

Haxe 3 Manual

Haxe Foundation November 20, 2013

Todo list

Check wording here (we don't want to address "you")	12
Took the ideas from old manual, but moved Neko in with C++ and co. Still Haxe 3 relevant?	12
make sure the types are right for inc, dec, negate, and bitwise negate	12
I don't really know how these work yet	21
resolve()	24
@:enum and @:expose	24
Mention toString()/String conversion somewhere in this chapter	40
Describe import a.*	46
Check "you" wording	68
Check if we can build GADTs this way	90
make sure this is true about static targets	97
Maybe we should introduce \api{type name}	97
I hope none of these are lies -C	97
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?	103

Contents

To	Todo list									
1	1.1 1.2 1.3 1.4	oduction What is Haxe?	7 7 8 8 9							
I	Lar	nguage Reference	10							
2	2.2 2.3 2.4 2.5	Basic Types 2.1.1 Numeric types 2.1.2 Overflow 2.1.3 Numeric Operators 2.1.4 Bool 2.1.5 Void Nullability Class Instance 2.3.1 Class constructor 2.3.2 Inheritance 2.3.3 Interfaces Enum Instance 2.4.1 Enum Constructor Anonymous Structure	11 12 12 12 13 14 14 16 17 17 18 19							
	2.62.72.8	2.5.1 JSON-Notation for Structure Values 2.5.2 Class Notation for Structure Types 2.5.3 Optional Fields 2.5.4 Impact on Performance Function 2.6.1 Optional Arguments 2.6.2 Default values Dynamic 2.7.1 Dynamic with Type Parameter 2.7.2 Implementing Dynamic	20 21 21 21 21 22 23 24 24 24							
	2.0	2.8.1Implicit Casts2.8.2Operator Overloading	24 26 28 29							

		2.8.4 Selective Functions	30
		2.8.5 Enum abstracts	30
		2.8.6 Exposing abstracts	31
			32
	2.9		32
3	Typ		33
	3.1		33
			34
	3.2		34
		3.2.1 Constraints	35
	3.3		36
		$O = I \cap I$	37
	3.4	Variance	38
	3.5	Unification	40
		3.5.1 Between Class/Interface	40
		3.5.2 Structural Subtyping	41
			41
		<u>.</u>	41
			41
	3.6		42
			43
		1	44
	3.7		44
			45
		71	45
		1	46
4	Clas		48
	4.1		48
	4.2		49
			51
			52
			53
	4.3	Method	54
			55
		4.3.2 Effects of variance and access modifiers	56
	4.4	Access Modifier	56
		4.4.1 Visibility	56
		4.4.2 Inline	58
		4.4.3 Dynamic	58
		4.4.4 Override	59
5			60
	5.1		60
	5.2		61
	5.3		61
	5.4		61
	5.5		61
	5.6	,	62
	5.7		62
	5.8	Function Call	62

	5.9	Closure	63
	5.10	var	63
	5.11	new	63
	5.12	for	63
	5.13	while	63
	5.14	if	63
	5.15	switch	63
	5.16	try/catch	63
	5.17	return	63
	5.18	break	64
	5.19	continue	64
	5.20	throw	65
	5.21	cast	65
6		guage Features	66
	6.1	Conditional Compilation	66
	6.2	Static Extension	67
		6.2.1 In the Haxe Standard Library	68
	6.3	Pattern Matching	68
		6.3.1 Introduction	68
		6.3.2 Enum matching	69
		6.3.3 Variable capture	69
		6.3.4 Structure matching	70
		6.3.5 Array matching	70
		6.3.6 Or patterns	70
		6.3.7 Guards	71
		6.3.8 Match on multiple values	71
		6.3.9 Extractors	71
		6.3.10 Exhaustiveness checks	73
		6.3.11 Useless pattern checks	73
	6.4	String Interpolation	73
	6.5	Array Comprehension	74
	6.6	Function Bindings	75
	6.7	Metadata	75
	6.8	Access Control	77
	6.9	Remoting	79
	6.10	Runtime Type Information	79
II	Co	ompiler Reference	80
7	Com	npiler Reference	81
	7.1	Compiler Metadata	81
8	Com	npiler Features	82
0	8.1	Dead Code Elimination	82
	8.2	Completion	83
	0.2	8.2.1 Field Access	83
		8.2.2 Call Arguments	83
		8.2.3 Usage	83
		8.2.4 Position	83
		8.2.5 Metadata	83
		U.Z.O ITICMOMM	

9	Mac		84
	9.1		84
	9.2	Arguments	85
		9.2.1 ExprOf	85
		9.2.2 Constant Expressions	86
			86
	9.3		87
			87
		1	88
		J 1	88
	9.4		88
	9.5		89
	9.5		90
		0	
	0.6		92
	9.6		92
			92
			92
			93
		9.6.4 Type Parameters	94
	9.7	Compiler Configuration	94
II	S	tandard Library	95
10		······································	96
			96
	10.2		96
			96
		10.2.2 List	97
		10.2.3 GenericStack	97
		10.2.4 Map	98
		*	99
	10.3		99
	10.0		.00
			.00
		10.3.3 Replace	
		10.3.4 Split	
		• • • • • • • • • • • • • • • • • • •	
		10.3.5 Map	
	40.4	10.3.6 Implementation Details	
	10.4	Math	
			.03
			03
			03
		10.4.4 Extensions	03
	10.5	Lambda	04
	10.6	Reflection	04
			04
			04
			04
		O -	04
			.05
	10.0	Xml	
	10.7	лии	UJ

10.10Input/Output 10.11Sys/sys															
IV Tools														10	6
10.12Haxelib	 	 		 										. 10)7
10.13Other Tools.	 	 		 										. 10)7

Chapter 1

Introduction

1.1 What is Haxe?

Haxe is a high-level, open source programming language and compiler. It allows compilation of programs written using an ECMAScript¹-oriented syntax to multiple target languages. Employing proper abstraction, it is possible to maintain a single code-base which compiles to multiple targets.

Haxe is strongly typed, but the typing system can be subverted where required. Utilizing type information, the Haxe type system can detect errors at compile-time which would only be noticeable at runtime in the target language. Furthermore, type information can be used by the target generators to generate optimized and robust code.

Currently, there are eight supported target languages which allow different use-cases:

Name	Output type	Main usages							
Javascript	Sourcecode	Desktop, Mobile, Server							
Actionscript 3	Sourcecode	Browser, Desktop, Mobile							
Flash	Bytecode	Browser, Desktop, Mobile							
Neko	Bytecode	Desktop, Server							
PHP	Sourcecode	Server							
C++	Sourcecode	Desktop, Mobile, Server							
Java	Sourcecode	Desktop, Server							
C#	Sourcecode	Desktop, Mobile, Server							

The remainder of section 1 gives a brief overview of what a Haxe program looks like, and how haxe evolved since its inception in 2005.

Types (Chapter 2) introduces the seven different kinds of types in Haxe and how they interact with each other. The discussion of types is continued in Type System (Chapter 3), where features like *unification*, type parameters and type inference are explained.

Class Fields (Chapter 4) is all about the structure of Haxe classes and, among other topics, deals with *properties*, *inline fields* and *generic functions*.

In Expressions (Chapter 5) we see how to actually get programs to do something by using *expressions*, plenty of which are used in the Haxe Standard Library described in Standard Library (Chapter 10).

Language Features (Chapter 6) describes some of the Haxe features in detail, such as *pattern matching*, *string interpolation* and *dead code elimination*.

Finally, we will venture to the exciting land of *haxe macros* in Macros (Chapter 9) to see how some common tasks can be greatly simplified.

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

1.2 About this Document

This document is the official manual for Haxe 3. As such, it is not a beginner's tutorial and does not teach programming. However, the topics are roughly designed to be read in order and there are references to topics "previously seen" and topics "yet to come". In some cases, an earlier section makes use of the information of a later section if it simplifies the explanation. These references are linked accordingly and it should generally not be a problem to read ahead on other topics.

We use a lot of Haxe source code to keep a practical connection of theoretical matters. These code examples are often complete programs that come with a main function and can be compiled as-is. However, sometimes only the most important parts are shown. Source code looks like this:

```
Haxe code here
```

Occasionally, we demonstrate how Haxe code is generated, for which we usually show the Javascript target.

Furthermore, we define a set of terms in this document. Most often, this is done when introducing a new type or when a term is specific to Haxe. We not define every new aspect we introduce, e.g. what a class is, in order to not clutter the text. A definition looks like this:

```
Definition: Definition name
Definition description
```

In a few places, this document has *trivia*-boxes. These include off-the-record information such as why certain decisions were made during Haxe's development or how a particular feature has been changed in past haxe versions. This information is generally not important and can be skipped as it is only meant to convey trivia:

```
Trivia: About Trivia
This is trivia.
```

1.3 Hello World

The following program prints "Hello World" after having been compiled and run:

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

This can be tested by saving the above code to a file named <code>HelloWorld.hx</code> and invoking the Haxe compiler like so: <code>haxe -main HelloWorld --interp</code>. This generates the following output: <code>HelloWorld.hx:3: Hello world</code>. There are several things to learn from this:

- Haxe programs are saved in files with an extension of .hx.
- The Haxe compiler is a command-line tool which can be invoked with parameters such as —main HelloWorld and —interp.
- Haxe programs have classes (HelloWorld, upper-case), which have functions (main, lower-case).

1.4 History

The Haxe project started on 22 October 2005 as a successor to the *Motion-Twin Action Script Compiler*, MTASC. French developer *Motion-Twin*² had developed MTASC as a popular ActionScript 2 compiler, and the next logical step for them was to design their very own programming language; thus, giving birth to haxe.

Being spelled *haXe* back then, its beta was released in February 2006 with the first supported targets being AVM³-bytecode and Motion-Twin's own *Neko* virtual machine⁴.

Nicolas Cannasse, who remains leader of the Haxe project to this date, kept on designing Haxe with a clear vision, subsequently leading to the Haxe 1.0 release in May 2006. This first major release came with support for Javascript code generation and already had some of the features that define Haxe today, such as type inference and structural sub-typing.

Haxe 1 saw several minor releases over the course of two years, adding the Flash 9 target along with the *haxelib*-tool in August 2006 and the Actionscript 3 target in March 2007. During these months, there was a strong focus on improving stability, which came in the form of several minor bug-fix releases.

Haxe 2.0 was released in July 2008, coming with the PHP target courtesy of *Franco Ponticelli*. A similar effort by *Hugh Sanderson* lead to the addition of the C++ target in July 2009 with the Haxe 2.04 release.

Just as with Haxe 1, what followed were several months of stability releases. In January 2011, Haxe 2.07 was released with the support of *macros*. Around that time, *Bruno Garcia* joined the team as maintainer of the Javascript target, which saw vast improvements in the subsequent 2.08 and 2.09 releases.

After the release of 2.09, *Simon Krajewski* joined the team and work towards Haxe 3 began. Furthermore, *Cauê Waneck's* Java and C# targets found their way into the Haxe builds. It was then decided to make one final Haxe 2 release, which happened in July 2012 with the release of Haxe 2.10.

In late 2012, the haxe 3 switch was flipped and the Haxe compiler team, now backed by the newly established *Haxe Foundation*⁵, focused on this next major version. Haxe 3 was subsequently released in May 2013.

²http://motion-twin.org

³Adobe Virtual Machine

⁴http://nekovm.org

⁵http://haxe-foundation.org

Part I Language Reference

Chapter 2

Types

The Haxe compiler employs a sophisticated typing system which helps detecting type-related errors in a program at compile-time. A type error is an invalid operation on a given type, such as dividing by a String, trying to access a field of an Integer or calling a function with not enough (or too many) arguments.

In some languages this additional safety comes at a price because programmers are forced to explicitly assign types to syntactic constructs:

```
var myButton:MySpecialButton = new MySpecialButton(); // As3
MySpecialButton* myButton = new MySpecialButton(); // C++
```

The explicit type annotations are not required in Haxe, because the compiler can *infer* the type:

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

We will explore type inference in detail later in Type Inference (Section 3.6). For now, it is sufficient to say that the variable myButton in the above code is known to be an *instance of class* MySpecialButton. A class instance is one of seven type groups:

Class instance: an object of a given class or interface

Enum instance: a value of a Haxe enumeration

Structure: an anonymous structure, i.e. a collection of named fields

Function: a compound type of several arguments and one return

Dynamic: a wildcard type which is compatible to any type

Abstract: an abstract value type

Monomorph: an unknown type, which may later become a different type

Definition: Compound Type

A compound type is a type which has sub-types. This includes any type with type parameters (3.2) and the function (2.6) type.

2.1 Basic Types

Check wording here (we don't want to address "you")

Basic Types include Bool, Float, and Int. The basic types are abstract types tagged with :coreType metadata. This means is they get replaced by platform specific types at compile-time. Don't worry about all of the implmentation details at the moment, but feel free to see Abstract (Section 2.8) and Metadata (Section 6.7) for more details. Just be aware that this abstraction can cause the Overflow (Section 2.1.2) and Nullability (Section 2.2) of Basic Types to behave differently on different target platforms. To reiterate: basic types can behave differently depending on the target platform.

2.1.1 Numeric types

Type: Float

Represents a double-precision IEEE 64bit floating point number.

Type: Int

Represents an integral number.

While every Int can be used where a Float is expected (that is, Int is assignable to or unifies with Float), the reverse is not true: Assigning a Float to an Int might lose precision and is not implicitly allowed.

2.1.2 Overflow

Took the ideas from old manual, but moved Neko in with C++ and co. Still Haxe 3 relevant?

For performance reasons, the Haxe compiler does not enforce any overflow behavior. The burden of checking for overflows falls to the target platform. Here are some platform specific notes on overflow behavior:

C++, Java, C#, Neko: 32-bit integers, with usual overflow practices

Flash, AVM2: 32-bit integers, but higher integers are memory boxed

PHP JS Flash 8: No native *Int* type. Overflows will occur if they reach their float limit (2^{52}) .

Alternatively, the *haxe.Int32* and *haxe.Int64* classes can be used to ensure correct overflow behavior regardless of the platform.

2.1.3 Numeric Operators

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

Arithmetic										
Operator	Operation	Argument 1	Argument 2	Return						
++	increment	Int	N/A	Int						
		Float	N/A	Float						
	decrement	Int	N/A	Int						
		Float	N/A	Float						
+	addition	Float	Float	Float						
		Float	Int							
		Int	Float	Float						
		Int	Int	Int						
_	subtraction	Float	Float	Float						
		Float	Int	Float						
		Int	Float	Float						
		Int	Int	Int						
*	multiplication	Float	Float	Float						
		Float	Int	Float						
		Int	Float	Float						
		Int	Int	Int						
/	division	Float	Float	Float						
		Float	Int	Float						
		Int	Float	Float						
		Int	Int	Float						
용	modulo	Float	Float	Float						
		Float	Int	Float						
		Int	Float	Float						
		Int	Int	Int						
	Cor	nparison								
Operator	Operation	Argument 1	Argument 2	Return						
~	negation	Float/Int	N/A	Bool						
==	equal	Float/Int	Float/Int	Bool						
! =	not equal	Float/Int	Float/Int	Bool						
<	less than	Float/Int	Float/Int	Bool						
<=	less than or equal	Float/Int	Float/Int	Bool						
>	greater than	Float/Int	Float/Int	nt Bool						
>=	great than or equal	Float/Int	Float/Int	Bool						
Bitwise										
Operator	Operation	Argument 1	Argument 2	Return						
!	bitwise negation	Int	N/A	Int						
&	bitwise and	Int	Int	Int						
	bitwise or	Int	Int	Int						
^	bitwise xor	Int	Int	Int						
<<	shift left	Int	Int	Int						
>>	shift right	Int	Int	Int						
>>>	unsigned shift right	Int	Int	Int						

2.1.4 Bool

Type: Bool Represents a value which can be either *true* or *false*.

Values of type Bool appear commonly in *conditions* such as if (section 5.14) and while (section 5.13). The following *operators* accept and return Bool values:

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

Haxe guarantees that compound boolean expressions are evaluated from left to right and only as far as necessary at runtime. For instance, an expression like A && B will evaluate A first and evaluate B only if the evaluation of A yielded true. Likewise, the expressions A $\mid \mid$ B will not evaluate B if the evaluation of A yielded true, because the value of B is irrelevant in that case. This is important in some cases such as this:

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

Accessing object.field if object is null would lead to a runtime error, but the check for object != null guards against it.

2.1.5 Void

Type: Void

Void denote the absence of a type. It is used to express that something (usually a function) has no value.

Void is a special case in the type system because it is not actually a type. It is used to express the absence of a type, which applies mostly to function arguments and return types. We have already "seen" Void in the initial "Hello World" example:

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

The function type will be explored in detail in section Function (Section 2.6), but a quick preview helps here: The type of function main in above example can be considered to be Void->Void, which reads as "it has no arguments and returns nothing". Haxe does not allow fields and variables of type Void, and will complain if an attempt is made at declaring such:

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

2.2 Nullability

Definition: nullable

A type in haxe is considered *nullable* if null is a valid value for it.

It is common for programming languages to have a single, clean definition for nullability. However, Haxe has to find a compromise in this regard. The reason for this is obvious when looking at the nature of Haxe's target languages: While some of them allow and, in fact, default to null

for anything, others do not even allow null for certain types. This necessitates the distinction of two types of target languages:

Definition: Static target

Static targets employ their own type system where null is not a valid value for basic types. This is true for the Flash 9+, C++, Java and C# targets.

Definition: Dynamic target

Dynamic targets are more lenient with their types and allow *null values for basic types*. They consist of JavaScript, PHP, neko and Flash 6-8.

There is nothing to worry about when working with null on dynamic targets, but static ones may require some thought. For starters, basic types are initialized to these values:

Int: 0

Float: NaN on Flash 9, 0.0 on other static targets

Bool: false

As a consequence, the Haxe compiler does not allow the assignment of null to a basic type on static targets. In order to achieve this, the basic type has to be wrapped as Null<T>:

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

Similarly, basic types cannot be compared to null unless wrapped:

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

This restriction extends to all situations where unification (3.5) is performed.

If a null-value is "hidden" in Null<T> or Dynamic and assigned to a basic type, the default value is used:

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

Optional Arguments and Nullability: Optional arguments also have to be accounted when considering nullability. In particular, there must be a distinction between *native* optional arguments which are not nullable and Haxe optional arguments which might be. The distinction is made by using the question-mark optional argument:

Trivia: Argument vs Parameter

In some other programming languages, argument and parameter are used interchangably. In Haxe, argument is used when referring to methods, and parameter refers to Type Parameters (Section 3.2).

```
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) {...}
```

2.3 Class Instance

Similar to many object-oriented languages, classes in Haxe are the primary data structure for the majority of programs. Each Haxe class has an explicit name, an implied path and zero or more class fields. Here we will focus on the general structure of classes and their relations, while leaving the details of class fields for Class Fields (Chapter 4).

The following code example serves as basis for the remainder of this section:

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

Semantically, this class represents a point in discrete 2-dimensional space, but this is not important here. Let us instead describe the structure:

- The keyword class denotes that we are declaring a class.
- Point is the name of the class and could be anything conforming to the rules for type identifiers.
- Enclosed in curly braces {} are the class fields,
- which consist of two variable fields x and y of type Int
- followed by a special function field named new, which is the constructor of the class,
- as well as a normal function toString

There is a special type in Haxe which is compatible with all classes:

Type: Class<T>

This type is compatible with all class types. At compile-time, Class<T> can be considered to be the common base type of all class types. However, this relation is not reflected in generated code.

2.3.1 Class constructor

Instances of classes are created by calling the class constructor, a process commonly referred to as *instantiation*. Another name for class instances is *object*, but we prefer the term class instance to emphasize the analogy between classes/class instances and enums/enum instances (2.4).

```
1 \text{ var p} = \text{new Point}(-1, 65);
```

This will yield an instance of class Point, which is assigned to a variable named p. The constructor of Point receives the two arguments -1 and 65 and assigns them to the instance variables x and y respectively (compare its definition in Class Instance (Section 2.3)). We will revisit the exact meaning of the new expression later in section 5.11. For now, we just consider it as calling the class constructor and returning the appropriate object.

2.3.2 Inheritance

Classes may inherit from other classes, which in haxe is denoted by the extends keyword:

```
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 }
```

This relation is often described as "is-a": Any instance of class Point3 is also an instance of Point. Point is then known as the *parent class* of Point3, which is a *child class* of Point. A class may have many child classes, but only one parent class. The term "a parent class of class X" usually refers to its direct parent class, the parent class of its parent class and so on.

The code above is very similar to the original Point class, with two new constructs being shown:

- extends Point denotes that this class inherits from class Point
- super (x, y) is the call to the constructor of the parent class, in this case Point.new

It is not necessary for child classes to define their own constructors, but if they do a call to super () is mandatory. Unlike some other object-oriented languages, this call can appear anywhere in the constructor code and does not have to be the first expression.

A class may override methods (4.3) of its base class, which requires the explicit override keyword. The effects and restrictions of this are detailed in Overriding Methods (Section 4.3.1).

2.3.3 Interfaces

An interface can be understood as the signature of a class because it describes the public fields of a class. Interfaces do not provide implementations, but rather pure structural information:

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

The syntax is similar to classes, with the following exceptions:

- interface keyword is used instead of class keyword
- functions do not have any expressions

• any field must have an explicit type

Interfaces, unlike structural subtyping, describe a *static relation* between classes. A given class is only considered to be compatible to an interface if it explicitly states so:

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

The implements keyword here denotes that Point has a "is-a" relationship to Printable, i.e. each instance of Point is also an instance of Printable. While a class may only have one parent class, it may implement multiple interfaces through multiple implements keywords.

The compiler checks if the implements assumption holds. That is, it makes sure the class actually does implement all the fields required by the interface. A field is considered implemented if the class or any of its parent classes provide an implementation.

Interface fields are not limited to methods, they can be variables and properties as well.

Trivia: Implements Syntax

Haxe versions prior to 3.0 required multiple implements keywords to be separated by a comma. We decided to adhere to the de-facto standard of Java and got rid of the comma. This was one of the breaking changes between Haxe 2 and 3.

2.4 Enum Instance

Haxe provides powerful enumeration (short: enum) types, which are actually an *algebraic data type* (ADT). While they cannot have any expressions, they are very useful for describing the structure of code logic:

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

Semantically, this enum describes a color which is either red, green, blue or a specified RGB value. The syntactic structure is as follows:

- The keyword enum denotes that we are declaring an enum.
- Color is the name of the enum and could be anything conforming to the rules for type identifiers.
- Enclosed in curly braces {} are the *enum constructors*,
- which are Red, Green and Blue taking no arguments,
- as well as Rgb taking three Int arguments named r, g and b.

The haxe type system provides a type which unifies with all enum types:

Type: Enum

This type is compatible with all enum types. At compile-time, Enum<T> can be considered to be the common base type of all enum types. However, this relation is not reflected in generated code.

2.4.1 Enum Constructor

Similar to classes and their constructors, enums provide a way of instantiating them by using one of their constructors. However, unlike classes, enums provide multiple constructors which can easily be used through their name:

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

In this code, the type of variables a, b and c is instance of enum Color. Variable c is initialized using the Rgb constructor with arguments.

All enum instances can be assigned to a special type named EnumValue.

Type: EnumValue

EnumValue is a special type which unifies with all enum instances. It is used by the standard library to provide some operations for all enum instances, and can be employed in user-code accordingly.

It is important to distinguish enum types and enum constructors, as this example demonstrates:

```
import Color;
1
2
3
  class EnumUnification {
      static public function main() {
4
5
          var ec:EnumValue = Red; // valid
          var en:Enum<Color> = Color; // valid
6
           //var x:Enum<Color> = Red; // Compile Error: Color should be
7
              Enum<Color>
      }
8
```

If line 7 is uncommented, the program does not compile because Red (an enum constructor) cannot be assigned to a variable of type Enum<Color> (an enum type). The relation is analogous to a class and its instance.

2.5 Anonymous Structure

Anonymous structures can be used to group data without explicitly creating a type. The following example creates a structure with two fields x and name, and initializes their values to 12 and "foo" respectively:

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

The general syntactic rules follow:

- 1. A structure is enclosed in curly braces {}.
- 2. Has a *comma-separated* list of key-value-pairs.

- 3. A *colon* separates the key, which must be a Expressions (Chapter 5), from the value.
- 4. The value can be any Haxe expression.

Rule 4 implies that structures can be nested and complex, e.g.:

```
1 var user = {
2    name : "Nicolas",
3    age : 32,
4    pos : [{ x : 0, y : 0 },{ x : 1, y : -1 }],
5 };
```

Fields of structures, like classes, are accessed using a *dot* (.) like so:

```
1 user.name; // get value of name, which is "Nicolas"
2 user.age = 33; // set value of age to 33
```

It is worth noting that using anonymous structures does not subvert the typing system. The compiler ensures that only available fields are accessed, which means the following program does not compile:

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

The error message indicates that the compiler knows the type of point: It is a structure with fields x and y of type Float. Since it has no field z the access fails. The type of point is known through type inference (3.6), which thankfully saves us from using explicit types for local variables. However, if point was a field, explicit typing would be necessary:

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

To avoid this kind of redundant type declaration, especially for more complex structures, it is advised to use a typedef (3.1):

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

2.5.1 JSON-Notation for Structure Values

It is also possible to use *ISON*-notation for structures by using *string literals* for the keys:

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

While any string literal is allowed, the field is only considered part of the type if it is a valid haxe identifier. Otherwise, haxe syntax does not allow expressing access to such a field, and reflection (10.6) has to be employed through the use of Reflect.field and Reflect.setField.

2.5.2 Class Notation for Structure Types

When defining a structure type, have allows using the same syntax as described in Class Fields (Chapter 4). The following typedef (3.1) declares a Point type with variable fields x and y of type Int:

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

2.5.3 Optional Fields

I don't really know how these work yet.

2.5.4 Impact on Performance

Using structures and by extension structural subtyping (3.5.2) has no impact on performance when compiling to dynamic targets (2.2). However, on static targets (2.2) a dynamic lookup has to be performed, which is typically slower than a static field access.

2.6 Function

The function type, along with the monomorph (2.9), is a type which is usually well-hidden from Haxe users, yet present everywhere. We can make it surface by using \$type, a special Haxe identifier which outputs the type its expression has during compilation:

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

There is a strong resemblance between the declaration of function test and the output of the first \$type expression, yet also a subtle difference:

- Function arguments are separated by the special arrow token -> instead of commas, and
- the *function return type* appears at the end after another ->.

In either notation, it is obvious that the function test accepts a first argument of type Int, a second argument of type String and returns a value of type Bool. If a call to this function is made, such as test (1, "foo") within the second \$type expression, the haxe typer checks if 1 can be assigned to Int and if "foo" can be assigned to String. The type of the call is then equal to the type of the value test returns, which is Bool.

2.6.1 Optional Arguments

Optional arguments are declared by prefixing an argument identifier with a question mark?:

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

Function test has two optional arguments i of type Int and s of String. This is directly reflected in the function type, which line 3 outputs. This example program calls test four times and prints its return value.

- 1. the first call is made without any arguments
- 2. the second call is made with a singular argument 1
- 3. the third call is made with two arguments 1 and "foo"
- 4. the fourth call is made with a singular argument "foo"

The output shows that optional arguments which are omitted from the call have a value of null. This implies that the type of these arguments must admit null as value, which raises the question of its nullability (2.2). The haxe compiler ensures that optional basic type arguments are nullable by inferring their type as Null<T> when compiling to a static target (2.2).

While the first three calls are intuitive, the fourth one might come as a surprise: It is indeed allowed to skip optional arguments if the supplied value is assignable to a later argument.

2.6.2 Default values

Haxe allows default values for arguments by assigning a *constant value* to them:

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

This example is very similar to the one from Optional Arguments (Section 2.6.1), with the only difference being that the values 12 and "bar" are assigned to the function arguments i and s respectively. The effect is that, instead of null, the default values are used should an argument be omitted from the call.

Default values in haxe are not part of the type and are not replaced at call-site (unless the function is inlined (4.4.2), which might be considered a more typical approach. On some targets, the compiler may still pass null for omitted argument values and generate code into the function which is similar to this:

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

This should be considered in performance-critical code, where a solution without default values may sometimes be more viable.

2.7 Dynamic

While haxe has a static type system, this type system can effectively be turned off by using the Dynamic type. A *dynamic value* can be assigned to anything, and anything can be assigned to it. This has several drawbacks:

- The compiler can no longer type-check assignments, function calls and other constructs where specific types are expected.
- Certain optimizations, in particular when compiling to static targets, can no longer be employed.
- Some common errors, e.g. a typo in a field access, can not be caught at compile-time and likely cause an error at runtime.
- Dead Code Elimination (Section 8.1) cannot detect used fields if they are used through Dynamic.

It is very easy to come up with examples where the usage of Dynamic can cause problems at runtime. Consider compiling the following two lines to a static target:

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

Trying to run a compiled program in the Flash Player yields an error Property foo not found on Number and there is no default value. Without Dynamic, this would have been detected at compile-time.

Trivia: Dynamic Inference before haxe 3

The haxe 3 compiler never infers a type to Dynamic, so users must be explicit about it. Previous haxe versions used to infer arrays of mixed types, e.g. [1, true, "foo"], as Array<Dynamic>. We found that this behavior introduced too many type problems and thus removed it for haxe 3.

Use of Dynamic should be minimized as there are better options in many situations, but sometimes it is just practical to use it. Parts of the haxe Reflection (Section 10.6) API use it, and

it is sometimes the best option when dealing with custom data structures that are not known at compile-time.

Dynamic behaves in a special way when being unified (3.5) with a monomorph (2.9), which is described in Monomorphs (Section 3.5.3).

Trivia: Dynamic in the Standard Library

Dynamic was quite frequent in the haxe standard library before haxe 3. With the continuous improvements of the haxe type system, the occurences of Dynamic were reduced over the releases leading to haxe 3.

2.7.1 Dynamic with Type Parameter

Dynamic is a special type because it allows explicit declaration with and without a type parameter (3.2). If such a type parameter is provided, the semantics described in Dynamic (Section 2.7) are constrained to all fields being compatible with the parameter type:

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

2.7.2 Implementing Dynamic

Classes can implement (2.3.3) Dynamic and Dynamic<T>, which enables arbitrary field access. In the former case, fields can have any type, in the latter they are constrained to be compatible with the parameter type:

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

Implementing Dynamic does not satisfy the requirements of other implemented interfaces. The expected fields still have to be implemented explicitly.

resolve()

2.8 Abstract

@:enum and @:expose

An abstract type is a type which is, at runtime, actually a different type. It is a compile-time feature which defines types "over" concrete types in order to modify or augment their behavior:

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

We can derive the following from this example:

- The keyword abstract denotes that we are declaring an abstract type.
- Abstract is the name of the abstract and could be anything conforming to the rules for type identifiers.
- Enclosed in parenthesis () is the underlying type Int.
- Enclosed in curly braces {} are the fields,
- which is a constructor function expressionnew accepting one argument i of type Int.

Definition: Underlying Type

The underlying type of an abstract is the type which is used to represent said abstract at runtime. It is usually a concrete (i.e. non-abstract) type, but could be another abstract type as well.

The syntax is reminiscent of classes and the semantics are indeed similar. In fact, everything in the "body" of an abstract (that is everything after the opening curly brace) is parsed as class fields. Abstracts may have method (4.3) fields and non-physical (4.2.3) property (4.2) fields.

Furthermore, abstracts can be instantiated and used just like classes:

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

As mentioned before, abstracts are a compile-time feature, so it is interesting to see what the above actually generates. A suitable target for this is Javascript, which tends to generate concise and clean code. Compiling the above (using haxe -main MyAbstract -js myabstract.js) shows this Javascript code:

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

The abstract type Abstract completely disappeared from the output and all that is left is a value of its underlying type, Int. This is because the constructor of Abstract is inlined, something we shall learn about later in section Inline (Section 4.4.2), and its inlined expression assigns a value to this. This might be shocking when thinking in terms of classes, but it is precisely what we want to express in the context of abstracts. Any *inlined member method* of an abstract can assign to this, and thus modify the "internal value".

A good question at this point is what happens if a member function is not declared inline, because the code obviously has to go somewhere. Haxe creates a private class, known to be the *implementation class*, which has all the abstract member functions as static functions accepting an additional first argument this of the underlying type. While this technically is an implementation detail, it can be used for selective functions (2.8.4).

Trivia: Basic Types and abstracts

Before the advent of abstract types, all basic types were implemented as extern classes or enums. While this nicely took care of some aspects such as Int being a "child class" of Float, it caused issues elsewhere. For instance, with Float being an extern class, it would unify with the empty structure {}, making it impossible to constrain a type to accepting only real objects.

2.8.1 Implicit Casts

Unlike classes, abstracts allow defining implicit casts. There are two kinds of implicit casts:

Direct: Allows direct casting of the abstract type to or from another type. This is defined by adding to and from rules to the abstract type, and is only allowed for types which unify with the underlying type of the abstract.

Class field: Allows casting via calls to special cast functions. These functions are defined using @:to and @:from metadata. This kind of cast is allowed for all types.

The following code example shows an example of *direct* casting:

```
abstract MyAbstract(Int) from Int to Int {
1
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;
           var b:Int = a;
10
11
12
```

We declare MyAbstract as being from Int and to Int, meaning it can be assigned from Int and assigned to Int. This is shown in lines 9 and 10, where we first assign the Int 12 to variable a of type MyAbstract (this works due to the from Int declaration) and then that abstract back to variable b of type Int (this works due to the to Int declaration).

Class field casts have the same semantic, but are defined completely differently:

```
abstract MyAbstract(Int) {
1
2
       inline function new(i:Int) {
3
           this = i;
4
5
       @:from static public function fromString(s:String) {
6
7
           return new MyAbstract(Std.parseInt(s));
8
9
10
       @:to public function toArray() {
           return [this];
11
12
13
14
```

```
15 class ImplicitCastField {
16    static public function main() {
17        var a:MyAbstract = "3";
18        var b:Array<Int> = a;
19        trace(b); // [3]
20    }
21 }
```

By adding @:from to a static function, that function qualifies as implicit cast function from its argument type to the abstract. These functions must return a value of the abstract type. They must also be declared static.

Similarly, adding @:to to a function qualifies it as implicit cast function from the abstract to its return type. These functions are typically member-functions, but they can be made static and then serve as selective function (2.8.4).

In the example, the method fromString allows the assignment of value "3" to variable a of type MyAbstract, while the method toArray allows assigning that abstract to variable b of type Array<Int>.

When using this kind of cast, calls to that functions are inserted were required. This becomes obvious when looking at the Javascript output:

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

This can be further optimized by inlining (4.4.2) both cast functions, turning the output to this:

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

The *selection algorithm* when assigning a type A to a type B with at least one of them being an abstract is simple:

- 1. If A is not an abstract, go to 3.
- 2. If A defines a *to*-conversions that admits B, go to 6.
- 3. If B is not an abstract, go to 5.
- 4. If B defines a *from*-conversions that admits A, go to A.
- 5. Stop, unification fails.
- 6. Stop, unification succeeds.

By design, implicit casts are *not transitive*, as the following example shows:

```
abstract A(Int) {
1
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
   abstract C(Int) {
11
       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
           var c:C = b; // valid, uses B.toC
19
20
           var c:C = a; // error, A should be C
21
       }
22
```

While the individual casts from A to B and from B to C are allowed, a transitive cast from A to C is not. This is to avoid ambiguous cast-paths and retain a simple selection algorithm.

2.8.2 Operator Overloading

Abstracts allow overloading of unary and binary operators by adding the @:op metadata to class fields:

```
1
   abstract MyAbstract(String) {
2
       public inline function new(s:String) {
3
           this = s;
4
5
       @:op(A * B) public function repeat(rhs:Int):MyAbstract {
6
           var s:StringBuf = new StringBuf();
7
           for (i in 0...rhs)
8
9
                s.add(this);
           return new MyAbstract(s.toString());
10
11
12
13
14
   class AbstractOperatorOverload {
       static public function main() {
15
           var a = new MyAbstract("foo");
16
17
           trace(a * 3); // foofoofoo
18
19
```

By defining @:op(A * B), the function repeat serves as operator method for the multiplication * operator when the type of the left value is MyAbstract and the type of the right value is Int. The usage is shown in line 17, which turns into this when compiled to Javascript:

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

Similar to implicit casts with class fields (2.8.1), a call to the overload method is inserted where required.

Note that the example repeat function is not commutative: While MyAbstract * Int works, Int * MyAbstract does not. If this should be allowed as well, the @:commutative metadata can be added. If it should work only for Int * MyAbstract, but not for MyAbstract * Int, the overload method can be made static, accepting Int and MyAbstract as first and second type respectively.

Overloading unary operators is analogous:

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

Note that both binary and unary operator overloads can return any type.

It is also possible to omit the method body of a @:op function, but only if the underlying type of the abstract allows the operation in question and if the resulting type can be assigned back to the abstract.

2.8.3 Array Access

Array access describes the particular syntax traditionally used to access the value in an array at a certain offset. This is usually only allowed with arguments of type Int, but with abstracts it is possible to define custom array access methods. The Haxe Standard Library (10) uses this in its Map type, where the following two methods can be found:

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

There are two kinds of array access methods:

- If an @:arrayAccess method accepts one argument, it is a getter.
- If an @:arrayAccess method accepts two arguments, it is a setter.

The methods get and arrayWrite seen above then allow the following usage:

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

At this point it should not be surprising to see that calls to the array access fields are inserted in the output:

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

2.8.4 Selective Functions

Since the compiler promotes abstract member functions to static functions, it is possible to define static functions by hand and use them on an abstract instance. The semantics here are similar to those of static extensions (6.2), where the type of the first function argument determines for which types a function is defined:

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

The method getString of abstract MyAbstract is defined to accept a first argument of MyAbstract<Stri. This causes it to be available on variable a on line 14 (because the type of a is MyAbstract<String>), but not on variable b whose type is MyAbstract<Int>.

Trivia: Accidental Feature

Selective functions were never actually designed, but rather discovered. After the idea was first mentioned, it required only minor adjustments in the compiler to make them work. Their discovery also lead to the introduction of multi-type abstracts, such as Map.

2.8.5 Enum abstracts

Since Haxe 3.1.0

By adding the : enum metadata to an abstract definition, that abstract can be used to define finite value sets:

```
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
       static function printStatus(status:HttpStatus) {
13
           return switch(status) {
14
15
              case NotFound: "Not found";
               case MethodNotAllowed: "Method not allowed";
16
17
           }
18
       }
19
```

The Haxe compiler replaces all field access to the HttpStatus abstract with their values, as evident in the Javascript output:

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

This is similar to accessing variables declared as inline (4.4.2), but has several advantages:

- The typer can ensure that all values of the set are typed correctly.
- When matching (6.3) an enum abstract, the pattern matcher checks for exhaustiveness (6.3.10).
- Defining fields requires less syntax.

2.8.6 Exposing abstracts

Since Haxe 3.1.0

When wrapping an underlying type, it is sometimes desirable to "keep" parts of its functionality. Because writing forwarding functions by hand is cumbersome, Haxe allows adding the :expose metadata to an abstract type:

```
1 @:expose(push, pop => read)
2 abstract MyArray<S>(Array<S>) {
3    public inline function new() {
4         this = [];
5    }
6 }
```

```
7
8 class Main {
9   static public function main() {
10     var myArray = new MyArray();
11     myArray.push(12);
12     myArray.read();
13   }
14 }
```

The MyArray abstract in this example wraps Array. Its :expose metadata has two arguments which demonstrate the two allowed ways of specifying exposed fields:

- With a plain identifier (here: push) to expose the field as is.
- Using the => notation (here: pop => read) to expose the underlying field on the left side (here: pop) as the field name on the right side (here: read).

In this example, the main function instantiates MyArray and accesses its pop and read function. As usual we can look at the Javascript output to see how the code is being generated:

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

Trivia: Implemented as macro

Both the :enumand :expose functionality are implemented using macros (9). The implementations can be found in the file haxe/macro/Build.hx. The compiler rewrites both metadata as a :build metadata which calls the implementations. See type building (9.5) for details.

2.8.7 Core-type abstracts

The Haxe Standard Library defines a set of basic types as core-type abstracts. They are identified by the :coreType metadata and the lack of an underlying type declaration. These abstracts can still be understood to represent a different type, but that type is native to the Haxe target.

Introducing custom core-type abstracts is rarely necessary in user code as it requires the target Haxe target to be able to make sense of it. However, there could be interesting use-cases for authors of macros and new Haxe targets.

In contrast to opaque abstracts, core-type abstracts have the following properties:

- They have no underlying type.
- They are considered nullable unless they are annotated with :notNull metadata.
- They are allowed to declare array access (2.8.3) functions without expressions.
- Operator overloading fields (2.8.2) that have no expression are not forced to adhere to the Haxe type semantics.

2.9 Monomorph

A monomorph is a type which may, through unification (3.5), morph into a different type later. We shall see details about this type when talking about type inference (3.6).

Chapter 3

Type System

We learned about the different kinds of types in Types (Chapter 2) and it is now time to see how they interact with each other. We start off easy by introducing typedef (3.1), a mechanism to give a name (or alias) to a more complex type. Among other things, this will come in handy when working with types having type parameters (3.2).

A lot of type-safety is achieved by checking if two given types of above type groups are compatible. That is, the compiler tries to perform *unification* between them, as detailed in Unification (Section 3.5).

All types are organized in *modules* and can be addressed through *paths*. Modules and Paths (Section 3.7) will give a detailed explanation of the related mechanics.

3.1 Typedef

We briefly looked at typedefs while talking about anonymous structures (2.5) and saw how we could shorten a complex structure type by giving it a name. This is precisely what typedefs are good for, and giving names to structure types might even be considered their primary use. In fact, it is so common that the distinction appears somewhat blurry and many haxe users consider typedefs to actually *be* the structure.

A typedef can give a name to any other type:

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

This enables us to use IA in places where we would normally use Array<Int>. While this saves only a few keystrokes in this particular case, it can make a difference for more complex, compound types. Again, this is why typedef and structures seem so connected:

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

A typedef is not a textual replacement, but actually a real type. It can even have type parameters (3.2) as the Iterable type from the standard library demonstrates:

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

3.1.1 Extensions

Extensions are used to express that a structure has all the fields of a given type in addition to some more:

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

The greater-than operator > denotes that an extension of Iterable<T> is being created, with the additional class fields following. In this case, a read-only property (4.2) length of type Int is required.

In order to be compatible with IterableWithLength<T>, a type then must be compatible with Iterable<T> and also provide a read-only length property of type Int. The example assigns an Array, which happens to fulfill these requirements.

There may only be a single extension on a structure, so extensions can be understood as an inheritance (2.3.2) mechanism for structures.

3.2 Type Parameters

Haxe allows parametrization of a number of types, as well as class fields (4) and enum constructors (2.4.1). Type parameters are defined by enclosing comma-separated type parameter names in angle brackets <>. A simple example from the standard library is Array:

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

Whenever an instance of Array is created, its type parameter T becomes a monomorph (2.9). That is, it can be bound to any type, but only one at a time. This binding can happen

```
\textbf{explicitly} \ \ \textbf{by invoking the constructor with explicit types (new \ \texttt{Array} < \texttt{String} > \texttt{())} \ \textbf{or}
```

implicitly by type inference (3.6), e.g. when invoking arrayInstance.push("foo").

Inside the definition of a class with type parameters, these type parameters are an unspecific type. Unless constraints (3.2.1) are added, the compiler has to assume that the type parameters could be used with any type. As a consequence, it is not possible to access fields of type parameters or cast (5.21) to a a type parameter type. It is also not possible to create a new instance of a type parameter type, unless the type parameter is generic (3.3) and constrained accordingly.

The following table shows where type parameters are allowed:

Parameter on	Bound upon	Notes
Class	instantiation	Can also be bound upon member field access.
Enum	instantiation	_
Enum Constructor	instantiation	
Function	invocation	Allowed for methods and named lvalue closures.
Structure	instantiation	

With function type parameters being bound upon invocation, such a type parameter (if unconstrained) accepts any type. However, only one type per invocation is accepted, which can be utilized if a function has multiple arguments:

```
class FunctionTypeParameter {
1
       static public function main() {
2
3
           equals(1, 1);
4
           equals("foo", "bar"); // runtime message: bar should be foo
5
           equals(1, "foo"); // compiler error: String should be Int
6
7
       static function equals<T>(expected:T, actual:T) {
8
9
           if (actual != expected) trace('$actual should be $expected');
10
11
```

Both arguments expected and actual of the equals function have type T. This implies that for each invocation of equals, the two arguments must be of the same type. The compiler admits the first call (both arguments being of Int) and the second call (both arguments being of String), but the third attempt causes a compiler error.

Trivia: Type parameters in expression syntax

We often get the question why a method with type parameters cannot be called as method<String>(). The error messages the compiler gives are not quite helpful, but there is a simple reason for that: The above code is parsed as if both < and > were binary operators, yielding (method < String) > ().

3.2.1 Constraints

Type parameters can be constrained to multiple types:

```
typedef Measurable = {
1
2
       public var length(default, null):Int;
3
5
   class Constraints {
       static public function main() {
6
           trace(test([]));
7
           trace(test(["bar", "foo"]));
8
           test("foo"); // String should be Iterable<String>
9
10
11
       static function test<T:(Iterable<String>, Measurable)>(a:T) {
12
           if (a.length == 0) return "empty";
13
14
           return a.iterator().next();
15
16
```

Type parameter T of method test is constrained to the types Iterable < String > and Measurable. The latter is defined using a typedef (3.1) for convenience and requires compatible types to have a read-only property (4.2) named length of type Int. The constraints then say that a type is compatible if

- it is compatible with Iterable<String> and
- has a length-property of type Int.

We can see that invoking test with an empty array in line 7 and an Array < String > in line 8 works fine. This is because Array has both a length-property and an iterator-method. However, passing a String as argument in line 9 fails the constraint check, because String is not compatible with Iterable < T >.

3.3 Generic

Usually, the Haxe compiler generates only a single class or function, even if it has type parameters. This results in a natural abstraction, where the code generator for the target language has to assume that a type parameter could be of any type. The generated code then might have to perform some type checks, which can be detrimental to performance.

A class or function can be made *generic* by attributing it with the :generic metadata (6.7). This causes the compiler to emit a distinct class/function per type parameter combination with mangled names. A specification like this can yield a boost in performance-critical code portions on static targets (2.2) at the cost of a larger output size:

```
@:generic
1
2
   class MyArray<T> {
       public function new() { }
3
4
5
   class Main {
6
7
      static public function main() {
           var a = new MyArray<String>();
8
9
10
  }
```

It seems unusual to see the explicit type MyArray<String> here as we usually let type inference (3.6) deal with this, but here it is indeed required. The compiler has to know the exact type of a generic class upon construction. The Javascript output shows the result:

```
(function () { "use strict";
2
   var Main = function() { }
   Main.main = function() {
3
       var a = new MyArray_String();
4
       var b = new MyArray_Int();
5
6
  var MyArray_Int = function() {
7
8
  var MyArray String = function() {
10
11 Main.main();
12 })();
```

We can identify that MyArray<String> and MyArray<Int> have become MyArray_-String and MyArray_Int respectively. This is similar for generic functions:

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

```
4          method(1);
5     }
6
7     @:generic static function method<T>(t:T) { }
8 }
```

Again, the Javascript output makes it obvious:

```
(function () { "use strict";
1
   var Main = function() { }
2
3
  Main.method_Int = function(t) {
4
   Main.method_String = function(t) {
5
6
  Main.main = function() {
7
      Main.method_String("foo");
8
9
      Main.method Int(1);
10
  Main.main();
11
12 })();
```

3.3.1 Construction of generic type parameters

Definition: Generic Type Parameter

A type parameter is said to be generic if its containing class or method is generic.

With normal type parameter, it is not possible to construct them, i.e. new T() is a compiler error. This is because haxe generates only a single function and the construct then makes no sense. This is different when the type parameter is generic: Since we know that the compiler will generate a distinct function for each type parameter combination, it is possible to replace the T new T() with the real type.

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

It should be noted that top-down inference (3.6.1) is used here to determine the actual type of T. There are two requirements for this kind of type parameter construction to work. The constructed type parameter must be

- 1. generic and
- 2. be explicitly constrained (3.2.1) to having a constructor (2.3.1).

Here, 1. is given by make having the @:generic metadata, and 2. by T being constrained to Constructible. The constraint holds for both String and haxe. Template as both have a constructor accepting a singular String argument. Sure enough, the relevant Javascript output looks as expected:

```
var Main = function()
1
  Main.__name__ = true;
   Main.make_haxe_Template = function() {
3
       return new haxe.Template("foo");
4
5
  Main.make_String = function() {
6
7
       return new String("foo");
8
9
  Main.main = function() {
       var s = Main.make_String();
10
       var t = Main.make_haxe_Template();
11
12
```

3.4 Variance

While variance is relevant in other places, it occurs particularly often with type parameters and often comes as a surprise in this context. It is also very easy to trigger variance errors:

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

Apparently, an Array<Child> cannot be assigned to an Array<Base>, even though Child can be assigned to Base. The reason for this might be somewhat unexpected: It is not allowed because arrays can be written to, e.g. via their push() method. It is easy to generate problems by ignoring variance errors:

```
1 class Base {
2   public function new() { }
3 }
4
5 class Child extends Base { }
```

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

What happens here is that we subvert the type checker by using a cast (5.21), thus allowing the assignment in line 12. With that we hold a reference bases to the original array, typed as Array<Base>. This allows pushing another type compatible with Base, OtherChild, onto that array. However, our original reference children is still of type Array<Child>, and things go bad when we encounter the OtherChild instance in one of its elements while iterating.

If Array had no push () method and no other means of modification, the assignment would be safe because no incompatible type could be added to it. We can achieve this in Haxe by restricting the type accordingly using structural subtyping (3.5.2):

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

With b being typed as MyArray<Base> and MyArray only having a pop() method, we can safely assign. There is no method defined on MyArray which could be used to add incompatible types, it is thus said to be *covariant*.

Definition: Covariance

A compound type (2) is considered covariant if its component types can be assigned to less specific components, i.e. if they are only read, but never written.

Definition: Contravariance

A compound type (2) is considered contravariant if its component types can be assigned to less generic components, i.e. if they are only written, but never read.

3.5 Unification

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

Unification is the heart of the type system and contributes immensely to the robustness of haxe programs. It describes the process of checking if a type is compatible to another type.

Definition: Unification

Unification between two types A and B is a directional process which answers the question if A *can be assigned to B*. It may *mutate* either type if it is or has a monomorph (2.9).

Unification errors are very easy to trigger:

```
1 class Main {
2    static public function main() {
3      var s:String = 1; // Int should be String
4    }
5 }
```

We try to assign a value of type Int to a variable of type String, which causes the compiler to try and *unify Int with String*. This is, of course, not allowed and makes the compiler emit the error Int should be String.

In this particular case, the unification is triggered by an *assignment*, a context in which the "is assignable to" definition is intuitive. It is one of several cases where unification is performed:

Assignment: If a is assigned to b, the type of a is unified with the type of b.

Function call: We have briefly seen this one while introducing the function (2.6) type. In general, the compiler tries to unify the first given argument type with the first expected argument type, the second given argument type with the second expected argument type and so on until all argument types are handled.

Function return: Whenever a function has a return expression, the type of e is unified with the function return type. If the function has no explicit return type, it is infered to the type of e and subsequent return expressions are infered against it.

Array declaration: The compiler tries to find a minimal type between all given types in an array declaration. Refer to Common Base Type (Section 3.5.5) for details.

Object declaration: If an object is declared "against" a given type, the compiler unifies each given field type with each expected field type.

Operator unification: Certain operators expect certain types which given types are unified against. For instance, the expression a && b unifies both a and b with Bool and the expression a == b unifies a with b.

3.5.1 Between Class/Interface

When defining unification behavior between classes, it is important to remember that unification is directional: We can assign a more specialized class (e.g. a child class) to a generic class (e.g. a base class), but the reverse is not valid.

The following assignments are allowed:

• child class to base class

- class to implementing interface
- interface to base interface

These rules are transitive, meaning that a child class can also be assigned to the base class of its base class, an interface its base class implements, the base interface of an implementing interface and so on.

3.5.2 Structural Subtyping

Definition: Structural Subtyping

Structural subtyping defines an implicit relation between types that have the same structure.

In haxe, structural subtyping is only possible when assigning a class instance to a structure. The following example is part of the Lambda class of the Haxe Standard Library (10):

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

The empty-method checks if an Iterable has an element. For this purpose, it is not necessary to know anything about the argument type other than the fact that it is considered an iterable. This allows calling the empty-method with any type that unifies with Iterable<T>, which applies to a lot of types in the Haxe Standard Library.

This kind of typing can be very convenient, but extensive use may be detrimental to performance on static targets, which is detailed in Impact on Performance (Section 2.5.4).

3.5.3 Monomorphs

Unification of types having or being a monomorph (2.9) is detailed in Type Inference (Section 3.6).

3.5.4 Function Return

Unification of function return types may involve the Void-type (2.1.5) and require a clear definition of what unifies with Void. With Void describing the absence of a type, it is not assignable to any other type, not even Dynamic. This means that if a function is explicitly declared as returning Dynamic, it must not return Void.

The opposite applies as well: If a function declares a return type of Void, it cannot return Dynamic or any other type. However, this direction of unification is allowed when assigning function types:

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

The right-hand function clearly is of type <code>Void->String</code>, yet we can assign it to variable <code>func</code> of type <code>Void->Void</code>. This is because the compiler can safely assume that the return type is irrelevant, given that it could not be assigned to any non-Void type.

3.5.5 Common Base Type

Given a set of multiple types, a common base type is a type which all types of the set unify against:

```
1
   class Base {
      public function new() { }
2
3
4
5
   class Child1 extends Base { }
   class Child2 extends Base { }
7
   class UnifyMin {
8
9
       static public function main() {
10
           var a = [new Child1(), new Child2()];
           $type(a); // Array<Base>
11
12
13
```

Although Base is not mentioned, the haxe compiler manages to infer it as the common type of Childl and Childl. The haxe compiler employs this kind of unification in the following situations:

- array declarations
- if/else
- cases of a switch

3.6 Type Inference

The effects of type inference have been seen throughout this document and will continue to be important. A simple example shows type inference at work:

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

The special construct \$type was previously mentioned in order to simplify the explanation of the Function (Section 2.6) type, so let us introduce it officially now:

Construct: type

type is a compile-time mechanism being called like a function, with a single argument. The compiler evaluates the argument expression and then outputs the type of that expression.

In the example above, the first type prints Unknown < 0>. This is a monomorph (2.9), a type that is not yet known. The next line x = "foo" assigns a String literal to x, which causes the unification (3.5) of the monomorph with String. We then see that the type of x indeed has changed to String.

Whenever a type other than Dynamic (Section 2.7) is unified with a monomorph, that monomorph *becomes* that type: it *morphs* into that type. Therefore it cannot morph into a different type afterwards, a property expressed in the *mono* part of its name.

Following the rules of unification, type inference can occur in compound types:

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

Variable x is first initialized to an empty Array. At this point we can tell that the type of x is an array, but we do not yet know the type of the array elements. Consequentially, the type of x is Array<Unknown<0>>. It is only after pushing a String onto the array that we know the type to be Array<String>.

3.6.1 Top-down Inference

Most of the time, types are inferred on their own and may then be unified with an expected type. In a few places, however, an expected type may be used to influence inference. We then speak of *top-down inference*.

Definition: Expected Type

Expected types occur when the type of an expression is known before that expression has been typed, e.g. because the expression is argument to a function call. They can influence typing of that expression through what is called top-down inference (3.6.1).

A good example are arrays of mixed types. As mentioned in Dynamic (Section 2.7), the compiler refuses [1, "foo"] because it cannot determine an element type. Employing top-down inference, this can be overcome:

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

Here, the compiler knows while typing [1, "foo"] that the expected type is Array<Dynamic>, so the element type is Dynamic. Instead of the usual unification behavior where the compiler would attempt (and fail) to determine a common base type (3.5.5), the individual elements are typed against and unified with Dynamic.

We have seen another interesting use of top-down inference when construction of generic type parameters (3.3.1) was introduced:

```
typedef Constructible = {
1
2
       public function new(s:String):Void;
3
4
  class Main {
5
       static public function main() {
6
7
           var s:String = make();
           var t:haxe.Template = make();
8
9
10
       @:generic
```

```
12    static function make<T:Constructible>():T {
13      return new T("foo");
14    }
15 }
```

The explicit types String and haxe. Template are used here to determine the return type of make. This works because the method is invoked as make(), so we know the return type will be assigned to the variables. Utilizing this information, it is possible to bind the unknown type T to String and haxe. Template respectively.

3.6.2 Limitations

Type inference saves a lot of manual type hints when working with local variables, but sometimes the type system still needs some help. In fact, it does not even try to infer the type of a variable (4.1) or property (4.2) field unless it has a direct initialization.

There are also some cases involving recursion where type inference has limitations. If a function calls itself recursively while its type is not (completely) known yet, type inference may infer a wrong, too specialized type.

3.7 Modules and Paths

Definition: Module

All haxe code is organized in modules, which are addressed using paths. In essence, each .hx file represents a module which may contain several types. A type may be private, in which case only its containing module can access it.

The distinction of a module and its containing type of the same name is blurry by design. In fact, addressing haxe.ds.StringMap<Int> can be considered shorthand for haxe.ds.StringMap.StringMap.The latter version consists of four parts:

- 1. the package haxe.ds
- 2. the module name StringMap
- 3. the type name StringMap
- 4. the type parameter Int

If the module and type name are equal, the duplicate can be removed, leading to the haxe.ds.StringMap<I short version. However, knowing about the extended version helps with understanding how module sub-types (3.7.1) are addressed.

Paths can be shortened further by using an import (3.7.2), which typically allows omitting the package part of a path. This may lead to usage of unqualified identifiers, for which understanding the resolution order (3.7.3) is required.

Definition: Type path

The (dot-)path to a type consists of the package, the module name and the type name. Its general form is pack1.pack1.pack1.ModuleName.TypeName.

3.7.1 Module Sub-Types

A module sub-type is a type declared in a module with a different name than that module. This allows a single .hx file to contain multiple types, which can be accessed unqualified from within the module, and by using package. Module. Type from other modules:

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

Here, the sub-type <code>ExprDef</code> within module <code>haxe.macro.Expr</code> is accessed. By default, module sub-types are publicly available, but their visibility can be constrained to their enclosing module by adding the <code>private</code> keyword:

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

The accessibility can be controlled more fine-grained by using access control (6.8).

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:

```
import haxe.ds.StringMap;

class Main {
    static public function main() {
        new StringMap(); // instead of: new haxe.ds.StringMap();
    }
}
```

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:

```
import haxe.macro.Expr;

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

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 several aspects worth knowing about importing:

- The bottommost import takes priority (detailed in Resolution Order (Section 3.7.3)).
- The static extension (6.2) 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 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();
    }
}
```

Describe import a.*

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.

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 delared local variables (5.10) (including function arguments)
- the imported (3.7.2) modules, types and statics
- the available static extensions (6.2)
- 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

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 (6.2), 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.

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 have 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 have. 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 closure (5.9) 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:

```
class VariableField {
    static var member:String = "bar";

public static function main() {
    trace(member);
    member = "foo";
    trace(member);
}
```

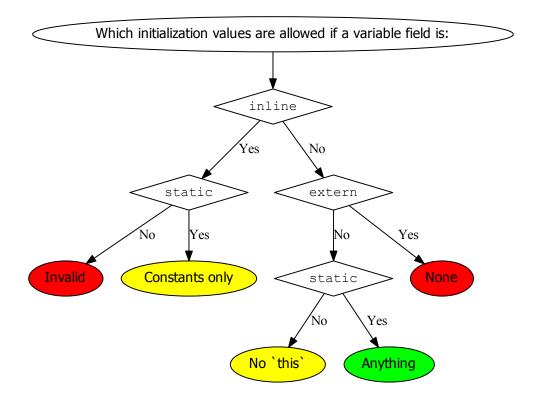


Figure 4.1: Variable initialization

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.

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:

```
class Main {
1
2
       // read from outside, write only within Main
       public var ro(default, null):Int;
3
4
5
       // write from outside, read only within Main
       public var wo(null, default):Int;
6
7
8
       // access through getter get_x and setter set_x
9
       public var x(get, set):Int;
10
       // read access through getter, no write access
11
12
       public var y(get, never):Int;
13
       // required by field x
14
15
       function get x() return 1;
16
       // required by field x
17
       function set_x(x) return x;
18
19
       // required by field y
20
       function get_y() return 1;
21
22
       function new() {
23
           var v = x;
24
25
           x = 2;
           x += 1;
26
27
28
29
       static public function main() {
           new Main();
30
31
       }
32
```

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

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

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

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

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

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

4.2.2 Impact on the type system

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

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

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

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

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 an 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 (6.7)

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

```
class Main {
   //Compiling fails with the error:
   //this field cannot be accessed because it is not a real variable
   public var x(get, set):Int;

function get_x() {
```

```
7         return x;
8     }
9
10     function set_x(x) {
11         return this.x = x;
12     }
13
14     static public function main() {}
15 }
```

If a physical field is indeed intended, it can be forced by attributing the field in question with the :isVar metadata (6.7):

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

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:

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

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

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

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

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

static function myFunc(f:String, i) {
        return true;
}
}
```

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

The important components here are

- the class Base which has a field method and a constructor,
- the class Child which extends Base and also has a field method 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 method() 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 field method 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):

```
class Base {
1
       public function new() { }
2
3
4
5
   class Child extends Base {
       private function method(obj:Child):Child {
6
7
           return obj;
8
9
10
   class ChildChild extends Child {
11
       public override function method(obj:Base):ChildChild {
12
13
           return null;
14
15
16
   class Main {
17
       static public function main() { }
18
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 {
       static public function available() {
2
3
           unavailable();
4
5
       static private function unavailable() { }
6
7
8
   class Main {
       static public function main() {
9
10
           MyClass.available();
           MyClass.unavailable(); // Cannot access private field
11
               unavailable
12
13
```

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.

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

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 id 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
2
       static inline function mid(s1:Int, s2:Int) {
           return (s1 + s2) / 2;
3
4
5
6
       static public function main() {
7
           var a = 1;
8
           var b = 2;
           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 of field mid was generated in place of the call to add (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.

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() {
```

```
return "original";

static public function main() {
    trace(test()); // original
    test = function() { return "new"; }
    trace(test()); // new
}

trace(test()); // new
}
```

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 compiletime, 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 as well.

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;
```

```
4 ...
5 exprN;
6 }
```

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 closures declared by function expressions (2.6). 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:

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

At runtime, blocks are evaluated from top to bottom. Control flow (e.g. exceptions (5.16) or return expressions (5.17)) 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:

```
1 var a = new Array();
2 a.push(1);
3 a.push(2);
4 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 6.5).

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 }.

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.

```
1 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 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
subject(e1, ..., eN); // call with multiple arguments
```

5.9 Closure

- 5.10 var
- 5.11 new

5.12 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 el 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.

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

5.13 while

5.14 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:

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

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

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

Here, expression2 may consist of another if expression:

```
1 if (condition1) expression1
2 else if(condition2) expression2
3 else expression3
```

If the value of an if expression is required, e.g. for var = 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.

5.15 switch

5.16 try/catch

5.17 return

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

```
1 return;
2 return expression;
```

It leaves the control-flow of the innermost function it is declared in, which has to be distinguished when closures (5.9) are involved:

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

The return leaves closure 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.

5.18 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 (5.15) is not supported in Haxe.

5.19 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:

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

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.

- **5.20** throw
- 5.21 cast

Chapter 6

Language Features

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 <code>-D key=value</code> or just <code>-D key</code>, 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 {
2
       public static function main() {
3
           #if !debug
                trace("ok");
4
           #elseif (debug_level > 3)
5
               trace(3);
7
                trace("debug level too low");
8
9
           #end
10
       }
11
```

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. As a consequence, these 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.

An exhaustive list of all built-in defines can be obtained by invoking the haxe compiler with the <code>--help-defines</code> argument.

6.2 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;
1
2
   class IntExtender {
3
       static public function triple(i:Int) {
4
           return i * 3;
5
6
7
8
9
   class Main {
       static public function main() {
10
11
           trace(12.triple());
12
13
```

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 Modules and Paths (Section 3.7)) as static extension brings all its types into context.

6.2.1 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.

6.3 Pattern Matching

Check "you" wording.

6.3.1 Introduction

Haxe 3 comes with improved options for pattern matching. Here we will explore the syntax for different patterns using this data structure as running example:

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

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:

6.3.2 Enum matching

As with haxe 2, enums can be matched by their constructors in a natural way. With haxe 3 pattern matching, this match can now be "deep":

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

The pattern matcher will check each case from top to bottom and pick the first one that matches the input value. If you are not too familiar with pattern matching, the following manual interpretation of each case rule might help:

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

6.3.3 Variable capture

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

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 y in that it matches anything, but with an identifier name like x it also binds the matched value to that variable.

6.3.4 Structure matching

It is now 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
```

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

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

6.3.5 Array matching

Arrays can be matched on fixed length:

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

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

6.3.6 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";
```

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

6.3.7 Guards

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

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

Note how 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.

6.3.8 Match on multiple values

Array syntax can also 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).

6.3.9 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:

```
1 enum Test {
2    TString(s:String);
3    TInt(i:Int);
4 }
5 
6 class Main {
7    static public function main() {
```

```
8
            var e = TString("f00");
9
            switch(e) {
10
                case TString(temp):
11
                     switch(temp.toLowerCase()) {
12
                         case "foo": true;
13
                         case _: false;
14
15
                case _: false;
            }
16
17
18
```

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

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. The way extractors are treated depends on the expression left of the => operator. If it is

- any identifier i, the generated code is equal to matchedValue.i(),
- otherwise for arbitrary expressions e, it is equal to e (matchedValue).

The distinction is made because interpreting a plain identifier like that can be quite convenient, as the toLowerCase() example above demonstrated. It also allows bringing extractors into context through static extensions (6.2).

Any expression can be used as extractor expression and the typer ensures that it is of function type $S \rightarrow T$, where S is the type of the currently matched value and T is equal to the type of the expression right of the => operator. With that, extractors can be combined with other features such as function binding (6.6):

```
1 enum Test {
2    TString(s:String);
3    TInt(i:Int);
4 }
```

```
6
   class Main {
7
       static public function main() {
           var e = TInt(4);
8
           var success = switch(e) {
9
10
                case TInt(lessThan.bind(_, 5) => true): true;
11
                case _: false;
12
            trace(success);
13
       }
14
15
       static function lessThan(lhs:Int, rhs:Int) {
16
17
            return lhs < rhs;</pre>
18
19
```

In this particular case, the extractor is called as lessThan(4, 5), yielding true.

A lcurrent imitation with regards to extractors is that they disable useless pattern checks (6.3.11).

6.3.10 Exhaustiveness checks

The compiler ensures that you do not forget a possible case for non value-only switches:

```
1 switch(true) {
2    case false:
3 } // This match is not exhaustive, these patterns are not matched:
    true
```

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

6.3.11 Useless pattern checks

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

6.4 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;
trace('The value of x is $x'); // The value of x is 12
```

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

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

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:

```
1 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.

6.5 Array Comprehension

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() {
           var a = [for (i in 0...10) i];
3
           trace(a); // [0,1,2,3,4,5,6,7,8,9]
4
           var i = 0;
6
           var b = [while(i < 10) i++];
           trace(b); // [0,1,2,3,4,5,6,7,8,9]
8
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++);</pre>
```

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)
        if (a % b == 0)
```

6.6 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>();
           var f = map.set.bind(_, "12");
4
           $type(map.set); // Int -> String -> Void
5
           $type(f); // Int -> Void
6
7
           f(1);
           f(2);
8
9
           f(3);
           trace(map.toString()); // {1 => 12, 2 => 12, 3 => 12}
10
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 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.

6.7 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:

```
@author("Nicolas")
   @debug
   class MyClass {
3
4
       @range(1, 8)
       var value:Int;
5
6
7
       @broken
8
       @:noCompletion
       static function method() { }
9
10
11
12
   class Main {
13
       static public function main() {
           trace(haxe.rtti.Meta.getType(MyClass)); // { author : ["
14
               Nicolas"], debug : null }
           trace(haxe.rtti.Meta.getFields(MyClass).value.range); // [1,8]
15
           trace(haxe.rtti.Meta.getStatics(MyClass).method); // { broken:
                null }
17
       }
18
```

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)

6.8 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 (6.7).

Forcing access: A target is forced to allow access to the given class or field by using the : access (target) metadata (6.7).

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:

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

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

- 6.9 Remoting
- 6.10 Runtime Type Information

Part II Compiler Reference

Chapter 7

Compiler Reference

7.1 Compiler Metadata

Chapter 8

Compiler Features

8.1 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 but Javascript.

-dce no: No DCE is performed.

-dce full: All classes are affected by DCE. This is the default setting when targeting Javascript.

The DCE-algorithm works well with typed code, but may fail when dynamic (2.7) or reflection (10.6) 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.

The compiler automatically defines the flag doe 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.

8.2 Completion

- 8.2.1 Field Access
- 8.2.2 Call Arguments
- 8.2.3 Usage
- 8.2.4 Position
- 8.2.5 Metadata

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.

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 Compiler Configuration (Section 9.7)).

Build Macros: These are defined for classes, enums and abstracts through the @:build or @:autoBuild metadata (6.7). 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 Compiler Configuration (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:

```
import haxe.macro.Expr;
1
2
3
   class Main {
       static public function main() {
4
           var x = 0;
5
           var b = add(x++);
6
7
           trace(x); // 2
8
       }
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 (6.2):

```
1
   import haxe.macro.Expr;
   using Main;
2
3
4
   class Main {
5
       static public function main() {
           identity("foo");
6
7
           identity(1);
           "foo".identity();
8
9
           //1.identity(); // Int has no field identity
10
11
       macro static function identity(e:ExprOf<String>) {
12
13
           return e;
14
15
```

The two direct calls to identity are accepted, even though the argument is declared as ExprOf<String>. It might come as a surprise that the Int 1 is accepted, but it is a logical consequence of what was explained about macro arguments (9.2): The argument expressions are never typed, so it is not possible for the compiler to check their compatibility by unifying (3.5).

This is different for the next two lines which are using static extensions (note the using Main): For these it is mandatory to type the left side ("foo" and 1) first in order to make sense of the identity field access. This makes it possible to check the types against the argument types, which causes 1.identity() to not consider Main.identity() as a suitable field.

9.2.2 Constant Expressions

A macro can be declared to expect constant (5.2) arguments:

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

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

9.2.3 Rest Argument

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

```
1 import haxe.macro.Expr;
2
```

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

9.3 Reification

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

9.3.1 Expression Reification

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

- **\$**{} and **\$e**{}: Expr -> Expr This can be used to compose expressions. The expression within the delimiting { } is executed, with its value being used in place.
- **\$a**{}: Expr -> Array<Expr> If used in a place where an Array<Expr> is expected (e.g. call arguments, block elements), \$a{} treats its value as that array. Otherwise it generates an array declaration.
- **\$b**{}: Array<Expr> -> Expr Generates a block expression from the given expression array.
- **\$i**{}: String -> Expr Generates an identifier from the given string.
- **\$p**{}: Array<String> → Expr Generates a field expression from the given string array.
- **\$v**{}: Dynamic -> Expr Generates an expression depending on the type of its argument. This is only guaranteed to work for basic types (2.1) and enum instances (2.4).

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

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

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

9.4 Tools

The Haxe Standard Library comes with a set of tool-classes to simplify working with macros. These classes work best as static extensions (6.2) 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.

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 tinkerbell¹ library for exactly the same purpose. Tinkerbell provided many useful macro tools long before they made it into the Haxe compiler and Haxe Standard Library. It remains the primary library for additional macro tools and offers other useful functionality as well.

9.5 Type Building

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

- They do not return expressions, but an array of class fields. Their return type must be set explicitly to Array<haxe.macro.Expr.Field>.
- Their context (9.1) has no local method and no local variables.
- Their context does have build fields, available from haxe.macro.Context.getBuildFields().
- They are not called directly, but are argument to a @:build or @:autoBuild metadata (6.7) on a class (2.3) or enum (2.4) declaration.

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

```
import haxe.macro.Context;
1
  import haxe.macro.Expr;
2
3
  class TypeBuildingMacro {
4
      macro static public function build(fieldName:String):Array<Field>
5
          var fields = Context.getBuildFields();
6
7
          var newField = {
              name: fieldName,
8
9
               doc: null,
```

```
meta: [],
10
                access: [AStatic, APublic],
11
                kind: FVar(macro: String, macro "my default"),
12
13
                pos: Context.currentPos()
14
            };
15
            fields.push (newField);
            return fields;
16
17
18
```

```
1 @:build(TypeBuildingMacro.build("myFunc"))
2 class Main {
3    static public function main() {
4         trace(Main.myFunc); // my default
5    }
6 }
```

The build method of TestBuildingMacro performs three steps:

- 1. It obtains the build fields using Context.getBuildFields().
- 2. It declares a new haxe.macro.expr.Field field using the funcName macro argument as field name. This field is a String variable (4.1) with a default value "my default" (from the kind field) and is public and static (from the access field).
- 3. It adds the new field to the build field array and returns it.

This macro is argument to the @:build metadata on the Main class. As soon as this type is required, the compiler does the following:

- 1. It parses the module file, including the class fields.
- 2. It sets up the type, including its relation to other types through inheritance (2.3.2) and interfaces (2.3.3).
- 3. It executes the type-building macro according to the @:build metadata.
- 4. It continues typing the class normally with the fields returned by the type-building macro.

This allows adding and modifying class fields at will in a type-building macro. In our example, the macro is called with a "myFunc" argument, making Main.myFunc a valid field access.

If a type-building macro should not modify anything, the macro can return null. This indicates to the compiler that no changes are intended and is preferable to returning Context.getBuildFields

9.5.1 Enum building

Building enums (2.4) is analogous to building classes with a simple mapping:

- Enum constructors without arguments are variable fields FVar.
- Enum constructors with arguments are method fields FFun.

Check if we can build GADTs this way.

```
1
  import haxe.macro.Context;
  import haxe.macro.Expr;
2
3
   class EnumBuildingMacro {
4
5
       macro static public function build():Array<Field> {
           var noArgs = makeEnumField("A", FVar(null, null));
6
7
           var fInt = switch (macro function(value:Int) { }).expr {
8
               case EFunction(_,f): f;
9
               case _: throw "false";
10
           var intArg = makeEnumField("B", FFun(fInt));
11
12
           return [noArgs, intArg];
13
14
       static function makeEnumField(name, kind) {
15
16
           return {
17
               name: name,
               doc: null,
18
19
               meta: [],
20
               access: [],
21
               kind: kind,
22
               pos: Context.currentPos()
23
       }
24
25
```

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

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

The main method proves the structure of our generated enum by matching (6.3) it. We can see that the generated type is equivalent to this:

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

9.5.2 @:autoBuild

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

```
import haxe.macro.Context;
1
2
  import haxe.macro.Expr;
3
  class AutoBuildingMacro {
4
       macro static public function fromInterface():Array<Field> {
5
           trace("fromInterface: " +Context.getLocalType());
6
           return null;
7
8
       }
9
       macro static public function fromBaseClass():Array<Field> {
10
           trace("fromBaseClass: " +Context.getLocalType());
11
           return null;
12
13
       }
14
```

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

This outputs during compilation:

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

9.6 Limitations

9.6.1 Macro-in-Macro

9.6.2 Static extension

The concepts or static extensions (6.2) 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:

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

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 (6.7) to the type in question:

```
1
  import haxe.macro.Context;
2
   import haxe.macro.Expr;
3
  #if !macro
  @:autoBuild(MyMacro.build())
5
6
   #end
7
  interface I1 { }
8
9
   #if !macro
  @:autoBuild(MyMacro.build())
10
11 #end
12
  interface I2 { }
13
  class C implements I1 implements I2 { }
14
15
   class MyMacro {
16
17
       macro static public function build():Array<Field> {
           var c = Context.getLocalClass().get();
18
19
           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 Compiler 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* for storing elements. It has one Type Parameters (Section 3.2) and all elements of the array must be of the specified type. Alternatively, arrays of mixed types are allowed if the type parameter is Dynamic (Section 2.7). See the below code snippet for an example.

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>.

The following example shows some basic examples of working with arrays:

```
class ArrayExample {
1
2
      static public function main() {
           //Compile error: Arrays of mixed types...
3
4
           //var myArray = [10, "Bob", false];
5
           //Dynamic Array with mixed types -- No Compile problems
6
           var myExplicitAray : Array<Dynamic> = [10, "Sally", true];
7
8
           //Initializing and iterating over an array
9
10
          var scores : Array<Int> = [110, 170, 35];
11
           var sum = 0;
           for (score in scores) {
```

```
13
                 sum += score;
14
            trace(sum); //315
15
16
17
            //Growing and accessing undefined indices
            var scores : Array\langle Int \rangle = [110, 170, 35];
18
            scores [5] = 92;
19
            trace(scores); //[110, 170, 35, null, null, 92]
20
            trace(scores[-5]); //null
21
22
        }
23
```

make sure this is true about static targets

In Haxe, arrays are unbounded: accessing or assigning an index outside the current size of the array does NOT result in an exception. When assigning, the array grows and the inbetween elements are assigned *null* (or the *static target* alternative: see Nullability (Section 2.2) for more details). When accessing, *null* is returned. Accessing a negative index in an array also returns *null*.

Arrays can be *iterated over* using a for (Section 5.12) loop. However removing elements while iterating over an array is error prone (but just fine with a *List*).

See Array Declaration (Section 5.5) for array initialization. New arrays can also be created by Array Comprehension (Section 6.5). The Array API has details about Array methods.

Maybe we should introduce \api{type name}

10.2.2 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:

```
1 class ListExample {
2    static public function main() {
3         var myList = new List<Int>();
4         for (ii in 0...5)
5             myList.add(ii);
6         trace(myList); //{0, 1, 2, 3, 4}
7    }
8 }
```

10.2.3 GenericStack

A GenericStack, like Array and List is a container for storing elements. It has one Type Parameters (Section 3.2) and all elements of the array 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.

I hope none of these are lies -C

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

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 Generic (Section 3.3) for more details.

10.2.4 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 MapExample{
1
       static public function main(){
2
           //Maps are initialized like arrays, but use '=>' operator
3
4
           //Maps can have their key value types defined explicity
           var map1 : Map<Int, String> = [1 => "one", 2=>"two"];
5
6
7
           //Or they can infer their key value types
           var map2 = ["one"=>1, "two"=>2, "three"=>3];
8
9
           $type(map2); // Map<String, Int>
10
           //Keys must be unique
11
           //var map3 = [1=>"dog", 1=>"cat"]; //Error: Duplicate Key
12
13
           //Maps values can be accessed using array accessors "[]"
14
           var map4 = ["M"=>"Monday", "T"=>"Tuesday"];
15
           trace(map4["M"]); //Monday
16
17
           //Maps iterate over their values by default
18
           var valueSum;
19
20
           for (value in map4)
               trace(value); //Monday \n Tuesday
21
22
           //Can iterate over keys by using the keys() method
23
           for (key in map4.keys())
24
25
               trace(key);//M \n T
```

```
//Like arrays, a new map can be made using comprehension
var map5 = [for (key in map4.keys()) key => "FRIDAY!!"];
trace(map5); //{M => FRIDAY!!, T => FRIDAY!!}
}
```

See the Map API for details of its methods.

Under the hood, a Map is an Abstract (Section 2.8) type. At compile time, it gets converted to one of several specialized types depending on the *key* type:

- haxe.ds.StringMap
- haxe.ds.IntMap
- haxe.ds.EnumValueMap
- haxe.ds.ObjectMap

So, at runtime, the Map type does not exist, and has been replaced with one of the above objects.

10.2.5 Option

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:

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

Alternatively, we can create regular expression with regular syntax:

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

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

We can use standard regular expression patterns such as:

- · . any character
- * repeat zero-or-more
- + repeat one-or-more
- ? optional zero-or-one
- [A-Z0-9] character ranges
- [^\r\n\t] character not-in-range
- (...) parenthesis to match groups of characters
- ^ beginning of the string (beginning of a line in multiline matching mode)
- \$ end of the string (end of a line in multiline matching mode)

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

• | "OR" statement.

For example, the following regular expression match a valid email address:

```
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 enable 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 match also 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:

```
class Main {
    static function main() {
        var r = ~/world/;
        var str = "hello world";
        trace(r.match(str)); // true : 'world' was found in the string
        trace(r.match("hello !")); // false
    }
}
```

10.3.2 Groups

Specific information can be extracted from a matched string by using *groups*. If match() returns true, we can get groups using the matched(X) method, where X is the number of a group defined by regular expression pattern:

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

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 {
1
2
    static function main() {
         var str = "abcdeeeeefghi";
3
          var r = ^{\sim}/e+/;
4
5
          r.match(str);
          trace(r.matched(0)); // "eeeee"
6
7
          trace(r.matchedPos()); // { pos : 4, len : 5 }
      }
8
9
```

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

10.3.3 Replace

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

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

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

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

10.3.4 Split

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

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

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.

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 Flash9, 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 fuctions operate on and return floats. However, a Int can be used where a Float is expected, and Haxe also converts Int to Float during most numeric operations. See Numeric Operators (Section 2.1.3) for more details on when operators convert Int to Float.

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

```
class MathExample {
1
2
       static public function main() {
3
           var x = 1/2;
4
           var y = 20.2;
5
6
           var z = -2;
7
8
           trace(Math.abs(z)); //2
9
           trace(Math.sin(x*Math.PI)); //1
10
           trace(Math.ceil(y)); //21
11
12
           //log is the natural logarithm
           trace(Math.log(Math.exp(5))); //5
13
```

10.4.1 Special Numbers

The math library has defitions 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 target. 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, for performance gains, wrap the operation in *Std.int()*. The Haxe compiler can then optimize for integer operations. An example:

```
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 6.2) 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;
    }
}
```

```
1 using MathStaticExtension;
2 class TestMath{
3    public static function main() {
4       var ang = 1/2*Math.PI;
```

```
5     trace(ang.toDegrees()); //90
6   }
7 }
```

10.5 Lambda

10.6 Reflection

10.7 Serialization

10.8 Json

Haxe provides built-in support for (de-)serializing $JSON^2$ data via haxe. Json class.

10.8.1 Parsing JSON

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

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

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

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

10.8.2 Encoding JSON

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

²http://en.wikipedia.org/wiki/JSON

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

10.8.3 Implementation details

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 (10.2.4) 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 in the HTML page one of the JSON implementations available on the internet or use <code>-D old_browser</code> compiler argument that will make <code>haxe.Json</code> try to use native JSON and fallback to its own implementation in case it's not available.

10.9 Xml

10.10 Input/Output

10.11 Sys/sys

Part IV

Tools

- 10.12 Haxelib
- 10.13 Other Tools