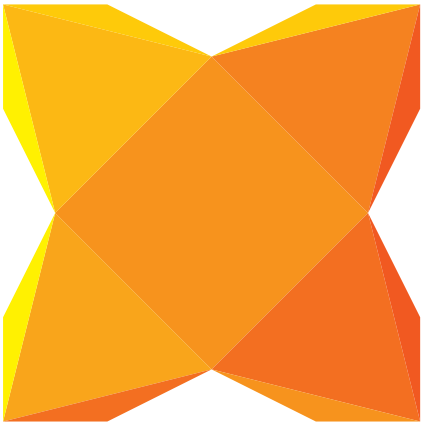

HAXE

Haxe 3 Manual

Haxe Foundation
September 12, 2014
(訳 : May 9, 2015)

Todo list

Contents

Chapter 1

導入

1.1 Haxeって何?

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

Haxeはオープンソースの高級プログラミング言語とコンパイラで構成されており、ECMAScript[1]を元にした構文で書いて、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です。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 (Chapter 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 (Chapter 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 (Chapter 3)へと続き、単一化(Unification)、型パラメータ、型推論についての解説がされます。

Class Fields (Chapter 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。

Expressions (Chapter 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。

Language Features (Chapter 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 (Chapter 7)で基本的な内容を、そして、Compiler Features (Chapter 8)で高度な機能をあつかいます。最後にMacros

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

(Chapter 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かを見ながら、刺激的なマクロの世界に挑んでいきます。

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

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

ソースコードは以下のように示されます:

```
1 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。

Definition: 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。

Trivia: 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で働く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 License

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

上記のコードは、HelloWorld.hxという名前で保存して、haxe -main HelloWorld --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HelloWorld.hx:3: Hello worldという出力がされるはずです。このことから以下のいくつかのことを学ぶことができます。

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

- Haxeのコードは.hxという拡張子で保存する。
- Haxeのコンパイラはコマンドラインツールであり、-main HelloWorld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(HelloWorld、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です(この場合だと、HelloWorld.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。

[2] http://nekovm.org

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

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

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

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

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

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

Part I

言語リファレンス

Chapter 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 (Section 3.6)で説明します。今のところは、上のコードの変数myButtonはMySpecialButtonのクラスインスタンスとわかると言っておけば十分でしょう。

Haxeの型システムは、以下の7つの型を認識します。

クラスインスタンス: クラスかインスタンスのオブジェクト

列挙型インスタンス: Haxeの列挙型(enum)の値

構造体: 匿名の構造体。例えば、連想配列。

関数: 引数と戻り値1つの型の複合型。

ダイナミック: あらゆる型に一致する、なんでも型。

抽象(abstract): 実行時には別の型となる、コンパイル時の型。

単相: 後で別の型が付けられる未知(Unknown)の型。

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

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

2.1 基本型

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

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

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

2.1.1 数値型

Type: Float
IEEEの64bit倍精度浮動小数点数を表します。

Type: Int
整数を表します。

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

2.1.2 オーバーフロー

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

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

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

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

2.1.3 数値の演算子

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

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

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

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す。

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

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

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

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

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

2.1.5 Void

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

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

please review, doubled content

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

review please, sounds weird

```
1 // Arguments and variables of type Void are not allowed
2 var x:Void;
```

2.2 Nullable(null許容型)

Definition: Nullable
Haxeでは、ある型が値としてnullをとる場合にNullable(null許容型)であるとみなす。

プログラミング言語は、Nullable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もともとの挙動に従うことで妥協しています。ターゲット言語のうちのいくつかは全てがデフォルト値としてnullをとり、その他は特定の型ではnullを許容しません。つまり、以下の2種類の言語を区別しなくてはいけません。

Definition: 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。

Definition: 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

Definition: 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いけません。

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

```
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.

Trivia: 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 (Section 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 (Chapter 4)の章で解説をします。

please review future tense

以下のサンプルコードが、この節で学ぶ基本になります。

```
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、

・ 通常の関数toStringでクラスフィールドが構成されています。

また、Haxeにはすべてのクラスと一致する特殊な型があります。

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

2.3.1 クラスのコンストラクタ

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

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

この例で、変数pに代入されたのがPointクラスのインスタンスです。Pointのコンストラクタは-1と65の2つの引数を受け取り、これらをそれぞれインスタンスのxとyの変数に代入しています(クラスインスタンス (Section 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 (Section 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。

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

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。

```
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。

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

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

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

```
interface Placeable {
  public var x:Float;
  public var y:Float;
}

class Main implements Placeable {
  public var x:Float;
  public var y:Float;
  static public function main() { }
}
```

> Trivia: 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] http://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 }
```

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

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

Haxの型システムには、すべての列挙型を統合する型があります。

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

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

2.4.1 列挙型のコンストラクタ

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

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

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。

```
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。

Trivia: Enum<T>の型パラメータを具体化する

このマニュアルのレビューアの一人は上のサンプルコードのColorとEnum<Color>の違いについて困惑しました。実際、型パラメータの具体化は意味のないもので、デモンストレーションのためのものでしかありませんでした。私たちはよく型を書くのを省いて、型についてあつかうのを型推論 (3.6)にまかせてしまいます。

しかし、型推論ではEnum<Color>ではないものが推論されます。コンパイラは、列挙型のコンストラクタをフィールドとしてみなした、仮の型を推論します。現在のHaxe3.2.0では、この仮の型について表現することは不可能であり、また表現する必要もありません。

2.4.2 列挙型を使う

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

```
1  enum Color {
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 Structure {
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。

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

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

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

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

2.5.1 JSONで構造体を書く

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

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

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

2.5.2 構造体の型のクラス記法

構造体の型を書く場合に、HaxeではClass Fields (Chapter 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。

```
typedef Point = {
    var x : Int;
    var y : Int;
}
```

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。

```
class FunctionType {
  static public function main() {
    // i : Int -> s : String -> Bool
    $type(test);
    $type(test(1, "foo")); // Bool
  }

  static function test(i:Int, s:String):Bool {
    return true;
  }
}
```

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。

```
class OptionalArguments {
  static public function main() {
    // ?i : Int -> ?s : String -> String
    $type(test);
    trace(test()); // i: null, s: null
    trace(test(1)); // i: 1, s: null
    trace(test(1, "foo")); // i: 1, s: foo
    trace(test("foo")); // i: null, s: foo
  }

  static function test(?i:Int, ?s:String) {
    return "i: " +i + ", s: " +s;
  }
}
```

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

この例では、関数を4回呼び出しその結果を出力しています。

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

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

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

2.6.2 デフォルト値

Haxeでは、引数のデフォルト値として定数値を割り当てることが可能です。

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

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

```
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 (Section 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。

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

実際のところDynamicは使ってしまいますが、多くの場面では他のもっと良い選択肢があるのでDynamicの使用は最低限にすべきです。例えば、HaxeのReflection (Section 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。

> Trivia: 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 (Section 2.7)のすべてのフィールドがパラメータの型であることが強制されます。

```
var att : Dynamic<String> = xml.attributes;
// valid, value is a String
att.name = "Nicolas";
// dito (this documentation is quite old)
att.age = "26";
// error, value is not a String
att.income = 0;
```

2.7.2　ダイナミックを実装(implements)する

クラスはDynamicとDynamic<T>を実装 (2.3.3)することができます。これにより任意のフィールドへのアクセスが可能になります。Dynamicの場合、フィールドはあらゆる型になる可能性があり、Dynamic<T>の場合、フィールドはパラメータの型と矛盾しない型のみに強制されます。

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

class Main {
  static public function main() {
    var c = new ImplementsDynamic();
    // valid, present is an existing field
    c.present = 1;
    // valid, assigned value is a String
    c.stringField = "foo";
    // error, Int should be String
    //c.intField = 1;
  }
}
```

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

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

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

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

```
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。

> Definition: 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 MyAbstract {
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 (Section 4.4.2)で学びます)。これは、クラスのように考えていた場合、驚くべきことかもしれません。しかし、これこそが抽象型を使って表現したいことそのものです。抽象型のすべてのインラインのメンバメソッドではthisへの代入が可能で、これにより“内部の値”が編集できます。

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

> Trivia: 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 ImplicitCastDirect {
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) {
```

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

  @:from
  static public function fromString(s:String) {
    return new MyAbstract(Std.parseInt(s));
  }

  @:to
  public function toArray() {
    return [this];
  }
}

class ImplicitCastField {
  static public function main() {
    var a:MyAbstract = "3";
    var b:Array<Int> = a;
    trace(b); // [3]
  }
}
```

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

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

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

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

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

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

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

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

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

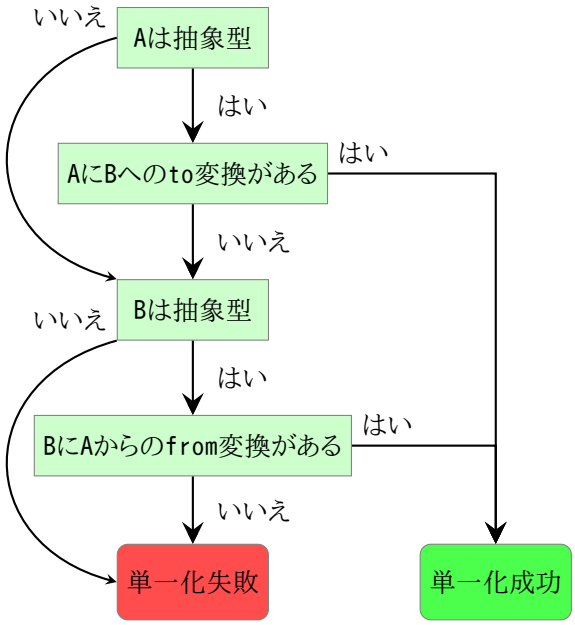

Figure 2.1: 選択アルゴリズムのフローチャート

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 AbstractOperatorOverload {
16   static public function main() {
17     var a = new MyAbstract("foo");
18     trace(a * 3); // foofoofoo
19   }
20 }
```

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

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

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

単項演算子の場合もこれによく似ています。

```
1  abstract MyAbstract(String) {
2    public inline function new(s:String) {
3      this = s;
4    }
5
6    @:op(++A) public function pre()
7      return "pre" + this;
8    @:op(A++) public function post()
9      return this + "post";
10 }
11
12 class AbstractUnopOverload {
13   static public function main() {
```

```
    var a = new MyAbstract("foo");
    trace(++a); // prefoo
    trace(a++); // foopost
  }
}
```

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。

Exposing underlying type operations　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。

```
abstract MyAbstractInt(Int) from Int to Int {
  // The following line exposes the (A > B) operation from the underlying Int
  // type. Note that no function body is used:
  @:op(A > B) static function gt( a:MyAbstractInt, b:MyAbstractInt ) : Bool;
}

class Main {
  static function main() {
    var a:MyAbstractInt = 42;
    if(a > 0) trace('Works fine, > operation implemented!');

    // The < operator is not implemented.
    // This will cause an 'Cannot compare MyAbstractInt and Int' error :
    if(a < 100) { }
  }
}
```

please review for correctness

2.8.3　配列アクセス

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

You have marked "Map" for some reason

```
@:arrayAccess
public inline function get(key:K) {
  return this.get(key);
}
@:arrayAccess
public inline function arrayWrite(k:K, v:V):V {
    this.set(k, v);
    return v;
}
```

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。

```
class Main {
  public static function main() {
    var map = new Map();
    map["foo"] = 1;
    trace(map["foo"]);
  }
}
```

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。

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

Order of array access resolving Due to a bug in Haxe versions before 3.2 the order of checked `:arrayAccess` fields was undefined. This was fixed for Haxe 3.2 so that the fields are now consistently checked from top to bottom:

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

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

The array access `a[0]` is resolved to the `getInt1` field, leading to lower case `f` being returned. The result might be different in Haxe versions before 3.2.

Fields which are defined earlier take priority even if they require an implicit cast (2.8.1).

2.8.4 選択的関数

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

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

  function get() return this;

```

```
6     static public function getString(v:MyAbstract<String>):String {
7       return v.get();
8     }
9   }
10
11  class SelectiveFunction {
12    static public function main() {
13      var a = new MyAbstract("foo");
14      a.getString();
15      var b = new MyAbstract(1);
16      // Int should be MyAbstract<String>
17      b.getString();
18    }
19  }
```

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。

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

2.8.5 抽象型列挙体

Since Haxe 3.1.0

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    }
```

```
}
```

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

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

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

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

2.8.6 抽象型フィールドの繰り上げ

Since Haxe 3.1.0

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

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

class Main {
  static public function main() {
    var myArray = new MyArray();
    myArray.push(12);
    myArray.pop();
    // MyArray<Int> has no field length
    //myArray.length;
  }
}
```

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

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

```
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。

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

2.8.7 コアタイプの抽象型

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

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

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

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

2.9 単相(モノモーフ)

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

Chapter 3

型システム

私たちは型 (Chapter 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、複雑な型に対して名前(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(ユニフィケーション) (Section 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 (Section 3.7)では、これらに関連した仕組みについて詳しい説明を行います。

3.1 typedef

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

typedefは他のあらゆる型に対して名前を与えることが可能です。

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

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

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

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

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

3.1.1 拡張

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

```
typedef IterableWithLength<T> = {
  > Iterable<T>,
  // read only property
  var length(default, null):Int;
}

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

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

IterableWithLength<T>に適合するためには、Iterable<T>にも適合してさらに読み込み専用のInt型のプロパティlengthを持ってなきゃいけません。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。

Since Haxe 3.1.0

複数の構造体を拡張することもできます。

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

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

class Extension2 {
  static public function main() {
    var array = [1, 2, 3];
    var t:IterableWithLengthAndPush<Int> = array;
  }
}
```

3.2 型パラメータ

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

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

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

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

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

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

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

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

```
1  class FunctionTypeParameter {
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。

Trivia: 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 Constraints {
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

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

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

```
1  @:generic
2  class MyValue<T> {
3    public var value:T;
4      public function new(value:T) {
5      this.value = value;
6    }
7  }
8
```

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

あまり使わない明示的なMyArray<String>の型宣言があり、よく使う型推論 (3.6)であつかっていますが、これが重要です。コンパイラは、コンストラクタの呼び出し時にジェネリッククラスの正確な型な型を知っている必要があります。このJavaScript出力は以下のような結果になります。

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

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

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

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

JavaScript出力を見れば明白です。

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

3.3.1 ジェネリック型パラメータのコンストラクト

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

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

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

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

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

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

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

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

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

3.4　変性(バリアンス)

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

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

class Child extends Base { }

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

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

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

class Child extends Base { }

class OtherChild extends Base { }

class Main {
  public static function main () {
    var children = [new Child()];
    // subvert type checker
    var bases:Array<Base> = cast children;
    bases.push(new OtherChild());
    for(child in children) {
      trace(child);
    }
  }
}
```

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

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

```
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。

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

Definition: 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(ユニフィケーション)

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

単一化は型システムの要であり、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。

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

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

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

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

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

3.5.1 クラスとインターフェースの単一化

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

以下のような、代入が許可されます。

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

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

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

3.5.2 構造的部分型付け

Definition: 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。

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

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

3.5.3 単相

単相 (2.9)である、あるいは単相を含む型についての単一化は型推論 (Section 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。

```
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。

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

class Child1 extends Base { }
class Child2 extends Base { }

class UnifyMin {
  static public function main() {
    var a = [new Child1(), new Child2()];
    $type(a); // Array<Base>
  }
}
```

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。

```
class TypeInference {
  public static function main() {
    var x = null;
    $type(x); // Unknown<0>
    x = "foo";
    $type(x); // String
  }
}
```

この特殊な構文$typeは、関数 (Section 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 (Section 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。

```
class TypeInference2 {
  public static function main() {
    var x = [];
    $type(x); // Array<Unknown<0>>
    x.push("foo");
    $type(x); // Array<String>
  }
}
```

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。

> Definition: 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 (Section 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。

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

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

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

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

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

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

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。

また、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場合があります。

A different kind of limitation involves the readability of code. If type inference is overused it might be difficult to understand parts of a program due to the lack of visible types. This is particularly true for method signatures. It is recommended to find a good balance between type inference and explicit type hints.

3.7 モジュールとパス

Definition: 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含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、モジュールの従属型 (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。

Definition: 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 モジュールの従属型

モジュール従属型とは、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の従属型ExprDefにアクセスしています。

従属型の関係は、実行時には影響を与えません。publicの従属型はそのパッケージのメンバーになります。このため、同じパッケージの別々のモジュールで同じ従属型を定義した場合に衝突を起こします。The sub-type relation is not reflected at run-time. That is, public sub-types become a member of their containing package, which could lead to conflicts if two modules within the same package tried to define the same sub-type. Naturally, the Haxe compiler detects these cases and reports them accordingly. In the example above ExprDef is generated as haxe.macro.ExprDef.

Sub-types can also be made private:

```
1 private class C { ... }
2 private enum E { ... }
3 private typedef T { ... }
4 private abstract A { ... }
```

Definition: Private type
A type can be made private by using the `private` modifier. As a result, the type can only be directly accessed from within the module (3.7) it is defined in.

Private types, unlike public ones, do not become a member of their containing package.

The accessibility of types can be controlled more fine-grained by using access control (6.10).

3.7.2 Import

If a type path is used multiple times in a .hx file, it might make sense to use an `import` to shorten it. This allows omitting the package when using the type:

```
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 }
```

With `haxe.ds.StringMap` being imported in the first line, the compiler is able to resolve the unqualified identifier `StringMap` in the `main` function to this package. The module `StringMap` is said to be imported into the current file.

In this example, we are actually importing a module, not just a specific type within that module. This means that all types defined within the imported module are available:

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

The type `Binop` is an enum (2.4) declared in the module `haxe.macro.Expr`, and thus available after the import of said module. If we were to import only a specific type of that module, e.g. `import haxe.macro.Expr.ExprDef`, the program would fail to compile with `Class not found : Binop`.

There are several aspects worth knowing about importing:

- The bottommost import takes priority (detailed in Resolution Order (Section 3.7.3)).
- The static extension (6.3) keyword `using` implies the effect of `import`.
- If an enum is imported (directly or as part of a module import), all its enum constructors (2.4.1) are also imported (this is what allows the `OpAdd` usage in the above example).

Furthermore, it is also possible to import static fields (4) of a class and use them unqualified:

```
import Math.random;

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

Special care has to be taken with field names or local variable names that conflict with a package name: Since they take priority over packages, a local variable named `haxe` blocks off usage the entire `haxe` package.

Wildcard import Haxe allows using `.*` to allow import of all modules in a package, all types in a module or all static fields in a type. It is important to understand that this kind of import only crosses a single level as we can see in the following example:

```
import haxe.macro.*;

class Main {
  static function main() {
    var expr:Expr = null;
    //var expr:ExprDef = null; // Class not found : ExprDef
  }
}
```

Using the wildcard import on `haxe.macro` allows accessing `Expr` which is a module in this package, but it does not allow accessing `ExprDef` which is a sub-type of the `Expr` module. This rule extends to static fields when a module is imported.

When using wildcard imports on a package the compiler does not eagerly process all modules in that package. This means that these modules are never actually seen by the compiler unless used explicitly and are then not part of the generated output.

Import with alias If a type or static field is used a lot in an importing module it might help to alias it to a shorter name. This can also be used to disambiguate conflicting names by giving them a unique identifier.

```
import String.fromCharCode in f;

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

Here we import `String.fromCharCode` as `f` which allows us to use `f(65)` and `f(66)`. While the same could be achieved with a local variable, this method is compile-time exclusive and guaranteed to have no run-time overhead.
Since Haxe 3.2.0

Haxe also allows the more natural `as` in place of `in`.

3.7.3 Resolution Order

Resolution order comes into play as soon as unqualified identifiers are involved. These are expressions (5) in the form of `foo()`, `foo = 1` and `foo.field`. The last one in particular includes module paths such as `haxe.ds.StringMap`, where `haxe` is an unqualified identifier.

We describe the resolution order algorithm here, which depends on the following state:

- the declared local variables (5.10) (including function arguments)
- the imported (3.7.2) modules, types and statics
- the available static extensions (6.3)
- the kind (static or member) of the current field
- the declared member fields on the current class and its parent classes
- the declared static fields on the current class
- the expected type (3.6.1)
- the expression being `untyped` or not

proper label and caption + code/identifier styling for diagram

Given an identifier `i`, the algorithm is as follows:

1. If `i` is `true`, `false`, `this`, `super` or `null`, resolve to the matching constant and halt.
2. If a local variable named `i` is accessible, resolve to it and halt.
3. If the current field is static, go to 6.
4. If the current class or any of its parent classes has a field named `i`, resolve to it and halt.
5. If a static extension with a first argument of the type of the current class is available, resolve to it and halt.
6. If the current class has a static field named `i`, resolve to it and halt.
7. If an enum constructor named `i` is declared on an imported enum, resolve to it and halt.
8. If a static named `i` is explicitly imported, resolve to it and halt.
9. If `i` starts with a lower-case character, go to 11.
10. If a type named `i` is available, resolve to it and halt.
11. If the expression is not in untyped mode, go to 14
12. If `i` equals `__this__`, resolve to the `this` constant and halt.
13. Generate a local variable named `i`, resolve to it and halt.
14. Fail

For step 10, it is also necessary to define the resolution order of types:

1. If a type named `i` is imported (directly or as part of a module), resolve to it and halt.

2. If the current package contains a module named `i` with a type named `i`, resolve to it and halt.
3. If a type named `i` is available at top-level, resolve to it and halt.
4. Fail

For step 1 of this algorithm as well as steps 5 and 7 of the previous one, the order of import resolution is important:

- Imported modules and static extensions are checked from bottom to top with the first match being picked.
- Within a given module, types are checked from top to bottom.
- For imports, a match is made if the name equals.
- For static extensions (6.3), a match is made if the name equals and the first argument unifies (3.5). Within a given type being used as static extension, the fields are checked from top to bottom.

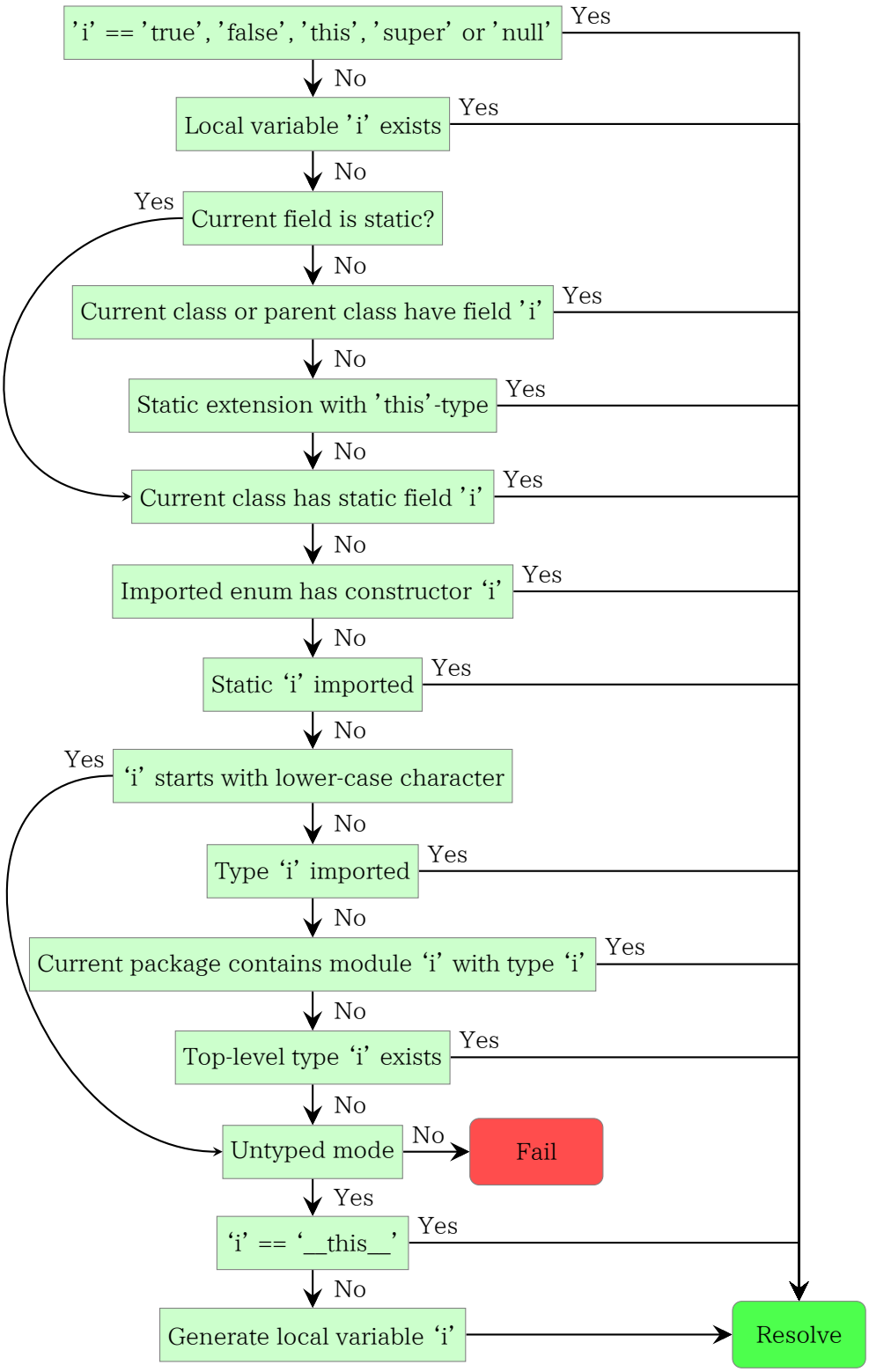

Figure 3.1: Resolution order of identifier 'i'

Chapter 4

Class Fields

> Definition: 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 VariableField {
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.

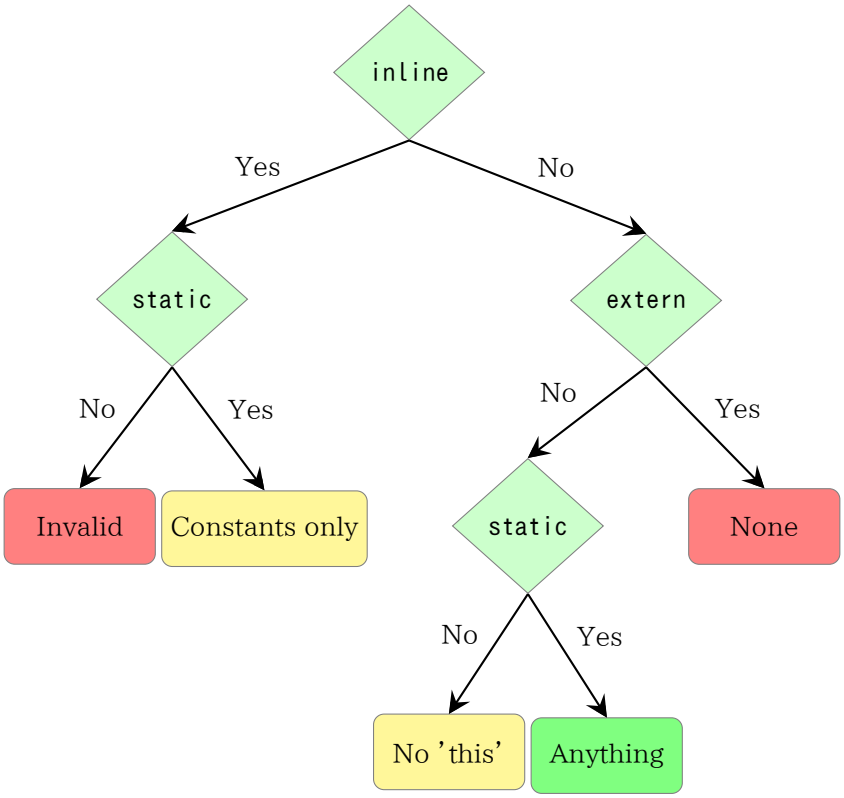

Figure 4.1: Initialization values of a variable field.

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:

Definition: 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.

Definition: 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.

Definition: 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`.

Trivia: 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:

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

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:

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

  public function new() { }

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

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:

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

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:

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

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:

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

class Main extends Base {
  // ok, get_x is declared by parent class
  public var x(get, null):Int;

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

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 me 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:

```
class Main {
  public var x(default, set):Int;

  function set_x(newX) {
    return x = newX;
  }

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

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

Definition: 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:

```
class Main {
```

```
  // This field cannot be accessed because it is not a real variable
  public var x(get, set):Int;

  function get_x() {
    return x;
  }

  function set_x(x) {
    return this.x = x;
  }

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

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

```
class Main {
  // @isVar forces the field to be physical allowing the program to
     compile.
  @:isVar public var x(get, set):Int;

  function get_x() {
    return x;
  }

  function set_x(x) {
    return this.x = x;
  }

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

Trivia: 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:

```
class HelloWorld {
  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 (Section 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`. It is then obvious that the field access is resolved dynamically at runtime.

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.

> Trivia: 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   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 (Section 4.3.1). This modifier is only allowed on method (4.3) fields.

Chapter 5

Expressions

Expressions in Haxe define what a program does. Most expressions are found in the body of a method (4.3), where they are combined to express what that method should do. This section explains the different kinds of expressions. Some definitions help here:

Definition: Name
A general name may refer to

- a type,
- a local variable,
- a local function or
- a field.

Definition: Identifier
Haxe identifiers start with an underscore _, a dollar `$`, a lower-case character `a-z` or an upper-case character `A-Z`. After that, any combination and number of _, `A-Z`, `a-z` and `0-9` may follow.
Further limitations follow from the usage context, which are checked upon typing:

- Type names must start with an upper-case letter `A-Z` or an underscore _.
- Leading dollars are not allowed for any kind of name (5) (dollar-names are mostly used for macro reification (9.3)).

5.1 Blocks

A block in Haxe starts with an opening curly brace { and ends with a closing curly brace }. A block may contain several expressions, each of which is followed by a semicolon ;. The general syntax is thus:

```
1 {
2     expr1;
3     expr2;
```

```
    ...
    exprN;
}
```

The value and by extension the type of a block-expression is equal to the value and the type of the last sub-expression.

Blocks can contain local variables declared by `var` expression (5.10), as well as local functions declared by `function` expressions (5.11). These are available within the block and within sub-blocks, but not outside the block. Also, they are available only after their declaration. The following example uses `var`, but the same rules apply to `function` usage:

```
{
    a; // error, a is not declared yet
    var a = 1; // declare a
    a; // ok, a was declared
    {
        a; // ok, a is available in sub-blocks
    }
  // ok, a is still available after
    // sub-blocks
    a;
}
a; // error, a is not available outside
```

At runtime, blocks are evaluated from top to bottom. Control flow (e.g. exceptions (5.18) or return expressions (5.19)) may leave a block before all expressions are evaluated.

5.2 Constants

The Haxe syntax supports the following constants:

Int: An integer (2.1.1), such as `0`, `1`, `97121`, `-12`, `0xFF0000`.

Float: A floating point number (2.1.1), such as `0.0`, `1.`, `.3`, `-93.2`.

String: A string of characters (10.1), such as `""`, `"foo"`, `''`, `'bar'`.

true,false: A boolean (2.1.4) value.

null: The null value.

Furthermore, the internal syntax structure treats identifiers (5) as constants, which may be relevant when working with macros (9).

5.3 Binary Operators

5.4 Unary Operators

5.5 Array Declaration

Arrays are initialized by enclosing comma `,` separated values in brackets `[]`. A plain `[]` represents the empty array, whereas `[1, 2, 3]` initializes an array with three elements `1`, `2` and `3`.

The generated code may be less concise on platforms that do not support array initialization. Essentially, such initialization code then looks like this:

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

This should be considered when deciding if a function should be inlined (4.4.2) as it may inline more code than visible in the syntax.

Advanced initialization techniques are described in Array Comprehension (Section 6.6).

5.6 Object Declaration

Object declaration begins with an opening curly brace `{` after which `key:value`-pairs separated by comma `,` follow, and which ends in a closing curly brace `}`.

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

Further details of object declaration are described in the section about anonymous structures (2.5).

5.7 Field Access

Field access is expressed by using the dot `.` followed by the name of the field.

```
object.fieldName
```

This syntax is also used to access types within packages in the form of `pack.Type`.

The typer ensures that an accessed field actually exist and may apply transformations depending on the nature of the field. If a field access is ambiguous, understanding the resolution order (3.7.3) may help.

5.8 Array Access

Array access is expressed by using an opening bracket `[` followed by the index expression and a closing bracket `]`.

```
expr[indexExpr]
```

This notation is allowed with arbitrary expressions, but at typing level only certain combinations are admitted:

- `expr` is of `Array` or `Dynamic` and `indexExpr` is of `Int`
- `expr` is an abstract type (2.8) which defines a matching array access (2.8.3)

5.9 Function Call

Functions calls consist of an arbitrary subject expression followed by an opening parenthesis `(`, a comma `,` separated list of expressions as arguments and a closing parenthesis `)`.

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

5.10 var

The `var` keyword allows declaring multiple variables, separated by comma `,`. Each variable has a valid identifier (5) and optionally a value assignment following the assignment operator `=`. Variables can also have an explicit type-hint.

```
var a; // declare local a
var b:Int; // declare variable b of type Int
// declare variable c, initialized to value 1
var c = 1;
// declare variable d and variable e
// initialized to value 2
var d,e = 2;
```

The scoping behavior of local variables is described in Blocks (Section 5.1).

5.11 Local functions

Haxe supports first-class functions and allows declaring local functions in expressions. The syntax follows class field methods (4.3):

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

We declare `myLocalFunction` inside the block expression (5.1) of the `main` class field. It takes one argument `i` and adds it to `value`, which is defined in the outside scope.

The scoping is equivalent to that of variables (5.10) and for the most part writing a named local function can be considered equal to assigning an unnamed local function to a local variable:

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

However, there are some differences related to type parameters and the position of the function. We speak of a “lvalue” function if it is not assigned to anything upon its declaration, and an “rvalue” function otherwise.

- Lvalue functions require a name and can have type parameters (3.2).
- Rvalue functions may have a name, but cannot have type parameters.

5.12 new

The `new` keyword signals that a class (2.3) or an abstract (2.8) is being instantiated. It is followed by the type path (3.7) of the type which is to be instantiated. It may also list explicit type parameters (3.2) enclosed in `<>` and separated by comma `,`. After an opening parenthesis `(` follow the constructor arguments, again separated by comma `,`, with a closing parenthesis `)` at the end.

```
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 }
```

Within the `main` method we instantiate an instance of `Main` itself, with an explicit type parameter `Int` and the arguments `12` and `"foo"`. As we can see, the syntax is very similar to the function call syntax (5.9) and it is common to speak of "constructor calls".

5.13 for

Haxe does not support traditional for-loops known from C. Its `for` keyword expects an opening parenthesis `(`, then a variable identifier followed by the keyword `in` and an arbitrary expression used as iterating collection. After the closing parenthesis `)` follows an arbitrary loop body expression.

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

The typer ensures that the type of `e1` can be iterated over, which is typically the case if it has an `iterator` method returning an `Iterator<T>`, or if it is an `Iterator<T>` itself.

Variable `v` is then available within loop body `e2` and holds the value of the individual elements of collection `e1`.

Haxe has a special range operator to iterate over intervals. It is a binary operator taking two `Int` operands: `min...max` returns an `IntIterator` instance that iterates from `min` (inclusive) to `max` (exclusive). Note that `max` may not be smaller than `min`.

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

The type of a `for` expression is always `Void`, meaning it has no value and cannot be used as right-side expression.

The control flow of loops can be affected by `break` (5.20) and `continue` (5.21) expressions.

5.14 while

A normal while loop starts with the `while` keyword, followed by an opening parenthesis `(`, the condition expression and a closing paranthesis `)`. After that follows the loop body expression:

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

The condition expression has to be of type `Bool`.

Upon each iteration, the condition expression is evaluated. If it evaluates to `false`, the loop stops, otherwise it evaluates the loop body expression.

```
class Main {
  static public function main() {
    var f = 0.0;
    while (f < 0.5) {
      trace(f);
      f = Math.random();
    }
  }
}
```

This kind of while-loop is not guaranteed to evaluate the loop body expression at all: If the condition does not hold from the start, it is never evaluated. This is different for do-while loops (5.15).

5.15 do-while

A do-while loop starts with the `do` keyword followed by the loop body expression. After that follows the `while` keyword, an opening parenthesis `(`, the condition expression and a closing parenthesis `)`:

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

The condition expression has to be of type `Bool`.

As the syntax suggests, the loop body expression is always evaluated at least once, unlike while (5.14) loops.

5.16 if

Conditional expressions come in the form of a leading `if` keyword, a condition expression enclosed in parentheses `()` and a expression to be evaluated in case the condition holds:

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

The condition expression has to be of type `Bool`.

Optionally, `expression` may be followed by the `else` keyword as well as another expression to be evaluated if the condition does not hold:

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

Here, `expression2` may consist of another `if` expression:

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

If the value of an `if` expression is required, e.g. for `var x = if(condition) expression1 else expression2`, the typer ensures that the types of `expression1` and `expression2` unify (3.5). If no `else` expression is given, the type is inferred to be `Void`.

5.17 switch

A basic switch expression starts with the `switch` keyword and the switch subject expression, as well as the case expressions between curly braces {}. Case expressions either start with the `case` keyword and are followed by a pattern expression, or consist of the `default` keyword. In both cases a colon `:` and an optional case body expression follows:

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

Case body expressions never "fall through", so the `break` (5.20) keyword is not supported in Haxe.

Switch expressions can be used as value; in that case the types of all case body expressions and the default expression must unify (3.5).

Further details on syntax of pattern expressions are detailed in Pattern Matching (Section 6.4).

5.18 try/catch

Haxe allows catching values using its `try/catch` syntax:

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

If during runtime the evaluation of `try-expression` causes a `throw` (5.22), it can be caught by any subsequent `catch` block. These blocks consist of

- a variable name which holds the thrown value,
- an explicit type annotation which determines which types of values to catch, and
- the expression to execute in that case.

Haxe allows throwing and catching any kind of value, it is not limited to types inheriting from a specific exception or error class. Catch blocks are checked from top to bottom with the first one whose type is compatible with the thrown value being picked.

This process has many similarities to the compile-time unification (3.5) behavior. However, since the check has to be done at runtime there are several restrictions:

- The type must exist at runtime: Class instances (2.3), enum instances (2.4), abstract core types (2.8.7) and Dynamic (2.7).
- Type parameters can only be Dynamic (2.7).

5.19 return

A `return` expression can come with or without an value expression:

```
return;
return expression;
```

It leaves the control-flow of the innermost function it is declared in, which has to be distinguished when local functions (5.11) are involved:

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

The `return` leaves local function `f2`, but not `f1`, meaning `expression` is still evaluated.

If `return` is used without a value expression, the typer ensures that the return type of the function it returns from is of `Void`. If it has a value expression, the typer unifies (3.5) its type with the return type (explicitly given or inferred by previous `return` expressions) of the function it returns from.

5.20 break

The `break` keyword leaves the control flow of the innermost loop (`for` or `while`) it is declared in, stopping further iterations:

```
while(true) {
    expression1;
    if (condition) break;
    expression2;
}
```

Here, `expression1` is evaluated for each iteration, but as soon as `condition` holds, `expression2` is not evaluated anymore.

The typer ensures that it appears only within a loop. The `break` keyword in `switch` cases (5.17) is not supported in Haxe.

5.21 continue

The `continue` keyword ends the current iteration of the innermost loop (`for` or `while`) it is declared in, causing the loop condition to be checked for the next iteration:

```
while(true) {
    expression1;
    if(condition) continue;
    expression2;
}
```

Here, `expression1` is evaluated for each iteration, but if `condition` holds, `expression2` is not evaluated for the current iteration. Unlike `break`, iterations continue.

The typer ensures that it appears only within a loop.

5.22 throw

Haxe allows throwing any kind of value using its `throw` syntax:

```
throw expr
```

A value which is thrown like this can be caught by `catch` blocks (5.18). If no such block catches it, the behavior is target-dependent.

5.23 cast

Haxe allows two kinds of casts:

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

5.23.1 unsafe cast

Unsafe casts are useful to subvert the type system. The compiler types `expr` as usual and then wraps it in a monomorph (2.9). This allows the expression to be assigned to anything.

Unsafe casts do not introduce any dynamic (2.7) types, as the following example shows:

```
class Main {
  public static function main() {
    var i = 1;
    $type(i); // Int
    var s = cast i;
    $type(s); // Unknown<0>
    Std.parseInt(s);
    $type(s); // String
  }
}
```

Variable `i` is typed as `Int` and then assigned to variable `s` using the unsafe cast `cast i`. This causes `s` to be of an unknown type, a monomorph. Following the usual rules of unification (3.5), it can then be bound to any type, such as `String` in this example.

These casts are called "unsafe" because the runtime behavior for invalid casts is not defined. While most dynamic targets (2.2) are likely to work, it might lead to undefined errors on static targets (2.2).

Unsafe casts have little to no runtime overhead.

5.23.2 safe cast

Unlike unsafe casts (5.23.1), the runtime behavior in case of a failing cast is defined for safe casts:

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

class Child1 extends Base {}

class Child2 extends Base {}

class Main {
    public static function main() {
    var child1:Base = new Child1();
    var child2:Base = new Child2();
```

```
13      cast(child1, Base);
14      // Exception: Class cast error
15      cast(child1, Child2);
16    }
17  }
```

In this example we first cast a class instance of type `Child1` to `Base`, which succeeds because `Child1` is a child class (2.3.2) of `Base`. We then try to cast the same class instance to `Child2`, which is not allowed because instances of `Child2` are not instances of `Child1`.

The Haxe compiler guarantees that an exception of type `String` is thrown (5.22) in this case. This exception can be caught using a `try/catch` block (5.18).

Safe casts have a runtime overhead. It is important to understand that the compiler already generates type checks, so it is redundant to add manual checks, e.g. using `Std.is`. The intended usage is to try the safe cast and catch the `String` exception.

5.24 type check

Since Haxe 3.1.0

It is possible to employ compile-time type checks using the following syntax:

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

The parentheses are mandatory. Unlike safe casts (5.23.2) this construct has no runtime impact. It has two compile-time implications:

1. Top-down inference (3.6.1) is used to type `expr` with type `type`.
2. The resulting typed expression is unified (3.5) with type `type`.

This has the usual effect of both operations such as the given type being used as expected type when performing unqualified identifier resolution (3.7.3) and the unification checking for abstract casts (2.8.1).

Chapter 6

Language Features

Abstract types (2.8):

An abstract type is a compile-time construct which is represented in a different way at runtime. This allows giving a whole new meaning to existing types.

Extern classes (6.2):

Externs can be used to describe target-specific interaction in a type-safe manner.

Anonymous structures (2.5):

Data can easily be grouped in anonymous structures, minimizing the necessity of small data classes.

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

Array Comprehension (6.6):

Create and populate arrays quickly using for loops and logic.

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

Classes, interfaces and inheritance (2.3):

Haxe allows structuring code in classes, making it an object-oriented language. Common related features known from languages such as Java are supported, including inheritance and interfaces.

Conditional compilation (6.1):

Conditional Compilation allows compiling specific code depending on compilation parameters. This is instrumental for abstracting target-specific differences, but can also be used for other purposes, such as more detailed debugging.

```
¥#if js
    js.Lib.alert("Hello");
¥#elseif sys
    Sys.println("Hello");
¥#end
```

(Generalized) Algebraic Data Types (2.4):

Structure can be expressed through algebraic data types (ADT), which are known as enums in the Haxe Language. Furthermore, Haxe supports their generalized variant known as GADT.

```
enum Result {
    Success(data:Array<Int>);
    UserError(msg:String);
```

```
    SystemError(msg:String, position:PosInfos);
}
```

Inlined calls (4.4.2):

Functions can be designed as being inline, allowing their code to be inserted at call-site. This can yield significant performance benefits without resorting to code duplication via manual inlining.

Iterators (6.7):

Iterating over a set of values, e.g. the elements of an array, is very easy in Haxe courtesy of iterators. Custom classes can quickly implement iterator functionality to allow iteration.

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

Local functions and closures (5.11):

Functions in Haxe are not limited to class fields and can be declared in expressions as well, allowing powerful closures.

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

Metadata (6.9):

Add metadata to fields, classes or expressions. This can communicate information to the compiler, macros, or runtime classes.

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

Static Extensions (6.3):

Existing classes and other types can be augmented with additional functionality through using static extensions.

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

String Interpolation (6.5):

Strings declared with a single quotes are able to access variables in the current context.

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

Partial function application (6.8):

Any function can be applied partially, providing the values of some arguments and leaving the rest to be filled in later.

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

Pattern Matching (6.4):

Complex structures can be matched against patterns, extracting information from an enum or a structure and defining specific operations for specific value combination.

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

Properties (4.2):

Variable class fields can be designed as properties with custom read and write access, allowing fine grained access control.

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

Access control (6.10):

The access control language feature uses the Haxe metadata syntax to force or allow access classes or fields.

Type Parameters, Constraints and Variance (3.2):

Types can be parametrized with type parameters, allowing typed containers and other complex data structures. Type parameters can also be constrained to certain types and respect variance rules.

```
class Main<A> {
    static function main() {
        new Main<String>("foo");
        new Main(12); // use type inference
    }

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

6.1 Conditional Compilation

Haxe allows conditional compilation by using `#if`, `#elseif` and `#else` and checking for compiler flags.

Definition: Compiler Flag
A compiler flag is a configurable value which may influence the compilation process. Such a flag can be set by invoking the command line with `-D key=value` or just `-D key`, in which case the value defaults to "1". The compiler also sets several flags internally to pass information between different compilation steps.

This example demonstrates usage of conditional compilation:

```
class ConditionalCompilation {
  public static function main(){
    #if !debug
      trace("ok");
    #elseif (debug_level > 3)
      trace(3);
    #else
      trace("debug level too low");
    #end
  }
}
```

Compiling this without any flags will leave only the `trace("ok");` line in the body of the `main` method. The other branches are discarded while parsing the file. These other branches must still contain valid Haxe syntax, but the code is not type-checked.

The conditions after `#if` and `#elseif` allow the following expressions:

- Any identifier is replaced by the value of the compiler flag by the same name. Note that `-D some-flag` from command line leads to the flags `some-flag` and `some_flag` to be defined.
- The values of `String`, `Int` and `Float` constants are used directly.
- The boolean operators `&&` (and), `||` (or) and `!` (not) work as expected.
- The operators `==`, `!=`, `>`, `>=`, `<`, `<=` can be used to compare values.
- Parentheses `()` can be used to group expressions as usual.

The Haxe parser does not parse `some-flag` as a single token and instead reads it as a subtraction binary operator `some - flag`. In cases like this the underscore version `some_flag` has to be used.

Built-in Compiler Flags An exhaustive list of all built-in defines can be obtained by invoking the Haxe Compiler with the `--help-defines` argument. The Haxe Compiler allows multiple `-D` flags per compilation.

See also the Compiler Flags list (6.1.1).

6.1.1 Global Compiler Flags

Starting from Haxe 3.0, you can get the list of supported compiler flags (6.1) by running `haxe --help-defines`

Flag	Description
absolute-path	Print absolute file path in trace output
advanced-telemetry	Allow the SWF to be measured with Monocle tool
as3	Defined when outputing flash9 as3 source code
check-xml-proxy	Check the used fields of the xml proxy
core-api	Defined in the core api context
cppia	Generate experimental cpp instruction assembly
dce	The current Dead Code Elimination (8.2) mode
dce-debug	Show Dead Code Elimination (8.2) log
debug	Activated when compiling with `-debug`
display	Activated during completion
dll-export	GenCPP experimental linking
dll-import	GenCPP experimental linking
doc-gen	Do not perform any removal/change in order to correctly gene
dump	Dump the complete typed AST for internal debugging
dump-dependencies	Dump the classes dependencies
fdb	Enable full flash debug infos for FDB interactive debugging
flash-strict	More strict typing for flash target
flash-use-stage	Keep the SWF library initial stage
format-warning	Print a warning for each formated string. for 2.x compatibility
gencommon-debug	GenCommon internal
haxe-boot	Given the name 'haxe' to the flash boot class instead of a gene
haxe-ver	The current Haxe version value
hxcpp-api-level	Provided to allow compatibility between hxcpp versions
include-prefix	prepend path to generated include files
interp	The code is compiled to be run with `--interp`
java-ver=[version:5-7]	Sets the Java version to be targeted
js-classic	Don't use a function wrapper and strict mode in JS output
js-es5	Generate JS for ES5-compliant runtimes
js-flatten	Generate classes to use fewer object property lookups
macro	Defined when we compile code in the macro context (9)
macro-times	Display per-macro timing when used with `--times`
neko-source	Output neko source instead of bytecode
neko-v1	Keep Neko 1.x compatibility
net-target=<name>	Sets the .NET target. Defaults to net. xbox, micro _(Micro Fram
net-ver=<version:20-45>	Sets the .NET version to be targeted
network-sandbox	Use local network sandbox instead of local file access one
no-compilation	Disable CPP final compilation
no-copt	Disable completion optimization _(for debug purposes)_
no-debug	Remove all debug macros from cpp output
no-deprecation-warnings	Do not warn if fields annotated with `@:deprecated` are used
no-flash-override	Change overrides on some basic classes into HX suffixed meth
no-inline	Disable inlining (4.4.2)
no-macro-cache	Disable macro context caching
no-opt	Disable optimizations
no-pattern-matching	Disable pattern matching (6.4)
no-root	GenCS internal
no-swf-compress	Disable SWF output compression
no-traces	Disable all `trace` calls
php-prefix	Compiled with `--php-prefix`
real-position	Disables haxe source mapping when targetting C#
replace-files	GenCommon internal
scriptable	GenCPP internal
shallow-expose	Expose types to surrounding scope of Haxe generated closure
source-map-content	Include the hx sources as part of the JS source map
swc	Output a SWC instead of a SWF
swf-compress-level=<level:1-9>	Set the amount of compression for the SWF output
swf-debug-password=<yourPassword>	Set a password for debugging. The password field is encrypted
swf-direct-blit	Use hardware acceleration to blit graphics
swf-gpu	Use GPU compositing features when drawing graphics

6.2 Externs

Externs can be used to describe target-specific interaction in a type-safe manner. They are defined like normal classes, except that

- the `class` keyword is preceded by the `extern` keyword,
- methods (4.3) have no expressions and
- all argument and return types are explicit.

A common example from the Haxe Standard Library (10) is the `Math` class, as an excerpt shows:

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

We see that externs can define both methods and variables (actually, `PI` is declared as a read-only property (4.2)). Once this information is available to the compiler, it allows field access accordingly and also knows the types:

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

This works because the return type of method `floor` is declared to be `Int`.

The Haxe Standard Library comes with many externs for the Flash and Javascript target. They allow accessing the native APIs in a type-safe manner and are instrumental for designing higher-level APIs. There are also externs for many popular native libraries on haxelib (11).

The Flash, Java and C# targets allow direct inclusion of native libraries from command line (7). Target-specific details are explained in the respective sections of Target Details (Chapter 12).

Some targets such as Python or JavaScript may require generating additional "import" code that loads an `extern` class from a native module. Haxe provides ways to declare such dependencies also described in respective sections Target Details (Chapter 12).

Rest arguments and type choices Since Haxe 3.2.0

The haxe.extern package provides two types that help mapping native semantics to Haxe:

`Rest<T>`: This type can be used as a final function argument to allow passing an arbitrary number of additional call arguments. The type parameter can be used to constrain these arguments to a specific type.

`EitherType<T1,T2>`: This type allows using either of its parameter types, thus representing a type choice. It can be nested to allow more than two different types.

We demonstrate the usage in this code sample:

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

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

class Main {
  static function main() {
    MyExtern.f1("foo", 1, 2, 3); // use 1, 2, 3 as rest argument
    MyExtern.f1("foo"); // no rest argument
    //MyExtern.f1("foo", "bar"); // String should be Int

    MyExtern.f2("foo");
    MyExtern.f2(12);
    //MyExtern.f2(true); // Bool should be EitherType<Int, String>
  }
}
```

6.3 Static Extension

> Definition: Static Extension
> A static extension allows pseudo-extending existing types without modifying their source. In Haxe this is achieved by declaring a static method with a first argument of the extending type and then bringing the defining class into context through `using`.

Static extensions can be a powerful tool which allows augmenting types without actually changing them. The following example demonstrates the usage:

```
using Main.IntExtender;

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

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

Clearly, `Int` does not natively provide a `triple` method, yet this program compiles and outputs `36` as expected. This is because the call to `12.triple()` is transformed into `IntExtender.triple(12)`. There are three requirements for this:

1. Both the literal `12` and the first argument of `triple` are of type `Int`.

2. The class `IntExtender` is brought into context through `using Main.IntExtender`.

3. `Int` does not have a `triple` field by itself (if it had, that field would take priority over the static extension).

Static extensions are usually considered syntactic sugar and indeed they are, but it is worth noting that they can have a dramatic effect on code readability: Instead of nested calls in the form of `f1(f2(f3(f4(x))))`, chained calls in the form of `x.f4().f3().f2().f1()` can be used.

Following the rules previously described in Resolution Order (Section 3.7.3), multiple `using` expressions are checked from bottom to top, with the types within each module as well as the fields within each type being checked from top to bottom. Using a module (as opposed to a specific type of a module, see モジュールとパス (Section 3.7)) as static extension brings all its types into context.

6.3.1 In the Haxe Standard Library

Several classes in the Haxe Standard Library are suitable for static extension usage. The next example shows the usage of `StringTools`:

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

While `String` does not have a `replace` functionality by itself, the `using StringTools` static extension provides one. As usual, the Javascript output nicely shows the transformation:

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

The following classes from the Haxe Standard Library are designed to be used as static extensions:

`StringTools`: Provides extended functionality on strings, such as replacing or trimming.

`Lambda`: Provides functional methods on iterables.

`haxe.EnumTools`: Provides type information functionality on enums and their instances.

`haxe.macro.Tools`: Provides different extensions for working with macros (see Tools (Section 9.4)).

Trivia: “using” using
Since the `using` keyword was added to the language, it has been common to talk about certain problems with “using using” or the effect of “using using”. This makes for awkward English in many cases, so the author of this manual decided to call the feature by what it actually is: Static extension.

6.4 Pattern Matching

6.4.1 Introduction

Pattern matching is the process of branching depending on a value matching given, possibly deep patterns. In Haxe, all pattern matching is done within a `switch` expression (5.17) where the individual `case` expressions represent the patterns. Here we will explore the syntax for different patterns using this data structure as running example:

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

Some pattern matcher basics include:

- Patterns will always be matched from top to bottom.
- The topmost pattern that matches the input value has its expression executed.
- A _ pattern matches anything, so `case _:` is equal to `default:`

6.4.2 Enum matching

Enums can be matched by their constructors in a natural way:

```
    var myTree = Node(Leaf("foo"), Node(Leaf("bar"), Leaf("foobar")));
    var match = switch(myTree) {
      // matches any Leaf
      case Leaf(_): "0";
      // matches any Node that has r = Leaf
      case Node(_, Leaf(_)): "1";
      // matches any Node that has has
      // r = another Node, which has
      // l = Leaf("bar")
      case Node(_, Node(Leaf("bar"), _)): "2";
      // matches anything
      case _: "3";
    }
    trace(match); // 2
```

The pattern matcher will check each case from top to bottom and pick the first one that matches the input value. The following manual interpretation of each case rule helps understanding the process:

`case Leaf(_)`: matching fails because `myTree` is a `Node`

`case Node(_, Leaf(_))`: matching fails because the right sub-tree of `myTree` is not a `Leaf`, but another `Node`

`case Node(_, Node(Leaf("bar"), _))`: matching succeeds

`case _`: this is not checked here because the previous line matched

6.4.3 Variable capture

It is possible to catch any value of a sub-pattern by matching it against an identifier:

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

This would return one of the following:

- If `myTree` is a `Leaf`, its name is returned.
- If `myTree` is a `Node` whose left sub-tree is a `Leaf`, its name is returned (this will apply here, returning `"foo"`).
- Otherwise `"none"` is returned.

It is also possible to use = to capture values which are further matched:

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

Here, `leafNode` is bound to `Leaf("foo")` if the input matches that. In all other cases, `myTree` itself is returned: `case x` works similar to `case _` in that it matches anything, but with an identifier name like `x` it also binds the matched value to that variable.

6.4.4 Structure matching

It is also possible to match against the fields of anonymous structures and instances:

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

In the second case we bind the matched `name` field to identifier `n` if `rating` matches `"awesome"`. Of course this structure could also be put into the `Tree` from the previous example to combine structure and enum matching.

A limitation with regards to class instances is that you cannot match against fields of their parent class.

6.4.5 Array matching

Arrays can be matched on fixed length:

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

This will trace `1` because `array[1]` matches `6`, and `array[0]` is allowed to be anything.

6.4.6 Or patterns

The `|` operator can be used anywhere within patterns to describe multiple accepted patterns:

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

If there is a captured variable in an or-pattern, it must appear in both its sub-patterns.

6.4.7 Guards

It is also possible to further restrict patterns with the `case ... if(condition):` syntax:

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

The first case has an additional guard condition `if (b > a)`. It will only be selected if that condition holds, otherwise matching continues with the next case.

6.4.8 Match on multiple values

Array syntax can be used to match on multiple values:

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

```
    }
    trace(s); // 2
```

This is quite similar to usual array matching, but there are differences:

- The number of elements is fixed, so patterns of different array length will not be accepted.
- It is not possible to capture the switch value in a variable, i.e. `case x` is not allowed (`case _` still is).

6.4.9 Extractors

Since Haxe 3.1.0

Extractors allow applying transformations to values being matched. This is often useful when a small operation is required on a matched value before matching can continue:

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

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

Here we have to capture the argument value of the `TString` enum constructor in a variable `temp` and use a nested switch on `temp.toLowerCase()`. Obviously, we want matching to succeed if `TString` holds a value of `"foo"` regardless of its casing. This can be simplified with extractors:

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

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

```
13            false;
14        }
15    }
16 }
```

Extractors are identified by the `extractorExpression => match` expression. The compiler generates code which is similar to the previous example, but the original syntax was greatly simplified. Extractors consist of two parts, which are separated by the `=>` operator:

1. The left side can be any expression, where all occurrences of underscore _ are replaced with the currently matched value.
2. The right side is a pattern which is matched against the result of the evaluation of the left side.

Since the right side is a pattern, it can contain another extractor. The following example "chains" two extractors:

```
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 }
```

This traces `12` as a result of the calls to `add(3, 1)`, where `3` is the matched value, and `mul(4, 3)` where `4` is the result of the `add` call. It is worth noting that the `a` on the right side of the second `=>` operator is a capture variable (6.4.3).

It is currently not possible to use extractors within or-patterns (6.4.6):

```
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  }
```

However, it is possible to have or-patterns on the right side of an extractor, so the previous example would compile without the parentheses.

6.4.10 Exhaustiveness checks

The compiler ensures that no possible cases are forgotten:

```
switch(true) {
    case false:
} // Unmatched patterns: true
```

The matched type `Bool` admits two values `true` and `false`, but only `false` is checked.

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

6.4.11 Useless pattern checks

In a similar fashion, the compiler detects patterns which will never match the input value:

```
switch(Leaf("foo")) {
    case Leaf(_)
        | Leaf("foo"): // This pattern is unused
    case Node(l,r):
    case _: // This pattern is unused
}
```

6.5 String Interpolation

With Haxe 3 it is no longer necessary to manually concatenate parts of a string due to the introduction of String Interpolation. Special identifiers, denoted by the dollar sign $ within a String enclosed by single-quote ' characters, are evaluated as if they were concatenated identifiers:

```
var x = 12;
// The value of x is 12
trace('The value of x is $x');
```

Furthermore, it is possible to include whole expressions in the string by using $\${expr}$, with `expr` being any valid Haxe expression:

```
var x = 12;
// The sum of 12 and 3 is 15
trace('The sum of $x and 3 is ${x + 3}');
```

String interpolation is a compile-time feature and has no impact on the runtime. The above example is equivalent to manual concatenation, which is exactly what the compiler generates:

```
trace("The sum of " + x +
  " and 3 is " + (x + 3));
```

Of course the use of single-quote enclosed strings without any interpolation remains valid, but care has to be taken regarding the $ character as it triggers interpolation. If an actual dollar-sign should be used in the string, $$ can be used.

> Trivia: String Interpolation before Haxe 3
> String Interpolation has been a Haxe feature since version 2.09. Back then, the macro `Std.format` had to be used, being both slower and less comfortable than the new string interpolation syntax.

6.6 Array Comprehension

Comprehensions are only listing Arrays, not Maps

Array comprehension in Haxe uses existing syntax to allow concise initialization of arrays. It is identified by `for` or `while` constructs:

```
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 }
```

Variable `a` is initialized to an array holding the numbers 0 to 9. The compiler generates code which adds the value of each loop iteration to the array, so the following code would be equivalent:

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

Variable `b` is initialized to an array with the same values, but through a different comprehension style using `while` instead of `for`. Again, the following code would be equivalent:

```
1  var i = 0;
2  var a = [];
3  while(i < 10) a.push(i++);
```

The loop expression can be anything, including conditions and nested loops, so the following works as expected:

```
1  class AdvArrayComprehension {
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 Iterators

With Haxe it is very easy to define custom iterators and iterable data types. These concepts are represented by the types `Iterator<T>` and `Iterable<T>` respectively:

```
1  typedef Iterator<T> = {
2      function hasNext() : Bool;
```

```
    function next() : T;
}

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

Any class (2.3) which structurally unifies (3.5.2) with one of these types can be iterated over using a for-loop (5.13). That is, if the class defines methods `hasNext` and `next` with matching return types it is considered an iterator, if it defines a method `iterator` returning an `Iterator<T>` it is considered an iterable type.

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

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

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

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

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

The type `MyStringIterator` in this example qualifies as iterator: It defines a method `hasNext` returning `Bool` and a method `next` returning `String`, making it compatible with `Iterator<String>`. The `main` method instantiates it, then iterates over it.

```
class MyArrayWrap<T> {
  var a:Array<T>;
  public function new(a:Array<T>) {
    this.a = a;
  }

  public function iterator() {
    return a.iterator();
  }
}

```

```
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 }
```

Here we do not setup a full iterator like in the previous example, but instead define that the `MyArrayWrap<T>` has a method `iterator`, effectively forwarding the iterator method of the wrapped `Array<T>` type.

6.8 Function Bindings

Haxe 3 allows binding functions with partially applied arguments. Each function type can be considered to have a `bind` field, which can be called with the desired number of arguments in order to create a new function. This is demonstrated here:

```
1  class Bind {
2    static public function main() {
3      var map = new Map<Int,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 }
```

Line 4 binds the function `map.set` to a variable named `f`, and applies `12` as second argument. The underscore _ is used to denote that this argument is not bound, which is shown by comparing the types of `map.set` and `f`: The bound `String` argument is effectively cut from the type, turning a `Int->String->Void` type into `Int->Void`.

A call to `f(1)` then actually invokes `map.set(1, "12")`, the calls to `f(2)` and `f(3)` are analogous. The last line proves that all three indices indeed are mapped to the value `"12"`.

The underscore _ can be skipped for trailing arguments, so the first argument could be bound through `map.set.bind(1)`, yielding a `String->Void` function that sets a new value for index `1` on invocation.

Trivia: Callback
Prior to Haxe 3, Haxe used to know a `callback`-keyword which could be called with a function argument followed by any number of binding arguments. The name originated from a common usage were a callback-function is created with the this-object being bound.
Callback would allow binding of arguments only from left to right as there was no support for the underscore _. The choice to use an underscore was controversial and several other suggestions were made, none of which were considered superior. After all,

the underscore _ at least looks like it's saying "fill value in here", which nicely describes its semantics.

6.9 Metadata

Several constructs can be attributed with custom metadata:

- `class` and `enum` declarations
- Class fields
- Enum constructors
- Expressions

These metadata information can be obtained at runtime through the `haxe.rtti.Meta` API:

```
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 }
```

We can easily identify metadata by the leading `@` character, followed by the metadata name and, optionally, by a number of comma-separated constant arguments enclosed in parentheses.

- Class `MyClass` has an `author` metadata with a single String argument `"Nicolas"`, as well as a `debug` metadata without arguments.
- The member variable `value` has a `range` metadata with two Int arguments `1` and `8`.
- The static method `method` has a `broken` metadata without arguments, as well as a `:noCompletion` metadata without arguments.

The `main` method accesses these metadata values using their API. The output reveals the structure of the obtained data:

- There is a field for each metadata, with the field name being the metadata name.
- The field values correspond to the metadata arguments. If there are no arguments, the field value is `null`. Otherwise the field value is an array with one element per argument.
- Metadata starting with `:` is omitted. This kind of metadata is known as compiler metadata.

Allowed values for metadata arguments are:

- Constants (5.2)
- Arrays declarations (5.5) (if all their elements qualify)
- Object declarations (5.6) (if all their field values qualify)

Built-in Compiler Metadata An exhaustive list of all defined metadata can be obtained by running `haxe --help-metas` from command line.

See also the Compiler Metadata list (8.1).

6.10 Access Control

Access control can be used if the basic visibility (4.4.1) options are not sufficient. It is applicable at class-level and at field-level and knows two directions:

Allowing access: The target is granted access to the given class or field by using the `:allow(target)` metadata (6.9).

Forcing access: A target is forced to allow access to the given class or field by using the `:access(target)` metadata (6.9).

In this context, a target can be the dot-path (3.7) to

- a class field,
- a class or abstract type, or
- a package.

If it is a class or abstract type, access modification extends to all fields of that type. Likewise, if it is a package, access modification extends to all types of that package and recursively to all fields of these types.

```
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 }
```

Here, `MyClass.foo` can be accessed from the `main`-method because `MyClass` is annotated with `@:allow(Main)`. This would also work with `@:allow(Main.main)` and both versions could alternatively be annotated to the field `foo` instead of the class `MyClass`:

```
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 }
```

If a type cannot be modified to allow this kind of access, the accessing method may force access:

```
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 }
```

The `@:access(MyClass.foo)` annotation effectively subverts the visibility of the `foo` field within the `main`-method.

Trivia: On the choice of metadata
The access control language feature uses the Haxe metadata syntax instead of additional language-specific syntax. There are several reasons for that:

- Additional syntax often adds complexity to the language parsing, and also adds (too) many keywords.
- Additional syntax requires additional learning by the language user, whereas metadata syntax is something that is already known.
- The metadata syntax is flexible enough to allow extension of this feature.
- The metadata can be accessed/generated/modified by Haxe macros.

Of course, the main drawback of using metadata syntax is that you get no error report in case you misspell either the metadata key (@:acesss for instance) or the class/package name. However, with this feature you will get an error when you try to access a private field that you are not allowed to, therefore there is no possibility for silent errors.

Since Haxe 3.1.0

If access is allowed to an interface (2.3.3), it extends to all classes implementing that interface:

```
class MyClass {
  @:allow(I)
  static private var foo: Int;
}

interface I { }

class Main implements I {
  static public function main() {
    MyClass.foo;
  }
}
```

This is also true for access granted to parent classes, in which case it extends to all child classes.

> Trivia: Broken feature
> Access extension to child classes and implementing classes was supposed to work in Haxe 3.0 and even documented accordingly. While writing this manual it was found that this part of the access control implementation was simply missing.

6.11 Inline constructors

Since Haxe 3.1.0

If a constructor is declared to be inline (4.4.2), the compiler may try to optimize it away in certain situations. There are several requirements for this to work:

- The result of the constructor call must be directly assigned to a local variable.
- The expression of the constructor field must only contain assignments to its fields.

The following example demonstrates constructor inlining:

```
class Point {
  public var x:Float;
  public var y:Float;

  public inline function new(x:Float, y:Float) {
    this.x = x;
    this.y = y;
  }
}

class Main {
  static public function main() {
    var pt = new Point(1.2, 9.3);
  }
}
```

A look at the Javascript output reveals the effect:

```
Main.main = function() {
    var pt_x = 1.2;
    var pt_y = 9.3;
};
```

Part II

Compiler Reference

Chapter 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 (Chapter 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.

Chapter 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`

Global meta	
Metadata	Description
@:abstract	Sets the underlying class implementation as abstract ty
@:access _(Target path)_	Forces private access to package type or field, see Acce
@:allow _(Target path)_	Allows private access from package type or field, see Ac
@:annotation	Annotation (`@interface`) definitions on `-java-lib` impor
@:arrayAccess	Allows Array access (2.8.3) on an abstract
@:autoBuild _(Build macro call)_	Extends `@:build` metadata to all extending and impleme
@:bind	Override Swf class declaration
@:bitmap _(Bitmap file path)_	_Embeds given bitmap data into the class (must extend
@:build _(Build macro call)_	Builds a class or enum from a macro. See Type Buildin
@:buildXml	
@:classCode	Used to inject platform-native code into a class
@:commutative	Declares an abstract operator as commutative
@:compilerGenerated	Marks a field as generated by the compiler. Shouldn't b
@:coreApi	Identifies this class as a core api class (forces Api chec
@:coreType	Identifies an abstract as core type (2.8.7) so that it requ
@:cppFileCode	
@:cppNamespaceCode	
@:dce	Forces Dead Code Elimination (8.2) even when not `-dc`
@:debug	Forces debug information to be generated into the Swf
@:decl	
@:defParam	
@:delegate	Automatically added by `-net-lib` on delegates
@:depend	
@:deprecated	Automatically added by `-java-lib` on class fields annot
@:event	Automatically added by `-net-lib` on events. Has no effe
@:enum	Defines finite value sets to abstract definitions. See enu
@:expose _(?Name=Class path)_	Makes the class available on the `window` object or `export`
@:extern	Marks the field as extern so it is not generated
@:fakeEnum _(Type name)_	Treat enum as collection of values of the specified type
@:file(File path)	Includes a given binary file into the target Swf and asso
@:final	Prevents a class from being extended
@:font _(TTF path Range String)_	Embeds the given TrueType font into the class (must ex
@:forward _(List of field names)_	Forwards field access (2.8.6) to underlying type
@:from	Specifies that the field of the abstract is a cast operation
@:functionCode	
@:functionTailCode	
@:generic	Marks a class or class field as generic (3.3) so each type
@:genericBuild	Builds instances of a type using the specified macro
@:getter _(Class field name)_	Generates a native getter function on the given field
@:hack	Allows extending classes marked as `@:final`
@:headerClassCode	
@:headerCode	
@:headerNamespaceCode	
@:hxGen	Annotates that an extern class was generated by Haxe
@:ifFeature _(Feature name)_	Causes a field to be kept by DCE (8.2) if the given featu
@:include	
@:initPackage	
@:internal	Generates the annotated field/class with `internal` acces
@:isVar	Forces a physical field to be generated for properties th
@:keep	Causes a field or type to be kept by DCE (8.2)
@:keepInit	Causes a class to be kept by DCE (8.2) even if all its fie
@:keepSub	Extends `@:keep` metadata to all implementing and exten
@:macro	_(deprecated)_
@:meta	Internally used to mark a class field as being the metad
@:multiType _(Relevant type parameters)_	Specifies that an abstract chooses its this-type from its
@:native _(Output type path)_	Rewrites the path of a class or enum during generation
@:nativeGen	Annotates that a type should be treated as if it were an
@:noCompletion	Prevents the compiler from suggesting completion (8.3)
@:noDebug	Does not generate debug information into the Swf even

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.macro.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).

Trivia: 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.

Trivia: 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.
- 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:

```
class Main {
  public static function main() {
    trace("Hello".|
  }
}
```

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):

```
<list>
<i n="length">
  <t>Int</t>
  <d>
    The number of characters in `this` String.
  </d>
</i>
<i n="charAt">
  <t>index : Int -&gt; String</t>
  <d>
    Returns the character at position `index` of `this` String.
    If `index` is negative or exceeds `this.length`, the empty String
    "" is returned.
  </d>
</i>
<i n="charCodeAt">
  <t>index : Int -&gt; Null&lt;Int&gt;</t>
  <d>
    Returns the character code at position `index` of `this` String.
    If `index` is negative or exceeds `this.length`, null is returned.
    To obtain the character code of a single character, "x".code can
    be used instead to inline the character code at compile time.
    Note that this only works on String literals of length 1.
  </d>
</i>
</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.

Since Haxe 3.2.0

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&lt;String&gt;
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>`.

> Trivia: 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>
```

```
<i n="PosInfos"><t></t><d></d></i>
<i n="Resource"><t></t><d></d></i>
<i n="Serializer"><t></t><d></d></i>
<i n="Template"><t></t><d></d></i>
<i n="Timer"><t></t><d></d></i>
<i n="Ucs2"><t></t><d></d></i>
<i n="Unserializer"><t></t><d></d></i>
<i n="Utf8"><t></t><d></d></i>
<i n="crypto"><t></t><d></d></i>
<i n="ds"><t></t><d></d></i>
<i n="extern"><t></t><d></d></i>
<i n="format"><t></t><d></d></i>
<i n="io"><t></t><d></d></i>
<i n="macro"><t></t><d></d></i>
<i n="remoting"><t></t><d></d></i>
<i n="rtti"><t></t><d></d></i>
<i n="unit"><t></t><d></d></i>
<i n="web"><t></t><d></d></i>
<i n="xml"><t></t><d></d></i>
<i n="zip"><t></t><d></d></i>
</list>
```

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):

```
import haxe.Unserializer.|
```

```
<list>
<i n="DEFAULT_RESOLVER">
  <t>haxe.TypeResolver</t>
  <d>
    This value can be set to use custom type resolvers.

    A type resolver finds a Class or Enum instance from a given String
       .
    By default, the haxe Type Api is used.

    A type resolver must provide two methods:

    1. resolveClass(name:String):Class&lt;Dynamic&gt; is called to
      determine a Class from a class name
    2. resolveEnum(name:String):Enum&lt;Dynamic&gt; is called to
      determine an Enum from an enum name

    This value is applied when a new Unserializer instance is created.
    Changing it afterwards has no effect on previously created
    instances.
  </d>
</i>
<i n="run">
  <t>v : String -&gt; Dynamic</t>
```

```
  <d>
    Unserializes `v` and returns the according value.

    This is a convenience function for creating a new instance of
    Unserializer with `v` as buffer and calling its unserialize()
    method once.
  </d>
</i>
<i n="TypeResolver"><t></t><d></d></i>
<i n="Unserializer"><t></t><d></d></i>
</list>
```

```
using haxe.Unserializer.|
```

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

```
using haxe.Unserializer.|
```

```
class Main {
  static public function main() {
    var x:haxe.Unserializer.|
  }
}
```

```
<list>
<i n="TypeResolver"><t></t><d></d></i>
<i n="Unserializer"><t></t><d></d></i>
</list>
```

8.3.5 Usage completion

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

```
class Main {
  public static function main() {
    var a = 1;
    var b = a + 1;
    trace(a);
    a.|
  }
}
```

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:

```
<list>
<pos>main.hx:4: characters 9-10</pos>
<pos>main.hx:5: characters 7-8</pos>
<pos>main.hx:6: characters 1-2</pos>
</list>
```

8.3.6 Position completion

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

```
class Main {
  static public function main() {
    "foo".split.|
}
```

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:

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

> Trivia: 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

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:

```
class Main {
  static public function main() {
    var a = 1;
  }
}

enum MyEnum {
  MyConstructor1;
  MyConstructor2(s:String);
}
```

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):

```
<il>
<i k="local" t="Int">a</i>
<i k="static" t="Void -&gt; Unknown&lt;0&gt;">main</i>
<i k="enum" t="MyEnum">MyConstructor1</i>
<i k="enum" t="s : String -&gt; MyEnum">MyConstructor2</i>
<i k="package">sys</i>
<i k="package">haxe</i>
```

```
<i k="type" p="Int">Int</i>
<i k="type" p="Float">Float</i>
<i k="type" p="MyEnum">MyEnum</i>
<i k="type" p="Main">Main</i>
</il>
```

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:

```
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 :

```
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.hxml" + newline);
8          s.write("¥textbackslash¥ 000");
9
10         var hasError = false;
11         for (line in s.read().split(newline))
12         {
13             switch (line.charCodeAt(0)) {
14                 case 0x01:
15                     neko.Lib.print(line.substr(1).split("¥
                           textbackslash¥ x01").join(newline));
16                 case 0x02:
17                     hasError = true;
18                 default:
19                     neko.io.File.stderr().writeString(line + newline);
20             }
21         }
22         if (hasError) neko.Sys.exit(1);
23     }
24 }
```

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:

what to use for listing of non-haxe code like hxml? [1]

```
-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.
- 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 (Section 8.5.1).
Since Haxe 3.2.0

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 (Section 8.5.1).

statics: The list of static class fields, described in Class field information (Section 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.

Chapter 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.

> Definition: 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.

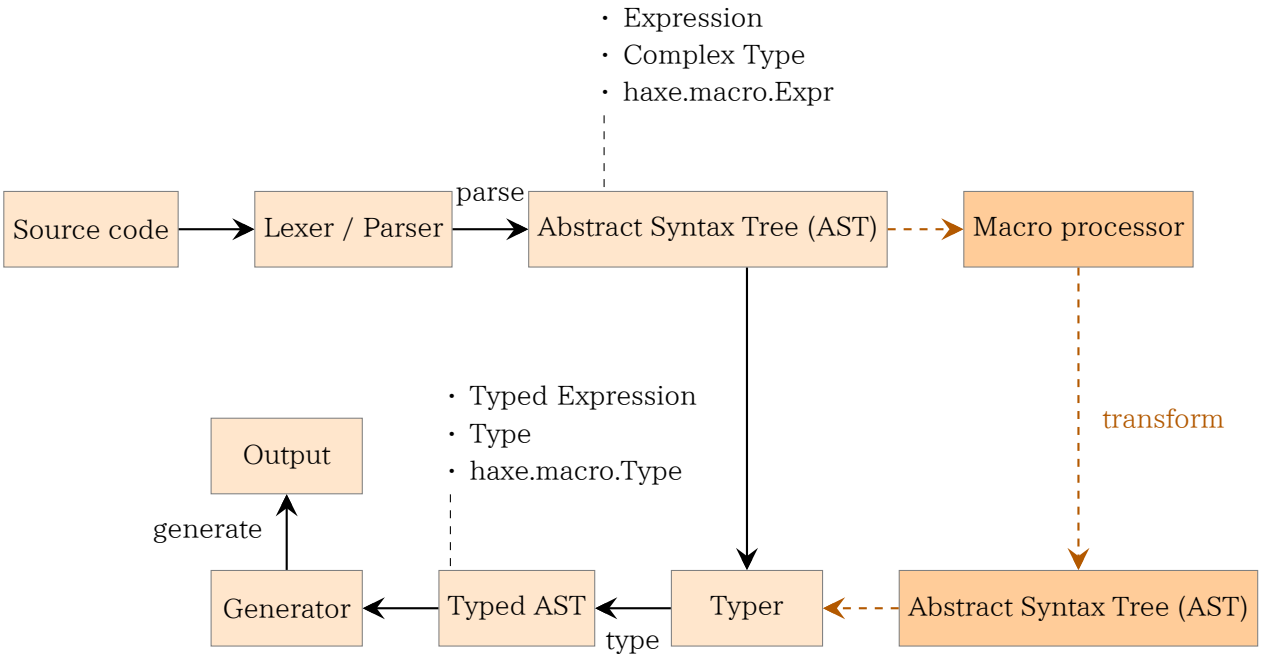

Figure 9.1: The role of macros during compilation.

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 (Section 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 (Section 9.5)).

Expression Macros: These are normal functions which are executed as soon as they are typed.

9.1 Macro Context

Definition: 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 (Section 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:

```
class Main {
  static public function main() {
    const("foo", 1, 1.5, true);
  }

  macro static function const(s:String, i:Int, f:Float, b:Bool) {
    trace(s);
    trace(i);
    trace(f);
    trace(b);
    return macro null;
  }
}
```

With these it is not necessary to detour over expressions as the compiler can use the provided constants directly.

9.2.3 Rest Argument

If the final argument of a macro is of type `Array<Expr>`, the macro accepts an arbitrary number of extra arguments which are available from that array:

```
import haxe.macro.Expr;

class Main {
  static public function main() {
    myMacro("foo", a, b, c);
  }

  macro static function myMacro(e1:Expr, extra:Array<Expr>) {
    for (e in extra) {
      trace(e);
    }
    return macro null;
  }
}
```

9.3 Reification

The Haxe Compiler allows reification of expressions, types and classes to simplify working with macros. The syntax for reification is `macro expr`, where `expr` is any valid Haxe expression.

9.3.1 Expression Reification

Expression reification is used to create instances of `haxe.macro.Expr` in a convenient way. The Haxe Compiler accepts the usual Haxe syntax and translates it to an expression object. It supports several escaping mechanisms, all of which are triggered by the `$` character:

`${}` and `$e{}`: `Expr -> Expr` This can be used to compose expressions. The expression within the delimiting `{ }` is executed, with its value being used in place.

`$a{}`: `Expr -> Array<Expr>` If used in a place where an `Array<Expr>` is expected (e.g. call arguments, block elements), `$a{}` treats its value as that array. Otherwise it generates an array declaration.

`$b{}`: `Array<Expr> -> Expr` Generates a block expression from the given expression array.

`$i{}`: `String -> Expr` Generates an identifier from the given string.

`$p{}`: `Array<String> -> Expr` Generates a field expression from the given string array.

`$v{}`: `Dynamic -> Expr` Generates an expression depending on the type of its argument. This is only guaranteed to work for basic types (2.1) and enum instances (2.4).

Additionally the metadata (6.9) `@:pos(p)` can be used to map the position of the annotated expression to `p` instead of the place it is reified at.

This kind of reification only works in places where the internal structure expects an expression. This disallows `object.${fieldName}`, but `object.$fieldName` works. This is true for all places where the internal structure expects a string:

- field access `object.$name`
- variable name `var $name = 1;`

Since Haxe 3.1.0

- field name `{ $name: 1}`
- function name `function $name() { }`
- catch variable name `try e() catch($name:Dynamic) {}`

9.3.2 Type Reification

Type reification is used to create instances of `haxe.macro.Expr.ComplexType` in a convenient way. It is identified by a `macro : Type`, where `Type` can be any valid type path expression. This is similar to explicit type hints in normal code, e.g. for variables in the form of `var x:Type`.

Each constructor of `ComplexType` has a distinct syntax:

```
TPath: macro : pack.Type

TFunction: macro : Arg1 -> Arg2 -> Return

TAnonymous: macro : { field: Type }

TParent: macro : (Type)

TExtend: macro : {> Type, field: Type }

TOptional: macro : ?Type
```

9.3.3 Class Reification

It is also possible to use reification to obtain an instance of `haxe.macro.Expr.TypeDefinition`. This is indicated by the `macro class` syntax as shown here:

```
class Main {
  macro static function generateClass(funcName:String) {
    var c = macro class MyClass {
      public function new() { }
      public function $funcName() {
        trace($v{funcName} + " was called");
      }
    }
    haxe.macro.Context.defineType(c);
    return macro new MyClass();
  }

  public static function main() {
    var c = generateClass("myFunc");
    c.myFunc();
  }
}
```

The generated `TypeDefinition` instance is typically passed to `haxe.macro.Context.defineType` in order to add a new type to the calling context (not the macro context itself).

This kind of reification can also be useful to obtain instances of `haxe.macro.Expr.Field`, which are available from the `fields` array of the generated `TypeDefinition`.

9.4 Tools

The Haxe Standard Library comes with a set of tool-classes to simplify working with macros. These classes work best as static extensions (6.3) and can be brought into context either individually or as a whole through `using haxe.macro.Tools`. These classes are:

`ComplexTypeTools`: Allows printing `ComplexType` instances in a human-readable way. Also allows determining the `Type` corresponding to a `ComplexType`.

`ExprTools`: Allows printing `Expr` instances in a human-readable way. Also allows iterating and mapping expressions.

`MacroStringTools`: Offers useful operations on strings and string expressions in macro context.

`TypeTools`: Allows printing `Type` instances in a human-readable way. Also offers several useful operations on types, such as unifying (3.5) them or getting their corresponding `ComplexType`.

Furthermore the `haxe.macro.Printer` class has public methods for printing various types as a human-readable format. This can be helpful when debugging macros.

Trivia: The tinkerbell library and why Tools.hx works
We learned about static extensions that using a module implies that all its types are brought into static extension context. As it turns out, such a type can also be a typedef (3.1) to another type. The compiler then considers this type part of the module, and extends static extension accordingly.
This "trick" was first used in Juraj Kirchheim's tinkerbell[1] library for exactly the same purpose. Tinkerbell provided many useful macro tools long before they made it into the Haxe Compiler and Haxe Standard Library. It remains the primary library for additional macro tools and offers other useful functionality as well.

9.5 Type Building

Type-building macros are different from expression macros in several ways:

- They do not return expressions, but an array of class fields. Their return type must be set explicitly to `Array<haxe.macro.Expr.Field>`.
- Their context (9.1) has no local method and no local variables.
- Their context does have build fields, available from `haxe.macro.Context.getBuildFields()`.
- They are not called directly, but are argument to a `@:build` or `@:autoBuild` metadata (6.9) on a class (2.3) or enum (2.4) declaration.

The following example demonstrates type building. Note that it is split up into two files for a reason: If a module contains a `macro` function, it has to be typed into macro context as well. This is often a problem for type-building macros because the type to be built could only be loaded in its incomplete state, before the building macro has run. We recommend to always define type-building macros in their own module.

```
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 {
```

```
  static public function main() {
    trace(Main.myFunc); // my default
  }
}
```

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.

```
import haxe.macro.Context;
import haxe.macro.Expr;

class EnumBuildingMacro {
  macro static public function build():Array<Field> {
    var noArgs = makeEnumField("A", FVar(null, null));
    var eFunc = macro function(value:Int) { };
    var fInt = switch (eFunc.expr) {
      case EFunction(_,f): f;
      case _: throw "false";
    }
```

```
    var intArg = makeEnumField("B", FFun(fInt));
    return [noArgs, intArg];
  }

  static function makeEnumField(name, kind) {
    return {
      name: name,
      doc: null,
      meta: [],
      access: [],
      kind: kind,
      pos: Context.currentPos()
    }
  }
}
```

```
@:build(EnumBuildingMacro.build())
enum E { }

class Main {
  static public function main() {
    switch(E.A) {
      case A:
      case B(v):
    }
  }
}
```

Because enum `E` is annotated with a `:build` metadata, the called macro builds two constructors `A` and `B` "into" it. The former is added with the kind being `FVar(null, null)`, meaning it is a constructor without argument. For the latter, we use reification (9.3.1) to obtain an instance of `haxe.macro.Expr.Function` with a singular `Int` argument.

The `main` method proves the structure of our generated enum by matching (6.4) it. We can see that the generated type is equivalent to this:

```
enum E {
    A;
    B(value:Int);
}
```

9.5.2 @:autoBuild

If a class has the `:autoBuild` metadata, the compiler generates `:build` metadata on all extending classes. If an interface has the `:autoBuild` metadata, the compiler generates `:build` metadata on all implementing classes and all extending interfaces. Note that `:autoBuild` does not imply `:build` on the class/interface itself.

```
import haxe.macro.Context;
import haxe.macro.Expr;

class AutoBuildingMacro {
  macro static public
```

```
  function fromInterface():Array<Field> {
    trace("fromInterface: " + Context.getLocalType());
    return null;
  }

  macro static public
  function fromBaseClass():Array<Field> {
    trace("fromBaseClass: " + Context.getLocalType());
    return null;
  }
}
```

```
@:autoBuild(AutoBuildingMacro.fromInterface())
interface I { }

interface I2 extends I { }

@:autoBuild(AutoBuildingMacro.fromBaseClass())
class Base { }

class Main extends Base implements I2 {
  static public function main() { }
}
```

This outputs during compilation:

```
AutoBuildingMacro.hx:6:
  fromInterface: TInst(I2,[])
AutoBuildingMacro.hx:6:
  fromInterface: TInst(Main,[])
AutoBuildingMacro.hx:11:
  fromBaseClass: TInst(Main,[])
```

It is important to keep in mind that the order of these macro executions is undefined, which is detailed in Build Order (Section 9.6.3).

9.5.3 @:genericBuild

Since Haxe 3.1.0

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:

```
// MyMacro.hx
import haxe.macro.Expr;
import haxe.macro.Context;
import haxe.macro.Type;

class MyMacro {
```

```
  static public function build() {
    switch (Context.getLocalType()) {
      case TInst(_, [t1]):
        trace(t1);
      case t:
        Context.error("Class expected", Context.currentPos());
    }
    return null;
  }
}

// Main.hx
@:genericBuild(MyMacro.build())
class MyType<T> { }

class Main {
  static function main() {
    var x:MyType<Int>;
    var x:MyType<String>;
  }
}
```

When running this example the compiler outputs `TAbstract(Int,[])` and `TInst(String,[])`, indicating that it is indeed aware of the concrete type parameters of `MyType`. The macro logic could use this information to generate a custom type (using `haxe.macro.Context.defineType`) or refer to an existing one. For brevity we return `null` here which asks the compiler to infer (3.6) the type.

In Haxe 3.1 the return type of a `@:genericBuild` macro has to be a `haxe.macro.Type`. Haxe 3.2 allows (and prefers) returning a `haxe.macro.ComplexType` instead, which is the syntactic representation of a type. This is easier to work with in many cases because types can simply be referenced by their paths.

Const type parameter Haxe allows passing constant expression (5.2) as a type parameter if the type parameter name is `Const`. This can be utilized in the context of `@:genericBuild` macros to pass information from the syntax directly to the macro:

```
import haxe.macro.Expr;
import haxe.macro.Context;
import haxe.macro.Type;

class MyMacro {
  static public function build() {
    switch (Context.getLocalType()) {
      case TInst(_,[TInst(_.get() => { kind: KExpr(macro $v{(s:String)
         }) },_)]):
        trace(s);
      case t:
        Context.error("Class expected", Context.currentPos());
    }
    return null;
  }
}
```

```
16
17 // Main.hx
18 @:genericBuild(MyMacro.build())
19 class MyType<Const> { }
20
21 class Main {
22   static function main() {
23     var x:MyType<"myfile.txt">;
24   }
25 }
```

Here the macro logic could load a file and use its contents to generate a custom type.

9.6 Limitations

9.6.1 Macro-in-Macro

9.6.2 Static extension

The concepts or static extensions (6.3) and macros are somewhat conflicting: While the former requires a known type in order to determine used functions, macros execute before typing on plain syntax. It is thus not surprising that combining these two features can lead to issues. Haxe 3.0 would try to convert the typed expression back to a syntax expression, which is not always possible and may lose important information. We recommend that it is used with caution.
Since Haxe 3.1.0

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   }
```

```
}
```

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:

```
import haxe.macro.Context;
import haxe.macro.Expr;

#if !macro
@:autoBuild(MyMacro.build())
#end
interface I1 { }

#if !macro
@:autoBuild(MyMacro.build())
#end
interface I2 { }

class C implements I1 implements I2 { }

class MyMacro {
  macro static public function build():Array<Field> {
    var c = Context.getLocalClass().get();
    if (c.meta.has(":processed")) return null;
    c.meta.add(":processed",[],c.pos);
    // process here
    return null;
  }
}

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

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.

Part III

Standard Library

Chapter 10

Standard Library

Standard library

10.1 String

Type: String
A String is a sequence of characters.

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:

```
class Main {
  static public function main() {
    var scores = [110, 170, 35];
    var sum = 0;
    for (score in scores) {
      sum += score;
    }
    trace(sum); // 315
  }
}
```

Haxe generates this optimized Javascript output:

```
Main.main = function() {
    var scores = [110,170,35];
    var sum = 0;
    var _g = 0;
    while(_g < scores.length) {
        var score = scores[_g];
        ++_g;
        sum += score;
    }
    console.log(sum);
};
```

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

```
class Main {
  static public function main() {
    // Compile Error: Arrays of mixed types are only allowed if the
       type is
    // forced to Array<Dynamic>
    //var myArray = [10, "Bob", false];

    // Array<Dynamic> with mixed types
    var myExplicitArray:Array<Dynamic> = [10, "Sally", true];
  }
}
```

Trivia: 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>`.

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.

```
class Main {
    static function main() {
    var vec = new haxe.ds.Vector(10);

    for (i in 0...vec.length) {
      vec[i] = i;
    }

    trace(vec[0]); // 0
    trace(vec[5]); // 5
    trace(vec[9]); // 9
    }
}
```

`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.

10.2.3 List

A `List` is a collection for storing elements. On the surface, a list is similar to an Array (Section 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.

See the List API for details about the list methods. A simple example for working with lists:

```
class ListExample {
  static public function main() {
    var myList = new List<Int>();
    for (ii in 0...5)
      myList.add(ii);
    trace(myList); //{0, 1, 2, 3, 4}
  }
}
```

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. See the GenericStack API for details about its methods. Here is a small example program for initializing and working with a `GenericStack`.

```
import haxe.ds.GenericStack;

class GenericStackExample {
    static public function main() {
        var myStack = new GenericStack<Int>();
        for (ii in 0...5)
            myStack.add(ii);
        trace(myStack); //{4, 3, 2, 1, 0}
        trace(myStack.pop()); //4
    }
}
```

> Trivia: 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 (Section 3.3) for more details.

10.2.5 Map

A `Map` is a container composed of key, value pairs. A `Map` is also commonly referred to as an associative array, dictionary, or symbol table. The following code gives a short example of working with maps:

```
class Main {
  static public function main() {
    // Maps are initialized like arrays, but
    // use '=>' operator. Maps can have their
    // key value types defined explicity
    var map1:Map<Int, String> =
      [1 => "one", 2=>"two"];

    // Or they can infer their key value types
    var map2 = [
      "one"=>1,
      "two"=>2,
      "three"=>3
    ];
    $type(map2); // Map<String, Int>

    // Keys must be unique
    // Error: Duplicate Key
```

```
19      //var map3 = [1=>"dog", 1=>"cat"];
20
21      // Maps values can be accessed using array
22      // accessors "[]"
23      var map4 = ["M"=>"Monday", "T"=>"Tuesday"];
24      trace(map4["M"]); //Monday
25
26      // Maps iterate over their values by
27      // default
28      var valueSum;
29      for (value in map4) {
30        trace(value); // Monday ¥n Tuesday
31      }
32
33      // Can iterate over keys by using the
34      // keys() method
35      for (key in map4.keys()) {
36        trace(key); // M ¥n T
37      }
38
39      // Like arrays, a new Map can be made using
40      // comprehension
41      var map5 = [
42        for (key in map4.keys())
43          key => "FRIDAY!!"
44      ];
45      // {M => FRIDAY!!, T => FRIDAY!!}
46      trace(map5);
47    }
48  }
```

See the Map API for details of its methods.

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.

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:

```
import haxe.ds.Option;

class Main {
  static public function main() {
    var result = trySomething();
    switch (result) {
      case None:
        trace("Got None");
      case Some(s):
        trace("Got a value: " +s);
    }
  }

  static function trySomething():Option<String> {
    if (Math.random() > 0.5) {
      return None;
    } else {
      return Some("Success");
    }
  }
}
```

10.3 Regular Expressions

Haxe has built-in support for regular expressions[1]. They can be used to verify the format of a string, transform a string or extract some regular data from a given text.

Haxe has special syntax for creating regular expressions. We can create a regular expression object by typing it between the ~/ combination and a single / character:

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

Alternatively, we can create regular expression with regular syntax:

```
var r = new EReg("haxe", "i");
```

First argument is a string with regular expression pattern, second one is a string with flags (see below).

We can use standard regular expression patterns such as:

- `.` any character
- `*` repeat zero-or-more
- `+` repeat one-or-more
- `?` optional zero-or-one
- `[A-Z0-9]` character ranges
- `[^¥r¥n¥t]` character not-in-range

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

- `(...)` 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 and Java targets only)
- `u` use UTF-8 matching (Neko and C++ targets only)

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:

```
class Main {
  static function main() {
    var str = "abcdeeeeefghi";
    var r = ~/e+/;
    r.match(str);
    trace(r.matched(0)); // "eeeee"
    // { pos : 4, len : 5 }
    trace(r.matchedPos());
  }
}
```

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

```
class Main {
  static function main() {
    var r = ~/b/;
    r.match("abc");
    trace(r.matchedLeft()); // a
    trace(r.matched(0)); // b
    trace(r.matchedRight()); // c
  }
}
```

10.3.3 Replace

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

```
class Main {
  static function main() {
    var str = "aaabcbcbcbz";
    // g : replace all instances
    var r = ~/b[^c]/g;
    // "aaabcbcbcxx"
    trace(r.replace(str,"xx"));
  }
}
```

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

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

10.3.4 Split

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

```
class Main {
  static function main() {
    var str = "XaaaYababZbbbW";
    var r = ~/[ab]+/g;
    // ["X","Y","Z","W"]
    trace(r.split(str));
  }
}
```

10.3.5 Map

The `map` method of a regular expression object can be used to replace matched substrings using a custom function. This function takes a regular expression object as its first argument so we may use it to get additional information about the match being done and do conditional replacement. For example:

```
class Main {
  static function main() {
    var r = ~/(dog|fox)/g;
    var s = "The quick brown fox jumped over the lazy dog.";
    var s2 = r.map(s, function(r) {
        var match = r.matched(0);
        switch (match) {
            case 'dog': return 'fox';
            case 'fox': return 'dog';
            default: throw 'Unknown animal: $match';
        };
    });
    trace(s2); // The quick brown dog jumped over the lazy fox.
  }
}
```

10.3.6 Implementation Details

Regular Expressions are implemented:

- in JavaScript, the runtime is providing the implementation with the object RegExp.
- in Neko and C++, the PCRE library is used
- in Flash, PHP, C# and Java, native implementations are used
- in Flash 6/8, the implementation is not available

10.4 Math

Haxe includes a floating point math library for some common mathematical operations. Most of the functions operate on and return `floats`. However, an `Int` can be used where

a `Float` is expected, and Haxe also converts `Int` to `Float` during most numeric operations (see 数値の演算子 (Section 2.1.3) for more details).

Here are some example uses of the math library. See the Math API for all available functions.

```
1  class MathExample {
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 }
```

10.4.1 Special Numbers

The math library has definitions for several special numbers:

- NaN (Not a Number): returned when a mathmatically incorrect operation is executed, e.g. Math.sqrt(-1)
- POSITIVE_INFINITY: e.g. divide a positive number by zero
- NEGATIVE_INFINITY: e.g. divide a negative number by zero
- PI : 3.1415...

10.4.2 Mathematical Errors

Although neko can fluidly handle mathematical errors, like division by zero, this is not true for all targets. Depending on the target, mathematical errors may produce exceptions and ultimately errors.

10.4.3 Integer Math

If you are targeting a platform that can utilize integer operations, e.g. integer division, it should be wrapped in Std.int() for improved performance. The Haxe Compiler can then optimize for integer operations. An example:

```
1      var intDivision = Std.int(6.2/4.7);
```

I think C++ can use integer operatins, but I don't know about any other targets. Only saw this mentioned in an old discussion thread, still true?

10.4.4 Extensions

It is common to see Static Extension (Section 6.3) used with the math library. This code shows a simple example:

```
class MathStaticExtension {
  /* Converts an angle in radians to degrees */
  inline public static function toDegrees(radians:Float):Float {
    return radians * 180 / Math.PI;
  }
}
```

```
using MathStaticExtension;

class TestMath{
  public static function main(){
    var ang = 1/2*Math.PI;
    trace(ang.toDegrees()); //90
  }
}
```

10.5 Lambda

> Definition: 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:

```
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:

```
using Lambda;
class Main {
    static function main() {
        var words = ['car', 'boat', 'cat', 'frog'];

        var isThreeLetters = function(word) return word.length == 3;
        var capitalize = function(word) return word.toUpperCase();

        // Three letter words and capitalized.
        trace(words.filter(isThreeLetters).map(capitalize)); // [CAR,
            CAT]
    }
}
```

This example demonstrates the Lambda count, has, foreach and fold function on a set of ints.

```
using Lambda;
class Main {
    static function main() {
        var numbers = [1, 3, 5, 6, 7, 8];

        trace(numbers.count()); // 6
        trace(numbers.has(4)); // false

        // test if all numbers are greater/smaller than 20
        trace(numbers.foreach(function(v) return v < 20)); // true
        trace(numbers.foreach(function(v) return v > 20)); // false

        // sum all the numbers
        var sum = function(num, total) return total += num;
```

```
15          trace(numbers.fold(sum, 0)); // 30
16      }
17  }
```

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::`.

```
<table>
    <tr>
        <th>Name</th>
        <th>Age</th>
    </tr>
    ::foreach users::
        <tr>
            <td>::name::</td>
            <td>::age::</td>
        </tr>
    ::end::
</table>
```

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

```
var users = [{name:"Mark", age:30}, {name:"John", age:45}];

var userTemplate = new haxe.Template("::foreach users:: ::name::(::age
    ::) ::end::");
var userOutput = userTemplate.execute({users: users});

var template = new haxe.Template("The users are ::users::");
var output = template.execute({users: userOutput});
trace(output);
```

The console will trace `The users are Mark(30) John(45)`.

Template macros To call custom functions while parts of the template are being rendered, provide a `macros` object to the argument of `Template.execute`. The key will act as the template variable name, the value refers to a callback function that should return a `String`. The first argument of this macro function is always a `resolve()` method, followed by the given arguments. The resolve function can be called to retrieve values from the template context. If `macros` has no such field, the result is unspecified.

The following example passes itself as macro function context and executes `display` from the template.

```
class Main {
  static function main() {
    new Main();
  }

  public function new() {
    var user = {name:"Mark", distance:3500};
    var sample = "The results: $$display(::user::,::time::)";
    var template = new haxe.Template(sample);
    var output = template.execute({user:user, time: 15}, this);
    trace(output);
  }

```

```
14    function display(resolve:String->Dynamic, user:User, time:Int) {
15      return user.name + " ran " + (user.distance/1000) + " kilometers
            in " + time + " minutes";
16    }
17  }
18  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.

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
```

```
  }
}
```

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

```
class Main {
  static function main() {
    // null
    trace(Type.resolveClass("haxe.Template"));
  }
}
```

The problem here is that the compiler never actually "sees" the type `haxe.Template`, so it does not compile it into the output. Furthermore, even if it were to see the type there could be issues arising from dead code elimitation (8.2) eliminating types or fields which are only used via reflection.

Another set of problems comes from the fact that, by design, several reflection functions expect arguments of type Dynamic (2.7), meaning the compiler cannot check if the passed in arguments are correct. The following example demonstrates a common mistake when working with `callMethod`:

```
class Main {
  static function main() {
    // wrong
    //Reflect.callMethod(Main, "f", []);
    // right
    Reflect.callMethod(Main,
      Reflect.field(Main, "f"), []);
  }

  static function f() {
    trace('Called');
  }
}
```

The commented out call would be accepted by the compiler because it assigns the string `"f"` to the function argument `func` which is specified to be `Dynamic`.

A good advice when working with reflection is to wrap it in a few functions within an application or API which are called by otherwise type-safe code. An example could look like this:

```
typedef MyStructure = {
  name: String,
  score: Int
}

class Main {
  static function main() {
    var data = reflective();
    // At this point data is nicely typed as MyStructure
  }

  static function reflective():MyStructure {
    // 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 interally work with reflection (and `Dynamic` for that matter) a lot, its return value is a typed structure which the callers can use in a type-safe manner.

10.8 Serialization

Many runtime values can be serialized and deserialized using the `haxe.Serializer` and `haxe.Unserializer` classes. Both support two usages:

1. Create an instance and continuously call the `serialize`/`unserialize` method to handle multiple values.
2. Call their static `run` method to serialize/deserialize a single value.

The following example demonstrates the first usage:

```
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 (Section 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:

```
import haxe.Serializer;
import haxe.Unserializer;

class Main {

    var x:Int;
    var y:Int;

    static function main() {
      var s = Serializer.run(new Main(1, 2));
      var c:Main = Unserializer.run(s);
      trace(c.x); // 1
      trace(c.y); // -1
    }

    function new(x, y) {
      this.x = x;
      this.y = y;
    }

    @:keep
    function hxSerialize(s:Serializer) {
        s.serialize(x);
    }

    @:keep
    function hxUnserialize(u:Unserializer) {
        x = u.unserialize();
        y = -1;
    }
}
```

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.

10.8.1 Serialization format

Each supported value is translated to a distinct prefix character, followed by the necessary data.

`null`: `n`

`Int`: `z` for zero, or `i` followed by the integer display (e.g. `i456`)

`Float`:

- `NaN`: `k`
- negative infinity: `m`
- positive infinity: `p`
- finite floats: `d` followed by the float display (e.g. `d1.45e-8`)

Bool: t for true, f for false

String: y followed by the url encoded string length, then : and the url encoded string (e.g. y10:hi%20there for "hi there".

name-value pairs: a serialized string representing the name followed by the serialized value

structure: o followed by the list of name-value pairs and terminated by g (e.g. oy1:xi2y1:kng for {x:2, k:null})

List: l followed by the list of serialized items, followed by h (e.g. lnnh for a list of two null values)

Array: a followed by the list of serialized items, followed by h. For multiple consecutive null values, u followed by the number of null values is used (e.g. ai1i2u4i7ni9h for [1,2,null,null,null,null,7,null,9])

Date: v followed by the date itself (e.g. v2010-01-01 12:45:10)

haxe.ds.StringMap: b followed by the name-value pairs, followed by h (e.g. by1:xi2y1:knh for {"x" => 2, "k" => null})

haxe.ds.IntMap: q followed by the key-value pairs, followed by h. Each key is represented as :<int> (e.g. q:4n:5i45:6i7h for {4 => null, 5 => 45, 6 => 7})

haxe.ds.ObjectMap: M followed by serialized value pairs representing the key and value, followed by h

haxe.io.Bytes: s followed by the length of the base64 encoded bytes, then : and the byte representation using the codes A-Za-z0-9% (e.g. s3:AAA for 2 bytes equal to 0, and s10:SGVsbG8gIQ for haxe.io.Bytes.ofString("Hello !"))

exception: x followed by the exception value

class instance: c followed by the serialized class name, followed by the name-value pairs of the fields, followed by g (e.g. cy5:Pointy1:xzy1:yzg for new Point(0, 0) (having two integer fields x and y)

enum instance (by name): w followed by the serialized enum name, followed by the serialized constructor name, followed by :, followed by the number of arguments, followed by the argument values (e.g. wy3:Fooy1:A: 0 for Foo.A (with no arguments), wy3:Fooy1:B:2i4n for Foo.B(4,null))

enum instance (by index): j followed by the serialized enum name, followed by :, followed by the constructor index (starting from 0), followed by :, followed by the number of arguments, followed by the argument values (e.g. wy3:Foo: 0:0 for Foo.A (with no arguments), wy3:Foo:1:2i4n for Foo.B(4,null))

cache references:

String: R followed by the corresponding index in the string cache (e.g. R456)

class, enum or structure r followed by the corresponding index in the object cache (e.g. r42)

custom: C followed by the class name, followed by the custom serialized data, followed by g

Cached elements and enum constructors are indexed from zero.

10.9 Json

Haxe provides built-in support for (de-)serializing JSON[2] data via the `haxe.Json` class.

10.9.1 Parsing JSON

Use the `haxe.Json.parse` static method to parse JSON data and obtain a Haxe value from it:

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

Note that the type of the object returned by `haxe.Json.parse` is `Dynamic`, so if the structure of our data is well-known, we may want to specify a type using anonymous structures (2.5). This way we provide compile-time checks for accessing our data and most likely more optimal code generation, because compiler knows about types in a structure:

```
typedef MyData = {
  var name:String;
  var tags:Array<String>;
}

class Main {
  static function main() {
    var s = '{
      "name": "Haxe",
      "tags": ["awesome"]
    }';
    var o:MyData = haxe.Json.parse(s);
    trace(o.name); // Haxe (a string)
    // awesome (a string in an array)
    trace(o.tags[0]);
  }
}
```

10.9.2 Encoding JSON

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

```
class Main {
    static function main() {
    var o = {rating: 5};
    var s = haxe.Json.stringify(o);
    trace(s); // {"rating":5}
    }
}
```

[2] http://en.wikipedia.org/wiki/JSON

10.9.3 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.

10.10 Xml

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.

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://
           localhost/");
4      cnx.setErrorHandler( function(err) trace('Error: $err'); } );
5      cnx.Server.foo.call([1,2], function(data) trace('Result: $data');)
           ;
6    }
7  }
```

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).

10.13.2 Implementation details

Javascript security specifics The html-page wrapping the js client must be served from the same domain as the one where the server is running. The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. The same-origin policy is used as a means to prevent some of the cross-site request forgery attacks.

To use the remoting across domain boundaries, CORS (cross-origin resource sharing) needs to be enabled by defining the header X-Haxe-Remoting in the .htaccess:

```
# Enable CORS
Header set Access-Control-Allow-Origin "*"
Header set Access-Control-Allow-Methods: "GET,POST,OPTIONS,DELETE,PUT"
Header set Access-Control-Allow-Headers: X-Haxe-Remoting
```

See same-origin policy for more information on this topic.

Also note that this means that the page can't be served directly from the file system "file:///C:/example/path/index.html".

Flash security specifics When Flash accesses a server from a different domain, set up a crossdomain.xml file on the server, enabling the X-Haxe headers.

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

Arguments types are not ensured There is no guarantee of any kind that the arguments types will be respected when a method is called using remoting. That means even if the arguments of function foo are typed to Int, the client will still be able to use strings while calling the method. This can lead to security issues in some cases. When in doubt, check the argument type when the function is called by using the Std.is method.

10.14 Unit testing

The Haxe Standard Library provides basic unit testing classes from the haxe.unit package.

Creating new test cases First, create a new class extending `haxe.unit.TestCase` and add own test methods. Every test method name must start with "`test`".

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

Running unit tests To run the test, an instance of `haxe.unit.TestRunner` has to be created. Add the `TestCase` using the `add` method and call `run` to start the test.

```
class Main {
  static function main() {
    var r = new haxe.unit.TestRunner();
    r.add(new MyTestCase());
    // add other TestCases here

    // finally, run the tests
    r.run();
  }
}
```

The result of the test looks like this:

```
Class: MyTestCase
.
OK 1 tests, 0 failed, 1 success
```

Test functions The `haxe.unit.TestCase` class comes with three test functions.

`assertEquals(a, b)` Succeeds if `a` and `b` are equal, where `a` is value tested and `b` is the expected value.

`assertTrue(a)` Succeeds if `a` is `true`

`assertFalse(a)` Succeeds if `a` is `false`

Setup and tear down To run code before or after the test, override the functions `setup` and `tearDown` in the `TestCase`.

`setup` is called before each test runs.

`tearDown` is called once after all tests are run.

```
class MyTestCase extends haxe.unit.TestCase {
  var value:String;

  override public function setup() {
    value = "foo";
  }

  public function testSetup() {
```

```
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  }
```

Part IV

Miscellaneous

Chapter 11

Haxelib

Haxelib is the library manager that comes with any Haxe distribution. Connected to a central repository, it allows submitting and retrieving libraries and has multiple features beyond that. Available libraries can be found at http://lib.haxe.org.

A basic Haxe library is a collection of `.hx` files. That is, libraries are distributed by source code by default, making it easy to inspect and modify their behavior. Each library is identified by a unique name, which is utilized when telling the Haxe Compiler which libraries to use for a given compilation.

11.1 Using a Haxe library with the Haxe Compiler

Any installed Haxe library can be made available to the compiler through the `-lib <library-name>` argument. This is very similiar to the `-cp <path>` argument, but expects a library name instead of a directory path. These commands are explained thoroughly in Compiler Usage (Chapter 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 (Section 11.4).

11.2 haxelib.json

Each Haxe library requires a `haxelib.json` file in which the following attributes are defined:

name: The name of the library. It must contain at least 3 characters among the following:

$$A - Za - z0 - 9\_.$$

. In particular, no spaces are allowed.

url: The URL of the library, i.e. where more information can be found.

license: The license under which the library is released. Can be `GPL`, `LGPL`, `BSD`, `Public` (for Public Domain) or `MIT`.

tags: An array of tag-strings which are used on the repository website to sort libraries.

description: The description of what the library is doing.

version: The version string of the library. This is detailed in Versioning (Section 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 (Section 11.2.2).

The following JSON is a simple example of a haxelib.json:

```
{
  "name": "useless_lib",
  "url" : "https://github.com/jasononeil/useless/",
  "license": "MIT",
  "tags": ["cross", "useless"],
  "description": "This library is useless in the same way on every
     platform.",
  "version": "1.0.0",
  "releasenote": "Initial release, everything is working correctly.",
  "contributors": ["Juraj","Jason","Nicolas"],
  "dependencies": {
    "tink_macro": "",
    "nme": "3.5.5"
  }
}
```

11.2.1 Versioning

Haxelib uses a simplified version of SemVer. The basic format is this:

```
major.minor.patch
```

These are the basic rules:

- Major versions are incremented when you break backwards compatibility - so old code will not work with the new version of the library.
- Minor versions are incremented when new features are added.
- Patch versions are for small fixes that do not change the public API, so no existing code should break.
- When a minor version increments, the patch number is reset to 0. When a major version increments, both the minor and patch are reset to 0.

Examples:

"0.0.1": A first release. Anything with a "0" for the major version is subject to change in the next release - no promises about API stability!

"0.1.0": Added a new feature! Increment the minor version, reset the patch version

"0.1.1": Realised the new feature was broken. Fixed it now, so increment the patch version

"1.0.0": New major version, so increment the major version, reset the minor and patch versions. You promise your users not to break this API until you bump to 2.0.0

"1.0.1": A minor fix

"1.1.0": A new feature

"1.2.0": Another new feature

"2.0.0": A new version, which might break compatibility with 1.0. Users are to upgrade cautiously.

If this release is a preview (Alpha, Beta or Release Candidate), you can also include that, with an optional release number:

```
1 major.minor.patch-(alpha/beta/rc).release
```

Examples:

"1.0.0-alpha": The alpha of 1.0.0 - use with care, things are changing!

"1.0.0-alpha.2": The 2nd alpha

"1.0.0-beta": Beta - things are settling down, but still subject to change.

"1.0.0-rc.1": The 1st release candidate for 1.0.0 - you shouldn't be adding any more features now

"1.0.0-rc.2": The 2nd release candidate for 1.0.0

"1.0.0": The final release!

11.2.2 Dependencies

As of Haxe 3.1.0, haxelib supports only exact version matching for dependencies. Dependencies are defined as part of the haxelib.json (11.2), with the library name serving as key and the expected version (if required) as value in the format described in Versioning (Section 11.2.1).

We have seen an example of this when introducing haxelib.json:

```
"dependencies": {
  "tink_macros": "",
  "nme": "3.5.5"
}
```

This adds two dependencies to the given Haxe library:

1. The library "tink_macros" can be used in any version. Haxelib will then always try to use the latest version.
2. The library "nme" is required in version "3.5.5". Haxelib will make sure that this exact version is used, avoiding potential breaking changes with future versions.

11.3 extraParams.hxml

If you add a file named `extraParams.hxml` to your library root (at the same level as `haxelib.json`), these parameters will be automatically added to the compilation parameters when someone use your library with `-lib`.

11.4 Using Haxelib

If the `haxelib` command is executed without any arguments, it prints an exhaustive list of all available arguments. Access the `http://lib.haxe.org` website to view all the libraries available.

The following commands are available:

Basic `haxelib install [project-name] [version]` installs the given project. You can optionally specify a specific version to be installed. By default, latest released version will be installed.

`haxelib update [project-name]` updates a single library to their latest version.

`haxelib upgrade` upgrades all the installed projects to their latest version. This command prompts a confirmation for each upgradeable project.

`haxelib remove project-name [version]` removes complete project or only a specified version if specified.

`haxelib list` lists all the installed projects and their versions. For each project, the version surrounded by brackets is the current one.

`haxelib set [project-name] [version]` changes the current version for a given project. The version must be already installed.

Information `haxelib search [word]` lists the projects which have either a name or description matching specified word.

`haxelib info [project-name]` gives you information about a given project.

`haxelib user [user-name]` lists information on a given Haxelib user.

`haxelib config` prints the Haxelib repository path. This is where Haxelib get installed by default.

`haxelib path [project-name]` prints paths to libraries and its dependencies (defined in `haxelib.xml`).

Development `haxelib submit [project.zip]` submits a package to Haxelib. If the user name is unknown, you'll be first asked to register an account. If the user already exists, you will be prompted for your password. If the project does not exist yet, it will be created, but no version will be added. You will have to submit it a second time to add the first released version. If you want to modify the project url or description, simply modify your `haxelib.xml` (keeping version information unchanged) and submit it again.

`haxelib register [project-name]` submits or update a library package.

`haxelib local [project-name]` tests the library package. Make sure everything (both installation and usage) is working correctly before submitting, since once submitted, a given version cannot be updated.

`haxelib dev [project-name] [directory]` sets a development directory for the given project. To set project directory back to global location, run command and omit directory.

`haxelib git [project-name] [git-clone-path] [branch] [subdirectory]` uses git repository as library. This is useful for using a more up-to-date development version, a fork of the original project, or for having a private library that you do not wish to post to Haxelib. When you use `haxelib upgrade` any libraries that are installed using GIT will automatically pull the latest version.

Miscellaneous `haxelib setup` sets the Haxelib repository path. To print current path use `haxelib config`.

`haxelib selfupdate` updates Haxelib itself. It will ask to run `haxe update.hxml` after this update.

`haxelib convertxml` converts `haxelib.xml` file to `haxelib.json`.

`haxelib run [project-name] [parameters]` runs the specified library with parameters. Requires a precompiled Haxe/ Neko `run.n` file in the library package. This is useful if you want users to be able to do some post-install script that will configure some additional things on the system. Be careful to trust the project you are running since the script can damage your system.

`haxelib proxy` setup the Http proxy.

Chapter 12

Target Details

12.1 Javascript

12.1.1 Getting started with Haxe/Javascript

Haxe can be a powerful tool for developing Javascript applications. Let's look at our first sample. This is a very simple example showing the toolchain.

Create a new folder and save this class as `Main.hx`.

```
import js.Lib;
import js.Browser;
class Main {
    static function main() {
        var button = Browser.document.createButtonElement();
        button.textContent = "Click me!";
        button.onclick = function(event) {
            Lib.alert("Haxe is great");
        }
        Browser.document.body.appendChild(button);
    }
}
```

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

```
haxe -js main-javascript.js -main Main -D js-flatten -dce full
```

Another possibility is to create and run (double-click) a file called `compile.hxml`. In this example the hxml-file should be in the same directory as the example class.

```
-js main-javascript.js
-main Main
-D js-flatten
-dce full
```

The output will be a main-javascript.js, which creates and adds a clickable button to the document body.

Run the Javascript To display the output in a browser, create an HTML-document called `index.html` and open it.

```
<!DOCTYPE html>
<html>
    <body>
        <script src="main-javascript.js">
    </body>
</html>
```

More information

- Haxe Javascript API docs
- MDN JavaScript Reference

12.1.2 Using external Javascript libraries

The Haxe Standard Library comes with many externs for the Javascript target. They provide access to the native APIs in a type-safe manner. The externs mechanism (6.2) assumes that the defined types exist at run-time but assumes nothing about how and where those types are defined. To clarify, in most cases we have to add a `<script>`-tag that links the external library manually to the HTML-document.

An example of an extern class is the jQuery class of the Haxe Standard Library. To illustrate, here is a part of this extern class:

```
@:initPackage
extern class JQuery implements ArrayAccess<Element> {
    function addClass( className : String ) : JQuery;
    function removeClass( ?className : String ) : JQuery;
    function hasClass( className : String ) : Bool;

    @:overload(function(html:String):JQuery{})
    @:overload(function(html:JQuery):JQuery{})
    function html() : String;
..
```

Note that functions can be overloaded to accept different types of arguments and return values, using the `@:overload` metadata. Function overloading works only in extern classes. Using this extern, we can use jQuery like this:

```
new JQuery("#my-div").addClass("brand-success").html("haxe is great!")
    ;
```

Beside externs, Typedefs (3.1) can be another great way to name (or alias) complex Javascript or JSON structures.

The package and class name of the extern class should be the same as defined in the external library. If that is not the case, rewrite the path of a class using `@:native`.

```
package my.application.media;

@:native('external.library.media.video')
extern class Video {
..
```

In Haxe, it is possible to call an exposed function thanks to the `untyped` keyword. This can be useful in some cases if we don't want to write externs. Anything untyped that is valid syntax will be generated as it is.

```
untyped window.trackEvent("page1");
```

Using the magic __js__ function we can inject pure Javascript code fragments into the output. This code can not be validated, so it accepts invalid code in the output, which is error-prone. This could, for example, write a Javascript comment in the output.

```
untyped __js__('// haxe is great!');
```

The standard compilation options also provide more Haxe sources to be added to the project:

- To add a Haxelib library (11), use `-lib <library-name>`. There are many externs for popular native libraries.
- To add another class path, use `-cp <directory>`.
- 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.

12.1.3 Javascript target Metadata

This is the list of Javascript specific metadata. For more information, see also the complete list of all Haxe built-in metadata (8.1).

Javascript metadata	
Metadata	Description
@:expose _(?Name=Class path)_	Makes the class available on the `window` object or `exports` for node

12.1.4 Exposing Haxe classes for Javascript

It is possible to make Haxe classes or static fields available for usage in plain Javascript. To expose, add the `@:expose` metadata to the desired class or static fields.

This example exposes the Haxe class `MyClass`.

```
@:expose
class MyClass {
  var name:String;
  function new(name:String) {
    this.name = name;
  }
  public function foo() {
    return 'Greetings from $name!';
  }
}
```

It generates the following Javascript output:

```
(function ($hx_exports) { "use strict";
var MyClass = $hx_exports.MyClass = function(name) {
    this.name = name;
};
MyClass.prototype = {
    foo: function() {
        return "Greetings from " + this.name + "!";
    }
```

```
};
})(typeof window != "undefined" ? window : exports);
```

By passing globals (like `window` or `exports`) as parameters to our anonymous function in the Javascript module, it becomes available which allows to expose the Haxe generated module.

In plain Javascript it is now possible to create an instance of the class and call its public functions.

```
// Javascript code
var instance = new MyClass('Mark');
console.log(instance.foo()); // logs a message in the console
```

The package path of the Haxe class will be completely exposed. To rename the class or define a different package for the exposed class, use `@:expose("my.package.MyExternalClass")`

Shallow expose When the code generated by Haxe is part of a larger Javascript project and wrapped in a large closure it is not always necessary to expose the Haxe types to global variables. Compiling the project using `-D shallow-expose` allows the types or static fields to be available for the surrounding scope of the generated closure only.

When the code is compiled using `-D shallow-expose`, the generated output will look like this:

```
var $hx_exports = $hx_exports || {};
(function () { "use strict";
var MyClass = $hx_exports.MyClass = function(name) {
    this.name = name;
};
MyClass.prototype = {
    foo: function() {
        return "Greetings from " + this.name + "!";
    }
};
})();
var MyClass = $hx_exports.MyClass;
```

In this pattern, a var statement is used to expose the module; it doesn't write to the `window` or `exports` object.

12.1.5 Loading extern classes using "require" function

Since Haxe 3.2.0

Modern JavaScript platforms, such as Node.js provide a way of loading objects from external modules using the "require" function. Haxe supports automatic generation of "require" statements for `extern` classes.

This feature can be enabled by specifying `@:jsRequire` metadata for the extern class. If our `extern` class represents a whole module, we pass a single argument to the `@:jsRequire` metadata specifying the name of the module to load:

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

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

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

The second argument is a dotted-path, so we can load sub-objects in any hierarchy.

If we need to load custom JavaScript objects in runtime, a `js.Lib.require` function can be used. It takes `String` as its only argument and returns `Dynamic`, so it is advised to be assigned to a strictly typed variable.

12.2 Flash

12.2.1 Getting started with Haxe/Flash

Developing Flash applications is really easy with Haxe. Let's look at our first code sample. This is a basic example showing most of the toolchain.

Create a new folder and save this class as `Main.hx`.

```
import flash.Lib;
import flash.display.Shape;
class Main {
    static function main() {
        var stage = Lib.current.stage;

        // create a center aligned rounded gray square
        var shape = new Shape();
        shape.graphics.beginFill(0x333333);
        shape.graphics.drawRoundRect(0, 0, 100, 100, 10);
        shape.x = (stage.stageWidth - 100) / 2;
        shape.y = (stage.stageHeight - 100) / 2;

        stage.addChild(shape);
    }
}
```

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

```
haxe -swf main-flash.swf -main Main -swf-version 15 -swf-header
   960:640:60:f68712
```

Another possibility is to create and run (double-click) a file called `compile.hxml`. In this example the hxml-file should be in the same directory as the example class.

```
-swf main-flash.swf
-main Main
-swf-version 15
-swf-header 960:640:60:f68712
```

The output will be a main-flash.swf with size 960x640 pixels at 60 FPS with an orange background color and a gray square in the center.

Display the Flash Run the SWF standalone using the Standalone Debugger FlashPlayer.

To display the output in a browser using the Flash-plugin, create an HTML-document called `index.html` and open it.

```
<!DOCTYPE html>
<html>
    <body>
        <embed src="main-flash.swf" width="960" height="640">
    </body>
</html>
```

More information

- Haxe Flash API docs
- Adobe Livedocs

12.2.2 Embedding resources

Embedding resources is different in Haxe compared to Actionscript 3. Instead of using

embed

(AS3-metadata) use Flash specific compiler metadata (12.2.4) like `@:bitmap`, `@:font`, `@:sound` or `@:file`.

```
import flash.Lib;
import flash.display.BitmapData;
import flash.display.Bitmap;

class Main {
  public static function main() {
    var img = new Bitmap( new MyBitmapData(0, 0) );
    Lib.current.addChild(img);
  }
}

@:bitmap("relative/path/to/myfile.png")
class MyBitmapData extends BitmapData { }
```

12.2.3 Using external Flash libraries

To embed external `.swf` or `.swc` libraries, use the following compilation options:

`-swf-lib <file>` Embeds the SWF library in the compiled SWF.

`-swf-lib-extern <file>` Adds the SWF library for type checking but doesn't include it in the compiled SWF.

The standard compilation options also provide more Haxe sources to be added to the project:

- To add another class path use `-cp <directory>`.

- To add a Haxelib library (11) use `-lib <library-name>`.
- To force a whole package to be included in the project, use `--macro include('mypackage')` which will include all the classes declared in the given package and subpackages.

12.2.4 Flash target Metadata

This is the list of Flash specific metadata. For a complete list see Haxe built-in metadata (8.1).

Flash metadata	
Metadata	Description
@:bind	Override Swf class declaration
@:bitmap _(Bitmap file path)_	_Embeds given bitmap data into the class (must extend `flash.di`
@:debug	Forces debug information to be generated into the Swf even with
@:file(File path)	Includes a given binary file into the target Swf and associates it v
@:font _(TTF path Range String)_	Embeds the given TrueType font into the class (must extend `flas`
@:getter _(Class field name)_	Generates a native getter function on the given field
@:noDebug	Does not generate debug information into the Swf even if `-debug`
@:ns	Internally used by the Swf generator to handle namespaces
@:setter _(Class field name)_	Generates a native setter function on the given field
@:sound _(File path)_	Includes a given _.wav_ or _.mp3_ file into the target Swf and ass

12.3 Neko

12.4 PHP

12.4.1 Getting started with Haxe/PHP

To get started with Haxe/PHP, create a new folder and save this class as `Main.hx`.

```
1 import php.Lib;
2
3 class Main {
4     static function main() {
5         Lib.println('Haxe is great!');
6     }
7 }
```

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

```
1 haxe -php bin -main Main
```

Another possibility is to create and run (double-click) a file called `compile.hxml`. In this example the hxml-file should be in the same directory as the example class.

```
1 -php bin
2 -main Main
```

The compiler outputs in the given bin-folder, which contains the generated PHP classes that prints the traced message when you run it. The generated PHP-code runs for version 5.1.0 and later.

More information

- Haxe PHP API docs
- PHP.net Documentation
- PHP to Haxe tool

12.5 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