# Haxe 3 Manual

Haxe Foundation October 10, 2014

# Todo list

Could we have a big Haxe logo in the First Manual Page (Introduction) under the menu (a bit like a book cover ?) It looks a bit empty now and is a landing page for ”Manual” . 7

This generates the following output: too many ’this’. You may like a passive sentence: the following output will be generated...though this is to be avoided, generally . . . . . . 9 make sure the types are right for inc, dec, negate, and bitwise negate . . . . . . . . . . . . . 13

While introducing the different operations, we should include that information as well, including how they differ with the ”C” standard, see http://haxe.org/manual/operators 13 please review, doubled content . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15 review please, sounds weird . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15 for starters...please review . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

Is there a difference between ?y : Int and y : Null*<*Int*>* or can you even do the latter? Some more explanation and examples with native optional and Haxe optional arguments and how they relate to nullability would be nice. . . . . . . . . . . . . . . . 17 please review future tense . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17 Same as in 2.2, what is Enum*<*T*>* syntax? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20 list arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20 please reformat . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22 I don’t really know how these work yet. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

It seems a bit convoluted explanations. Should we maybe start by ”decoding” the meaning of Void -¿ Void, then Int -¿ Bool -¿ Float, then maybe have samples using $type . . . . 24 please review your use of “this” and try to vary somewhat to avoid too much word repetition 31 please review for correctness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34 You have marked “Map” for some reason . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34 Mention toString()/String conversion somewhere in this chapter. . . . . . . . . . . . . . . . 46

”parent class” should probably be used here, but I have no idea what it means, so I will refrain from changing it myself. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47 Describe import a.\* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53 proper label and caption + code/identifier styling for diagram . . . . . . . . . . . . . . . . . 53 Figure out wtf our rules are now for when this is checked. . . . . . . . . . . . . . . . . . . . 86 Comprehensions are only listing Arrays, not Maps . . . . . . . . . . . . . . . . . . . . . . . . 87 what to use for listing of non-haxe code like hxml? . . . . . . . . . . . . . . . . . . . . . . . . 98 Check if we can build GADTs this way. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

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? . . . . . . . . . . . . . . . . . . . 124

# Contents

**Todo list 1**

1. **Introduction 7**
   1. What is Haxe? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2. About this Document . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.2.1 Authors and contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

* 1. Hello World . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  2. History . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

**I Language Reference 11**

1. **Types 12**
   1. Basic Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      1. Numeric types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      2. Overflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      3. Numeric Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      4. Bool . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
      5. Void . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   2. Nullability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
      1. Optional Arguments and Nullability . . . . . . . . . . . . . . . . . . . . . . . 17
   3. Class Instance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      1. Class constructor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
      2. Inheritance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
      3. Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
   4. Enum Instance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
      1. Enum Constructor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
      2. Using enums . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
   5. Anonymous Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      1. JSON for Structure Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
      2. Class Notation for Structure Types . . . . . . . . . . . . . . . . . . . . . . . . 24
      3. Optional Fields . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
      4. Impact on Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
   6. Function Type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
      1. Optional Arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
      2. Default values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
   7. Dynamic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
      1. Dynamic with Type Parameter . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      2. Implementing Dynamic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   8. Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
      1. Implicit Casts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
      2. Operator Overloading . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
      3. Array Access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
      4. Selective Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
      5. Enum abstracts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
      6. Forwarding abstract fields . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
      7. Core-type abstracts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
   9. Monomorph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
2. **Type System 39**
   1. Typedef . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
      1. Extensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
   2. Type Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
      1. Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
   3. Generic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
      1. Construction of generic type parameters . . . . . . . . . . . . . . . . . . . . 44
   4. Variance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
   5. Unification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
      1. Between Class/Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
      2. Structural Subtyping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
      3. Monomorphs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
      4. Function Return . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
      5. Common Base Type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   6. Type Inference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
      1. Top-down Inference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
      2. Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
   7. Modules and Paths . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
      1. Module Sub-Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
      2. Import . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
      3. Resolution Order . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
3. **Class Fields 56**
   1. Variable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
   2. Property . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
      1. Common accessor identifier combinations . . . . . . . . . . . . . . . . . . . 59
      2. Impact on the type system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
      3. Rules for getter and setter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
   3. Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
      1. Overriding Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
      2. Effects of variance and access modifiers . . . . . . . . . . . . . . . . . . . . . 64
   4. Access Modifier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
      1. Visibility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
      2. Inline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
      3. Dynamic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
      4. Override . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4. **Expressions 68**
   1. Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
   2. Constants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
   3. Binary Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
   4. Unary Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
   5. Array Declaration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
   6. Object Declaration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
   7. Field Access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
   8. Array Access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
   9. Function Call . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
   10. var . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
   11. Local functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
   12. new . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
   13. for . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
   14. while . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
   15. do-while . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
   16. if . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
   17. switch . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
   18. try/catch . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
   19. return . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
   20. break . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
   21. continue . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
   22. throw . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
   23. cast . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
       1. unsafe cast . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
       2. safe cast . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
   24. type check . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5. **Language Features 78**
   1. Conditional Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
   2. Externs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
   3. Static Extension . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
      1. In the Haxe Standard Library . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
   4. Pattern Matching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
      1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
      2. Enum matching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
      3. Variable capture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
      4. Structure matching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
      5. Array matching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
      6. Or patterns . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
      7. Guards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
      8. Match on multiple values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
      9. Extractors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
      10. Exhaustiveness checks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
      11. Useless pattern checks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
   5. String Interpolation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
   6. Array Comprehension . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
   7. Iterators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
   8. Function Bindings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
   9. Metadata . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
   10. Access Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
   11. Inline constructors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
   12. Remoting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

**II Compiler Reference 95**

1. **Compiler Usage 96**
2. **Compiler Features 97**
   1. Compiler Metadata . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
   2. Dead Code Elimination . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
   3. Completion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
      1. Field Access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
      2. Call Arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
      3. Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
      4. Position . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
      5. Toplevel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
   4. Resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
      1. Embedding resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
      2. Retrieving text resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
      3. Retrieving binary resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
      4. Implementation details . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
   5. Runtime Type Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
      1. RTTI structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
3. **Macros 102**
   1. Macro Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
   2. Arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
      1. ExprOf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
      2. Constant Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
      3. Rest Argument . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
   3. Reification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
      1. Expression Reification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
      2. Type Reification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
      3. Class Reification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
   4. Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
   5. Type Building . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
      1. Enum building . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
      2. @:autoBuild . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
   6. Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
      1. Macro-in-Macro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
      2. Static extension . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
      3. Build Order . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
      4. Type Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
   7. Initialization macros . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

**III Standard Library 114**

1. **Standard Library 115**
   1. String . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
   2. Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
      1. Array . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
      2. Vector . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
      3. List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
      4. GenericStack . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
      5. Map . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
      6. Option . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
   3. Regular Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
      1. Matching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
      2. Groups . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
      3. Replace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
      4. Split . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
      5. Map . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
      6. Implementation Details . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
   4. Math . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
      1. Special Numbers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
      2. Mathematical Errors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
      3. Integer Math . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
      4. Extensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
   5. Lambda . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
   6. Reflection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
   7. Serialization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
      1. Serialization format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
   8. Json . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
      1. Parsing JSON . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
      2. Encoding JSON . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
      3. Implementation details . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
   9. Xml . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

10.10Input/Output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

10.11Sys/sys . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

**IV Miscellaneous 132**

1. **Haxelib 133**
   1. Using a Haxe library with the Haxe Compiler . . . . . . . . . . . . . . . . . . . . . . 133
   2. haxelib.json . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
      1. Versioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
      2. Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
   3. extraParams.hxml . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
   4. Using Haxelib . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
2. **Target Details 137**
   1. Javascript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
      1. Loading extern classes using ”require” function . . . . . . . . . . . . . . . . 137
   2. Flash . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
   3. Neko . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
   4. PHP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
   5. C++ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
      1. Using C++ Defines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
      2. Using C++ Pointers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
   6. Java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
   7. C# . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140

Could we have a big

**Chapter 1**

# Introduction

**1.1 What is Haxe?**

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

Haxe consists of a high-level, open source programming language and a compiler. It allows compilation of programs, written using an ECMAScript[[1]](#footnote-1)-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 run-time in the target language. Furthermore, type information can be used by the target generators to generate optimized and robust code.

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

|  |  |  |
| --- | --- | --- |
| Name | Output type | Main usages |
| Javascript | Sourcecode | Browser, Desktop, Mobile, Server |
| Neko | Bytecode | Desktop, Server |
| PHP | Sourcecode | Server |
| Python | Sourcecode | Desktop, Server |
| C++ | Sourcecode | Desktop, Mobile, Server |
| Actionscript 3 | Sourcecode | Browser, Desktop, Mobile |
| Flash | Bytecode | Browser, Desktop, Mobile |
| 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*.

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

We continue with the Haxe compiler reference, which first handles the basics in Compiler Usage (Chapter 7) before getting into the advanced features in Compiler Features (Chapter 8). Finally, we will venture into the exciting land of *haxe macros* in Macros (Chapter 9) to see how some common tasks can be greatly simplified.

In the following chapter, Standard Library (Chapter 10), we explore important types and concepts from the Haxe Standard Library. We then learn about Haxe’s package manager Haxelib in Haxelib (Chapter 11).

Haxe abstracts away many target differences, but sometimes it is important to interact with a target directly, which is the subject of Target Details (Chapter 12).

## 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 |

1

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. Predominantly, this is done when introducing a new type or when a term is specific to Haxe. We do not define every new aspect we introduce, e.g. what a class is, to avoid cluttering 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.

### 1.2.1 Authors and contributions

Most of this document’s content was written by Simon Krajewski while working for the Haxe Foundation. We would like to thank these people for their contributions:

* Dan Korostelev: Additional content and editing
* Caleb Harper: Additional content and editing
* Josefiene Pertosa: Editing
* Miha Lunar: Editing
* Nicolas Cannasse: Haxe creator

## 1.3 Hello World

The following program prints “Hello World” after being compiled and run:

|  |
| --- |
| class HelloWorld { static public function main():Void { trace("Hello World");  }  } |

1

2

3

4

5

This can be tested by saving the above code to a file named HelloWorld.hx and invoking the

Haxe Compiler like so: haxe -main HelloWorld --interp. It then generates the following This generates the fol- output: HelloWorld.hx:3: Hello world. There are several things to learn from this:

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

* 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, lowercase).

## 1.4 History

The Haxe project was started on 22 October 2005 by French developer *Nicolas Cannasse* as a successor to the popular open-source ActionScript 2 compiler *MTASC* (Motion-Twin Action Script Compiler) and the in-house *MTypes* language, which experimented with the application of type inference to an object oriented language. Nicolas’ long-time passion for programming language design and the rise of new opportunies to mix different technologies as part of his game developer work at *Motion-Twin* led to the creation of a whole new language.

Being spelled *haXe* back then, its beta version was released in February 2006 with the first supported targets being AVM[[2]](#footnote-2)-bytecode and Nicolas’ own *Neko* virtual machine[[3]](#footnote-3).

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 AVM2 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 resulted in several minor bug-fix releases.

Haxe 2.0 was released in July 2008, including 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, *Caueˆ 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*[[4]](#footnote-4), focused on this next major version. Haxe 3 was subsequently released in May 2013.

**Part I**

# Language Reference

**Chapter 2**

# Types

The Haxe Compiler employs a rich type 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++ |

1

2

3

4

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

|  |
| --- |
| var myButton = new MySpecialButton(); // Haxe |

1

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.

The Haxe type system knows 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 with any type

***Abstract*:** a compile-time type which is represented by a different type at runtime

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

We will describe each of these type groups and how they relate to each other in the next chapters.

## 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

*Basic types* are Bool, Float and Int. They can easily be identified in the syntax by values such as

* true and false for Bool, • 1, 0, -1 and 0xFF0000 for Int and
* 1.0, 0.0, -1.0, 1e10 for Float.

Basic types are not classes (2.3) in Haxe. They are implemented as abstract types (2.8) and are tied to the compiler’s internal operator-handling as described in the following sections.

### 2.1.1 Numeric types

**Type: Float**

Represents a double-precision IEEE 64-bit 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 allowed implicitly.

### 2.1.2 Overflow

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, Flash:** 32-bit signed integers with usual overflow practices

**PHP, JS, Flash 8:** No native *Int* type, loss of precision will occur if they reach their float limit (252)

Alternatively, the *haxe.Int32* and *haxe.Int64* classes can be used to ensure correct overflow behavior regardless of the platform at the cost of additional computations depending on the platform.

### 2.1.3 Numeric Operators

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

While introducing the different operations, we should include that information as well, including how they differ with the ”C” standard, see

http://haxe.org/manual/operators

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 | Float |
|  |  | 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 |
| % | modulo | Float | Float | Float |
|  |  | Float | Int | Float |
|  |  | Int | Float | Float |
|  |  | Int | Int | Int |
|  | Comparison | |  |  |
| Operator | Operation | Argument 1 | Argument 2 | Return |
| == | 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 | 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 |
| ˆ | 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 are a common occurence in *conditions* such as if (5.16) and while (5.14). 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 run-time. 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 || 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 cases such as this:

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

1

2

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

### 2.1.5 Void

#### Type: Void

Void denotes 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

|  |
| --- |
| class HelloWorld { static public function main():Void { trace("Hello World");  }  } |

please review, doubled already “seen” Void in the initial “Hello World” example: content

1

2

3

4

5

The function type will be explored in detail in the section Function Type (Section 2.6) but a quick preview helps here: The type of the function main in the example above is Void->Void, which reads as “it has no arguments and returns nothing”. Haxe does not allow fields and

|  |
| --- |
| // Arguments and variables of type Void  // are not allowed var x:Void; |

review please, sounds variables of type Void and will complain if an attempt at declaring such is made: weird

1

2

3

## 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 due to 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, C++, Java and C# targets.

### Definition: Dynamic target

Dynamic targets are more lenient with their types and allow null values for basic types. This applies to the JavaScript, PHP, Neko and Flash 6-8 targets.

There is nothing to worry about when working with null on dynamic targets; however, static

for starters...please re- ones may require some thought. For starters, basic types are initialized to their default values. view

**Definition: Default values**

Basic types have the following default values on static targets:

### Int: 0

**Float:** NaN on Flash, 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*>*:

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

1

2

3

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

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

1

2

3

4

5

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

### Type: Null<T>

On static targets the types Null<Int>, Null<Float> and Null<Bool> can be used to allow null as a value. On dynamic targets this has no effect. Null<T> can also be used with other types in order to document that null is an allowed value.

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 |

1

2

3

### 2.2.1 Optional Arguments and Nullability

Optional arguments also have to be accounted for when considering nullability.

In particular, there must be a distinction between *native* optional arguments which are not nullable and Haxe-specific optional arguments which might be. The distinction is made by using the question-mark optional argument:

1. // x is a native Int (not nullable)
2. function foo(x : Int = 0) {...}
3. // y is Null<Int> (nullable)
4. function bar( ?y : Int) {...}
5. // z is also Null<Int>
6. function opt( ?z : Int = -1) {...}

Is there a difference

between ?y: Int and

#### y: Null*<*Int*>* or Trivia: Argument vs. Parameter

can you even do the In some other programming languages, *argument* and *parameter* are used interchangeably. In latter? Some more Haxe, *argument* is used when referring to methods and *parameter* refers to Type Parameters explanation and ex- (Section 3.2).

amples with native optional and Haxe optional arguments and

how they relate to nul- **2.3 Class Instance** lability would be nice.

Similar to many object-oriented languages, classes are the primary data structure for the majority of programs in Haxe. 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

please review future leaving the details of class fields for Class Fields (Chapter 4). tense The following code example serves as basis for the remainder of this section:

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

1

2

3

4

5

6

7

8

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 (5).
* 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 which means that all classes (not their instances) can be assigned to it. At compile-time, Class<T> is the common base type of all class types. However, this relation is not reflected in generated code.

This type is useful when an API requires a value to be *a* class, but not a specific one. This applies to several methods of the Haxe reflection API (10.6).

### 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*. Nevertheless, we prefer the term class instance to emphasize the analogy between classes/class instances and enums/enum instances

(2.4).

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

1

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 the section 5.12. For now, we just think of 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:

|  |
| --- |
| class Point3 extends Point { var z : Int; public function new(x,y,z) { super(x,y); this.z = z;  }  } |

1

2

3

4

5

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 parent 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 pure structural information:

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

1

2

3

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

* interface keyword is used instead of class keyword
* functions do not have any expressions (5)
* every field must have an explicit type

Interfaces, unlike structural subtyping (3.5.2), describe a *static relation* between classes. A given class is only considered to be compatible to an interface if it explicitly states so:

|  |
| --- |
| class Point implements Printable { } |

1

Here, the implements keyword 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:

|  |
| --- |
| class Point implements Printable implements Serializable |

1

2

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:

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

1

2

3

4

5

6

7

8

9

10

#### 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)[[5]](#footnote-5). While they cannot have any expressions (5), they are very useful for describing data structures:

|  |
| --- |
| enum Color {  Red;  Green;  Blue;  Rgb(r:Int, g:Int, b:Int);  } |

1

2

3

4

5

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 (5).
* 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*<*T*>*

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

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

### 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 Color. Variable c is initialized using the Rgb All enum instances can be assigned to a special type named EnumValue.

constructor with arguments.

list arguments

#### Type: EnumValue

EnumValue is a special type which unifies with all enum instances. It is used by the Haxe Standard Library to provide certain operations for all enum instances and can be employed in user-code accordingly in cases where an API requires *an* enum instance, but not a specific one.

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

|  |
| --- |
| enum Color {  Red;  Green;  Blue;  Rgb(r:Int, g:Int, b:Int);  }  class Main { static public function main() {  var ec:EnumValue = Red; // valid var en:Enum<Color> = Color; // valid  // Error: Color should be Enum<Color>  //var x:Enum<Color> = Red;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

If the commented line 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.

#### Trivia: Concrete type parameter for Enum*<*T*>*

One of the reviewers of this manual was confused about the difference between Color and Enum<Color> in the example above. Indeed, using a concrete type parameter there is pointless and only serves the purpose of demonstration. Usually we would omit the type there and let type inference (3.6) deal with it.

However, the inferred type would be different from Enum<Color>. The compiler infers a pseudo-type which has the enum constructors as “fields”. As of Haxe 3.2.0, it is not possible to express this type in syntax but also, it is never necessary to do so.

### 2.4.2 Using enums

Enums are a good choice if only a finite set of values should be allowed. The individual constructors (2.4.1) then represent the allowed variants and enable the compiler to check if all possible values are respected. This can be seen here:

|  |
| --- |
| enum Color {  Red;  Green;  Blue;  Rgb(r:Int, g:Int, b:Int);  } |

1

2

3

4

5

6

7

|  |
| --- |
| class Main { static function main() { var color = getColor(); switch (color) { case Red: trace("Color was red"); case Green: trace("Color was green"); case Blue: trace("Color was blue"); case Rgb(r, g, b):  trace("Color had a red value of " +r); } }  static function getColor():Color { return Rgb(255, 0, 255);  }  } |

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

After retrieving the value of color by assigning the return value of getColor() to it, a switch expression (5.17) is used to branch depending on the value. The first three cases Red, Green and Blue are trivial and correspond to the constructors of Color that have no arguments. The final case Rgb(r, g, b) shows how the argument values of a constructor can be extracted: they are available as local variables within the case body expression, just as if a var expression (5.10) had been used.

Advanced information on using the switch expression will be explored later in the section on pattern matching (6.4).

## 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:

|  |
| --- |
| class Structure { static public function main() {  var myStructure = { x: 12, name: "foo"}; }  } |

1

2

3

4

5

The general syntactic rules follow:

1. A structure is enclosed in curly braces {} and
2. Has a *comma-separated* list of key-value-pairs.
3. A *colon* separates the key, which must be a valid identifier (5), from the value.
4. The value can be any Haxe expression.

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

please reformat

|  |
| --- |
| var user = { name : "Nicolas",  age : 32, |

1

2

3

|  |
| --- |
| pos : [  { x : 0, y : 0 },  { x : 1, y : -1 }  ],  }; |

4

5

6

7

8

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

|  |
| --- |
| // get value of name, which is "Nicolas" user.name;  // set value of age to 33 user.age = 33; |

1

2

3

4

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:

|  |
| --- |
| class Test { static public function main() {  var point = { x: 0.0, y: 12.0 }; // { y : Float, x : Float } has no field z point.z;  }  } |

1

2

3

4

5

6

7

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:

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

1

2

3

4

5

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

|  |
| --- |
| typedef Point = { x : Int, y : Int }  class Path { var start : Point; var target : Point; var current : Point;  } |

1

2

3

4

5

6

7

### 2.5.1 JSON for Structure Values

It is also possible to use *JavaScript Object Notation* for structures by using *string literals* for the keys:

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

1

While any string literal is allowed, the field is only considered part of the type if it is a valid Haxe identifier (5). 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, Haxe 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 Type

It seems a bit convo-

luted explanations.

Should we maybe

start by ”decoding”

the meaning of Void -¿

Void, then Int -¿ Bool

-

¿ Float, then maybe

have samples using

$type

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

{

2

static

public

function

main() {

3

//

i

:

Int

->

s

:

String

->

Bool

4

$type(test);

5

$type(test(1,

"foo"

;

))

//

Bool

6

}

7

8

static

function

test(i:Int,

s:String):Bool

{

9

return

true

;

10

}

11

}

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, such as test(1, "foo"), is made 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.

If a function type has other function types as argument or return type, parentheses can be used to group them correctly. For example, Int -> (Int -> Void) -> Void represents a function which has a first argument of type Int, a second argument of function type Int -> Void and a return of Void.

### 2.6.1 Optional Arguments

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

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

Function test has two optional arguments: i of type Int and s of String. This is directly reflected in the function type output by line 3. 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:

|  |
| --- |
| class DefaultValues { static public function main() {  // ?i : Int -> ?s : String -> String  $type(test);  trace(test()); // i: 12, s: bar trace(test(1)); // i: 1, s: bar trace(test(1, "foo")); // i: 1, s: foo |

1

2

3

4

5

6

7

|  |
| --- |
| trace(test("foo")); // i: 12, s: foo  }  static function test(i = 12, s = "bar") {  return "i: " +i + ", s: " +s;  }  } |

8

9

10

11

12

13

14

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 the default values are used instead of null 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 can be considered as a more typical approach. On some targets the compiler may still pass null for omitted argument values and generate code similar to this into the function:

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

1

2

3

4

5

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, in effect, 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 8.2) 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; |

1

2

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). Monomorphs are never bound to Dynamic which can have surprising results in examples such as this:

|  |
| --- |
| class Main { static function main() {  var jsonData = ’[1, 2, 3]’; var json = haxe.Json.parse(jsonData);  $type(json); // Unknown<0> for (i in 0...json.length) {  // Array access is not allowed on  // {+ length : Int }  trace(json[0]);  }  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

Although the return type of Json.parse is Dynamic, the type of local variable json is not bound to it and remains a monomorph. It is then inferred as an anonymous structure (2.5) upon the json.length field access, which causes the following json[0] array access to fail. In order to avoid this, the variable json can be explicitly typed as Dynamic by using var json:Dynamic.

### 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;  // valid, value is a String att.name = "Nicolas";  // dito (this documentation is quite old) att.age = "26";  // error, value is not a String att.income = 0; |

1

2

3

4

5

6

7

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

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

Classes that implement Dynamic (with or without type parameter) can also utilize a special method named resolve. If a read access (4.2) is made and the field in question does not exist, the resolve method is called with the field name as argument:

|  |
| --- |
| class Resolve implements Dynamic<String> { public var present:Int; public function new() {} function resolve(field:String) {  return "Tried to resolve " +field; }  }  class Main { static public function main() {  var c = new Resolve(); c.present = 2; trace(c.present); trace(c.resolveMe);  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

## 2.8 Abstract

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

|  |
| --- |
| abstract AbstractInt(Int) { |

1

|  |
| --- |
| inline public function new(i:Int) {  this = i;  }  } |

2

3

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 are a constructor function new 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  }  } |

1

2

3

4

5

6

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); |

1

2

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 the section Inline (Section 4.4.2) - and its inlined expression assigns a value to this. This might be surprising when thinking in terms of classes. However, 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 technically this 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 { inline function new(i:Int) {  this = i;  } }  class ImplicitCastDirect { static public function main() {  var a:MyAbstract = 12; var b:Int = a;  }  } |

1

2

3

4

5

6

7

8

9

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 Int12 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 semantics, but are defined completely differently:

|  |
| --- |
| abstract MyAbstract(Int) { inline function new(i:Int) {  this = i;  }  @:from static public function fromString(s:String) { return new MyAbstract(Std.parseInt(s));  }  @:to public function toArray() { return [this]; |

1

2

3

4

5

6

7

8

9 10

11

12

13

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

14

15

16

17

18

19

20

21

22

23

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 the cast-functions are inserted where required. This becomes obvious when looking at the Javascript output:

|  |
| --- |
| var a = \_ImplicitCastField.MyAbstract\_Impl\_  .fromString("3"); var b = \_ImplicitCastField.MyAbstract\_Impl\_  .toArray(a); |

1

2

3

4

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

please review your following: use of “this” and try var a = Std.parseInt("3");

1

to vary somewhat to var b = [a];

2 avoid too much word

repetition 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 6.
5. Stop, unification fails.
6. Stop, unification succeeds.

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

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

1

2

3

4

5

6

A

is abstract

A

defines

to

for

B

B

is abstract

B

defines

from

for

A

Unification fails

No

Yes

No

Yes

Unification succeeds

No

No

Yes

Yes

Figure 2.1: Selection algorithm flow chart.

|  |
| --- |
| public function new() this = 0;  @:to public function toC() return new C();  }  abstract C(Int) {  public function new() this = 0;  }  class Main { static public function main() {  var a = new A(); var b:B = a; // valid, uses A.toB var c:C = b; // valid, uses B.toC var c:C = a; // error, A should be C  }  } |

7

8

9

10

11

12

13

14

15

16

17

18

19

20

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:

|  |
| --- |
| abstract MyAbstract(String) { public inline function new(s:String) {  this = s;  } |

1

2

3

4

|  |
| --- |
| @:op(A \* B) public function repeat(rhs:Int):MyAbstract { var s:StringBuf = new StringBuf(); for (i in 0...rhs)  s.add(this);  return new MyAbstract(s.toString()); } }  class AbstractOperatorOverload { static public function main() { var a = new MyAbstract("foo"); trace(a \* 3); // foofoofoo  }  } |

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

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:

|  |
| --- |
| console.log(\_AbstractOperatorOverload.  MyAbstract\_Impl\_.repeat(a,3)); |

1

2

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

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:

|  |
| --- |
| abstract MyAbstract(String) { public inline function new(s:String) {  this = s;  }  @:op(++A) public function pre() return "pre" + this;  @:op(A++) public function post() return this + "post";  }  class AbstractUnopOverload { static public function main() { var a = new MyAbstract("foo"); trace(++a); // prefoo trace(a++); // foopost  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

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

please review for cor- the abstract. rectness

### 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. Nevertheless, 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:

You have marked

“Map” for some rea-

son

1

@:arrayAccess

public

inline

function

2

get(key:K)

return

this

.get(key);

3

@:arrayAccess

public

inline

function

4

arrayWrite(k:K,

v:V):V

{

5

this

.set(k,

v);

6

return

v;

7

}

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:

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

1

2

3

4

5

6

7

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

|  |
| --- |
| map.set("foo",1); 1; console.log(map.get("foo")); |

1

2

3

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

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

1

2

3

4

5

6

7

8

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

9

10

11

12

13

14

15

16

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

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

### 2.8.4 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 (6.3), where the type of the first function argument determines for which types a function is defined:

|  |
| --- |
| abstract MyAbstract<T>(T) from T {  public function new(t:T) this = t;  function get() return this;  static public function getString(v:MyAbstract<String>):String { return v.get();  } }  class SelectiveFunction { static public function main() { var a = new MyAbstract("foo"); a.getString(); var b = new MyAbstract(1);  // Int should be MyAbstract<String> b.getString();  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

The method getString of abstract MyAbstract is defined to accept a first argument of MyAbstract*<*Strin 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

Rather than having actually been designed, selective functions were 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:

|  |
| --- |
| @:enum abstract HttpStatus(Int) {  var NotFound = 404; var MethodNotAllowed = 405;  }  class Main { static public function main() { var status = HttpStatus.NotFound; var msg = printStatus(status);  }  static function printStatus(status:HttpStatus) { return switch(status) { case NotFound:  "Not found"; case MethodNotAllowed:  "Method not allowed";  }  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

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

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

1

2

3

4

5

6

7

8

9

10

11

12

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.
* The pattern matcher checks for exhaustiveness (6.4.10) when matching (6.4) an enum abstract.
* Defining fields requires less syntax.

### 2.8.6 Forwarding abstract fields

#### 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 :forward metadata to an abstract type:

|  |
| --- |
| @:forward(push, pop) abstract MyArray<S>(Array<S>) { public inline function new() {  this = [];  }  }  class Main { static public function main() {  var myArray = new MyArray(); myArray.push(12); myArray.pop();  // MyArray<Int> has no field length  //myArray.length;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

The MyArray abstract in this example wraps Array. Its :forward metadata has two arguments which correspond to the field names to be forwarded to the underlying type. In this example, the main method instantiates MyArray and accesses its push and pop methods. The commented line demonstrates that the length field is not available.

As usual we can look at the Javascript output to see how the code is being generated:

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

1

2

3

4

5

It is also possible to use :forward without any arguments in order to forward all fields. Of course the Haxe Compiler still ensures that the field actually exists on the underlying type.

#### Trivia: Implemented as macro

Both the :enum and :forward functionality were originally implemented using build macros (9.5). While this worked nicely in non-macro code, it caused issues if these features were used from within macros. The implementation was subsequently moved to the compiler.

### 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. Still, that type is native to the Haxe target.

Introducing custom core-type abstracts is rarely necessary in user code as it requires the 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 the type groups above are compatible. Meaning, 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 (2.5) by giving it a name. This is precisely what typedefs are good for. 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:

|  |
| --- |
| typedef IA = Array<Int>; |

1

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 much bigger difference for more complex, compound types. Again, this is why typedef and structures seem so connected:

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

1

2

3

4

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 Haxe Standard Library demonstrates:

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

1

2

3

### 3.1.1 Extensions

Extensions are used to express that a structure has all the fields of a given type as well as some additional fields of its own:

|  |
| --- |
| typedef IterableWithLength<T> = {  > Iterable<T>, // read only property var length(default, null):Int;  }  class Extension { static public function main() {  var array = [1, 2, 3]; var t:IterableWithLength<Int> = array; }  } |

1

2

3

4

5

6

7

8

9

10

11

12

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. **Since Haxe 3.1.0**

It is also possible to extend multiple structures:

|  |
| --- |
| typedef WithLength = { var length(default, null):Int;  }  typedef IterableWithLengthAndPush<T> = {  > Iterable<T>, > WithLength, function push(a:T):Int;  }  class Extension2 { static public function main() {  var array = [1, 2, 3];  var t:IterableWithLengthAndPush<Int> = array;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

## 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 Haxe Standard Library is Array:

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

1

2

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 **explicitly** by invoking the constructor with explicit types (new Array*<*String*>*()) 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 (5.23) to 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 local lvalue functions. |
| 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. This can be utilized if a function has multiple arguments:

|  |
| --- |
| class FunctionTypeParameter { static public function main() { equals(1, 1);  // runtime message: bar should be foo equals("foo", "bar");  // compiler error: String should be Int equals(1, "foo");  }  static function equals<T>(expected:T, actual:T) {  if (actual != expected) { trace(’$actual should be $expected’); }  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

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>(x). The error messages the compiler gives are not very helpful. However, there is a simple reason for that: The above code is parsed as if both < and > were binary operators, yielding (method < String) > (x).

### 3.2.1 Constraints

Type parameters can be constrained to multiple types:

|  |
| --- |
| typedef Measurable = { public var length(default, null):Int;  }  class Constraints { static public function main() { trace(test([])); trace(test(["bar", "foo"])); // String should be Iterable<String>  //test("foo"); }  static function test<T:(Iterable<String>, Measurable)>(a:T) {  if (a.length == 0) return "empty"; return a.iterator().next();  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

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 (6.9). 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 class MyArray<T> {  public function new() { }  }  class Main { static public function main() { var a = new MyArray<String>(); var b = new MyArray<Int>();  }  } |

1

2

3

4

5

6

7

8

9

10

11

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

|  |
| --- |
| (function () { "use strict"; var Main = function() { }  Main.main = function() { var a = new MyArray\_String(); var b = new MyArray\_Int();  } var MyArray\_Int = function() {  }; var MyArray\_String = function() {  };  Main.main();  })(); |

1

2

3

4

5

6

7

8

9

10

11

12

We can identify that MyArray<String> and MyArray<Int> have become MyArray\_String and MyArray\_Int respectively. This is similar for generic functions:

|  |
| --- |
| class Main { static public function main() { method("foo"); method(1);  }  @:generic static function method<T>(t:T) { }  } |

1

2

3

4

5

6

7

8

Again, the Javascript output makes it obvious:

|  |
| --- |
| (function () { "use strict"; var Main = function() { }  Main.method\_Int = function(t) {  }  Main.method\_String = function(t) {  }  Main.main = function() {  Main.method\_String("foo");  Main.method\_Int(1); |

1

2

3

4

5

6

7

8

9

|  |
| --- |
| }  Main.main();  })(); |

10

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.

It is not possible to construct normal type parameters, e.g. new T() is a compiler error. The reason for this is that Haxe generates only a single function and the construct makes no sense in that case. 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 Tnew T() with the real type.

|  |
| --- |
| typedef Constructible = { public function new(s:String):Void;  }  class Main { static public function main() { var s:String = make(); var t:haxe.Template = make();  }  @:generic static function make<T:Constructible>():T { return new T("foo");  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

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() { }  Main.\_\_name\_\_ = true;  Main.make\_haxe\_Template = function() { return new haxe.Template("foo");  }  Main.make\_String = function() { return new String("foo"); |

1

2

3

4

5

6

7

|  |
| --- |
| }  Main.main = function() { var s = Main.make\_String(); var t = Main.make\_haxe\_Template();  } |

8

9

10

11

12

## 3.4 Variance

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

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

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:

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

Here we subvert the type checker by using a cast (5.23), 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. In Haxe, we can achieve this by restricting the type accordingly using structural subtyping (3.5.2):

|  |
| --- |
| class Base {  public function new() { }  }  class Child extends Base { }  typedef MyArray<T> = { public function pop():T;  }  class Main {  public static function main () {  var a = [new Child()]; var b:MyArray<Base> = a;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

We can safely assign with b being typed as MyArray<Base> and MyArray only having a pop() method. 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 con- Unification is the heart of the type system and contributes immensely to the robustness of version somewhere in Haxe programs. It describes the process of checking if a type is compatible to another type. this chapter.

### 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:

|  |
| --- |
| class Main { static public function main() {  // Int should be String var s:String = 1;  }  } |

1

2

3

4

5

6

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 e expression, the type of e is unified with the function return type. If the function has no explicit return type, it is inferred to the type of e and subsequent return expressions are inferred 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 the 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 parent class) but the reverse is not valid.

The following assignments are allowed:

* child class to parent 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 ”parent class” should and so on.

probably be used

### here, but I have no 3.5.2 Structural Subtyping

idea what it means, so I will refrain from changing it myself.

#### 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();  } |

1

2

3

4

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 requires 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 cannot 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:

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

1

The right-hand function clearly is of type Void->String, yet we can assign it to the variable func of type Void->Void. 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:

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

1

2

3

4

5

6

7

8

9 10

11

|  |
| --- |
| }  } |

12

13

Although Base is not mentioned, the Haxe Compiler manages to infer it as the common type of Child1 and Child2. 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  }  } |

1

2

3

4

5

6

7

8

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

### 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>  }  } |

1

2

3

4

5

6

7

8

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:

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

1

2

3

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 = { public function new(s:String):Void;  }  class Main { static public function main() { var s:String = make(); var t:haxe.Template = make();  }  @:generic static function make<T:Constructible>():T { return new T("foo");  }  } |

1

2

3

4

5

6

7

8

9 10

11

12

13

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.

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

## 3.7 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.StringMa 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<In 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.pack2.packN.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; |

1

Here the sub-type ExprDef within module haxe.macro.Expr is accessed.

The sub-type relation is not reflected at run-time. That is, public sub-types become a member of their containing package, which could lead to conflicts if two modules within the same package tried to define the same sub-type. Naturally, the Haxe compiler detects these cases and reports them accordingly. In the example above ExprDef is generated as haxe.macro.ExprDef. Sub-types can also be made private:

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

1

2

3

4

#### Definition: Private type

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

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

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

### 3.7.2 Import

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

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

1

2

3

4

5

6

7

8

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

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

|  |
| --- |
| import haxe.macro.Expr;  class Main { static public function main() {  var e:Binop = OpAdd;  }  } |

1

2

3

4

5

6

7

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

There are several aspects worth knowing about importing:

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

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

|  |
| --- |
| import Math.random;  class Main { static public function main() { random();  }  } |

1

2

3

4

5

6

7

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

proper label and caption + code/identifier Given an identifier i, the algorithm is as follows:

styling for diagram

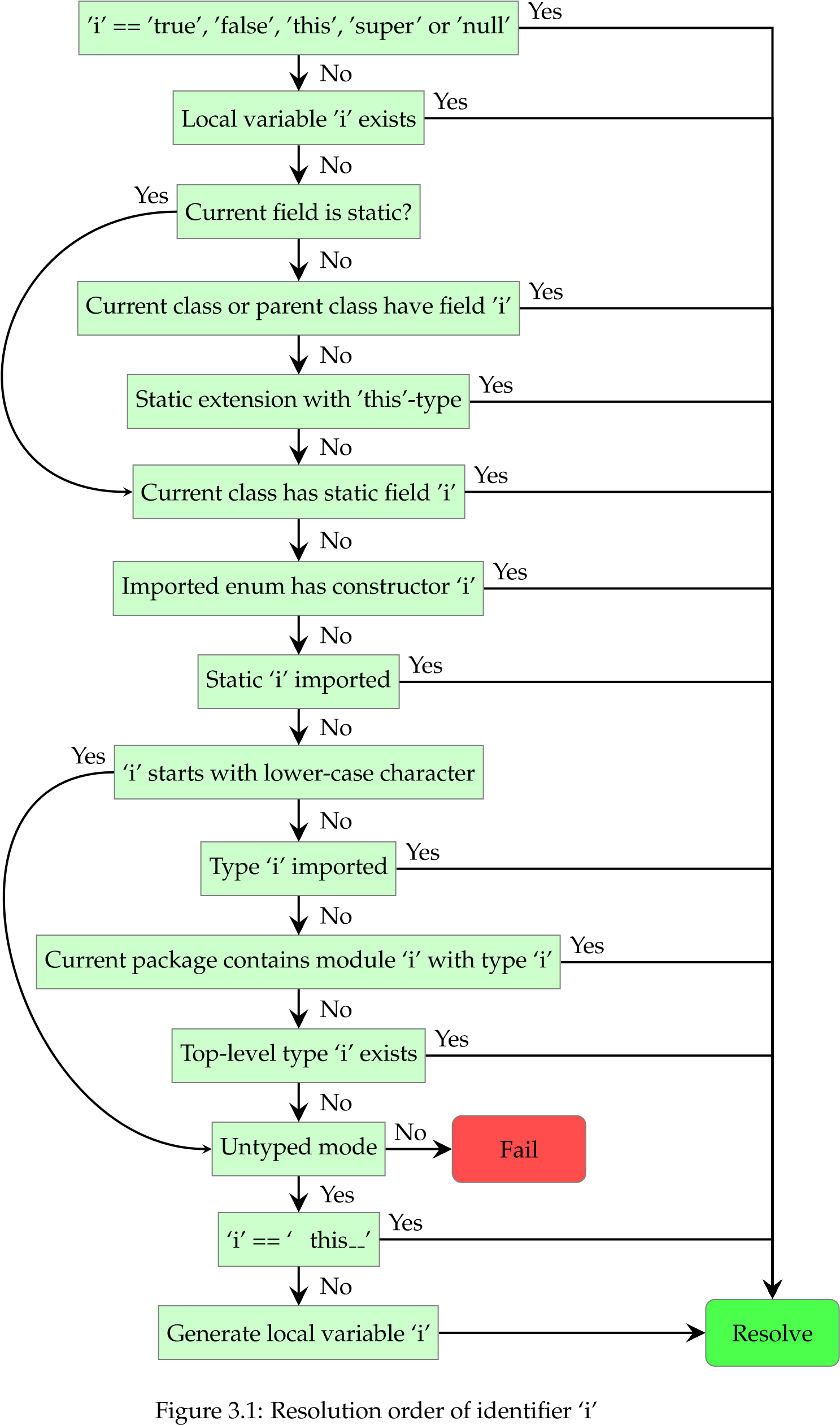
1. If i is true, false, this, super or null, resolve to the matching constant and halt.
2. If a local variable named i is accessible, resolve to it and halt.
3. If the current field is static, go to 6.
4. If the current class or any of its parent classes has a field named i, resolve to it and halt.
5. 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.
12. Generate a local variable named i, resolve to it and halt.
13. Fail

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

1. If a type named i is imported (directly or as part of a module), resolve to it and halt.
2. If the current package contains a module named i with a type named i, resolve to it and halt.
3. If a type named i is available at top-level, resolve to it and halt.
4. Fail

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

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



**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 nonstatic. Non-static fields are referred to as *member* fields, so we speak of e.g. a *static method* or a *member variable*.

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

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

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

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

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

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

## 4.1 Variable

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

|  |
| --- |
| class VariableField { static var member:String = "bar";  public static function main() { trace(member); member = "foo"; trace(member);  }  } |

1

2

3

4

5

6

7

8

9

We can learn from this that a variable

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

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

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

inline

extern

None

static

Anything

No ’this’

No

Yes

No

Yes

static

Constants only

Invalid

No

Yes

Yes

No

Figure 4.1: Initialization values of a variable field.

## 4.2 Property

Next to variables (4.1), properties are the second option for dealing with data on a class. Unlike variables however, they offer more control of which kind of field access should be allowed and how it should be generated. Common use cases include:

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

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

### Definition: Read Access

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

### Definition: Write Access

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

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

|  |
| --- |
| class Main { public var x(default, null):Int; static public function main() { }  } |

1

2

3

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 {  // read from outside, write only within Main public var ro(default, null):Int;  // write from outside, read only within Main public var wo(null, default):Int;  // access through getter get\_x and setter  // set\_x public var x(get, set):Int;  // read access through getter, no write  // access public var y(get, never):Int;  // required by field x function get\_x() return 1;  // required by field x function set\_x(x) return x;  // required by field y function get\_y() return 1;  function new() {  var v = x; x = 2; x += 1;  }  static public function main() { new Main();  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

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

|  |
| --- |
| var Main = function() { var v = this.get\_x(); this.set\_x(2); var \_g = this;  \_g.set\_x(\_g.get\_x() + 1);  }; |

1

2

3

4

5

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:

|  |
| --- |
| class Main { public var x(get, set):Int; function get\_x() return 1; function set\_x(x) return x;  public function new() { }  static public function main() {  new Main().x += 1;  }  } |

1

2

3

4

5

6

7

8

9

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:

|  |
| --- |
| Main.main = function() { var \_g = new Main();  \_g.set\_x(\_g.get\_x() + 1);  } |

1

2

3

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:

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

1

2

3

4

5

6

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

|  |
| --- |
| class Base { public function get\_x() return 1;  } |

1

2

3

|  |
| --- |
| class Main extends Base {  // ok, get\_x is declared by parent class public var x(get, null):Int;  static public function main() {}  } |

4

5

6

7

8

9

10

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

existence

### 4.2.3 Rules for getter and setter

Visibility of accessor methods has no effect on the accessibility of its property. That is, if a property is public and defined to have a getter, that getter may 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() {}  } |

1

2

3

4

5

6

7

8

9

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

|  |
| --- |
| **Definition: Physical field**  A field is considered to be *physical* if it is either   * a variable (4.1) * a property (4.2) with the read-access or write-access identifier being default or null * a property (4.2) with :isVar metadata (6.9) |

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

|  |
| --- |
| class Main {  // This field cannot be accessed because it  // is not a real variable public var x(get, set):Int;  function get\_x() { return x;  } |

1

2

3

4

5

6

7

8

9

|  |
| --- |
| function set\_x(x) {  return this.x = x;  }  static public function main() {}  } |

10

11

12

13

14

15

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

|  |
| --- |
| class Main {  // @isVar forces the field to be physical // allowing the program to compile. @:isVar public var x(get, set):Int;  function get\_x() { return x;  }  function set\_x(x) {  return this.x = x;  }  static public function main() {}  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

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 rightside expressions. Given a chain like x = y = 1, it is evaluated as x = (y = 1). In order to assign the result of y = 1 to x, the former must have a value. If y had a setter returning Void, this would not be possible.

## 4.3 Method

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

|  |
| --- |
| class HelloWorld { static public function main():Void { trace("Hello World");  }  } |

1

2

3

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(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;  }  } |

1

2

3

4

5

6

7

8

9

Arguments are given by an opening parenthesis ( after the field name, a comma , separated list of argument specifications and a closing parenthesis ). Additional information on the argument specification is described in Function Type (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:

|  |
| --- |
| class Base {  public function new() { } public function myMethod() { return "Base";  }  }  class Child extends Base { public override function myMethod() { return "Child";  }  }  class Main { static public function main() { var child:Base = new Child(); trace(child.myMethod()); // Child }  } |

1

2

3

4

5

6

7

8

9 10

11

12

13

14

15

16

17

18

19

The important components here are

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

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

### 4.3.2 Effects of variance and access modifiers

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

|  |
| --- |
| class Base {  public function new() { }  }  class Child extends Base { private function method(obj:Child):Child { return obj;  } }  class ChildChild extends Child { public override function method(obj:Base):ChildChild { return null;  }  }  class Main {  static public function main() { }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

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.

|  |
| --- |
| class MyClass { static public function available() { unavailable();  } static private function unavailable() { }  }  class Main { static public function main() {  MyClass.available();  // Cannot access private field unavailable MyClass.unavailable();  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

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

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

|  |
| --- |
| class Base {  public function new() { } private function baseField() { }  }  class Child1 extends Base { private function child1Field() { }  }  class Child2 extends Base { public function child2Field() {  var child1 = new Child1(); child1.baseField();  // Cannot access private field child1Field child1.child1Field();  }  }  class Main {  static public function main() { }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

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

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

1. If the class is declared as extern.
2. If the field 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 objectoriented 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:

|  |
| --- |
| class Main { static inline function mid(s1:Int, s2:Int) {  return (s1 + s2) / 2;  }  static public function main() {  var a = 1; var b = 2; var c = mid(a, b);  }  } |

1

2

3

4

5

6

7

8

9

10

11

The generated Javascript output reveals the effect of inline:

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

1

2

3

4

5

6

7

8

9

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

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

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

|  |
| --- |
| class Main { public static function main () { }  static function test() { if (Math.random() > 0.5) { return "ok";  } else {  error("random failed");  } }  static inline function error(s:String) { throw s;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

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

### 4.4.3 Dynamic

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

|  |
| --- |
| class Main { static dynamic function test() { return "original";  }  static public function main() { trace(test()); // original test = function() { return "new"; } trace(test()); // new  }  } |

1

2

3

4

5

6

7

8

9

10

11

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

String"new".

Dynamic fields cannot be inline for obvious reasons: While inlining is done at 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:

|  |
| --- |
| { expr1; expr2; |

1

2

3

|  |  |
| --- | --- |
| } | ... exprN; |

4

5

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

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

1

2

3

4

5

6

7

8

9

10

11

12

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

## 5.2 Constants

The Haxe syntax supports the following constants:

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

**Float:** A floating point number (2.1.1), such as 0.0, 1., .3, -93.2. **String:** A string of characters (10.1), such as "", "foo", ’’, ’bar’. **true,false:** A boolean (2.1.4) value.

**null:** The null value.

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

**5.3 Binary Operators**

**5.4 Unary Operators**

## 5.5 Array Declaration

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

3.

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

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

1

2

3

4

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

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

## 5.6 Object Declaration

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

|  |  |
| --- | --- |
| {  } | key1:value1, key2:value2, ...  keyN:valueN |

1

2

3

4

5

6

Further details of object declaration are described in the section about anonymous structures

(2.5).

## 5.7 Field Access

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

|  |
| --- |
| object.fieldName |

1

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

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

## 5.8 Array Access

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

|  |
| --- |
| expr[indexExpr] |

1

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

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

## 5.9 Function Call

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

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

1

2

3

4

5

## 5.10 var

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

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

1

2

3

4

5

6

7

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

## 5.11 Local functions

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

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

1

2

3

4

5

6

7

8

9

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

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

|  |
| --- |
| var myLocalFunction = function(a) { } |

1

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

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

## 5.12 new

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

|  |
| --- |
| class Main<T> { static public function main() { new Main<Int>(12, "foo");  }  function new(t:T, s:String) { }  } |

1

2

3

4

5

6

7

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

## 5.13 for

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

|  |
| --- |
| for (v in e1) e2; |

1

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

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

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

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

## 5.14 while

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

|  |
| --- |
| while(condition) expression; |

1

The condition expression has to be of type Bool.

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

|  |
| --- |
| class Main { static public function main() {  var f = 0.0; while (f < 0.5) { |

1

2

3

4

|  |
| --- |
| trace(f); f = Math.random();  }  }  } |

5

6

7

8

9

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

(5.15).

## 5.15 do-while

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

|  |
| --- |
| do expression while(condition); |

1

The condition expression has to be of type Bool.

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

## 5.16 if

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

|  |
| --- |
| if (condition) expression; |

1

The condition expression has to be of type Bool.

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

|  |
| --- |
| if (condition) expression1 else expression2; |

1

Here, expression2 may consist of another if expression:

|  |
| --- |
| if (condition1) expression1 else if(condition2) expression2 else expression3 |

1

2

3

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

## 5.17 switch

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

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

1

2

3

4

5

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

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

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

## 5.18 try/catch

Haxe allows catching values using its try/catch syntax:

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

1

2

3

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

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

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

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

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

## 5.19 return

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

|  |
| --- |
| return; return expression; |

1

2

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

|  |
| --- |
| function f1() { function f2() { return;  } f2(); |

1

2

3

4

5

|  |  |
| --- | --- |
| } | expression; |

6

7

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

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

## 5.20 break

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

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

1

2

3

4

5

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

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

## 5.21 continue

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

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

1

2

3

4

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.

## 5.22 throw

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

|  |
| --- |
| throw expr |

1

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

## 5.23 cast

Haxe allows two kinds of casts:

|  |
| --- |
| cast expr; // unsafe cast cast (expr, Type); // safe cast |

1

2

### 5.23.1 unsafe cast

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

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

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

1

2

3

4

5

6

7

8

9

10

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

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

Unsafe casts have little to no runtime overhead.

### 5.23.2 safe cast

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

|  |
| --- |
| class Base {  public function new() { }  }  class Child1 extends Base {}  class Child2 extends Base {}  class Main { public static function main() { var child1:Base = new Child1(); var child2:Base = new Child2(); cast(child1, Base);  // Exception: Class cast error cast(child1, Child2);  }  } |

1

2

3

4

5

6

7

8

9 10

11

12

13

14

15

16

17

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

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

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

## 5.24 type check

### Since Haxe 3.1.0

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

|  |
| --- |
| (expr : type) |

1

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

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

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

**Chapter 6**

# Language Features

## 6.1 Conditional Compilation

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

### Definition: Compiler Flag

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

This example demonstrates usage of conditional compilation:

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

1

2

3

4

5

6

7

8

9

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 --help-defines argument. The Haxe Compiler allows multiple -D flags per compilation.

## 6.2 Externs

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

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

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

|  |
| --- |
| extern class Math  { static var PI(default,null) : Float; static function floor(v:Float):Int;  } |

1

2

3

4

5

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

|  |
| --- |
| class Main { static public function main() { var pi = Math.floor(Math.PI);  $type(pi); // Int  }  } |

1

2

3

4

5

6

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

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

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

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

## 6.3 Static Extension

### Definition: Static Extension

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

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

|  |
| --- |
| using Main.IntExtender;  class IntExtender { static public function triple(i:Int) {  return i \* 3;  }  }  class Main { static public function main() { trace(12.triple());  }  } |

1

2

3

4

5

6

7

8

9

10

11

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.

### 6.3.1 In the Haxe Standard Library

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

|  |
| --- |
| using StringTools;  class Main { static public function main() { "adc".replace("d", "b");  }  } |

1

2

3

4

5

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:

|  |
| --- |
| Main.main = function() {  StringTools.replace("adc","d","b");  } |

1

2

3

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

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

**Lambda:** Provides functional methods on iterables.

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

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

#### Trivia: “using” using

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

## 6.4 Pattern Matching

### 6.4.1 Introduction

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

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

1

2

3

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:

### 6.4.2 Enum matching

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

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

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

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

**case Node(\_, Leaf(\_)):** matching fails because the right sub-tree of myTree is not a Leaf, but another Node **case Node(\_, Node(Leaf("bar"), \_)):** matching succeeds

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

### 6.4.3 Variable capture

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

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

1

2

3

4

5

6

7

8

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; |

1

2

3

|  |
| --- |
| case x: x;  } trace(node); // Leaf(foo) |

4

5

6

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

### 6.4.4 Structure matching

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

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

1

2

3

4

5

6

7

8

9

10

11

12

13

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

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

### 6.4.5 Array matching

Arrays can be matched on fixed length:

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

1

2

3

4

5

6

7

8

9

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

### 6.4.6 Or patterns

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

|  |
| --- |
| var match = switch(7) {  case 4 | 1: "0"; |

1

2

|  |
| --- |
| case 6 | 7: "1"; case \_: "2";  } trace(match); // 1 |

3

4

5

6

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

### 6.4.7 Guards

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

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

1

2

3

4

5

6

7

8

9

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

### 6.4.8 Match on multiple values

Array syntax can be used to match on multiple values:

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

1

2

3

4

5

6

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

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

\_ still is).

### 6.4.9 Extractors

#### Since Haxe 3.1.0

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

|  |
| --- |
| enum Test {  TString(s:String);  TInt(i:Int);  }  class Main { |

1

2

3

4

5

6

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

7

8

9

10

11

12

13

14

15

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 {  TString(s:String);  TInt(i:Int);  }  class Main { static public function main() { var e = TString("fOo"); var success = switch(e) { case TString(\_.toLowerCase() => "foo"):  true; case \_:  false;  }  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

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

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

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

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

1

2

3

4

5

6

|  |
| --- |
| }  static function add(i1:Int, i2:Int) {  return i1 + i2;  }  static function mul(i1:Int, i2:Int) {  return i1 \* i2;  }  } |

7

8

9

10

11

12

13

14

15

16

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

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

|  |
| --- |
| class Main { static public function main() { switch("foo") {  // Extractors in or patterns are not  // allowed case (\_.toLowerCase() => "foo") | "bar": }  }  } |

1

2

3

4

5

6

7

8

9

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

### 6.4.10 Exhaustiveness checks

The compiler ensures that no possible cases are forgotten:

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

1

2

3

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

Figure out wtf our rules are now for when this is checked. **6.4.11 Useless pattern checks**

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

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

1

2

3

4

5

6

## 6.5 String Interpolation

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

1. var x = 12;
2. // The value of x is 12
3. trace(’The value of x is $x’);

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

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

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

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

1

2

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

### Trivia: String Interpolation before Haxe 3

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

## 6.6 Array Comprehension

Comprehensions are only listing Arrays, not Array comprehension in Haxe uses existing syntax to allow concise initialization of arrays. It

Maps is identified by for or while constructs:

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

1

2

3

4

5

6

7

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:

|  |
| --- |
| var i = 0; var a = [];  while(i < 10) a.push(i++); |

1

2

3

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

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

1

2

3

4

5

6

7

8

9

10

11

12

## 6.7 Iterators

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

|  |
| --- |
| typedef Iterator<T> = { function hasNext() : Bool; function next() : T;  }  typedef Iterable<T> = { function iterator() : Iterator<T>;  } |

1

2

3

4

5

6

7

8

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

|  |
| --- |
| class MyStringIterator { var s:String; var i:Int;  public function new(s:String) {  this.s = s; i = 0;  }  public function hasNext() {  return i < s.length; |

1

2

3

4

5

6

7

8

9 10

11

|  |
| --- |
| }  public function next() { return s.charAt(i++);  }  }  class Main { static public function main() { var myIt = new MyStringIterator("string"); for (chr in myIt) { trace(chr);  }  }  } |

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

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

|  |
| --- |
| class MyArrayWrap<T> { var a:Array<T>; public function new(a:Array<T>) {  this.a = a;  }  public function iterator() { return a.iterator();  }  }  class Main { static public function main() {  var myWrap = new MyArrayWrap([1, 2, 3]); for (elt in myWrap) { trace(elt);  }  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

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

## 6.8 Function Bindings

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

|  |
| --- |
| class Bind { static public function main() { var map = new Map<Int,String>(); |

1

2

3

|  |
| --- |
| var f = map.set.bind(\_, "12");  $type(map.set); // Int -> String -> Void  $type(f); // Int -> Void f(1); f(2); f(3);  trace(map); // {1 => 12, 2 => 12, 3 => 12}  }  } |

4

5

6

7

8

9

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.

## 6.9 Metadata

Several constructs can be attributed with custom metadata:

* class and enum declarations
* Class fields
* Enum constructors
* Expressions

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

|  |
| --- |
| import haxe.rtti.Meta;  @author("Nicolas") @debug class MyClass { @range(1, 8) var value:Int;  @broken |

1

2

3

4

5

6

7

8

9

|  |
| --- |
| @:noCompletion static function method() { }  }  class Main { static public function main() {  // { author : ["Nicolas"], debug : null } trace(Meta.getType(MyClass));  // [1,8] trace(Meta.getFields(MyClass).value.range);  // { broken: null }  trace(Meta.getStatics(MyClass).method);  }  } |

10

11

12

13

14

15

16

17

18

19

20

21

22

23

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

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

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

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

Allowed values for metadata arguments are:

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

## 6.10 Access Control

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

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

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

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

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

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

|  |
| --- |
| @:allow(Main) class MyClass { static private var foo: Int;  }  class Main { static public function main() { MyClass.foo;  }  } |

1

2

3

4

5

6

7

8

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 {  @:allow(Main.main) static private var foo: Int;  }  class Main { static public function main() { MyClass.foo;  }  } |

1

2

3

4

5

6

7

8

9

10

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

|  |
| --- |
| class MyClass {  static private var foo: Int;  }  class Main {  @:access(MyClass.foo) static public function main() { MyClass.foo;  }  } |

1

2

3

4

5

6

7

8

9

10

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

|  |
| --- |
| **Trivia: On the choice of metadata**  The access control language feature uses the Haxe metadata syntax instead of additional language-specific syntax. There are several reasons for that:   * Additional syntax often adds complexity to the language parsing, and also adds (too) many keywords. * Additional syntax requires additional learning by the language user, whereas metadata syntax is something that is already known. * The metadata syntax is flexible enough to allow extension of this feature. * The metadata can be accessed/generated/modified by Haxe macros.   Of course, the main drawback of using metadata syntax is that you get no error report in case you misspell either the metadata key (@:acesss for instance) or the class/package name. However, with this feature you will get an error when you try to access a private field that you are not allowed to, therefore there is no possibility for silent errors. |

### Since Haxe 3.1.0

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

|  |
| --- |
| class MyClass { @:allow(I) static private var foo: Int;  }  interface I { }  class Main implements I { static public function main() { MyClass.foo;  }  } |

1

2

3

4

5

6

7

8

9

10

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 implementation was simply missing.

## 6.11 Inline constructors

### Since Haxe 3.1.0

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

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

The following example demonstrates constructor inlining:

|  |
| --- |
| class Point { public var x:Float; public var y:Float;  public inline function new(x:Float, y:Float) {  this.x = x; this.y = y;  }  }  class Main { static public function main() { var pt = new Point(1.2, 9.3); }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

A look at the Javascript output reveals the effect:

|  |
| --- |
| Main.main = function() { var pt\_x = 1.2; var pt\_y = 9.3;  }; |

1

2

3

4

**6.12 Remoting**

**Part II**

# Compiler Reference

**Chapter 7**

**Compiler Usage**

**Chapter 8**

# Compiler Features

**8.1 Compiler Metadata**

## 8.2 Dead Code Elimination

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

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

**-dce std:** Only classes in the Haxe Standard Library are affected by DCE. This is the default setting on all targets 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.

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

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

### Trivia: DCE-rewrite

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

### Trivia: DCE and try.haxe.org

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

## 8.3 Completion

**8.3.1 Field Access 8.3.2 Call Arguments 8.3.3 Usage 8.3.4 Position 8.3.5 Toplevel**

## 8.4 Resources

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

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

### 8.4.1 Embedding resources

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

what to use for listing

of non-haxe code like

hxml?

1

-

resource

hello\_message.txt@welcome

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

### 8.4.2 Retrieving text resources

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

|  |
| --- |
| class Main { static function main() { trace(haxe.Resource.getString("welcome"));  }  } |

1

2

3

4

5

The code above will display the content of the *hello message.txt* file that we included earlier using *welcome* as the identifier.

### 8.4.3 Retrieving binary resources

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

|  |
| --- |
| class Main { static function main() {  var bytes =  haxe.Resource.getBytes("welcome");  trace(bytes.readString(0, bytes.length)); }  } |

1

2

3

4

5

6

7

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

### 8.4.4 Implementation details

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

* *Flash* resources are embedded as ByteArray definitions
* *C#* resources are included in the compiled assembly
* *Java* resources are packed in the resulting JAR file
* *C++* resources are stored in global byte array constants.
* *JavaScript* resources are serialized in Haxe serialization format and stored in a static field of haxe.Resource class.
* *Neko* resources are stored as strings in a static field of haxe.Resource class.

## 8.5 Runtime Type Information

The Haxe compiler generates runtime type information (RTTI) for classes that are annotated or extend classes that are annotated with the :rtti metadata. This information is stored as a XML string in a static field \_\_rtti and can be processed through haxe.rtti.XmlParser. The resulting structure is described in RTTI structure (Section 8.5.1).

### Since Haxe 3.2.0

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

|  |
| --- |
| @:rtti class Main { var x:String; static function main() { var rtti = haxe.rtti.Rtti.getRtti(Main); |

1

2

3

4

5

|  |
| --- |
| trace(rtti);  }  } |

6

7

8

### 8.5.1 RTTI structure

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

**file:** The full slash path of the .hx file containing the type. This might be null in case there is no such file, e.g. if the type is defined through a macro (9). **params:** An array of strings representing the names of the type parameters (3.2) the type has. As of Haxe 3.2.0, this does not include the constraints (3.2.1).

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

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

**platforms:** A list of strings representing the targets where the type is available. **meta:** The meta data the type was annotated with.

**Class type information isExtern:** Whether or not the class is extern (6.2). **isInterface:** Whether or not the class is actually an interface (2.3.3). **superClass:** The class’ parent class defined by its type path and list of type parameters. **interfaces:** The list of interfaces defined by their type path and list of type parameters.

**fields:** The list of member class fields (4), described in Class field information (Section 8.5.1). **statics:** The list of static class fields, described in Class field information (Section 8.5.1).

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

**Enum type information isExtern:** Whether or not the enum is extern (6.2). **constructors:** The list of enum constructors.

**Abstract type information to:** An array containing the defined implicit to casts (2.8.1). **from:** An array containing the defined implicit from casts (2.8.1). **impl:** The class type information (8.5.1) of the implementing class. **athis:** The underlying type (2.8) of the abstract.

**Class field information name:** The name of the field. **type:** The type of the field. **isPublic:** Whether or not the field is public (4.4.1). **isOverride:** Whether or not the field overrides (4.4.4) another field. **doc:** The documentation of the field. This information is only available if the compiler flag (6.1) -D use\_rtti\_doc was in place. Otherwise, or if the field has no documentation, the value is null.

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

**params:**

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

**platforms:** A list of strings representing the targets where the field is available. **meta:** The meta data the field was annotated with.

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

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

**Enum constructor information name:** The name of the constructor. **args:** The list of arguments the constructor has, or null if no arguments are available.

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

**platforms:** A list of strings representing the targets where the constructor is available. **meta:** The meta data the constructor was annotated with.

**Chapter 9**

# Macros

Macros are without a doubt the most advanced feature in Haxe. They are often perceived as dark magic that only a select few are capable of mastering, yet there is nothing magical (and certainly nothing dark) about them.

## Definition: Abstract Syntax Tree (AST)

The AST is the result of *parsing* Haxe code into a typed structure. This structure is exposed to macros through the types defined in the file haxe/macro/Expr.hx of the Haxe Standard Library.

• Expression

Source code

Lexer / Parser

Abstract Syntax Tree (AST)

•

Complex Type

•

haxe.macro.Expr

Macro processor

Abstract Syntax Tree (AST)

Typer

Typed AST

•

Typed Expression

•

Type

•

haxe.macro.Type

Generator

Output

parse

transform

type

generate

Figure 9.1: The role of macros during compilation.

A basic macro is a *syntax-transformation*. It receives zero or more expressions (5) and also returns an expression. If a macro is called, it effectively inserts code at the place it was called from. In that respect, it could be compared to a preprocessor like #define in C++, but a Haxe macro is not a textual replacement tool.

We can identify different kinds of macros, which are run at specific compilation stages:

**Initialization Macros:** These are provided by command line using the --macro compiler parameter. They are executed after the compiler arguments were processed and the *typer context* has been created, but before any typing was done (see Initialization macros (Section 9.7)).

**Build Macros:** These are defined for classes, enums and abstracts through the @:build or @:autoBuild metadata (6.9). They are executed per-type, after the type has been set up (including its relation to other types, such as inheritance for classes) but before its fields are typed (see Type Building (Section 9.5)).

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

## 9.1 Macro Context

### Definition: Macro Context

The macro context is the environment in which the macro is executed. Depending on the macro type, it can be considered to be a class being built or a function being typed. Contextual information can be obtained through the haxe.macro.Context API.

Haxe macros have access to different contextual information depending on the macro type. Other than querying such information, the context also allows some modifications such as defining a new type or registering certain callbacks. It is important to understand that not all information is available for all macro kinds, as the following examples demonstrate:

* Initialization macros will find that the Context.getLocal\*() methods return null. There is no local type or method in the context of an initialization macro.
* Only build macros get a proper return value from Context.getBuildFields(). There are no fields being built for the other macro kinds.
* Build macros have a local type (if incomplete), but no local method, so Context.getLocalMethod() returns null.

The context API is complemented by the haxe.macro.Compiler API detailed in Initialization macros (Section 9.7). While this API is available to all macro kinds, care has to be taken for any modification outside of initialization macros. This stems from the natural limitation of undefined build order (9.6.3), which could cause e.g. a flag definition through Compiler.define() to take effect before or after a conditional compilation (6.1) check against that flag.

## 9.2 Arguments

Most of the time, arguments to macros are expressions represented as an instance of enum Expr. As such, they are parsed but not typed, meaning they can be anything conforming to Haxe’s syntax rules. The macro can then inspect their structure, or (try to) get their type using haxe.macro.Context.typeof().

It is important to understand that arguments to macros are not guaranteed to be evaluated, so any intended side-effect is not guaranteed to occur. On the other hand, it is also important to understand that an argument expression may be duplicated by a macro and used multiple times in the returned expression:

|  |
| --- |
| import haxe.macro.Expr;  class Main { static public function main() {  var x = 0; var b = add(x++); trace(x); // 2  }  macro static function add(e:Expr) {  return macro $e + $e;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

The macro add is called with x++ as argument and thus returns x++ + x++ using expression reification (9.3.1), causing x to be incremented twice.

### 9.2.1 ExprOf

Since Expr is compatible with any possible input, Haxe provides the type haxe.macro.ExprOf<T>. For the most part, this type is identical to Expr, but it allows constraining the type of accepted expressions. This is useful when combining macros with static extensions (6.3):

|  |
| --- |
| import haxe.macro.Expr; using Main;  class Main { static public function main() { identity("foo"); identity(1); "foo".identity();  // Int has no field identity  //1.identity(); }  macro static function identity(e:ExprOf<String>) { return e;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

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 Int1 is accepted, but it is a logical consequence of what was explained about macro arguments (9.2): The argument expressions are never typed, so it is not possible for the compiler to check their compatibility by unifying (3.5).

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

### 9.2.2 Constant Expressions

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

|  |
| --- |
| class Main { static public function main() { const("foo", 1, 1.5, true);  }  macro static function const(s:String, i:Int, f:Float, b:Bool) { trace(s); trace(i); trace(f); trace(b); return macro null;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

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

### 9.2.3 Rest Argument

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

|  |
| --- |
| import haxe.macro.Expr;  class Main { static public function main() { myMacro("foo", a, b, c);  }  macro static function myMacro(e1:Expr, extra:Array<Expr>) {  for (e in extra) { trace(e);  } return macro null;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

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

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

## 9.4 Tools

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

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

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

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

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

### Trivia: The tinkerbell library and why Tools.hx works

We learned about static extensions that using a *module* implies that all its types are brought into static extension context. As it turns out, such a type can also be a typedef (3.1) to another type. The compiler then considers this type part of the module, and extends static extension accordingly.

This “trick” was first used in Juraj Kirchheim’s *tinkerbell*1 library for exactly the same purpose. Tinkerbell provided many useful macro tools long before they made it into the Haxe Compiler and Haxe Standard Library. It remains the primary library for additional macro tools and offers other useful functionality as well.

## 9.5 Type Building

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

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

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

|  |
| --- |
| import haxe.macro.Context; import haxe.macro.Expr;  class TypeBuildingMacro { macro static public function build(fieldName:String):Array<Field> { var fields = Context.getBuildFields(); var newField = { name: fieldName, doc: null, meta: [], access: [AStatic, APublic], kind: FVar(macro : String, macro "my default"),  pos: Context.currentPos()  }; fields.push(newField); return fields;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

|  |
| --- |
| @:build(TypeBuildingMacro.build("myFunc")) class Main { static public function main() { trace(Main.myFunc); // my default |

1

2

3

4

1. }
2. }

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;

2

import

haxe.

macro

.Expr;

3

4

class

EnumBuildingMacro

{

5

macro

static

public

function

6

build():Array<Field> {

7

var

noArgs

=

8

makeEnumField(

"A"

,

FVar(

null

,

null

))

;

9

var

eFunc

=

macro

function

(

value:Int) { };

10

var

fInt

=

switch

(

eFunc.expr

){

11

case

EFunction(\_,f):

f;

12

case

\_:

throw

"false"

;

13

}

14

var

intArg

=

15

makeEnumField(

"B"

,

FFun(fInt));

16

return

[

noArgs,

intArg];

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

17

18

19

20

21

22

23

24

25

26

27

28

29

|  |
| --- |
| @:build(EnumBuildingMacro.build())  enum E { }  class Main { static public function main() { switch(E.A) { case A:  case B(v):  }  }  } |

1

2

3

4

5

6

7

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 (6.4) it. We can see that the generated type is equivalent to this:

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

1

2

3

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; import haxe.macro.Expr;  class AutoBuildingMacro { macro static public function fromInterface():Array<Field> { trace("fromInterface: "  + Context.getLocalType()); |

1

2

3

4

5

6

7

8

1. return null;
2. }

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

11

12

13

14

15

16

17

18

|  |
| --- |
| @:autoBuild(AutoBuildingMacro.fromInterface()) interface I { }  interface I2 extends I { }  @:autoBuild(AutoBuildingMacro.fromBaseClass()) class Base { }  class Main extends Base implements I2 {  static public function main() { }  } |

1

2

3

4

5

6

7

8

9

10

11

This outputs during compilation:

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

1

2

3

4

5

6

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

#### Since Haxe 3.1.0

The combination of static extensions and macros was reworked for the 3.1.0 release. The Haxe Compiler does not even try to find the original expression for the macro argument and instead passes a special @:this this expression. While the structure of this expression conveys no information, the expression can still be typed correctly:

|  |
| --- |
| import haxe.macro.Context; import haxe.macro.Expr;  using Main; using haxe.macro.Tools;  class Main { static public function main() { "foo".test(); }  macro static function test(e:ExprOf<String>) { trace(e.toString()); // @:this this  // TInst(String,[]) trace(Context.typeof(e)); return e;  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

### 9.6.3 Build Order

The build order of types is unspecified and this extends to the execution order of build-macros (9.5). While certain rules can be determined, we strongly recommend to not rely on the execution order of build-macros. If type building requires multiple passes, this should be reflected directly in the macro code. In order to avoid multiple build-macro execution on the same type, state can be stored in static variables or added as metadata (6.9) to the type in question:

|  |
| --- |
| import haxe.macro.Context; import haxe.macro.Expr;  #if !macro  @:autoBuild(MyMacro.build())  #end interface I1 { }  #if !macro  @:autoBuild(MyMacro.build())  #end interface I2 { }  class C implements I1 implements I2 { }  class MyMacro { macro static public function build():Array<Field> { var c = Context.getLocalClass().get(); if (c.meta.has(":processed")) return null; c.meta.add(":processed",[],c.pos); // process here return null; |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

1. }
2. }

|  |
| --- |
| class Main {  static public function main() { }  } |

26

27

28

29

With both interfaces I1 and I2 having :autoBuild metadata, the build macro is executed twice for class C. We guard against duplicate processing by adding a custom :processed metadata to the class, which can be checked during the second macro execution.

**9.6.4 Type Parameters**

## 9.7 Initialization macros

Initialization macros are invoked from command line by using the --macro callExpr(args) command. This registers a callback which the compiler invokes after creating its context, but before typing what was argument to -main. This then allows configuring the compiler in some ways.

If the argument to --macro is a call to a plain identifier, that identifier is looked up in the class haxe.macro.Compiler which is part of the Haxe Standard Library. It comes with several useful initialization macros which are detailed in its [API.](http://api.haxe.org//haxe/macro/Compiler.html)

As an example, the include macro allows inclusion of an entire package for compilation, recursively if necessary. The command line argument for this would then be --macro include(’some.pack’ true).

Of course it is also possible to define custom initialization macros to perform various tasks before the real compilation. A macro like this would then be invoked via --macro some.Class.theMacro(ar For instance, as all macros share the same context (9.1), an initialization macro could set the value of a static field which other macros use as configuration.

**Part III**

# Standard Library

**Chapter 10**

# Standard Library

Standard library

## 10.1 String

**Type: String**

A String is a sequence of characters.

## 10.2 Data Structures

### 10.2.1 Array

An Array is a collection of elements. It has one type parameter (3.2) which corresponds to the type of these elements. Arrays can be created in three ways:

1. By using their constructor: new Array()
2. By using array declaration syntax (5.5): [1, 2, 3]
3. By using array comprehension (6.6): [for (i in 0...10) if (i % 2 == 0) i]

Arrays come with an [API](http://api.haxe.org/Array.html) to cover most use-cases. Additionally they allow read and write array access (5.8):

|  |
| --- |
| class Main { static public function main() {  var a = [1, 2, 3]; trace(a[1]); // 2 a[1] = 1; trace(a[1]); // 1  }  } |

1

2

3

4

5

6

7

8

Since array access in Haxe is unbounded, i.e. it is guaranteed to not throw an exception, this requires further discussion:

* If a read access is made on a non-existing index, a target-dependent value is returned.
* If a write access is made with a positive index which is out of bounds, null (or the default value (2.2) for basic types (2.1) on static targets (2.2)) is inserted at all positions between the last defined index and the newly written one.
* If a write access is made with a negative index, the result is unspecified.

Arrays define an iterator (6.7) over their elements. This iteration is typically optimized by the compiler to use a while loop (5.14) with array index:

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

1

2

3

4

5

6

7

8

9

10

Haxe generates this optimized Javascript output:

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

1

2

3

4

5

6

7

8

9

10

11

Haxe does not allow arrays of mixed types unless the parameter type is forced to Dynamic

(2.7):

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

1

2

3

4

5

6

7

8

9 10

11

12

#### Trivia: Dynamic Arrays

In Haxe 2, mixed type array declarations were allowed. In Haxe 3, arrays can have mixed types only if they are explicitly declared as Array<Dynamic>.

### 10.2.2 Vector

A Vector is an optimized fixed-length *collection* of elements. Much like Array (10.2.1), it has one type parameter (3.2) and all elements of a vector must be of the specified type, it can be *iterated over* using a for loop (5.13) and accessed using array access syntax (2.8.3). However, unlike Array and List, vector length is specified on creation and cannot be changed later.

|  |
| --- |
| class Main { static function main() { var vec = new haxe.ds.Vector(10);  for (i in 0...vec.length) { vec[i] = i;  }  trace(vec[0]); // 0 trace(vec[5]); // 5 trace(vec[9]); // 9  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

haxe.ds.Vector is implemented as an abstract type (2.8) over a native array implementation for given target and can be faster for fixed-size collections, because the memory for storing its elements is pre-allocated.

### 10.2.3 List

A List is a *collection* for storing elements. On the surface, a list is similar to an Array (Section 10.2.1). However, the underlying implementation is very different. This results in several functional differences:

1. A list can not be indexed using square brackets, i.e. [0].
2. A list can not be initialized.
3. There are no list comprehensions.
4. A list can freely modify/add/remove elements while iterating over them.

See the [List API](http://api.haxe.org/List.html) for details about the list methods. A simple example for working with lists:

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

1

2

3

4

5

6

7

8

### 10.2.4 GenericStack

A GenericStack, like Array and List is a container for storing elements. It has one type parameter (3.2) and all elements of the stack must be of the specified type. See the [GenericStack API](http://api.haxe.org/haxe/ds/GenericStack.html) for details about its methods. Here is a small example program for initializing and working with a GenericStack.

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

1

2

3

4

5

6

7

8

9

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.

### 10.2.5 Map

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

|  |
| --- |
| class Main { static public function main() {  // Maps are initialized like arrays, but  // use ’=>’ operator. Maps can have their // key value types defined explicity var map1:Map<Int, String> = [1 => "one", 2=>"two"];  // Or they can infer their key value types var map2 = [  "one"=>1,  "two"=>2,  "three"=>3  ];  $type(map2); // Map<String, Int>  // Keys must be unique  // Error: Duplicate Key  //var map3 = [1=>"dog", 1=>"cat"];  // Maps values can be accessed using array  // accessors "[]" var map4 = ["M"=>"Monday", "T"=>"Tuesday"]; trace(map4["M"]); //Monday |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

|  |
| --- |
| // Maps iterate over their values by  // default var valueSum; for (value in map4) { trace(value); // Monday \n Tuesday  }  // Can iterate over keys by using the  // keys() method for (key in map4.keys()) {  trace(key); // M \n T  }  // Like arrays, a new Map can be made using  // comprehension var map5 = [ for (key in map4.keys()) key => "FRIDAY!!"  ];  // {M => FRIDAY!!, T => FRIDAY!!} trace(map5);  }  } |

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

See the [Map API](http://api.haxe.org/Map.html) for details of its methods.

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

* String: haxe.ds.StringMap
* Int: haxe.ds.IntMap
* EnumValue: haxe.ds.EnumValueMap
* {}: haxe.ds.ObjectMap

The Map type does not exist at runtime and has been replaced with one of the above objects. Map defines array access (2.8.3) using its key type.

### 10.2.6 Option

An option is an enum (2.4) in the Haxe Standard Library which is defined like so:

|  |
| --- |
| enum Option<T> {  Some(v:T);  None;  } |

1

2

3

4

It can be used in various situations, such as communicating whether or not a method had a valid return and if so, what value it returned:

|  |
| --- |
| import haxe.ds.Option;  class Main { static public function main() { |

1

2

3

4

|  |
| --- |
| var result = trySomething(); switch (result) { case None: trace("Got None");  case Some(s):  trace("Got a value: " +s); } }  static function trySomething():Option<String> {  if (Math.random() > 0.5) { return None;  } else {  return Some("Success");  }  }  } |

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

## 10.3 Regular Expressions

Haxe has built-in support for *regular expressions*[[6]](#footnote-6). They can be used to verify the format of a string, transform a string or extract some regular data from a given text.

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

|  |
| --- |
| var r = ˜/haxe/i; |

1

Alternatively, we can create regular expression with regular syntax:

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

1

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
* [ˆ\r\n\t] character not-in-range
* (...) parenthesis to match groups of characters
* ˆ beginning of the string (beginning of a line in multiline matching mode)
* $ end of the string (end of a line in multiline matching mode)
* | ”OR” statement.

For example, the following regular expression matches valid email addresses:

|  |
| --- |
| ˜/[A-Z0-9.\_\%-]+@[A-Z0-9.-]+\.[A-Z][A-Z][A-Z]?/i; |

1

Please notice that the i at the end of the regular expression is a *flag* that enables case-insensitive matching.

The possible flags are the following:

* i case insensitive matching
* g global replace or split, see below
* m multiline matching, ˆ and $ represent the beginning and end of a line
* s the dot . will also match newlines *(Neko, C++, PHP and Java targets only)*
* u use UTF-8 matching *(Neko and C++ targets only)*

### 10.3.1 Matching

Probably one of the most common uses for regular expressions is checking whether a string matches the specific pattern. The match method of a regular expression object can be used to do that:

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

1

2

3

4

5

6

7

8

9

### 10.3.2 Groups

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

|  |
| --- |
| class Main { static function main() {  var str = "Nicolas is 26 years old"; var r =  ˜/([A-Za-z]+) is ([0-9]+) years old/;  r.match(str);  trace(r.matched(1)); // "Nicolas" trace(r.matched(2)); // "26" }  } |

1

2

3

4

5

6

7

8

9

10

Note that group numbers start with 1 and r.matched(0) will always return the whole matched substring.

The r.matchedPos() will return the position of this substring in the original string:

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

1

2

3

4

5

6

7

8

9

10

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

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

1

2

3

4

5

6

7

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"; // g : replace all instances var r = ˜/b[ˆc]/g; // "aaabcbcbcxx" trace(r.replace(str,"xx"));  }  } |

1

2

3

4

5

6

7

8

9

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

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

1

2

3

4

5

6

7

8

### 10.3.4 Split

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

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

1

2

3

4

5

6

7

8

### 10.3.5 Map

The map method of a regular expression object can be used to replace matched substrings using a custom function:

|  |
| --- |
| class Main { static function main() { var r = ˜/world/; var s = "Hello, world!"; var s2 = r.map(s, function(r) { return "Haxe";  }); trace(s2); // Hello, Haxe!  }  } |

1

2

3

4

5

6

7

8

9

10

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 Flash, PHP, C# and Java, native implementations are used
* in Flash 6/8, the implementation is not available

## 10.4 Math

Haxe includes a floating point math library for some common mathematical operations. Most of the fuctions operate on and return floats. However, an Int can be used where a Float is expected, and Haxe also converts Int to Float during most numeric operations (see Numeric Operators (Section 2.1.3) for more details).

Here are some example uses of the math library. See the [Math API](http://api.haxe.org/Math.html) for all available functions.

|  |
| --- |
| class MathExample { static public function main() {  var x = 1/2; var y = 20.2; |

1

2

3

4

5

|  |
| --- |
| var z = -2;  trace(Math.abs(z)); //2 trace(Math.sin(x\*Math.PI)); //1 trace(Math.ceil(y)); //21  // log is the natural logarithm trace(Math.log(Math.exp(5))); //5  // Output for neko target, may vary  // depending on platform trace(1/0); //inf trace(-1/0); //-inf trace(Math.sqrt(-1)); //nan  }  } |

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

### 10.4.1 Special Numbers

The math library has definitions for several special numbers:

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

### 10.4.2 Mathematical Errors

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

### 10.4.3 Integer Math

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

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

I think C++ can use

integer operatins, but I

don’t know about any **10.4.4 Extensions**

other targets. Only saw this mentioned It is common to see Static Extension (Section 6.3) used with the math library. This code shows a in an old discussion simple example: thread, still true?

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

1

2

3

4

5

6

7

8

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

1

2

3

4

5

6

7

**10.5 Lambda**

## 10.6 Reflection

Haxe supports runtime reflection of types and fields. Special care has to be taken here because runtime representation generally varies between targets. In order to use reflection correctly it is necessary to understand what kind of operations are supported and what is not. Given the dynamic nature of reflection, this can not always be determined at compile-time. The reflection API consists of two classes:

**Reflect:** A lightweight API which work best on anonymous structures (2.5), with limited support for classes (2.3).

**Type:** A more robust API for working with classes and enums (2.4).

The available methods are detailed in the API for [Reflect](http://api.haxe.org//Reflect.html) and [Type.](http://api.haxe.org//Type.html)

Reflection can be a powerful tool, but it is important to understand why it can also cause problems. As an example, several functions expect a String (10.1) argument and try to resolve it to a type or field. This is vulnerable to typing errors:

|  |
| --- |
| class Main { static function main() { trace(Type.resolveClass("Mian")); // null }  } |

1

2

3

4

5

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

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

1

2

3

4

5

6

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

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

|  |
| --- |
| class Main { static function main() {  // wrong  //Reflect.callMethod(Main, "f", []);  // right  Reflect.callMethod(Main,  Reflect.field(Main, "f"), []);  }  static function f() { trace(’Called’);  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

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

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

|  |
| --- |
| typedef MyStructure = { name: String, score: 12  }  class Main { static function main() { var data = reflective();  // At this point data is nicely typed as  // MyStructure  }  static function reflective():MyStructure { // Work with reflection here to get some // values we want to return.  return { name: "Reflection", score: 0  }  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

While the method reflective could interally work with reflection (and Dynamic for that matter) a lot, its return value is a typed structure which the callers can use in a type-safe manner.

## 10.7 Serialization

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

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

The following example demonstrates the first usage:

|  |
| --- |
| import haxe.Serializer; import haxe.Unserializer;  class Main { static function main() { var serializer = new Serializer(); serializer.serialize("foo"); serializer.serialize(12); var s = serializer.toString(); trace(s); // y3:fooi12  var unserializer = new Unserializer(s); trace(unserializer.unserialize()); // foo trace(unserializer.unserialize()); // 12 }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

The result of the serialization (here stored in local variable s) is a String (10.1) and can be passed around at will, even remotely. Its format is described in Serialization format (Section 10.7.1).

### Supported values

* null
* Bool, Int and Float (including infinities and NaN)
* String
* Date
* haxe.io.Bytes (encoded as base64)
* Array (10.2.1) and List (10.2.3)
* haxe.ds.StringMap, haxe.ds.IntMap and haxe.ds.ObjectMap
* anonymous structures (2.5)
* Haxe class instances (2.3) (not native ones)
* enum instances (2.4)

**Serialization configuration** Serialization can be configured in two ways. For both a static variable can be set to influence all haxe.Serializer instances, and a member variable can be set to only influence a specific instance:

**USE\_CACHE, useCache:** If true, repeated objects are serialized by reference. This can avoid infinite loops for recursive data at the expense of longer serialization time. By default, this value is false.

**USE\_ENUM\_INDEX, useEnumIndex:** If true, enum constructors are serialized by their index instead of their name. This can make the serialization string shorter but breaks if enum constructors are inserted into the type before deserialization. By default, this value is false.

**Deserialization behavior** If the serialization result is stored and later used for deserialization, care has to be taken to maintain compatibility when working with class and enum instances. It is then important to understand exactly how unserialization is implemented.

* The type has to be available in the runtime where the deserialization is made. If dead code elimination (8.2) is active, a type which is used only through serialization might be removed.
* Each Unserializer has a member variable resolver which is used to resolve classes and enums by name. Upon creation of the Unserializer this is set to Unserializer.DEFAULT\_RESOLVER. Both that and the instance member can be set to a custom resolver.
* Classes are resolved by name using resolver.resolveClass(name). The instance is then created using Type.createEmptyInstance, which means that the class constructor is not called. Finally, the instance fields are set according to the serialized value.
* Enums are resolved by name using resolver.resolveEnum(name). The enum instance is then created using Type.createEnum, using the serialized argument values if available. If the constructor arguments were changed since serialization, the result is unspecified.

**Custom (de)serialization** If a class defines the member method hxSerialize, that method is called by the serializer and allows custom serialization of the class. Likewise, if a class defines the member method hxUnserialize it is called by the deserializer:

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

1

2

3

4

5

6

7

8

9 10

11

12

13

14

15

16

17

|  |  |
| --- | --- |
| } | this.y = y;  }  @:keep function hxSerialize(s:Serializer) {  s.serialize(x);  }  @:keep function hxUnserialize(u:Unserializer) { x = u.unserialize(); y = -1;  } |

18

19

20

21

22

23

24

25

26

27

28

29

30

31

In this example we decide that we want to ignore the value of member variable y and do not serialize it. Instead we default it to -1 in hxUnserialize. Both methods are annotated with the :keep metadata to prevent dead code elimination (8.2) from removing them as they are never properly referenced in the code.

### 10.7.1 Serialization format

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

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

**Float: NaN:** k

#### negative infinity: m positive infinity: p

**normal Float:** d followed by the float display (e.g. d1.45e-8)

**Bool:** t for true, f for false

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

**String (cached):** R followed by the string cache ID (e.g. R456). String caching is always enabled.

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

**structure:** o followed by the list of name-value pairs, followed by g (e.g. oy1:xi2y1:kng for

{x:2, k:null})

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

**Array:** a followed by the list of serialized items, followed by h. For multiple consecutive null values, u followed by the number of null values is used (e.g. ai1i2u4i7ni9h for

[1,2,null,null,null,null,7,null,9])

**Date:** v followed by the date itself (e.g. d2010-01-01 12:45:10)

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

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

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

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

**exception:** x followed by the exception value

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

**enum instance (by name):** w followed by the serialized enum name, followed by the serialized constructor name, followed by the number of arguments, followed by the argument values

(e.g. wy3:Fooy1:A0 for Foo.A (with no arguments), wy3:Fooy1:B2i4n for Foo.B(4,null))

**enum instance (by index):** j followed by the serialized enum name, followed by :, followed by the constructor index, followed by the number of arguments, followed by the argument values (e.g. wy3:Foo0:0 for Foo.A (with no arguments), wy3:Foo1:2i4n for

Foo.B(4,null)) **custom:** C followed by the class name, followed by the custom serialized data, followed by g **cache references:** r followed by the cache index

## 10.8 Json

Haxe provides built-in support for (de-)serializing *JSON*[[7]](#footnote-7) data via the 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:

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

1

2

3

4

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

### 10.8.2 Encoding JSON

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

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

1

2

3

4

5

6

7

### 10.8.3 Implementation details

The haxe.Json API automatically uses native implementation on targets where it is available, i.e. *JavaScript*, *Flash* and *PHP* and provides its own implementation for other targets.

Usage of Haxe own implementation can be forced with -D haxeJSON compiler argument. This will also provide serialization of enums (2.4) by their index, maps (10.2.5) with string keys and class instances.

Older browsers (Internet Explorer 7, for instance) may not have native *JSON* implementation. In case it’s required to support them, we can include one of the JSON implementations available on the internet in the HTML page. Alternatively, a -D old\_browser compiler argument that will make haxe.Json try to use native JSON and fallback to its own implementation in case it’s not available can be used.

**10.9 Xml**

**10.10 Input/Output**

## 10.11 Sys/sys

**Part IV Miscellaneous**

**Chapter 11**

# Haxelib

Haxelib is the library manager that comes with any Haxe distribution. Connected to a central repository, it allows submitting and retrieving libraries and has multiple features beyond that.

Available libraries can be found at [http://lib.haxe.org.](http://lib.haxe.org)

A basic Haxe library is a collection of .hx files. That is, libraries are distributed by source code by default, making it easy to inspect and modify their behavior. Each library is identified by a unique name, which is utilized when telling the Haxe Compiler which libraries to use for a given compilation.

## 11.1 Using a Haxe library with the Haxe Compiler

Any installed Haxe library can be made available to the compiler through the -lib <library-name> argument. This is very similiar to the -cp <path> argument, but expects a library name instead of a directory path. These commands are explained thoroughly in Compiler Usage (Chapter 7).

For our exemplary usage we chose a very simple Haxe library called “random”. It provides a set of static convenience methods to achieve various random effects, such as picking a random element from an array.

|  |
| --- |
| class Main { static public function main() { var elt = Random.fromArray([1, 2, 3]); trace(elt);  }  } |

1

2

3

4

5

6

Compiling this without any -lib argument causes an error message along the lines of Unknown identifier : Random. This shows that installed Haxe libraries are not available to the compiler by default unless they are explicitly added. A working command line for above program is haxe -lib random -main Main --interp.

If the compiler emits an error Error: Library random is not installed : run ’haxelib install random’ the library has to be installed via the haxelib command first. As the error message suggests, this is achieved through haxelib install random. We will learn more about the haxelib command in Using Haxelib (Section 11.4).

## 11.2 haxelib.json

Each Haxe library requires a haxelib.json file in which the following attributes are defined:

**name:** The name of the library. It must contain at least 3 characters among the following:

*A* − *Za* − *z*0−9−*.*

. In particular, no spaces are allowed.

**url:** The URL of the library, i.e. where more information can be found.

**license:** The license under which the library is released. Can be GPL, LGPL, BSD, Public (for Public Domain) or MIT.

**tags:** An array of tag-strings which are used on the repository website to sort libraries.

**description:** The description of what the library is doing.

**version:** The version string of the library. This is detailed in Versioning (Section 11.2.1). **releasenote:** The release notes of the current version.

**contributors:** An array of user names which identify contributors to the library.

**dependencies:** An object describing the dependencies of the library. This is detailed in Dependencies (Section 11.2.2).

The following JSON is a simple example of a haxelib.json:

|  |
| --- |
| {  "name": "useless\_lib", "url" :  "https://github.com/jasononeil/useless/",  "license": "MIT",  "tags": ["cross", "useless"], "description":  "This library is useless in the same way on every platform", "version": "1.0.0", "releasenote":  "Initial release, everything is working correctly",  "contributors": ["Juraj","Jason","Nicolas"],  "dependencies": {  "tink\_macros": "",  "nme": "3.5.5"  }  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

### 11.2.1 Versioning

Haxelib uses a simplified version of [SemVer.](http://semver.org/) The basic format is this:

|  |
| --- |
| major.minor.patch |

1

These are the basic rules:

* Major versions are incremented when you break backwards compatibility - so old code will not work with the new version of the library.
* Minor versions are incremented when new features are added.
* Patch versions are for small fixes that do not change the public API, so no existing code should break.
* When a minor version increments, the patch number is reset to 0. When a major version increments, both the minor and patch are reset to 0.

Examples:

**”0.0.1”:** A first release. Anything with a ”0” for the major version is subject to change in the next release - no promises about API stability!

**”0.1.0”:** Added a new feature! Increment the minor version, reset the patch version **”0.1.1”:** Realised the new feature was broken. Fixed it now, so increment the patch version

**”1.0.0”:** New major version, so increment the major version, reset the minor and patch versions.

You promise your users not to break this API until you bump to 2.0.0

**”1.0.1”:** A minor fix

**”1.1.0”:** A new feature

**”1.2.0”:** Another new feature

**”2.0.0”:** A new version, which might break compatibility with 1.0. Users are to upgrade cautiously.

If this release is a preview (Alpha, Beta or Release Candidate), you can also include that, with an optional release number:

|  |
| --- |
| major.minor.patch-(alpha/beta/rc).release |

1

Examples:

**”1.0.0-alpha”:** The alpha of 1.0.0 - use with care, things are changing!

#### ”1.0.0-alpha.2”: The 2nd alpha

**”1.0.0-beta”:** Beta - things are settling down, but still subject to change.

**”1.0.0-rc.1”:** The 1st release candidate for 1.0.0 - you shouldn’t be adding any more features now

**”1.0.0-rc.2”:** The 2nd release candidate for 1.0.0 **”1.0.0”:** The final release!

### 11.2.2 Dependencies

As of Haxe 3.1.0, haxelib supports only exact version matching for dependencies. Dependencies are defined as part of the haxelib.json (11.2), with the library name serving as key and the expected version (if required) as value in the format described in Versioning (Section 11.2.1). We have seen an example of this when introducing haxelib.json:

|  |
| --- |
| "dependencies": {  "tink\_macros": "",  "nme": "3.5.5"  } |

1

2

3

4

This adds two dependencies to the given Haxe library:

1. The library “tink macros” can be used in any version. Haxelib will then always try to use the latest version.
2. The library “nme” is required in version “3.5.5”. Haxelib will make sure that this exact version is used, avoiding potential breaking changes with future versions.

## 11.3 extraParams.hxml

If you add a file named extraParams.hxml to your library root (at the same level as haxelib.json), these parameters will be automatically added to the compilation parameters when someone use your library with -lib.

## 11.4 Using Haxelib

If the haxelib command is executed without any arguments, it prints an exhaustive list of all available arguments.

**Chapter 12**

# Target Details

## 12.1 Javascript

### 12.1.1 Loading extern classes using ”require” function

#### Since Haxe 3.2.0

Modern JavaScript platforms, such as Node.js provide a way of loading objects from external modules using the ”require” function. Haxe supports automatic generation of ”require” statements for extern classes.

This feature can be enabled by specifying @:jsRequire metadata for the extern class. If our extern class represents a whole module, we pass a single argument to the @:jsRequire metadata specifying the name of the module to load:

|  |
| --- |
| @:jsRequire("fs") extern class FS { static function readFileSync(path:String, encoding:String):String;  } |

1

2

3

4

5

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

|  |
| --- |
| @:jsRequire("http", "Server") extern class HTTPServer { function new();  } |

1

2

3

4

The second argument is a dotted-path, so we can load sub-objects in any hierarchy.

If we need to load custom JavaScript objects in runtime, a js.Lib.require function can be used. It takes String as its only argument and returns Dynamic, so it is advised to be assigned to a strictly typed variable.

**12.2 Flash**

**12.3 Neko**

**12.4 PHP**

## 12.5 C++

### 12.5.1 Using C++ Defines

* ANDROIDHOST
* ANDROIDNDK DIR
* ANDROIDNDK ROOT
* BINDIR
* DEVELOPER DIR
* HXCPP
* HXCPP 32
* HXCPP COMPILE CACHE
* HXCPP COMPILE THREADS
* HXCPP CONFIG
* HXCPP CYGWIN
* HXCPP DEPENDS OK
* HXCPP EXIT ON ERROR
* HXCPP FORCE PDB SERVER
* HXCPP M32
* HXCPP M64
* HXCPP MINGW
* HXCPP MSVC
* HXCPP MSVC CUSTOM
* HXCPP MSVC VER
* HXCPP NO COLOR
* HXCPP NO COLOUR
* HXCPP VERBOSE
* HXCPP WINXP COMPAT
* IPHONE VER
* LEGACY MACOSX SDK
* LEGACY XCODE LOCATION
* MACOSX VER
* MSVC VER
* NDKV
* NO AUTO MSVC
* PLATFORM
* QNX HOST
* QNX TARGET
* TOOLCHAIN VERSION
* USE GCC FILETYPES
* USE PRECOMPILED HEADERS
* android
* apple
* blackberry • cygwin
* dll import
* dll import include
* dll import link
* emcc
* emscripten
* gph
* hardfp
* haxe ver
* ios
* iphone
* iphoneos
* iphonesim
* linux
* linux host
* mac host
* macos
* mingw
* rpi
* simulator
* tizen
* toolchain • webos
* windows
* windows host
* winrt
* xcompile

**12.5.2 Using C++ Pointers**

**12.6 Java**

## 12.7 C#

1. http://www.ecma-international.org/publications/standards/Ecma-327.htm [↑](#footnote-ref-1)
2. Adobe Virtual Machine [↑](#footnote-ref-2)
3. http://nekovm.org [↑](#footnote-ref-3)
4. http://haxe-foundation.org [↑](#footnote-ref-4)
5. <http://en.wikipedia.org/wiki/Algebraic_data_type> [↑](#footnote-ref-5)
6. http://en.wikipedia.org/wiki/Regular expression [↑](#footnote-ref-6)
7. http://en.wikipedia.org/wiki/JSON [↑](#footnote-ref-7)