macro static function test(e:ExprOf<String>) {
16         trace(e.toString()); // @:this this
17         // TInst(String,[])
18         trace(Context.typeof(e));
19         return e;
20     }
21 }

```

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

Standard Library

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:

```
1 "#".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](#)):

```
1 class Main {  
2   static public function main() {  
3     var a = [1, 2, 3];  
4     trace(a[1]); // 2  
5     a[1] = 1;  
6     trace(a[1]); // 1  
7   }  
8 }
```


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:

```
1 class Main {
2   static public function main() {
3     var scores = [110, 170, 35];
4     var sum = 0;
5     for (score in scores) {
6       sum += score;
7     }
8     trace(sum); // 315
9   }
10 }
```

Haxe generates this optimized JavaScript output:

```
1 Main.main = function() {
2   var scores = [110, 170, 35];
3   var sum = 0;
4   var _g = 0;
5   while(_g < scores.length) {
6     var score = scores[_g];
7     ++_g;
8     sum += score;
9   }
10  console.log(sum);
11 };
```

Haxe does not allow arrays of mixed types unless the parameter type is forced to `Dynamic` (2.7):

```
1 class Main {
2   static public function main() {
3     // Compile Error: Arrays of mixed types are only allowed if the
4     // type is
5     // forced to Array<Dynamic>
6     //var myArray = [10, "Bob", false];
7
8     // Array<Dynamic> with mixed types
9     var myExplicitArray:Array<Dynamic> = [10, "Sally", true];
10  }
11 }
```

トリア: 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)
```

```

5     myList.add(ii);
6     trace(myList); //{0, 1, 2, 3, 4}
7 }
8 }

```

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`.

```

1 import haxe.ds.GenericStack;
2
3 class Main {
4     static public function main() {
5         var myStack = new GenericStack<Int>();
6         for (ii in 0...5)
7             myStack.add(ii);
8         trace(myStack); //{4, 3, 2, 1, 0}
9         trace(myStack.pop()); //4
10    }
11 }

```

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:

```

1 class Main {
2     static public function main() {
3         // Maps are initialized like arrays, but
4         // use the map literal syntax with the
5         // '=>' operator. Maps can have their
6         // key value types defined explicitly
7         var map1:Map<Int, String> =
8             [1 => "one", 2=>"two"];
9
10        // Or they can infer their key value types
11        var map2 = [

```

```

12     "one"=>1,
13     "two"=>2,
14     "three"=>3
15 ];
16 $type(map2); // Map<String, Int>
17
18 // Keys must be unique
19 // Error: Duplicate Key
20 //var map3 = [1=>"dog", 1=>"cat"];
21
22 // Maps values can be accessed using array
23 // accessors "[]"
24 var map4 = ["M"=>"Monday", "T"=>"Tuesday"];
25 trace(map4["M"]); //Monday
26
27 // Maps iterate over their values by
28 // default
29 var valueSum;
30 for (value in map4) {
31     trace(value); // Monday ¶n Tuesday
32 }
33
34 // Can iterate over keys by using the
35 // keys() method
36 for (key in map4.keys()) {
37     trace(key); // M ¶n T
38 }
39
40 // Like arrays, a new Map can be made using
41 // comprehension
42 var map5 = [
43     for (key in map4.keys())
44         key => "FRIDAY!!"
45 ];
46 // {M => FRIDAY!!, T => FRIDAY!!}
47 trace(map5);
48 }
49 }

```

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:

```
1 enum Option<T> {  
2     Some(v:T);  
3     None;  
4 }
```

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:

```
1 import haxe.ds.Option;  
2  
3 class Main {  
4     static public function main() {  
5         var result = trySomething();  
6         switch (result) {  
7             case None:  
8                 trace("Got None");  
9             case Some(s):  
10                 trace("Got a value: " +s);  
11         }  
12     }  
13  
14     static function trySomething():Option<String> {  
15         if (Math.random() > 0.5) {  
16             return None;  
17         } else {  
18             return Some("Success");  
19         }  
20     }  
21 }
```

10.3 Regular Expressions

Haxe has built-in support for regular expressions¹. 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:

```
1 var r = ~/haxe/i;
```

Alternatively, we can create regular expression with regular syntax:

```
1 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

¹http://en.wikipedia.org/wiki/Regular_expression

- * repeat zero-or-more
- + repeat one-or-more
- ? optional zero-or-one
- [A-Z0-9] character ranges
- [^rnrnt] 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:

```
1 ~/[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:

```
1 class Main {
2     static function main() {
3         var r = ~/world/;
4         var str = "hello world";
5         // true : 'world' was found in the string
6         trace(r.match(str));
7         trace(r.match("hello !")); // false
8     }
9 }
```

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:

```
1 class Main {
2     static function main() {
3         var str = "Nicolas is 26 years old";
4         var r =
5             ~/([A-Za-z]+) is ([0-9]+) years old/;
6         r.match(str);
7         trace(r.matched(1)); // "Nicolas"
8         trace(r.matched(2)); // "26"
9     }
10 }
```

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:

```
1 class Main {
2     static function main() {
3         var str = "abcdeeeefghi";
4         var r = ~/e+/;
5         r.match(str);
6         trace(r.matched(0)); // "eeeeee"
7         // { pos : 4, len : 5 }
8         trace(r.matchedPos());
9     }
10 }
```

Additionally, `r.matchedLeft()` and `r.matchedRight()` can be used to get substrings to the left and to the right of the matched substring:

```
1 class Main {
2     static function main() {
3         var r = ~/b/;
4         r.match("abc");
5         trace(r.matchedLeft()); // a
6         trace(r.matched(0)); // b
7         trace(r.matchedRight()); // c
8     }
9 }
```

10.3.3 Replace

A regular expression can also be used to replace a part of the string:

```
1 class Main {
2     static function main() {
3         var str = "aaabcbcbcbz";
4         // g : replace all instances
5         var r = ~/b[^c]/g;
```

```

6 // "aaabcbcbcx"
7 trace(r.replace(str, "xx"));
8 }
9 }

```

We can use `$X` to reuse a matched group in the replacement:

```

1 class Main {
2   static function main() {
3     var str = "{hello} {0} {again}";
4     var r = ~/{([a-z]+)}/g;
5     // "*hello* {0} *again*"
6     trace(r.replace(str, "*$1*"));
7   }
8 }

```

10.3.4 Split

A regular expression can also be used to split a string into several substrings:

```

1 class Main {
2   static function main() {
3     var str = "XaaaYababZbbbW";
4     var r = ~/[ab]+/g;
5     // ["X","Y","Z","W"]
6     trace(r.split(str));
7   }
8 }

```

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:

```

1 class Main {
2   static function main() {
3     var r = ~/ (dog|fox) /g;
4     var s = "The quick brown fox jumped over the lazy dog.";
5     var s2 = r.map(s, function(r) {
6       var match = r.matched(0);
7       switch (match) {
8         case 'dog': return 'fox';
9         case 'fox': return 'dog';
10        default: throw 'Unknown animal: $match';
11      };
12    });
13    trace(s2); // The quick brown dog jumped over the lazy fox.
14  }
15 }

```


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 is the natural logarithm
12        trace(Math.log(Math.exp(5))); //5
13
14        // Output for neko target, may vary
15        // depending on platform
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 mathematically 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 operations, 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 靜的擴張 (節 6.3) used with the math library. This code shows a simple example:

```
1 class MathStaticExtension {
2     /* Converts an angle in radians to degrees */
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,
11                       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
6         trace(numbers.count()); // 6
7         trace(numbers.has(4)); // false
8
9         // test if all numbers are greater/smaller than 20
10        trace(numbers.foreach(function(v) return v < 20)); // true
11        trace(numbers.foreach(function(v) return v > 20)); // false
12
13        // sum all the numbers
14        var sum = function(num, total) return total += num;
15        trace(numbers.fold(sum, 0)); // 30
16    }
17 }
```

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:

```
1 class Main {
2     static function main() {
3         var sample = "My name is <strong>::name::</strong>, <em>::age::</em> years old";
4         var user = {name:"Mark", age:30};
5         var template = new haxe.Template(sample);
6         var output = template.execute(user);
7         trace(output);
8     }
9 }
```

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::`.

```
1 ::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.

```
1 ::if (points == 10):: Great! ::end::
```

To compare to a string, use double quotes " in the template.

```
1 ::if (name == "Mark"):: Hi Mark ::end::
```

Iterating Iterate on a structure by using `::foreach::`. End the loop with `::end::`.

```
1 <table>
2   <tr>
3     <th>Name</th>
4     <th>Age</th>
5   </tr>
6   ::foreach users::
7     <tr>
8       <td>::name::</td>
9       <td>::age::</td>
10    </tr>
11  ::end::
12 </table>
```

Sub-templates To include templates in other templates, pass the sub-template result string as a parameter.

```
1 var users = [{name:"Mark", age:30}, {name:"John", age:45}];
2
3 var userTemplate = new haxe.Template("::foreach users:: ::name::(:age
4   ::) ::end::");
5
6 var userOutput = userTemplate.execute({users: users});
7
8 var template = new haxe.Template("The users are ::users::");
9 var output = template.execute({users: userOutput});
10 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.

```

1 class Main {
2   static function main() {
3     new Main();
4   }
5
6   public function new() {
7     var user = {name:"Mark", distance:3500};
8     var sample = "The results: $$display(::user::,::time::)";
9     var template = new haxe.Template(sample);
10    var output = template.execute({user:user, time: 15}, this);
11    trace(output);
12  }
13
14  function display(resolve:String->Dynamic, user:User, time:Int) {
15    return user.name + " ran " + (user.distance/1000) + " kilometers
16      in " + time + " minutes";
17  }
18 }
19 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`.

```

1 class Main {
2   static function main() {
3     var sample = haxe.Resource.getString("my_sample");
4     var user = {name:"Mark", age:30};
5     var template = new haxe.Template(sample);
6     var output = template.execute(user);
7     trace(output);
8   }
9 }

```

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:

```
1 class Main {
2     static function main() {
3         trace(Type.resolveClass("Mian")); // null
4     }
5 }
```

However, even if there are no typing errors it is easy to come across unexpected behavior:

```
1 class Main {
2     static function main() {
3         // null
4         trace(Type.resolveClass("haxe.Template"));
5     }
6 }
```

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 elimination (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**:

```
1 class Main {
2     static function main() {
3         // wrong
4         //Reflect.callMethod(Main, "f", []);
5         // right
6         Reflect.callMethod(Main,
7             Reflect.field(Main, "f"), []);
8     }
9
10    static function f() {
11        trace('Called');
12    }
13 }
```

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:

```

1 typedef MyStructure = {
2     name: String,
3     score: Int
4 }
5
6 class Main {
7     static function main() {
8         var data = reflective();
9         // At this point data is nicely typed as MyStructure
10    }
11
12    static function reflective():MyStructure {
13        // Work with reflection here to get some values we want to return.
14        return {
15            name: "Reflection",
16            score: 0
17        }
18    }
19 }

```

While the method `reflective` could internally 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:

```

1 import haxe.Serializer;
2 import haxe.Unserializer;
3
4 class Main {
5     static function main() {
6         var serializer = new Serializer();
7         serializer.serialize("foo");
8         serializer.serialize(12);
9         var s = serializer.toString();
10        trace(s); // y3:fooi12
11
12        var unserializer = new Unserializer(s);
13        trace(unserializer.unserialize()); // foo
14        trace(unserializer.unserialize()); // 12
15    }
16 }

```


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:kn` 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:kn` 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 XML² 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.

```
1 for (child in xml) {
2     // iterate on all children.
3 }
4 for (elt in xml.elements()) {
5     // iterate on all elements.
6 }
7 for (user in xml.elementsNamed("user")) {
8     // iterate on all elements with a nodeName "user".
9 }
10 for (att in xml.attributes()) {
11     // iterator on all attributes.
12 }
```

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.

```
1 var xml = Xml.parse('<root>Haxe is great!</root>').firstElement();
2 trace(xml.firstChild().nodeValue);
```

10.9.3 Encoding Xml

The method [xml.toString\(\)](#) can be used to obtain the String representation.

```
1 var xml = Xml.createElement('root');
2 xml.addChild(Xml.createElement('child1'));
3 xml.addChild(Xml.createElement('child2'));
4
5 trace(xml.toString()); // <root><child1/><child2/></root>
```

10.10 Json

Haxe provides built-in support for (de-)serializing JSON³ 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:

```
1 class Main {
2     static function main() {
3         var s = '{"rating": 5}';
4         var o = haxe.Json.parse(s);
5         trace(o); // { rating: 5 }
```

³<http://en.wikipedia.org/wiki/JSON>

```

6   }
7 }

```

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:

```

1 typedef MyData = {
2     var name:String;
3     var tags:Array<String>;
4 }
5
6 class Main {
7     static function main() {
8         var s = '{
9             "name": "Haxe",
10            "tags": ["awesome"]
11        }';
12         var o:MyData = haxe.Json.parse(s);
13         trace(o.name); // Haxe (a string)
14         // awesome (a string in an array)
15         trace(o.tags[0]);
16     }
17 }

```

10.10.2 Encoding JSON

Use the `haxe.Json.stringify` static method to encode a Haxe value into a JSON string:

```

1 class Main {
2     static function main() {
3         var o = {rating: 5};
4         var s = haxe.Json.stringify(o);
5         trace(s); // {"rating":5}
6     }
7 }

```

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:

```
1 class Server {
2     function new() { }
3     function foo(x, y) { return x + y; }
4
5     static function main() {
6         var ctx = new haxe.remoting.Context();
7         ctx.addObject("Server", new Server());
8
9         if(haxe.remoting.HttpConnection.handleRequest(ctx))
10        {
11            return;
12        }
13
14        // handle normal request
15        trace("This is a remoting server !");
16    }
17 }
```

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.

```
1 class Client {
2     static function main() {
3         var cnx = haxe.remoting.HttpAsyncConnection.urlConnect("http://
4             localhost/");
5         cnx.setErrorHandler( function(err) trace('Error: $err'); } );
6         cnx.Server.foo.call([1,2], function(data) trace('Result: $data');)
7         ;
8     }
9 }
```

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`:

```
1 # Enable CORS
2 Header set Access-Control-Allow-Origin "*"
3 Header set Access-Control-Allow-Methods: "GET, POST, OPTIONS, DELETE, PUT"
4 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.

```
1 <cross-domain-policy>
2   <allow-access-from domain="*" /> <!-- or the appropriate domains
3     -->
4   <allow-http-request-headers-from domain="*" headers="X-Haxe*" />
5 </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"`.

```
1 class MyTestCase extends haxe.unit.TestCase {
2   public function testBasic() {
3     assertEquals("A", "A");
4   }
5 }
```

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.

```
1 class Main {
2   static function main() {
3     var r = new haxe.unit.TestRunner();
4     r.add(new MyTestCase());
5     // add other TestCases here
6
7     // finally, run the tests
8     r.run();
9   }
10 }
```

The result of the test looks like this:

```
1 Class: MyTestCase
2 .
3 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.

```
1 class MyTestCase extends haxe.unit.TestCase {
2   var value:String;
3
4   override public function setup() {
5     value = "foo";
6   }
7
8   public function testSetup() {
9     assertEquals("foo", value);
10  }
11 }
```

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 similar to the `-cp <path>` argument, but expects a library name instead of a directory path. These commands are explained thoroughly in [Compiler Usage \(章 7\)](#).

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:

$A - Z a - z 0 - 9 _$.

. 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`:

```
1 {  
2   "name": "useless_lib",  
3   "url" : "https://github.com/jasononeil/useless/",  
4   "license": "MIT",  
5   "tags": ["cross", "useless"],  
6   "description": "This library is useless in the same way on every  
7     platform.",  
8   "version": "1.0.0",  
9   "releasenote": "Initial release, everything is working correctly.",  
10  "contributors": ["Juraj", "Jason", "Nicolas"],  
11  "dependencies": {  
12    "tink_macro": "",  
13    "nme": "3.5.5"  
14  }
```

11.2.1 Versioning

Haxelib uses a simplified version of [SemVer](#). The basic format is this:

```
1 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`:

```
1 "dependencies": {  
2   "tink_macros": "",  
3   "nme": "3.5.5"  
4 }
```

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.html

If you add a file named `extraParams.html` 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`.

```
1 import js.Lib;
2 import js.Browser;
3 class Main {
4     static function main() {
5         var button = Browser.document.createElement();
6         button.textContent = "Click me!";
7         button.onclick = function(event) {
8             Lib.alert("Haxe is great");
9         }
10        Browser.document.body.appendChild(button);
11    }
12 }
```

To compile, either run the following from the command line:

```
1 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.html`. In this example the `html`-file should be in the same directory as the example class.

```
1 -js main-javascript.js
2 -main Main
3 -D js-flatten
4 -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.

```

1 <!DOCTYPE html>
2 <html>
3   <body>
4     <script src="main-javascript.js"></script>
5   </body>
6 </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:

```

1 package js.jquery;
2 @:native("$") extern class JQuery {
3   /**
4     Creates DOM elements on the fly from the provided string of
5     raw HTML.
6     OR
7     Accepts a string containing a CSS selector which is then used
8     to match a set of elements.
9     OR
10    Binds a function to be executed when the DOM has finished
11    loading.
12    */
13    @:selfCall
14    @:overload(function(element:js.html.Element):Void { })
15    @:overload(function(selection:js.jquery.JQuery):Void { })
16    @:overload(function(callback:haxe.Constraints.Function):Void { })
17    @:overload(function(selector:String, ?context:haxe.extern.
18      EitherType<js.html.Element, js.jquery.JQuery>):Void { })
19    public function new():Void;
20
21    /**
22     Adds the specified class(es) to each element in the set of
23     matched elements.
24    */
25    @:overload(function(_function:Int -> String -> String):js.jquery.
26      JQuery { })
27    public function addClass(className:String):js.jquery.JQuery;
28
29    /**
30     Get the HTML contents of the first element in the set of
31     matched elements.

```

```

25         OR
26         Set the HTML contents of each element in the set of matched
           elements.
27     **/
28     @:overload(function(htmlString:String):js.jquery.JQuery { })
29     @:overload(function(_function:Int -> String -> String):js.jquery.
           JQuery { })
30     public function html():String;
31 }

```

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:

```

1 import js.jquery.*;
2 ..
3 new JQuery("#my-div").addClass("brand-success").html("haxe is great!")
4 ;
5 ..

```

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`.

```

1 package my.application.media;
2
3 @:native('external.library.media.video')
4 extern class Video {
5 ..

```

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:

```

1 extern class Functor {
2     public function new():Void;
3     @:selfCall function call():Void;
4 }
5
6 class Test {
7     static function main() {
8         var f = new Functor();
9         f.call(); // will be outputted as `f();`
10    }
11 }

```

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><code>js.jquery.*</code></code>
3.2-	jQuery 1.6.4	<code><code>js.JQuery</code></code>
3.3	SWFObject 2.3	<code><code>js.swfobject.*</code></code>
3.2-	SWFObject 1.5	<code><code>js.SWFObject</code></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.

```
1 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.

```
1 untyped __js__('alert("Haxe is great!")');
2 // output: alert("Haxe is great!");
3
4 var myMessage = "Haxe is great!";
5 untyped __js__('alert({0})', myMessage);
6 // output:
7 //   var myMessage = "Haxe is great!";
8 //   alert(myMessage);
9
10 var myVar:Bool = untyped __js__('confirm({0})', "Are you sure?");
11 // output: var myVar = confirm("Are you sure?");
12
13 var hexString:String = untyped __js__('({0}).toString({1})', 255, 16);
14 // output: var hexString = (255).toString(16);
```

`untyped __instanceof__(o,cl)` Same as `o instanceof cl` in JavaScript.

```
1 var myString = new String("Haxe is great");
2 var isString = untyped __instanceof__(myString, String);
3 output: var isString = (myString instanceof String);
```

`untyped __typeof__(o)` Same as `typeof o` in JavaScript.

```
1 var isNodeJS = untyped __typeof__(window) == null;
2 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").

```
1 var a = "0";
2 var b = 0;
3 var isEqual = untyped __strict_eq__(a, b);
4 output: var isEqual = ((a) === b);
```

`untyped __strict_neq__(a,b)` Same as `a !== b` in JavaScript, tests on negative strict equality.

```
1 var a = "0";
2 var b = 0;
3 var isntEqual = untyped __strict_neq__(a, b);
4 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.

```
1 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.

```
1 // Haxe code:
2 var myVar = untyped __js__('myObject.myJavaScriptFunction({0}, {1})',
    "Mark", 31);
```

This will generate the following JavaScript code:

```
1 // JavaScript Code
2 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 <code>window</code> object or <code>exports</code> 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`.

```
1 @:expose
2 class MyClass {
3     var name:String;
4     function new(name:String) {
5         this.name = name;
6     }
7     public function foo() {
8         return 'Greetings from $name!';
9     }
10 }
```

It generates the following JavaScript output:

```
1 (function ($hx_exports) { "use strict";
2 var MyClass = $hx_exports.MyClass = function(name) {
3     this.name = name;
4 };
5 MyClass.prototype = {
6     foo: function() {
7         return "Greetings from " + this.name + "!";
8     }
9 };
10 })(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.

```
1 // JavaScript code
2 var instance = new MyClass('Mark');
3 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:

```
1 var $hx_exports = $hx_exports || {};  
2 (function () { "use strict";  
3 var MyClass = $hx_exports.MyClass = function(name) {  
4     this.name = name;  
5 };  
6 MyClass.prototype = {  
7     foo: function() {  
8         return "Greetings from " + this.name + "!";  
9     }  
10 };  
11 })();  
12 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:

```
1 @:jsRequire("fs")  
2 extern class FS {  
3     static function readFileSync(path:String, encoding:String):String;  
4 }
```

In case our **extern** class represents an object within a module, second `@:jsRequire` argument specifies an object to load from a module:

```
1 @:jsRequire("http", "Server")  
2 extern class HTTPServer {  
3     function new();  
4 }
```

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`.

```
1 import flash.Lib;
2 import flash.display.Shape;
3 class Main {
4     static function main() {
5         var stage = Lib.current.stage;
6
7         // create a center aligned rounded gray square
8         var shape = new Shape();
9         shape.graphics.beginFill(0x333333);
10        shape.graphics.drawRoundRect(0, 0, 100, 100, 10);
11        shape.x = (stage.stageWidth - 100) / 2;
12        shape.y = (stage.stageHeight - 100) / 2;
13
14        stage.addChild(shape);
15    }
16 }
```

To compile this, either run the following from the command line:

```
1 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.html`. In this example the `html`-file should be in the same directory as the example class.

```
1 -swf main-flash.swf
2 -main Main
3 -swf-version 15
4 -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.

```
1 <!DOCTYPE html>
2 <html>
3     <body>
4         <embed src="main-flash.swf" width="960" height="640">
5     </body>
6 </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 <code>flash.display.BitmapData</code>)
@:debug	Forces debug information to be generated into the Swf even with <code>-debug</code>
@:file(File path)	Includes a given binary file into the target Swf and associates it with a given name
@:font _(TTF path Range String)_	Embeds the given TrueType font into the class (must extend <code>flash.text.TextField</code>)
@:getter _(Class field name)_	Generates a native getter function on the given field
@:noDebug	Does not generate debug information into the Swf even if <code>-debug</code> is used
@: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 <code>_.wav_</code> or <code>_.mp3_</code> file into the target Swf and associates it with a given name

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.html`. In this example the `html`-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).

```
1 var value:String = "test";
2 untyped __php__("echo '<pre>'; print_r($value); echo '</pre>';");
3 // 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.

```
1 var value = untyped __call__("array", 1,2,3);
2 // 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.

```
1 var value : String = untyped __var__('_SERVER', 'REQUEST_METHOD')
2 // output: $value = $_SERVER['REQUEST_METHOD']
```

`untyped __physeq__(val1, val2)` Strict equals test between the two values. Returns a Bool.

```
1 var isFalse = untyped __physeq__(false, value);
2 // 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