The Language to rule them all..

Language Reference

Table of contents

Basic Types

The haXe syntax is Java/ActionScript/C++ like.

A source code file is composed of an optional package name followed by several imports and type declarations. By convention, package names are composed of several identifiers each of which start with a lowercase letter and are separated from one another by periods ".", while type identifiers always start with an uppercase letter.

There are several kinds of types. The two important ones are *classes* and *enums*. Here are some of the basic types as declared in the standard library :

```
enum Void {
}

class Float {
}

class Int extends Float {
}

enum Bool {
    true;
    false;
}

enum Dynamic<T> {
}
```

Let's see each type one by one :

- Void is declared as an *enum*. An enumeration lists a number of valid *constructors*. An empty enumeration such as `Void` does not have any constructor. However, it's still a valid type that can be used.
- Float is a floating point number class. It doesn't have any method so it can be greatly optimized on some platforms.
- Int is an integer. It doesn't have methods either but it inherits from `Float`, so it means that everywhere a `Float` is requested, you can use an `Int`, while the contrary is not true. And that seems pretty correct.
- Bool is an enumeration, like `Void`, but it has two instances `true` and `false`. As you can see, even *standard* types can be defined easily using the haXe type system. It also means you can use it to define your own types.
- Dynamic is an enum with a *type parameter*. We will explain how to use type parameters later in this document.

Syntax

In haXe, all expressions have the same level. It means that you can nest them together recursively without any problem. For example : `foo(if (x == 3) 5 else 8)`. As this example shows, it also means that every expression *returns* a value of a given *type*.

Constants

The following constant values can be used :

```
0; // Int
-134; // Int
0xFF00; // Int

123.0; // Float
.14179; // Float
13e50; // Float
-1e-99; // Float

"hello"; // String
"hello \"world\" !"; // String
'hello "world" !'; // String

true; // Bool
false; // Bool

null; // Unknown<0>

~/[a-z]+/i; // EReg : regular expression
```

You will notice that `null` has a special value that can be used for any type, and has different behavior than `Dynamic`. It will be explained in detail when introducing *type inference*.

Operations

The following operations can be used, in order of priority :

- `v = e` : assign a value to an expression, returns `e`
- `+= -= *= /= %= &= |= ^= <<= >>= >>>=` : assign after performing the corresponding operation
- `e1 || e2` : If `e1` is `true` then `true` else evaluate `e2` . Both `e1` and `e2` must be `Bool`.
- `e1 && e2` : If `e1` is `false` then `false` else evaluate `e2` . Both `e1` and `e2` must be `Bool`.
- `e1...e2` : Build an integer iterator (see later about Iterators).
- `== != > < >= <= === !==` : perform normal or physical comparisons between two expressions sharing a common type. Returns `Bool`.
- `| & ^` : perform bitwise operations between two `Int` expressions. Returns `Int`.
- `<< >> >>>` : perform bitwise shifts between two `Int` expressions. Returns `Int`.
- `e1 + e2` : perform addition. If both expressions are `Int` then return `Int` else if both expressions are either `Int` or `Float` then return `Float` else return `String`.

- `e1 - e2` : perform subtraction between two `Int` or `Float` expressions. Return `Int` if both are `Int` and return `Float` if they are either `Float` and `Int`.
- `e1 * e2` : multiply two numbers, same return type as subtract.
- `e1 / e2` : divide two numbers, return `Float`.
- `e1 % e2` : modulo of two numbers, same return type as subtract.

Unary operations

The following unary operations are available :

- `!` : boolean *not*. Inverse the expression `Bool` value.
- `-` : negative number, change the sign of the `Int` or `Float` value.
- `++` and `----` can be used before or after an expression. When used before, they first increment the corresponding variable and then return the incremented value. When used after, they increment the variable but return the value it had before incrementation. Can only be used with `Int` or `Float` values.
- ~ : ones-complement of an `Int`.

Note: ~ is usually used for 32-bit integers, so it will not provide expected results with Neko 31-bits integers, that is why it does not work on Neko.

Parentheses

Expressions can be delimited with parentheses in order to give a specific priority when performing operations. The type of `( e )` is the same as `e` and they both evaluate to the same value.

Blocks

Blocks can execute several expressions. The syntax of a block is the following :

```
{
    e1;
    e2;
        // ...
    eX;
}
```

A block evaluates to the type and value of the *last expression* of the block. For example :

```
{ f(); x = 124; true; }
```

This block is of type `Bool` and will evaluate to `true`.

As an exception, the empty block `{ }` evaluates to `Void`.

Local Variables

Local variables can be declared in blocks using `var`, as the following samples show:

```
{
    var x;
    var y = 3;
    var z : String;
    var w : String = "";
    var a, b : Bool, c : Int = 0;
}
```

A variable can be declared with an optional type and an optional initial value. If no value is given then the variable is `null` by default. If no type is given, then the variable type is `Unknown` but will still be strictly typed. This will be explained in details when introducing *type inference*.

Several local variables can be declared in the same `var` expression.

Local variables are only defined until the block they're declared in is closed. They can not be accessed outside the block in which they are declared.

Identifiers

When a variable identifier is found, it is *resolved* using the following order :

- local variables, last declared having priority
- class members (current class and inherited fields)
- current class static fields
- enum constructors that have been either declared in this file or *imported*

```
enum Axis {
    x;
    y;
    z;
}

class C {
    static var x : Int;
    var x : Int;

    function new() {
        {
            // x at this point means member variable
            // this.x
            var x : String;
            // x at this point means the local
            // variable
        }
    }

    function f(x : String) {
```

```
            // x at this point means the function
           // parameter
        }

        static function f() {
            // x at this point means the class static
             // variable
        }
    }

    class D {
        function new() {
            // x means the x Axis
        }
    }
```

Type identifiers are resolved according to the imported packages, as we will explain later.

Field access

Object access is done using the traditional dot-notation :

```
o.field
```

Calls

You can call functions using parentheses and commas in order to delimit arguments. You can call methods by using dot access on objects :

```
f(1,2,3);
object.method(1,2,3);
```

New

The `new` keyword is used in expressions to create a class instance. It needs a class name and can take parameters :

```
a = new Array();
s = new String("hello");
```

Arrays

You can create arrays directly from a list of values by using the following syntax :

```
var a : Array<Int> = [1,2,3,4];
```

Please notice that the type Array takes one *type parameter* that is the type of items stored into the Array. This way all operations on arrays are safe. As a consequence, all items in a given Array must be of the same type.

You can read and write into an Array by using the following traditional bracket access :

```
first = a[0];
a[1] = value;
```

The array index must be of type `Int`.

If

Here are some examples of `if` expressions :

```
if (life == 0) destroy();
if (flag) 1 else 2;
```

Here's the generic syntax of `if` expressions :

```
if( expr-cond ) expr-1 [else expr-2]
```

First `expr-cond` is evaluated. It must be of type `Bool`. Then if `true` then `expr-1` is evaluated, otherwise, if there is an `expr-2` then it is evaluated instead.

If there is no `else`, and the `if` expression is false, then the entire expression has type `Void`. If there is an `else`, then `expr-1` and `expr-2` must be of the same type and this will be the type of the `if` expression :

```
var x : Void = if( flag ) destroy();
var y : Int = if( flag ) 1 else 2;
```

In haXe, `if` is similar to the ternary C `a?b:c` syntax.

As an exception, if an `if` block is not supposed to return any value (like in the middle of a Block) then both `expr-1` and `expr-2` can have different types and the `if` block type will be `Void`.

While

While are standard loops that use a precondition or postcondition :

```
while( expr-cond ) expr-loop;
do expr-loop while( expr-cond );
```

For example :

```
var i = 0;
while( i < 10 ) {
    // ...
    i++;
}
```

Or using `do...while` :

```
var i = 0;
do {
    // ...
    i++;
} while( i < 10 );
```

Like with `if`, the `expr-cond` in a while-loop type must be of type `Bool`.

Another useful example will produce a loop to count from 10 to 1:

```
var i = 10;
while( i > 0 ) {
.......
i--;
}
```

For

For loops are little different from traditional C `for` loops. They're actually used for *iterators*, which will be introduced later in this document. Here's an example of a for loop :

```
for( i in 0...a.length ) {
    foo(a[i]);
}
```

Return

In order to exit from a function before the end or to return a value from a function, you can use the `return` expression :

```
function odd( x : Int ) : Bool {
    if( x % 2 != 0 )
        return true;
    return false;
}
```

The `return` expression can be used without argument if the function does not require a value to be returned :

```
function foo() : Void {
    // ...
    if( abort )
        return;
    // ....
}
```

Break and Continue

These two keywords are useful to exit early a `for` or `while` loop or to go to the next iteration of a loop :

```
var i = 0;
while( i < 10 ) {
    if( i == 7 )
        continue; // skip this iteration.
        // do not execute any more statements in this
         // block,
        // BUT go back to evaluating the "while"
         // condition.
    if( flag )
        break; // stop early.
        // Jump out of the "while" loop, and continue
```

```
            // execution with the statement following the
           // while loop.
    }
```

Exceptions

Exceptions are a way to do non-local jumps. You can `throw` an exception and `catch` it from any calling function on the stack :

```
    function foo() {
        // ...
        throw new Error("invalid foo");
    }

    // ...

    try {
        foo();
    } catch( e : Error ) {
        // handle exception
    }
```

There can be several `catch` blocks after a `try`, in order to catch different types of exceptions. They're tested in the order they're declared. Catching `Dynamic` will catch all exceptions :

```
    try {
        foo();
    } catch( e : String ) {
        // handle this kind of error
    } catch( e : Error ) {
        // handle another kind of error
    } catch( e : Dynamic ) {
        // handle all other errors
    }
```

All the `try` and the `catch` expressions must have the same return type except when no value is needed (same as `if`).

Switch

Switches are a way to express multiple `if...else if... else if` test on the same value:

```
    if( v == 0 )
        e1
    else if( v == foo(1) )
        e2
    else if( v == 65 )
        e3
    else
        e4;
```

Will translate to the following `switch` :

```
switch( v ) {
case 0:
    e1;
case foo(1):
    e2;
case 65:
    e3;
default:
    e4;
}
```

Switches in haXe are different from traditional switches : all *cases* are separate expressions so after one case expression is executed the switch block is automatically exited. As a consequence, `break` can't be used in a `switch` and the position of the `default` case is not important.

On some *platforms*, switches on constant values (especially constant integers) might be optimized for better speed.

Switches can also be used on `enums` with a different semantic. It will be explained later in this document.

Local Functions

Local functions are declared using the `function` keyword but they can't have a name. They're *values* just like literal integers or strings :

```
var f = function() { /* ... */ };
f(); // call the function
```

Local functions can access their parameters, the current class statics and also the local variables that were declared before it :

```
var x = 10;
var add = function(n) { x += n; };
add(2);
add(3);
// now x is 15
```

However, local functions declared in methods cannot access the `this` value. You then need to declare a local variable such as `me` :

```
class C {

    var x : Int;

    function f() {
        // WILL NOT COMPILE
        var add = function(n) { this.x += n };
    }

    function f2() {

        // will compile
        var me = this;
        var add = function(n) { me.x += n };
```

```
        }
    }
```

Anonymous Objects

Anonymous objects can be declared using the following syntax :

```
    var o = { age : 26, name : "Tom" };
```

Please note that because of the *type inference*, anonymous objects are also strictly typed.

Type Inference

Type Inference means that the type information is not only checked in the program, it's also *carried* when typing, so it doesn't have to be resolved immediatly. For example a local variable can be declared without any type (it will have the type `Unknown`) and when first used, its type will be set to the corresponding one.

Printing a Type

Anywhere in your program, you can use the `type` operation to know the type of a given expression. At compilation, the `type` operation will be removed and only the expression will remain :

```
    var x : Int = type(0);
```

This will print `Int` at compilation, and compile the same program as if `type` was not used.

This is useful to quickly get a type instead of looking at the class or some documentation.

Local variable inference

Type Inference enables the whole program to be strictly typed without any need to put types everywhere. In particular, local variables does not need to be typed, their types will be inferred when they are first accessed for reading or writing :

```
    var loc;
    type(loc); // print Unknown<0>
    loc = "hello";
    type(loc); // print String
```

Function types inference

Declaring the type of parameter passed to a class method or local function is also optional. The first time the function is used, the type of the parameter will be set to the type it was used with, just like local variables. This can be tricky since it will depend on the order in which the program is executed. Here's an example that shows the problem :

```
    function f( posx ) {
```

```
        // ....
    }
    // ...

    f(134);
    f(12.2); // Error : Float should be Int
```

The first call to `f` sets the type of `posx` to `Int`. The second call to `f` causes a compilation error because `f` is now expecting an `Int`, not a `Float`. However if we reverse the two calls to `f`, the value type is set to `Float` first. A second call using an `Int` does not fail since `Int` is a subtype of `Float`.

```
    function f( posx ) {
        // ....
    }
    // ...

    f(12.2); // Sets the parameter type to Float
    f(134); // Success
```

In this example the two calls are near each other so it's quite easy to understand and fix. In larger programs with more complex cases, fixing such compilation problems can be tricky. The easiest solution is to explicitly set the type of the function. Then the call that was responsible for the problem will be displayed when recompiling.

Drawing from the first example:

```
    function f( posx : Int ) {
        // ....
    }
    // ...

    f(134);
    f(12.2); // Failure will point to this line
```

User Choice

Using type inference is a choice. You can simply not type your variables and functions and let the compiler *infer* the types for you, or you can type all of them in order to have more control on the process. The best is maybe in the middle, by adding some typing in order to improve code documentation and still be able to write quickly some functions without typing everything.

In all cases, and unless you use *dynamics* (they will be introduced later), your program will be strictly typed and any wrong usage will be detected instantly at compilation.

Object Oriented Programming

Classes

We will quickly introduce the structure of classes that you might already be familiar with if you've done some OO programming before :

```
package my.pack;
/*
    this will define the class my.pack.MyClass
*/
class MyClass {
    // ....
}
```

A Class can have several *variables* and *methods*.

```
package my.pack;

class MyClass {

    var id : Int;

    static var name : String = "MyString";

    function foo() : Void {
    }

    static function bar( s : String, v : Bool ) : Void {
    }
}
```

Variables and methods can have the following *flags* :

- static : the field belongs to the Class itself and not to *instances* of this class. Static identifiers can be used directly in the class itself. Outside of the class, it must be used with the class name (for example : `my.pack.MyClass.name`).
- public : the field can be accessed by other classes. **By default, all fields are `private`.**
- private : the field access is restricted to the class itself and to classes that subclass or `extends` it. This ensures that the class internal state is not accessible.

All class variables must be declared with a type (you can use `Dynamic` if you don't know which type to use). Function arguments and return types are optional but are still stricly checked as we will see when introducing *type inference*.

Static variables *can* have an initial value although it's not required.

Constructor

The class can have only one constructor, which is the not-static function called `new`. This is a keyword that can also be used to name a class function :

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

        public function new() {
            this.x = 0;
```

```
            this.y = 0;
        }

    }
```

Constructor parametrization & overloading :

```
         public function new (x : Int, ?y : Int){
             this.x = x;
             this.y = (y == null) ? 0 : y; // "y" is
optional
    }
```

Class Inheritance

When declared, it's possible that a class *extends* one class and *implements* several classes or interfaces. This means the class will inherit several *types* at the same time, and can be treated as such. For example :

```
class D extends A, implements B, implements C {
}
```

Every *instance* of D will have the *type* D but you will also be able to use it where an instance of *type* A, B or C is required. This means that every *instance* of D also has the *types* A , B and C.

Extends

When *extending* a class, your class *inherits* from all *public* and *private* not-static fields. You can then use them in your class as if they where declared here. You can also *override* a method by redefining it with the same number of arguments and types as its *superclass*. Your class *can not* inherit *static* fields.
When a *method* is *overridden*, then you can still access the *superclass* method using `super` :

```
class B extends A {
    function foo() : Int {
        return super.foo() + 1;
    }
}
```

In your class constructor you can call the *superclass* constructor using also `super` :

```
class B extends A {
    function new() {
        super(36,"");
    }
}
```

Implements

When *implementing* a class or an interface, your class is *required* to implement all the fields *declared* or *inherited* by the class it implements, with same type and name. However the field might already be *inherited* from a superclass.

Interfaces

Interface are classes prototypes. They are declared using the `interface` keyword. By default, all interfaces fields are *public*. Interfaces cannot be instantiated.

```
interface PointProto {
    var x : Int;
    var y : Int;
    function length() : Int;
}
```

An interface can also *implements* one or several interfaces :

```
interface PointMore implements PointProto {
    function distanceTo( p : PointProto ) : Float;
}
```

Type Parameters

A class can have several *type parameters* that can be used to get *extensible behavior*. For example, the `Array` class have one type parameter :

```
    class Array<T> {

        function new() {
            // ...
        }

        function get( pos : Int ) : T {
            // ...
        }

        function set( pos : Int, val : T ) : Void {
            // ...
        }
    }
```

Inside the `Array` class, the type `T` is *abstract* and then its fields and methods are not accessible. However when you declare an array you need to specify its type : `Array<Int>` or `Array<String>` for example. This will act the same as if you had replaced all types `T` in `Array` declaration by the type you're specifying.

Type parameter is very useful in order to get strict typing of *containers* such as `Array`, `List`, `Tree`... You can define your own *parameterized* classes with several *type parameters* for your own usage when you need it.

Constraint Parameters

While it's nice to be able to define *abstract* parameters, it is also possible to define several *constraints* on them in order to be able to use them in the class implementation. For example :

```
class EvtQueue<T : (Event, EventDispatcher)> {
    var evt : T;
    // ...
}
```

In this class, the field `evt`, although it's a class parameter, the typer knows that it has both types `Event` and `EventDispatcher` so it can actually access it like if it was *implementing* both classes. Later, when an `EvtQueue` is created, the typer will check that the *type parameter* will either *extends* or *implements* the two types `Event` and `EventDispatcher`. When multiple constraint parameters are defined for a single class parameter, as in the example above, they should be placed within parenthesis in order to disambiguate from cases where more class parameters are to follow.

Type parameters constraints are a powerful advanced feature, that can be really useful to write generic code that can be reused in different applications.

Enums

Enums are another type than *classes* and are declared with a finite number of *constructors*. Here's a small sample :

```
enum Color {
    red;
    green;
    blue;
}

class Colors {
    static function toInt( c : Color ) : Int {
        return switch( c ) {
            case red: 0xFF0000;
            case green: 0x00FF00;
            case blue: 0x0000FF;

        }
    }
}
```

When you want to ensure that only a fixed number of values are used then *enum*s are the best thing since they guarantee that other values cannot be constructed.

Constructors parameters

The previous `Color` sample shows three *constant constructors* for an *enum*. It is also possible to have parameters for constructors :

```
enum Color2 {
    red;
    green;
    blue;
```

```
        grey( v : Int );
        rgb( r : Int, g : Int, b : Int );
    }
```

This way, there is an infinite number of `Color2` possible, but they are five different constructors possible for it. The following values are all `Color2` :

```
    red;
    green;
    blue;
    grey(0);
    grey(128);
    rgb( 0x00, 0x12, 0x23 );
    rgb( 0xFF, 0xAA, 0xBB );
```

We can also have a recursive type, for example to add `alpha` :

```
    enum Color3 {
        red;
        green;
        blue;
        grey( v : Int );
        rgb( r : Int, g : Int, b : Int );
        alpha( a : Int, col : Color3 );
    }
```

The following are valid `Color3` values :

```
    alpha( 127, red );
    alpha( 255, rgb(0,0,0) );
```

Switch on Enum

Switch have a special semantic when used on an *enum*. If there is no `default` case then it will check that all *enum constructor* are used, and you'll get an error if not. For example, using the first `Color` *enum* :

```
    switch( c ) {
        case red: 0xFF0000;
        case green: 0x00FF00;
    }
```

This will cause an *compile error* telling that the constructor `blue` is not used. In that case you can either add a case for it or add a default case that does something. It's very useful since when you add a new constructor to your *enum*, compiler errors will display in your program the places where the new constructor have to be handled.

Switch with Constructor Parameters

If enum constructor have parameters, they *must* be listed as variable names in a *switch* case. This way all the variables will be locally accessible in the `case` expression and correspond to the type of the enum constructor parameter. For example, using the `Color3` enum :

```
    class Colors {
        static function toInt( c : Color3 ) : Int {
            return switch( c ) {
                case red: 0xFF0000;
                case green: 0x00FF00;
                case blue: 0x0000FF;
                case grey(v): (v << 16) | (v << 8) | v;
                case rgb(r,g,b): (r << 16) | (g << 8) |
b;
                case alpha(a,c): (a << 24) | (toInt(c) &
0xFFFFFF);
            }
        }

    }
```

Using `switch` is the only possible way to access the enum constructors parameters.

Enum Type Parameters

Enum, as classes, can also have type parameters. The syntax is the same so here's a small sample of a parameterized linked `List` using an *enum* to store the cells :

```
    enum Cell<T> {
        empty;
        cons( item : T, next : Cell<T> );
    }

    class List<T> {
        var head : Cell<T>;

        public function new() {
            head = empty;
        }

        public function add( item : T ) {
            head = cons(item,head);
        }

        public function length() : Int {
            return cell_length(head);
        }

      private function cell_length( c : Cell<T> ) : Int {
            return switch( c ) {
            case empty : 0;
            case cons(item,next): 1 + cell_length(next);
            }
        }

    }
```

Using both *enum* and *classes* together can be pretty powerful in some cases.

Packages and Imports

Each file can contain several *classes*, *enums* and *imports*. They are all part of the *package* declared at the beginning of the class. If *package* is not declared than the default empty package is used. Each *type* has then a *path* corresponding to the *package* name followed by the *type* name.

```
// file my/pack/C.hx
package my.pack;

enum E {
}

class C {
}
```

This file declares two *types* : `my.pack.E` and `my.pack.C`. It's possible to have several classes in the same file, but the type name must be *unique* in the whole application, so conflicts can appear if you're not using packages enough (this does not mean that you have to use long packages names everywhere).

When using packages, your files should be placed into subdirectories having the same name of it. In general the name of the file is the one of the main *class* defined into it.

The file extension for haXe is `.hx`.

Imports

Imports can be used to have access to all the types of a file without needing to specify the package name.

```
package my.pack2;
class C2 extends my.pack.C {
}
```

Is identical to the following :

```
package my.pack2;
import my.pack.C;

class C2 extends C {
}
```

The only difference is that when using import you can use *enum constructors* that were declared in the `my/pack/C.hx` file.

Type Lookup

When a type name is found, the following lookup is performed, in this order:

- current class type parameters
- standard types

- types declared in the current file
- types declared in imported files (if the searched package is empty)
- if not found, the corresponding file is loaded and the type is searched inside it

Enum Constructors

In order to use enum constructors, the file in which the *enum* is declared must first be imported, or you can use the full type path to access constructors as if they were static fields of the *enum* type.

```
var c : my.pack.Color = my.pack.Color.red;
```

As an exception, in `switch`: if the type of the *enum* is known at compile time, then you can use constructors directly without the need to import.

Dynamic

When you want to get some *dynamically typed* behavior and break free from the type system, you can use the `Dynamic` type which can be used in place of *any* type without any compiler type-checking:

```
var x : Dynamic = "";
x = true;
x = 1.744;
x = new Array();
```

Also, a `Dynamic` variable has an infinite number of fields, all having the type `Dynamic`; it can be used as an `Array` for bracket syntax, etc...

While this can be useful sometimes, please be careful not to break your program safety by using too many dynamic variables.

Note that an *untyped* variable is of type `Unknown` and not `Dynamic`. That is, an *untyped* variable does not have a type until it is determined by *type inference*. A `Dynamic` variable has a type, an *any* type.

Dynamic Parameter

As it was said when listing standard library types, `Dynamic` *can* also take a type parameter. When you use `Dynamic<String>`, it has different behavior.

`Dynamic<String>` cannot be used in place of any other type. However, it has an infinite number of fields which all have the type `String`. This is useful to encode Hashtables where items are accessed using *dot* syntax :

```
var att : Dynamic<String> = xml.attributes;
x.name = "Nicolas";
x.age = "26";
// ...
```

Implementing Dynamic

Any class can also *implement* `Dynamic` with or without a type parameter. In the first case, the class fields are typed when they exist, otherwise they have a dynamic type:

```
class C implements Dynamic<Int> {
    public var name : String;
    public var address : String;
}
// ...
var c = new C();
var n : String = c.name; // ok
var a : String = c.address; // ok
var i : Int = c.phone; // ok : use Dynamic
var c : String = c.country; // ERROR
// c.country is an Int because of Dynamic<Int>
```

Dynamic behavior is *inherited* by subclasses. When several classes are implementing different `Dynamic` types in a class hierarchy, the last Dynamic definition is used.

Type Casting

You can cast from one type to another by using the `cast` keyword.

```
     var a : A = ....
     var b : B = cast(a,B);
```

This will either return the value `a` with the type `B` if `a` is an instance of `B` or it will throw an exception `"Class cast error"`.

Untyped

One other way to do dynamic things is to use the `untyped` keyword. When an expression is said untyped, no type-check will be done so you can do a lot of dynamic operations at once :

```
    untyped { a["hello"] = 0; }
```

Be careful to use untyped expressions only when you really need them and when you know what you're doing.

Unsafe Cast

Untyped is pretty powerful but it allows all kind of invalid syntax on the right side of the `untyped` keyword. One other possibility is to do an unsafe `cast` which is similar to the standard `cast` except that no type is specified. That means that the `cast` call will not result in any runtime check, but will allow you to "loose" one type.

```
    var y : B = cast 0;
```

Cast is somehow the equivalent of storing a value in a temporary `Dynamic` variable :

```
    var tmp : Dynamic = 0;
    var y : B = tmp;
```

Advanced Types

Up to now, we have seen the following types :

- class instance
- enum instance
- dynamic

There are several additional important types.

Anonymous

An anonymous type is the type of an anonymously declared object. It is also the type of a Class identifier (corresponding to all the static fields) or an Enum identifier (listing all the constructors). Here's an example that shows it.

```
enum State {
    on;
    off;
    disable;
}

class C {
    static var x : Int;
    static var y : String;

    function f() {
        // print { id : Int, city : String }
        type({ id : 125, city : "Kyoto" });
    }

    function g() {
        // print { on : State, off : State, disable : State }
        type(State);
    }

    function h() {
        // print { x : Int, y : String }
        type(C);
    }
}
```

Anonymous types are structural, so it's possible to have more fields in the value than in the type :

```
var p : { x : Int, y : Int } = { x : 0, y : 33, z : -45 };
```

Typedef

You can define type definitions which are some kind of type-shortcut that can be used to give a name to an anonymous type or a long type that you don't want to repeat everywhere in your program :

```
typedef User = {
    var age : Int;
    var name : String;
}
// ....
var u : User = { age : 26, name : "Tom" };


// PointCube is a 3-dimensional array of points
typedef PointCube = Array<Array<Array<Point>>>
```

Typedef are not classes, they are only used for typing.

Functions

When you want to define function types to use them as variables, you can define them by listing the arguments followed by the return type and separated with arrows. For example `Int -> Void` is the type of a function taking an Int as argument and returning Void. And `Color -> Color ->Int` takes two Color arguments and returns an Int.

```
    class C {
        function f(x : String) : Int {
            // ...
        }

        function g() {
            type(f); // print String -> Int
            var ftype : String -> String = f;
            // ERROR : should be String -> Int
        }
    }
```

Unknown

When a type is not declared, it is used with the type `Unknown`. The first time it is used with another type, it will change to it. This was explained in more details in *type inference*. The `id` printed with the `Unknown` type is used to differentiate several unknowns when printing a complex type.

```
    function f() {
        var x;
        type(x); // print Unknown<0>
        x = 0;
        type(x); // print Int
    }
```

The diversity of types expressible with haXe enable more powerful models of programming by providing high-level abstractions that don't need complex classes relationships to be used.

Extensions

Extensions can be used to extend either a typedef representing an anonymous type or to extend a class on-the-fly.

Here's an example of anonymous typedef extension :

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

// define 'p' as a Point with an additional field z
var p : {> Point, z : Int }
p = { x : 0, y : 0, z : 0 }; // works
p = { x : 0, y : 0 }; // fails
```

For classes, since they don't define types, you need to use a `cast` when assigning, it's unsafe so be careful :

```
var p : {> flash.MovieClip, tf : flash.TextField };
p = flash.Lib._root; // fails
p = cast flash.Lib._root; // works, but no typecheck !
```

You can also use extensions to create *cascading typedefs* :

```
typedef Point = {
    var x : Int;
    var y : Int;
}
typedef Point3D = {> Point,
    var z : Int;
}
```

In that case, every `Point3D` will of course be a `Point` as well.

Iterators

An iterator is an object which follow the `Iterator` typedef (The type `T` is the iterated type) :

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

You can use the `for` syntax in order to execute iterators. The most simple iterator is the `IntIter` iterator which can easily be built using the operator `...` (three dots). For example this will list all numbers from 0 to 9 :

```
for( i in 0...10 ) {
    // ...
}
```

Or the usual `for` loop :

```
for( i in 0...arr.length ) {
```

```
        foo(arr[i]);
    }
```

You don't need to declare the variable `i` before using a for, since it will be automatically declared. This variable will only be available inside the `for` loop.

Implementing Iterator

You can also define you own iterators. You can simply follow the `Iterator` typedef in your class by implementing the `hasNext` and `next` methods. Here's for example the `IntIter` class that is part of the standard library :

```
class IntIter {
    var min : Int;
    var max : Int;

    public function new( min : Int, max : Int ) {
        this.min = min;
        this.max = max;
    }

    public function hasNext() {
        return( min < max );
    }

    public function next() {
        return min++;
    }
}
```

Once your iterator is implemented, you can simply use it with the `for...in` syntax, this way :

```
    var iter = new IntIter(0,10);
    for( i in iter ) {
        // ...
    }
```

The variable name in the `for` is automatically declared and its
type is bound to the iterator type. It is not accessible after the iteration is done.

Iterable Objects

If an object has a method `iterator()` taking no arguments and returning an iterator, it is said *iterable*. It doesn't have to implement any type. You can use such class directly into a `for` expression without the need to call the `iterator()` method :

```
    var a : Array<String> =
["hello","world","I","love","haXe","!"];
    for( txt in a ) {
        tf.text += txt + " ";
    }
```

This sample will build the string by listing an array elements using an iterator. It is same as calling `a.iterator()` in the `for` expression.

Properties

Properties are a specific way to declare class fields, and can be used to implement several kind of features such as read-only/write-only fields or fields accessed through getter-setter methods.

Here's a property declaration example :

```
class C {

 public var x(getter,setter) : Int;

}
```

The values for `getter` and `setter` can be one of the following :

- a method name that will be used as getter/setter
- `null` if the access is restricted
- `default` if the access is a classic field access
- `dynamic` if the access is done through a runtime-generated method

Sample

Here's a complete example :

```
class C {
   public var ro(default,null) : Int;
   public var wo(null,default) : Int;
   public var x(getX,setX) : Int;

   private var my_x : Int;

   private function getX() {
       return my_x;
   }

   private function setX( v : Int ) {
      if( v >= 0 )
         my_x = v;
      return my_x;
   }

}
```

Using properties declaration, we declare three public fields in the class `C` :

- outside the class code, the field `ro` is read only
- outside the class code, the field `wo` is write only

- [x] the field `x` is accessed through a pair of getter/setter methods

For instance, the following two functions are equivalent, although the methods `getX` and `setX` are private and then can't be accessed directly as in `f2` :

```
    var c : C;
    function f1() {
        c.x *= 2;
    }
    function f2() {
        c.setX(c.getX()*2);
    }
```

Important Remarks

It's important to know that such features are only working if the type of the class is known. **There is no runtime properties handling**, so for instance the following code will always trace `null` since the method `getX` will **never** be called :

```
    var c : Dynamic = new C();
    trace(c.x);
```

The same goes for read-only and write-only properties. They can always be modified if the type of the class is unknown.

Also, please note that you have to return a value from the setter function. The compiler will complain otherwise.

Dynamic Property

The additional `dynamic` access can be used to add methods at runtime, it's quite a specific feature that should be used with care. When a `dynamic` field `x` is accessed for reading, the `get_x` method is called, when accessed for writing the `set_x` method is called :

```
class C {
    public var age(dynamic,dynamic) : Int;
    public function new() {
    }
}

class Test {
    static function main() {
        var c = new C();
        var my_age = 26;
        Reflect.setField(c,"get_age",function() { return
my_age; });
        Reflect.setField(c,"set_age",function(a) { my_age
= a; return my_age; });
        trace(c.age); // 26
        c.age++; // will call c.set_age(c.get_age()+1)
        trace(c.age); // 27
```

```
    }
}
```

Optional Arguments

Some function parameters can be made optional by using a question mark `?` before the parameter name :

```
class Test {
    static function foo( x : Int, ?y : Int ) {
        trace(x+","+y);
    }
    static function main() {
        foo(1,2); // trace 1,2
        foo(3); // trace 3,null
    }
}
```

Default values

When not specified, an optional parameter will have the default value `null`. It's also possible to define another default value, by using the following syntax :

```
    static function foo( x : Int, ?y : Int = 5 ) {
        // ...
    }
```

Thanks to type inference, you can also shorten the syntax to the following :

```
    static function foo( x : Int, y = 5 ) {
        // ...
    }
```

Ordering

Although it is advised to put optional parameters at the end of the function parameters, you can use them in the beginning or in the middle also.

Also, optional parameters are *independent* in haXe. It means that one optional parameter can be used without providing a previous one :

```
    function foo( ?x : A, ?y : B ) {
    }

    foo(new A()); // same as foo(new A(),null);
    foo(new B()); // same as foo(null, new B());
    foo(); // same as foo(null,null);
    foo(new C()); // compile-time error
    foo(new B(),new A()); // error : the order must be
preserved
```

However, such usages of optional arguments could be considered quite advanced.

Conditional Compilation

Sometimes you might want to have a single library using specific API depending on the platform it is compiled on. At some other time, you might want to do some optimizations only if you turn a flag ON. For all that, you can use *conditional compilation macros* (AKA preprocessor macros):

Here's an example of multiplaform code :

```
    #if flash8
    // haXe code specific for flash player 8
    #elseif flash
    // haXe code specific for flash platform (any
version)
    #elseif js
    // haXe code specific for javascript plaform
    #elseif neko
    // haXe code specific for neko plaform
    #else
    // do something else
        #error  // will display an error "Not implemented
on this platform"
    #end
```

Here's another example for turning on some logs only if `mydebug` flag is used when compiling the code (using -D mydebug):

```
    #if mydebug
    trace("Some debug infos");
    #end
```

You can define your own variables by using the **haXe** compiler commandline options.

Inline

The *inline* keyword can be used in two ways: for static variables and for any kind of method.

Inlining Static Variables

For static variables, it's pretty simple. Every time the variable is used, its value will be used instead of the variable access. Example :

```
class Test {
    static inline var WIDTH = 500;
    static function main() {
        trace(WIDTH);
    }
}
```

Using "inline" adds a few restrictions :

- the variable must be initialized when declared
- the variable value cannot be modified

The main advantage of using "inline" is that you can use as many variables as you want without slowing down your code with these variables accesses since the value is directly replaced in the compiled/generated code.

Inlining Methods

The principle is the same for a method. The less expensive function call is the one that is never done. In order to achieve that for small methods that are often called, you can "inline" the method body a the place the method is called.

Let's have a look at one example :

```
class Point {
   public var x : Float;
   public var y : Float;
   public function new(x,y) { this.x = x; this.y = y; }
   public inline function add(x2,y2) { return new Point(x
+x2,y+y2); }
}
class Main {
   static function main() {
       var p = new Point(1,2);
       var p2 = p.add(2,3);
       // is the same as writing :
       var p2 = new Point(p.x+2,p.y+3);
   }
}
```

Again, there's some limitations to inline functions :

- they cannot be redefined at runtime
- they cannot be overridden in subclasses
- an function containing "super" accesses or declare another function cannot be inlined
- if the inline function has arguments, then the arguments evaluation order is undefined and some arguments might even be not evaluated, which is good unless they have some expected side-effects
- if the inline arguments are modified in the inline function, then they might be modified as well in the original function, for example :

```
inline function setX(x) { x = 3; }
inline function foo() {
    var z = 0;
    setX(z);
    trace(z); // 3
}
```

- if the inline returns a value, then only "final returns" are accepted, for example :

```
inline function foo(flag) { return flag?0:1; } //
accepted
inline function bar(flag) { if( flag ) return 0; else
return 1; } // accepted
```

```
inline function baz(flag) { if( flag ) return 0;
return 1; } // refused
```

Apart from these few restrictions, using inline increase the compiled/generated codesize but brings a lot of additional speed for small methods. Please note that it's also possible to inline static methods.