Haxe

Haxe is a web-oriented language that provides platform support for C++, C#, Swf/ActionScript, Javascript, Java, and Neko byte code (also written by the Haxe author). Note that this guide is for Haxe version 3. Some of the guide may be applicable to older versions, but it is recommended to use other references.

```
/*
   Welcome to Learn Haxe 3 in 15 minutes.  http://www.haxe.org
   This is an executable tutorial.  You can compile and run it using the haxe
   compiler, while in the same directory as LearnHaxe.hx:
   $> haxe -main LearnHaxe3 -x out

   Look for the slash-star marks surrounding these paragraphs.  We are inside
   a "Multiline comment".  We can leave some notes here that will get ignored
   by the compiler.

   Multiline comments are also used to generate javadoc-style documentation for
   haxedoc.  They will be used for haxedoc if they immediately precede a class,
   class function, or class variable.

 */

// Double slashes like this will give a single-line comment


/*
   This is your first actual haxe code coming up, it's declaring an empty
   package.  A package isn't necessary, but it's useful if you want to create a
   namespace for your code (e.g. org.yourapp.ClassName).

   Omitting package declaration is the same as declaring an empty package.
 */
package; // empty package, no namespace.

/*
   Packages are directories that contain modules. Each module is a .hx file
   that contains types defined in a package. Package names (e.g. org.yourapp)
   must be lower case while module names are capitalized. A module contain one
   or more types whose names are also capitalized.

   E.g, the class "org.yourapp.Foo" should have the folder structure org/module/Foo.hx,
   as accessible from the compiler's working directory or class path.

   If you import code from other files, it must be declared before the rest of
   the code.  Haxe provides a lot of common default classes to get you started:
 */
import haxe.ds.ArraySort;

// you can import many classes/modules at once with "*"
import haxe.ds.*;

// you can import static fields
import Lambda.array;
```

```haxe
// you can also use "*" to import all static fields
import Math.*;

/*
   You can also import classes in a special way, enabling them to extend the
   functionality of other classes like a "mixin".  More on 'using' later.
 */
using StringTools;

/*
   Typedefs are like variables... for types.  They must be declared before any
   code.  More on this later.
 */
typedef FooString = String;

// Typedefs can also reference "structural" types, more on that later as well.
typedef FooObject = { foo: String };

/*
   Here's the class definition.  It's the main class for the file, since it has
   the same name (LearnHaxe3).
 */
class LearnHaxe3{
    /*
       If you want certain code to run automatically, you need to put it in
       a static main function, and specify the class in the compiler arguments.
       In this case, we've specified the "LearnHaxe3" class in the compiler
       arguments above.
     */
    static function main(){

        /*
           Trace is the default method of printing haxe expressions to the
           screen.  Different targets will have different methods of
           accomplishing this.  E.g., java, c++, c#, etc. will print to std
           out.  Javascript will print to console.log, and flash will print to
           an embedded TextField.  All traces come with a default newline.
           Finally, It's possible to prevent traces from showing by using the
           "--no-traces" argument on the compiler.
         */
        trace("Hello World, with trace()!");

        /*
           Trace can handle any type of value or object.  It will try to print
           a representation of the expression as best it can.  You can also
           concatenate strings with the "+" operator:
         */
        trace( " Integer: " + 10 + " Float: " + 3.14 + " Boolean: " + true);

        /*
           In Haxe, it's required to separate expressions in the same block with
           semicolons.  But, you can put two expressions on one line:
         */
        trace('two expressions..'); trace('one line');
```

```
////////////////////////////////////////////////////////////////////
// Types & Variables
////////////////////////////////////////////////////////////////////
trace("***Types & Variables***");

/*
   You can save values and references to data structures using the
   "var" keyword:
 */
var an_integer:Int = 1;
trace(an_integer + " is the value for an_integer");


/*
   Haxe is statically typed, so "an_integer" is declared to have an
   "Int" type, and the rest of the expression assigns the value "1" to
   it.  It's not necessary to declare the type in many cases.  Here,
   the haxe compiler is inferring that the type of another_integer
   should be "Int".
 */
var another_integer = 2;
trace(another_integer + " is the value for another_integer");

// The $type() method prints the type that the compiler assigns:
$type(another_integer);

// You can also represent integers with hexadecimal:
var hex_integer = 0xffffff;

/*
   Haxe uses platform precision for Int and Float sizes.  It also
   uses the platform behavior for overflow.
   (Other numeric types and behavior are possible using special
   libraries)
 */

/*
   In addition to simple values like Integers, Floats, and Booleans,
   Haxe provides standard library implementations for common data
   structures like strings, arrays, lists, and maps:
 */

var a_string = "some" + 'string';  // strings can have double or single quotes
trace(a_string + " is the value for a_string");

/*
   Strings can be "interpolated" by inserting variables into specific
   positions.  The string must be single quoted, and the variable must
   be preceded with "$".  Expressions can be enclosed in ${...}.
 */
var x = 1;
var an_interpolated_string = 'the value of x is $x';
```

```
var another_interpolated_string = 'the value of x + 1 is ${x + 1}';

/*
   Strings are immutable, instance methods will return a copy of
   parts or all of the string.
   (See also the StringBuf class).
 */
var a_sub_string = a_string.substr(0,4);
trace(a_sub_string + " is the value for a_sub_string");

/*
   Regexes are also supported, but there's not enough space to go into
   much detail.
 */
var re = ~/foobar/;
trace(re.match('foo') + " is the value for (~/foobar/.match('foo')))");

/*
   Arrays are zero-indexed, dynamic, and mutable.  Missing values are
   defined as null.
 */
var a = new Array<String>(); // an array that contains Strings
a[0] = 'foo';
trace(a.length + " is the value for a.length");
a[9] = 'bar';
trace(a.length + " is the value for a.length (after modification)");
trace(a[3] + " is the value for a[3]"); //null

/*
   Arrays are *generic*, so you can indicate which values they contain
   with a type parameter:
 */
var a2 = new Array<Int>(); // an array of Ints
var a3 = new Array<Array<String>>(); // an Array of Arrays (of Strings).

/*
   Maps are simple key/value data structures.  The key and the value
   can be of any type.
 */
var m = new Map<String, Int>();  // The keys are strings, the values are Ints.
m.set('foo', 4);
// You can also use array notation;
m['bar'] = 5;
trace(m.exists('bar') + " is the value for m.exists('bar')");
trace(m.get('bar') + " is the value for m.get('bar')");
trace(m['bar'] + " is the value for m['bar']");

var m2 =  ['foo' => 4, 'baz' => 6]; // Alternative map syntax
trace(m2 + " is the value for m2");

/*
   Remember, you can use type inference.  The Haxe compiler will
   decide the type of the variable the first time you pass an
   argument that sets a type parameter.
```

```
 */
var m3 = new Map();
m3.set(6, 'baz'); // m3 is now a Map<Int,String>
trace(m3 + " is the value for m3");

/*
   Haxe has some more common datastructures in the haxe.ds module, such as
   List, Stack, and BalancedTree
 */


//////////////////////////////////////////////////////////////////
// Operators
//////////////////////////////////////////////////////////////////
trace("***OPERATORS***");

// basic arithmetic
trace((4 + 3) + " is the value for (4 + 3)");
trace((5 - 1) + " is the value for (5 - 1)");
trace((2 * 4) + " is the value for (2 * 4)");
trace((8 / 3) + " is the value for (8 / 3) (division always produces Floats)");
trace((12 % 4) + " is the value for (12 % 4)");


//basic comparison
trace((3 == 2) + " is the value for 3 == 2");
trace((3 != 2) + " is the value for 3 != 2");
trace((3 >  2) + " is the value for 3 > 2");
trace((3 <  2) + " is the value for 3 < 2");
trace((3 >= 2) + " is the value for 3 >= 2");
trace((3 <= 2) + " is the value for 3 <= 2");

// standard bitwise operators
/*
~       Unary bitwise complement
<<      Signed left shift
>>      Signed right shift
>>>     Unsigned right shift
&       Bitwise AND
^       Bitwise exclusive OR
|       Bitwise inclusive OR
*/

//increments
var i = 0;
trace("Increments and decrements");
trace(i++); //i = 1. Post-Incrementation
trace(++i); //i = 2. Pre-Incrementation
trace(i--); //i = 1. Post-Decrementation
trace(--i); //i = 0. Pre-Decrementation

//////////////////////////////////////////////////////////////////
// Control Structures
//////////////////////////////////////////////////////////////////
```

```
        trace("***CONTROL STRUCTURES***");

        // if statements
        var j = 10;
        if (j == 10){
            trace("this is printed");
        } else if (j > 10){
            trace("not greater than 10, so not printed");
        } else {
            trace("also not printed.");
        }

        // there is also a "ternary" if:
        (j == 10) ?  trace("equals 10") : trace("not equals 10");

        /*
           Finally, there is another form of control structures that operates
           at compile time:  conditional compilation.
         */
#if neko
        trace('hello from neko');
#elseif js
        trace('hello from js');
#else
        trace('hello from another platform!');
#end
        /*
           The compiled code will change depending on the platform target.
           Since we're compiling for neko (-x or -neko), we only get the neko
           greeting.
         */


        trace("Looping and Iteration");

        // while loop
        var k = 0;
        while(k < 100){
            // trace(counter); // will print out numbers 0-99
            k++;
        }

        // do-while loop
        var  l = 0;
        do{
            trace("do statement always runs at least once");
        } while (l > 0);

        // for loop
        /*
           There is no c-style for loop in Haxe, because they are prone
           to error, and not necessary.  Instead, Haxe has a much simpler
           and safer version that uses Iterators (more on those later).
         */
```

```
var m = [1,2,3];
for (val in m){
    trace(val + " is the value for val in the m array");
}

// Note that you can iterate on an index using a range
// (more on ranges later as well)
var n = ['foo', 'bar', 'baz'];
for (val in 0...n.length){
    trace(val + " is the value for val (an index for n)");
}


trace("Array Comprehensions");

// Array comprehensions give you the ability to iterate over arrays
// while also creating filters and modifications.
var filtered_n = [for (val in n) if (val != "foo") val];
trace(filtered_n + " is the value for filtered_n");

var modified_n = [for (val in n) val += '!'];
trace(modified_n + " is the value for modified_n");

var filtered_and_modified_n = [for (val in n) if (val != "foo") val += "!"];
trace(filtered_and_modified_n + " is the value for filtered_and_modified_n");

////////////////////////////////////////////////////////////////////
// Switch Statements (Value Type)
////////////////////////////////////////////////////////////////////
trace("***SWITCH STATEMENTS (VALUE TYPES)***");

/*
   Switch statements in Haxe are very powerful.  In addition to working
   on basic values like strings and ints, they can also work on the
   generalized algebraic data types in enums (more on enums later).
   Here's some basic value examples for now:
 */
var my_dog_name = "fido";
var favorite_thing  = "";
switch(my_dog_name){
    case "fido" : favorite_thing = "bone";
    case "rex"  : favorite_thing = "shoe";
    case "spot" : favorite_thing = "tennis ball";
    default     : favorite_thing = "some unknown treat";
    // case _   : favorite_thing = "some unknown treat"; // same as default
}
// The "_" case above is a "wildcard" value
// that will match anything.

trace("My dog's name is" + my_dog_name
        + ", and his favorite thing is a: "
        + favorite_thing);

////////////////////////////////////////////////////////////////////
```

```haxe
// Expression Statements
////////////////////////////////////////////////////////////////
trace("***EXPRESSION STATEMENTS***");

/*
   Haxe control statements are very powerful because every statement
   is also an expression, consider:
*/

// if statements
var k = if (true) 10 else 20;

trace("k equals ", k); // outputs 10

var other_favorite_thing = switch(my_dog_name) {
    case "fido" : "teddy";
    case "rex"  : "stick";
    case "spot" : "football";
    default     : "some unknown treat";
}

trace("My dog's name is" + my_dog_name
        + ", and his other favorite thing is a: "
        + other_favorite_thing);

////////////////////////////////////////////////////////////////
// Converting Value Types
////////////////////////////////////////////////////////////////
trace("***CONVERTING VALUE TYPES***");

// You can convert strings to ints fairly easily.

// string to integer
Std.parseInt("0"); // returns 0
Std.parseFloat("0.4"); // returns 0.4;

// integer to string
Std.string(0); // returns "0";
// concatenation with strings will auto-convert to string.
0 + "";  // returns "0";
true + ""; // returns "true";
// See documentation for parsing in Std for more details.


////////////////////////////////////////////////////////////////
// Dealing with Types
////////////////////////////////////////////////////////////////

/*

   As mentioned before, Haxe is a statically typed language.  All in
   all, static typing is a wonderful thing.  It enables
   precise autocompletions, and can be used to thoroughly check the
   correctness of a program.  Plus, the Haxe compiler is super fast.
```

```
   *HOWEVER*, there are times when you just wish the compiler would let
   something slide, and not throw a type error in a given case.

   To do this, Haxe has two separate keywords.  The first is the
   "Dynamic" type:
 */
var dyn: Dynamic = "any type of variable, such as this string";

/*
   All that you know for certain with a Dynamic variable is that the
   compiler will no longer worry about what type it is. It is like a
   wildcard variable:  You can pass it instead of any variable type,
   and you can assign any variable type you want.

   The other more extreme option is the "untyped" keyword:
 */

    untyped {
        var x:Int = 'foo'; // this can't be right!
        var y:String = 4; // madness!
    }

/*
   The untyped keyword operates on entire *blocks* of code, skipping
   any type checks that might be otherwise required. This keyword should
   be used very sparingly, such as in limited conditionally-compiled
   situations where type checking is a hinderance.

   In general, skipping type checks is *not* recommended.  Use the
   enum, inheritance, or structural type models in order to help ensure
   the correctness of your program.  Only when you're certain that none
   of the type models work should you resort to "Dynamic" or "untyped".
 */

//////////////////////////////////////////////////////////////////
// Basic Object Oriented Programming
//////////////////////////////////////////////////////////////////
trace("***BASIC OBJECT ORIENTED PROGRAMMING***");


/*
   Create an instance of FooClass.  The classes for this are at the
   end of the file.
 */
var foo_instance = new FooClass(3);

// read the public variable normally
trace(foo_instance.public_any + " is the value for foo_instance.public_any");

// we can read this variable
trace(foo_instance.public_read + " is the value for foo_instance.public_read");
// but not write it
// foo_instance.public_read = 4; // this will throw an error if uncommented:
```

```
        // trace(foo_instance.public_write); // as will this.

        // calls the toString method:
        trace(foo_instance + " is the value for foo_instance");
        // same thing:
        trace(foo_instance.toString() + " is the value for foo_instance.toString()");


        /*
           The foo_instance has the "FooClass" type, while acceptBarInstance
           has the BarClass type.  However, since FooClass extends BarClass, it
           is accepted.
         */
        BarClass.acceptBarInstance(foo_instance);

        /*
           The classes below have some more advanced examples, the "example()"
           method will just run them here.
         */
        SimpleEnumTest.example();
        ComplexEnumTest.example();
        TypedefsAndStructuralTypes.example();
        UsingExample.example();

    }

}

/*
   This is the "child class" of the main LearnHaxe3 Class
 */
class FooClass extends BarClass implements BarInterface{
    public var public_any:Int; // public variables are accessible anywhere
    public var public_read (default, null): Int; // enable only public read
    public var public_write (null, default): Int; // or only public write
    public var property (get, set): Int; // use this style to enable getters/setters

    // private variables are not available outside the class.
    // see @:allow for ways around this.
    var _private:Int; // variables are private if they are not marked public

    // a public constructor
    public function new(arg:Int){
        // call the constructor of the parent object, since we extended BarClass:
        super();

        this.public_any = 0;
        this._private = arg;

    }

    // getter for _private
    function get_property() : Int {
        return _private;
```

```
    }

    // setter for _private
    function set_property(val:Int) : Int {
        _private = val;
        return val;
    }

    // special function that is called whenever an instance is cast to a string.
    public function toString(){
        return _private + " with toString() method!";
    }

    // this class needs to have this function defined, since it implements
    // the BarInterface interface.
    public function baseFunction(x: Int) : String{
        // convert the int to string automatically
        return x + " was passed into baseFunction!";
    }
}

/*
    A simple class to extend
*/
class BarClass {
    var base_variable:Int;
    public function new(){
        base_variable = 4;
    }
    public static function acceptBarInstance(b:BarClass){
    }
}

/*
    A simple interface to implement
*/
interface BarInterface{
    public function baseFunction(x:Int):String;
}

////////////////////////////////////////////////////////////////
// Enums and Switch Statements
////////////////////////////////////////////////////////////////

/*
   Enums in Haxe are very powerful.  In their simplest form, enums
   are a type with a limited number of states:
 */

enum SimpleEnum {
    Foo;
    Bar;
    Baz;
}
```

```
//   Here's a class that uses it:

class SimpleEnumTest{
    public static function example(){
        var e_explicit:SimpleEnum = SimpleEnum.Foo; // you can specify the "full" name
        var e = Foo; // but inference will work as well.
        switch(e){
            case Foo: trace("e was Foo");
            case Bar: trace("e was Bar");
            case Baz: trace("e was Baz"); // comment this line to throw an error.
        }

        /*
           This doesn't seem so different from simple value switches on strings.
           However, if we don't include *all* of the states, the compiler will
           complain.  You can try it by commenting out a line above.

           You can also specify a default for enum switches as well:
         */
        switch(e){
            case Foo: trace("e was Foo again");
            default : trace("default works here too");
        }
    }
}

/*
    Enums go much further than simple states, we can also enumerate
    *constructors*, but we'll need a more complex enum example
 */
enum ComplexEnum{
    IntEnum(i:Int);
    MultiEnum(i:Int, j:String, k:Float);
    SimpleEnumEnum(s:SimpleEnum);
    ComplexEnumEnum(c:ComplexEnum);
}
// Note: The enum above can include *other* enums as well, including itself!
// Note: This is what's called *Algebraic data type* in some other languages.

class ComplexEnumTest{
    public static function example(){
        var e1:ComplexEnum = IntEnum(4); // specifying the enum parameter
        /*
           Now we can switch on the enum, as well as extract any parameters
           it might of had.
         */
        switch(e1){
            case IntEnum(x) : trace('$x was the parameter passed to e1');
            default: trace("Shouldn't be printed");
        }

        // another parameter here that is itself an enum... an enum enum?
        var e2 = SimpleEnumEnum(Foo);
```

```
        switch(e2){
            case SimpleEnumEnum(s): trace('$s was the parameter passed to e2');
            default: trace("Shouldn't be printed");
        }

        // enums all the way down
        var e3 = ComplexEnumEnum(ComplexEnumEnum(MultiEnum(4, 'hi', 4.3)));
        switch(e3){
            // You can look for certain nested enums by specifying them explicitly:
            case ComplexEnumEnum(ComplexEnumEnum(MultiEnum(i,j,k))) : {
                trace('$i, $j, and $k were passed into this nested monster');
            }
            default: trace("Shouldn't be printed");
        }
        /*
           Check out "generalized algebraic data types" (GADT) for more details
           on why these are so great.
         */
    }
}

class TypedefsAndStructuralTypes {
    public static function example(){
        /*
           Here we're going to use typedef types, instead of base types.
           At the top we've declared the type "FooString" to mean a "String" type.
         */
        var t1:FooString = "some string";

        /*
           We can use typedefs for "structural types" as well.  These types are
           defined by their field structure, not by class inheritance.  Here's
           an anonymous object with a String field named "foo":
         */

        var anon_obj = { foo: 'hi' };

        /*
           The anon_obj variable doesn't have a type declared, and is an
           anonymous object according to the compiler.  However, remember back at
           the top where we declared the FooObj typedef?  Since anon_obj matches
           that structure, we can use it anywhere that a "FooObject" type is
           expected.
         */

        var f = function(fo:FooObject){
            trace('$fo was passed in to this function');
        }
        f(anon_obj); // call the FooObject signature function with anon_obj.

        /*
           Note that typedefs can have optional fields as well, marked with "?"

           typedef OptionalFooObj = {
```

```
            ?optionalString: String,
            requiredInt: Int
        }
     */

    /*
        Typedefs work well with conditional compilation.  For instance,
        we could have included this at the top of the file:

#if( js )
        typedef Surface = js.html.CanvasRenderingContext2D;
#elseif( nme )
        typedef Surface = nme.display.Graphics;
#elseif( !flash9 )
        typedef Surface = flash8.MovieClip;
#elseif( java )
        typedef Surface = java.awt.geom.GeneralPath;
#end

        That would give us a single "Surface" type to work with across
        all of those platforms.
        */
    }
}

class UsingExample {
    public static function example() {

        /*
            The "using" import keyword is a special type of class import that
            alters the behavior of any static methods in the class.

            In this file, we've applied "using" to "StringTools", which contains
            a number of static methods for dealing with String types.
         */
        trace(StringTools.endsWith("foobar", "bar") + " should be true!");

        /*
            With a "using" import, the first argument type is extended with the
            method.  What does that mean?  Well, since "endsWith" has a first
            argument type of "String", that means all String types now have the
            "endsWith" method:
         */
        trace("foobar".endsWith("bar") + " should be true!");

        /*
            This technique enables a good deal of expression for certain types,
            while limiting the scope of modifications to a single file.

            Note that the String instance is *not* modified in the run time.
            The newly attached method is not really part of the attached
            instance, and the compiler still generates code equivalent to a
            static method.
         */
```

```
        }

}
```

We're still only scratching the surface here of what Haxe can do. For a formal overiew of all Haxe features, checkout the online manual, the online api, and "haxelib", the [haxe library repo] (http://lib.haxe.org/).

For more advanced topics, consider checking out:

- Abstract types
- Macros, and Compiler Macros
- Tips and Tricks

Finally, please join us on the mailing list, on IRC #haxe on freenode, or on Google+.