FREE eBook

LEARNING haxe

Free unaffiliated eBook created from
Stack Overflow contributors.

#haxe

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: haxe

It is an unofficial and free haxe ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official haxe.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with haxe

Remarks

Haxe is an open source toolkit that is capable of compiling to many different target languages and platforms.

It consists of:

- the Haxe programming language - a modern, high-level, and strictly typed programming language
- the Haxe standard library - a collection of general purpose, system, and target-specific APIs
- the Haxe compiler - a fast, optimising cross-compiler with metadata support, dead code elimination (DCE), completion mode, resource embedding, runtime type information (RTTI), static analyzer, macros, and more

Haxe has been used to create games, web, mobile, desktop, and command-line applications, as well as cross-platform APIs.

As of Haxe 3.3.0-rc.1, Haxe can compile to sources / bytecode of the following languages: ActionScript 3, C#, C++, Flash, HL, Lua, Java, JavaScript, Neko, PHP, and Python.

Haxe has a package manager, Haxelib, which is bundled with Haxe. It also has a custom build file format, `.hxml`, which offers an easier way of passing arguments passed to the Haxe compiler.

References

- Haxe documentation

Examples

Installation

Haxe is available on Windows, Linux, and OS X. It is distributed in two forms:

- as an **installer**, providing an optional Neko VM dependency and configuring `haxe` and `haxelib` environment variables;
- as **binaries**, providing only the Haxe compiler and package manager.

Windows

Installer and binaries are available from the Haxe website.

Linux

Binaries (32-bit and 64-bit) are available from the Haxe website.

The Haxe Foundation also officially participates in the maintenance of Haxe and Neko packages for popular Linux distributions. It is recommended to use those packages if available.

Ubuntu

It is recommended to use the Haxe PPA which provides latest Haxe and Neko releases for all currently supported Ubuntu versions. The PPA can also be used for Ubuntu-based distributions.

```
sudo add-apt-repository ppa:haxe/releases -y
sudo apt-get update
sudo apt-get install haxe -y
mkdir ~/haxelib && haxelib setup ~/haxelib
```

Note that Neko is installed as a dependency of Haxe.

Debian

To install the currently available stable versions, run the following commands:

```
sudo apt-get install haxe -y
mkdir ~/haxelib && haxelib setup ~/haxelib
```

Note that Neko will be installed as a dependency of Haxe.

To install newer releases from the unstable channel, do the following:

1. In `/etc/apt/sources.list`, add

   ```
   deb http://httpredir.debian.org/debian unstable main contrib non-free
   ```

2. In `/etc/apt/preferences.d/`, create a new file named `unstable` with the following content:

   ```
   Package: *
   Pin: release a=unstable
   Pin-Priority: 100

   Package: haxe neko libneko*
   Pin: release a=unstable
   Pin-Priority: 999
   ```

3. Pull package index files from the newly added source:

   ```
   sudo apt-get update
   ```

4. Install Haxe (and Neko):

   ```
   sudo apt-get install haxe -y
   ```

Fedora

The Haxe Foundation maintains the Haxe and Neko RPM packages in the Fedora repository. The packages are up-to-date most of the time. However, when a new version of Haxe is released, it will take a few days, up to 2 weeks, to push an updated package to the stable releases of Fedora. The update activities can be tracked in the Bodhi Fedora Update System.

To install the currently available versions of Haxe and Neko, run the following commands:

```
sudo dnf install haxe -y
mkdir ~/haxelib && haxelib setup ~/haxelib
```

Note that Neko is installed as a dependency of Haxe.

openSuse

The Haxe Foundation maintains the Haxe and Neko RPM packages in the openSUSE:Factory repository. The packages are up-to-date most of the time. However, when a new version of Haxe is released, it will take a few days, up to 2 weeks, to be accepted by openSUSE:Factory.

To install currently available versions of Haxe and Neko, run the following commands:

```
sudo zypper install haxe
mkdir ~/haxelib && haxelib setup ~/haxelib
```

Note that Neko is installed as a dependency of Haxe.

To get the lastest Haxe version that may not available to openSUSE:Factory or an openSUSE release, use the devel:languages:haxe project in the openSUSE Build Service. Visit the Haxe package page, click "Download package" at the top-right corner and follow the instructions. Again, Neko will also be installed as a dependency of Haxe.

Arch Linux

There are Haxe and Neko packages in the Arch Linux community repository. The Haxe Foundation will continue to help keep the packages up-to-date. However, when a new version of Haxe is released, it will take time to update the package, depended on the availability of the package maintainer.

For currently available versions of Haxe and Neko, check the following pages:

- Haxe in Arch Linux
- Neko in Arch Linux

To install the currently available versions of Haxe and Neko, run the following commands:

```
sudo pacman -S haxe
mkdir ~/haxelib && haxelib setup ~/haxelib
```

Note that Neko is installed as a dependency of Haxe.

OS X

Installer and binaries are available from the Haxe website.

It is also possible to install the current stable Haxe version through the Brew package manager.

```
brew install haxe
```

References

- "Downloads", Haxe website
- "Linux Software Packages", Haxe website

Hello World

Requirements

1. A version of the Haxe toolkit must be installed
2. Haxe must be present in your system path
3. Command line must be accessible

Code

Navigate to a desired project directory and create a `Test.hx` source file with the following content:

```
class Test {
    static function main() {
        trace("Hello world");
    }
}
```

Haxe source files are called **modules**. A module *should* define a type (`abstract`, `class`, `enum`, `interface`, or `typedef`) with the same identifier as the module name - in this case the `Test` class. Once that requirement is met, a module can define an arbitrary number of different types.

Haxe programs require an **entry point**, as denoted by the static `main` function. The class implementing the entry point is the **startup class** or main class. Again, in this case the main class is the `Test` class.

The `trace()` function is a general purpose logging function exposed to the global namespace for the sake of convenience. It outputs to the target language's standard output handle (e.g. browser console for JavaScript, command line for C++). See the API documentation for more information.

Execution

Navigate to the project folder from your command line. Test to see if Haxe is configured in your

environment by calling:

```
haxe --help
```

The Haxe interpreter can be used to test code that does not rely on any specific target language API. Use the interpreter by calling:

```
haxe -main Test --interp
```

Remember, the `Test` module contains the `Test` startup class, which is why `-main Test` is passed to the compiler.

Haxe sources can compile (*transpile*) to sources / bytecodes of several different languages. The following table displays the target language, compiler flag, argument type, and compilation result. Use it by calling:

```
haxe -main Test [flag] [argument].
```

Language	Flag	Argument	Result
ActionScript 3	-as3	Directory	Source
C#	-cs	Directory	Source + optional bytecode (.exe)
C++	-cpp	Directory	Source + optional binary (native)
Flash	-swf	File	Bytecode (.swf)
HL	-hl	File	Source
Lua	-lua	File	Source
Java	-java	Directory	Source + optional bytecode (.jar)
JavaScript	-js	File	Source
Neko	-neko	File	Bytecode (.n)
PHP	-php	Directory	Source
Python	-python	File	Source
HashLink	-hl	File	Bytecode (.hl)

Note that the path arguments here are relative to the path `haxe` was called from. The optional bytecode/binary outputs can be opt-outed by adding the `-D no-compilation` flags, in order to avoid an additional compilation step involving calling the target language's compiler.

References

- API documentation for `haxe.Log`
- "Hello world" entry in the Haxe Code Cookbook

Read Getting started with haxe online: https://riptutorial.com/haxe/topic/2593/getting-started-with-haxe

Chapter 2: Abstracts

Syntax

- abstract *identifier*(*underyling type*) { ... }
- abstract *identifier*(*underlying type*) from *typeA* from *typeB* ... to *typeA* to *typeB* { ... }

Remarks

An **abstract type** is a *compile-time* type which resolves to the **underlying type** at *run-time*. This means that thee abstract type *does not exist* in the source code generated by the Haxe compiler. In its stead are placed the underlying type, or types defined for implicit casting.

Abstracts are denoted by the `abstract` keyword, followed by an identifier, and underlying type in parentheses.

Abstracts may only define method fields and non-physical property fields. Non-inlined method fields are declared as static functions in a private **implementation class**, accepting as an additional first argument the underlying type of the abstract.

Note that operator overloading is only possible for abstract types.

Examples

Abstracts for data validation

The following abstract defines an `EmailAddress` type based on the `String` type which will use a regular expression to validate the passed argument as an e-mail address. If the address isn't valid, an exception will be thrown.

```
abstract EmailAddress(String) {
  static var ereg = ~/^[\w-\.]{2,}@[\w-\.]{2,}\.[a-z]{2,6}$/i;

  inline public function new(address:String) {
    if (!ereg.match(address)) throw "EmailAddress "$address" is invalid";
    this = address.toLowerCase();
  }
}
```

Use the abstract as follows.

```
var emailGood = new EmailAddress("john@doe.com");
var emailBad = new EmailAddress("john.doe.com");
```

Try the example on try.haxe.org.

References

- "EmailAddress", Haxe Code Cookbook

Operator overloading

Operator overloading is only possible with abstract types.

The following abstract defines a `Vec2i` type based on the `Array<Int>` type. This is a two-component vector with integer values. Operator overloading is made possible my the `@:op` **compiler metadata** . Only the available numeric operators can be overloaded - custom operators are not allowed to be specified.

```
abstract Vec2i(Array<Int>) {
    public inline function getX() : Int {
        return this[0];
    }

    public inline function getY() : Int {
        return this[1];
    }

    public inline function new(x : Int, y : Int) {
        this = [x, y];
    }

    @:op(A + B)
    public inline function add(B : Vec2i) : Vec2i {
        return new Vec2i(
            getX() + B.getX(),
            getY() + B.getY()
        );
    }
}
```

Use the abstract as follows.

```
var v1 = new Vec2i(1, 2);
var v2 = new Vec2i(3, 4);
v1 + v2;
v1.add(v2);
```

Try the example on try.haxe.org.

References

- "EmailAddress", Haxe Code Cookbook

Read Abstracts online: https://riptutorial.com/haxe/topic/4162/abstracts

Chapter 3: Branching

Syntax

- if (*condition*) { ... }
- if (*condition*) { ... } else { ... }
- if (*condition*) { ... } else if (*condition*) { ... } else { ... }
- // Braces are optional for single line statements
  if (*condition*) ... else if (*condition*) ... else ...
- switch (*expression*) { case *pattern*: ... default: ... }
- *condition* ? *expression if true* : *expression if false*;

Remarks

All branching expressions make it possible to return evaluated expressions. This means branching results can be assigned to variables. In this case, **all expressions that can be evaluated by a successful condition test must pass type unification**. If no `else` expression is given, the type is inferred to be `Void`.

Examples

If / else if / else

```
if (a > b) {
    trace("You win!");
} else if (a == b) {
    trace("It's a draw!");
} else {
    trace("You lose!");
}

// Assigning the evaluated expression to a variable
var message =  if (a > b) {
    "You win!";
} else if (a == b) {
    "It's a draw!";
} else {
    "You lose!";
}
trace(message);
```

Reference

- "If", Haxe manual

Ternary operator

```
n % 2 == 0 ? trace("n is even!") : trace("n is odd!");

// Assigning the evaluated expression to a variable
var message = n % 2 == 0 ? "n is even!" : "n is odd!";
trace(message);
```

Reference

- "If", Haxe manual

Switch

```
switch (n % 2) {
    case 0: trace("n is even!");
    case 1: trace("n is odd!");
    default: trace("I don't know!");
}

// Assigning the evaluated expression to a variable
var message = switch (n % 2) {
    case 0: "n is even!";
    case 1: "n is odd!";
    default: "I don't know!";
}
trace(message);
```

Note that **`case` body expressions never fall through**, so using the `break` expression in this context isn't supported by Haxe.

Reference:

- "Switch", Haxe manual

Read Branching online: https://riptutorial.com/haxe/topic/6265/branching

Chapter 4: Enums

Syntax

- enum *identifier* { *constructors* }

Examples

Overview

Haxe's enumeration types are **algebraic data types** (ADT). Their primary use is for describing data structures. Enums are denoted by the `enum` keyword and contain one or more **enum constructors**.

```
enum Color {
    Red;
    Green;
    Blue;
    RGB(r : Int, g : Int, b : Int);
}
```

The above enum can be instantiated as follows:

```
var c1 = Color.Red;
var c2 = Color.RGB(255, 0, 0);
```

Try the example on try.haxe.org.

References

- "Enum instance", Haxe manual

Capturing enum values

Values passed as enum constructor arguments can be captured into variables by use of **pattern matching**.

Assume the following enum:

```
enum Color {
    RGB(r : Int, g : Int, b : Int);
    HSV(h : Int, s : Float, v : Float);
}
```

The red channel value can be captured as follows:

```
var color = Color.RGB(255, 127, 0);
var red = switch (color) {
    // Match the Color.RGB constructor and capture value into `r`
    case Color.RGB(r, _, _):
        // Return the captured red value
        r;
    // Catch-all for matching remaining constructors
    case _:
        // Return -1
        -1;
}
```

Try the example on try.haxe.org.

References

- "Pattern matching", Haxe manual
- "Variable capture", Haxe manual

Matching enum constructors

Enum constructors can be matched using **pattern matching**.

Assume the following enum:

```
enum Color {
    Red;
    Green;
    Blue;
    RGB(r : Int, g : Int, b : Int);
}
```

Colours with only a green channel value can be matched as follows:

```
var color = Color.RGB(0, 127, 0);
var isGreenOnly = switch (color) {
    // Match Green or RGB with red and blue values at 0
    case Color.RGB(0, _, 0) | Color.Green: true;
    case _: false;
}
```

Try the example on try.haxe.org.

References

- "Pattern matching", Haxe manual
- "Enum matching", Haxe manual
- "Or patterns", Haxe manual

Read Enums online: https://riptutorial.com/haxe/topic/4667/enums

Chapter 5: Loops

Syntax

- for (*variable identifier* in *iterating collection*) { *expression* }
- while (*condition*) { *expression* }
- do { *expression* } while (*condition*);
- break;
- continue;

Examples

For

For-loops iterate over an **iterating collection**. An iterating collection is any class which structurally unifies with `Iterator<T>` or `Iterable<T>` types from the Haxe standard library.

A for-loop which logs numbers in range 0 to 10 (exclusive) can be written as follows:

```
for (i in 0...10) {
    trace(i);
}
```

The variable identifier `i` holds the individual value of elements in the iterating collection. This behaviour is similar to for-each in other languages.

A for-loop which logs elements in an array can therefore be written as follows:

```
for (char in ['a', 'b', 'c', 'd']) {
    trace(char);
}
```

Try the example on try.haxe.org.

References

- "For", Haxe manual
- "Iterators", Haxe manual

While

While-loops execute a body expression as long as the loop condition evaluates to `true`.

A while-loop which logs numbers in range 9 to 0 (inclusive) can be written as follows:

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

Try the example on try.haxe.org.

References

- "While", Haxe manual

Do-while

Do-while-loops execute a body expression at least once, and then keep executing it as long as the loop condition evaluates to `true`.

A do-while-loop which logs numbers in range 10 to 0 (inclusive) can be written as follows:

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

Try the example on try.haxe.org.

References

- "Do-while", Haxe manual

Flow control

The flow or execution of a loop can be controlled by use of `break` and `continue` expressions.

Break

`break` exits the current loop. In case the loop is nested inside another loop, the parent loop is unaffected.

```
for (i in 0...10) {
    for (j in 0...10) {
        if (j == 5) break;
        trace(i, j);
    }
}
```

Try the example on try.haxe.org.

Continue

`continue` skips the current iteration of the loop at the point of the expression. In case the loop is nested inside another loop, the parent loop is unaffected.

```
for (i in 0...10) {
    for (j in 0...10) {
        if (j == 5) continue;
        trace(i, j);
    }
}
```

Try the example on try.haxe.org.

References

- "Break", Haxe manual
- "Continue", Haxe manual

Read Loops online: https://riptutorial.com/haxe/topic/4409/loops

Chapter 6: Pattern matching

Remarks

Pattern matching is the process of branching depending on provided patterns. All pattern matching is done within a `switch` expression, and individual `case` expressions represent the patterns.

The fundamental rules of pattern matching are:

- 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:`.

When all possible cases are handled, the catch-all _ pattern or `default` case is not required.

Examples

Enum matching

Assume the following enum:

```
enum Operation {
    Multiply(left : Int, right : Int);
}
```

Enum matching can be performed as follows:

```
var result = switch(Multiply(1, 3)) {
    case Multiply(_, 0):
        0;
    case Multiply(0, _):
        0;
    case Multiply(l, r):
        l * r;
}
```

References

- "Enum matching", Haxe manual

Structure matching

Assume the following structure:

```
var dog = {
    name : "Woofer",
    age : 7
```

```
};
```

Enum matching can be performed as follows:

```
var message = switch(dog) {
    case { name : "Woofer" }:
        "I know you, Woofer!";
    case _:
        "I don't know you, sorry!";
}
```

References

- "Structure matching", Haxe manual

Array matching

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

References

- "Array matching", Haxe manual

Or patterns

The | operator can be used anywhere within patterns to describe multiple accepted patterns. If there is a captured variable in an or-pattern, it must appear in both its sub-patterns.

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

References

- "Or patterns", Haxe manual

Guards

It is also possible to further restrict patterns with guards. These are defined by 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";
}
```

References

- "Guards", Haxe manual

Extractors

Extractors are identified by the `extractorExpression => match` expression. 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:

```
static public function main() {
    switch(3) {
        case add(_, 1) => mul(_, 3) => a:
            trace(a); // mul(add(3 + 1), 3)
    }
}

static function add(i1:Int, i2:Int) {
    return i1 + i2;
}

static function mul(i1:Int, i2:Int) {
    return i1 * i2;
}
```

It is currently not possible to use extractors within or-patterns. However, it is possible to have or-patterns on the right side of an extractor.

References

- "Extractors", Haxe manual

Read Pattern matching online: https://riptutorial.com/haxe/topic/6436/pattern-matching

Credits

S. No	Chapters	Contributors
1	Getting started with haxe	5Mixer, ali_o_kan, Andy Li, Community, Domagoj, KevinResoL, YsenGrimm
2	Abstracts	Domagoj
3	Branching	Domagoj, Kev, Mark Knol
4	Enums	Domagoj
5	Loops	Domagoj
6	Pattern matching	Domagoj