Part I
The Core Language

1

Introducing haXe

The Internet is a fantastic invention. As developers, we determine its future and shape its usefulness through our own developments. This book will hopefully help to refine this path, as we seek to make the Internet ever more perfect.

In this chapter, you will acquire a broad perspective of:

- Current issues with developing for the Internet.
- What are haXe and Neko?
- How haXe and Neko can alleviate issues with Internet development.

A Problem with Internet Development

The IT developer, regardless of skills and title, can often be perceived as sitting somewhere along a scale depicting the transition from the creative to the logical. On the creative end of the scale, you might see those who design website layouts, digital illustrations, animations or presentations, while on the purely logical side, you might see the database administrators or server-side architects. Of course, most people located under the developer umbrella sit somewhere in between and cover more than a little speck along this line.

The term *developer* tends to describe those who deal with some form of programming language. The highly creative ones might use ActionScript or JavaScript to accomplish their development needs, while the very logical may be restricted to using PL/SQL or C, perhaps. Yet, regardless of which languages or development skills you claim to have, the diversity of the tasks you are charged to accomplish require that you use numerous technologies.

As an example, if you were to look at the technologies that may be required for the development of a web application, you might decide to use the services of a Flash designer, an HTML and

JavaScript developer, a server-side applications developer, and a database architect. This will certainly cover a lot of the requirements for many web-based projects, and of course, many developers can handle more than one of these points very well. However, software development demands that the various layers that constitute the final application will function and communicate well with each other and you cannot always guarantee that the members of a development team will understand the roles and technologies of the other team members with the depth required to facilitate this level of reliability. Similarly, when a single developer is charged with providing his or her services to build such an application, it is likely they would much prefer to spend the time they have been paid for in developing the application as opposed to having to learn the various new technologies required to complete the project.

This is where haXe makes its entrance.

What Is haXe and Why Was It Created?

Over recent years, we have seen the introduction of numerous new technologies and languages that make applications development a little bit more exciting. Languages such as Python and Ruby have appeared that, while not necessarily providing groundbreaking features, lend some support to providing alternative ways to producing software. Ruby on Rails, or RoR, is one such feature that has enabled developers to produce web-based applications with remarkable speed and ease, making the development of common application functionality less painful.

This is a great move in the right direction, and many developers have since made the transition to such languages to further lessen the strain that was ever present with languages such as C++ or Java, but it doesn't provide a be-all-and-end-all for applications development as it still means having to learn a new technology. What's more, despite the surge of new technologies that have been appearing, you can see that they are only really of benefit to desktop or server-side web development, while the client side for web applications remain firm with Flash, HTML, JavaScript, and VBScript, for the simple reason that older browser technologies have a greater coverage on users' machines. This means that, while we now have a larger choice of tools for creating our applications, we are perhaps further segregating our development team members and forcing possible rifts into our software stability and production timescales.

So why are we suffering these issues and what can be done to dispose of these rifts we keep making for ourselves while continuing to embrace technology developments? The issues here are not about how such technologies interact, or about the functionality they provide, but about how we develop for them.

Think about that for a second. If humans, like technology, needed to interact with one another, we would use an agreed form of communication; written English in the case of this book. If we were to seek employment in a country whose entire population didn't speak our language, we are sure this would hinder our progress very much, as it would be hard enough to even convey that we were seeking employment, let alone be capable of performing many of the available work positions on offer. Yet, in respect of many programming languages and technologies available, they often communicate very well with each other using processes of their own, but the way in which we interact with them can be very diverse.

When new languages are created, the syntax they support is often chosen to reflect the flavor of programmer the language is likely to attract, the features the language provides, the structure of the compiler, and the general taste of the language's author, to name a few. However, a compiler is often just a clever program that analyzes your code and converts it to a more machine readable syntax, so it is just as plausible to create a compiler that can compile from one language into another. This is exactly what Nicolas Cannasse thought, and what prompted him to create the haXe compiler.

The haXe Compiler

The original purpose of the haXe compiler was to encompass several web technologies under a single language. Supporting Flash, JavaScript and Neko, web developers can use the haXe compiler to target all three technologies without having to learn a new syntax. Also, as haXe, Flash, JavaScript and Neko can function on Windows, Mac OS and Linux, many developers will be able to embrace haXe without having to change their preferred operating system.

As time goes on, further bridges to other technologies are being developed. With an emphasis on Neko, the haXe language can be further enhanced without having to modify the compiler in any way. This has led to other exciting developments such as the SWHX (ScreenWeaver for haXe) framework for creating functional desktop applications using a Flash GUI (Graphical User Interface) layer or our own Neko Media Engine (NME), which wraps the Simple DirectMedia Layer (SDL) framework and functions as a great 2D games API.

The point here is that the haXe compiler is not a new technology, merely a program that converts a new language to numerous existing technologies.

The benefits to using haXe are astronomical, but can be summarized by the following points:

- Use of existing technologies are stretched to their fullest capabilities.
- Boundaries between developers are lowered and more ground is provided for them to collaborate.
- Knowledge of the development team is increased as they write code that extends to the technologies of their peers.
- Projects are built rapidly, with less errors and platform concerns.
- Projects become easier to maintain, as all team members are able to understand the language syntax regardless of deployment technology.
- The haXe classes developed for one technology can later be used to compile for other technologies.

Compiling to Flash

Flash is a fantastic platform. So fantastic in fact, that no other tool has been able to knock it out of its position as the most popular multimedia platform for the Web, and it will take a lot of effort to find a viable contender. With its ability to run on numerous machines and operating systems, it is safe to say that Flash will be around for some time to come, despite rumors that Microsoft's new developments will be a threat to Flash.

The Flash IDE, synonymous to developers who create Flash movies, contains its own version of a Flash file compiler, as does the new Flex Builder application from Adobe. These tools are probably the forerunners in commercial Flash development, but are not needed to create complete Flash applications.

Since the introduction of ActionScript 2.0 — the scripting language behind Flash movies — developers have had the ability to write applications using pure code. Unfortunately, though, the Flash IDE is not the most ideal environment for building Flash files in this way, as it is aimed primarily at designers. Flex provides a better solution, but still has drawbacks.

If you contemplate both Flash and Flex and their relative language syntax — ActionScript 1 and 2 for the Flash IDE and MXML & ActionScript 3 for Flex Builder — you will see two very different programs and languages compile to the same platform. Granted, the Flash IDE compiles to Flash versions 9 and below (ActionScript 1, 2, and 3) whereas the Flex Builder IDE compiles to Flash version 9 alone (ActionScript 3), but they inherently perform the same feat. haXe is able to perform the same routine of compiling from a source language syntax to the Flash byte code specification, much like the Flash IDE and Flex Builder, except that haXe is able to compile a single language to both the complete Flash 6 to 8 specifications and the Flash 9 specification. That's pretty impressive in our book (pun intended).

haXe makes all of this possible by providing a language that is loosely coupled to the output, which is why it is able to support so many platforms. All haXe needs to understand is which equivalent structures for each platform map to the haXe structures and how it should be serialized in the output file.

The Flash SWF file, which is one such output, is a document containing a series of codes in the form of bytes. Each byte represents media, functions, variables, or properties that make up the content of the movie and describe to the Flash virtual machine (or player) the exact content and functionality of the movie. When compiling to Flash, haXe produces the same SWF output and provides all of the same features as the official Flash compilers produced by Adobe, though of course, certain functions within the haXe library may not be supported depending on which version of Flash you are compiling against. You can handle this using *compiler directives* that allow different portions of code to be compiled depending on the target technology.

> *For those of you who are used to the MTASC compiler by Nicolas Cannasse for ActionScript versions 8 and below, haXe steps in as the successor, reducing further development of the MTASC compiler to bug fixes only.*

When MTASC (Motion-Twin ActionScript Compiler) was released several years ago, many developers saw, for the first time, their first break into the world of Flash. Previously, many developers would complain that the Flash IDE was far too geared toward the designer and left little leverage for the developer. MTASC changed all of that by offering an all-code entry point using tools they were already familiar with.

haXe follows this developer-friendly route, though with its powerful yet friendly syntax, it also offers designers the chance to tinker in the world of the developer.

Compiling to JavaScript

JavaScript has been around for some time, but for client-side browser scripting, there is no competition. While Internet Explorer provides access to the Visual Basic scripting interpreter, JavaScript is still the only scripting language supported by the majority of browsers, and so is the only choice for thousands of developers worldwide.

Each of the well-known browsers supports quite a variation of the JavaScript API. The most noted differences are those between the Internet Explorer JavaScript engine and the Mozilla JavaScript engine, which have been the source of much pulling of hair for web developers everywhere for a number of years. When you build applications that rely heavily on client-side scripting over various browser types, it is a necessity to include numerous hacks and tricks to avoid facilitating functionality that performs well on some browsers, yet poor on others. When compiling to JavaScript with the haXe compiler, haXe provides a set of functions that form a small framework for maintaining suitable cross-browser JavaScript functionality.

Compiling to Neko

haXe compiles to the Neko byte code format, in a similar way to how it handles Flash byte code. The compiled files are used by the Neko virtual machine in a very similar way to the Flash virtual machine, though any player support must come from a third-party library.

Usually, one would write for the Neko virtual machine for the purpose of creating server-side web logic using the `mod_neko` module or to create a desktop application for use in a command console or batch file. Using third-party modules, it is also possible to create desktop applications and full network ready server frameworks of any scale.

Although not much has been mentioned yet about the Neko language, compiler, and virtual machine, they do form a fairly substantial part of this book. The Neko framework is discussed in detail in Chapter 9, "Building Websites with haXe."

The haXe Language

New languages pop their heads out of the woodwork at a staggering rate. It seems, every time we browse the Web, we find a new language that claims some fantastic new capability or style. The haXe language, however, was designed to unite existing technologies under a single language, so the syntax of the haXe language was its most scrutinized characteristic.

haXe is, by definition, a high-level language. The primary benefits for using haXe are its simplicity and consistency. Where most languages force a user to program in a certain fashion, haXe provides a hybrid nature of many features, but at the same time, haXe strives to produce the best of all features. For example, haXe is a statically typed language, so it is important that data containers within a haXe program maintain a set data type, thus maintaining security and good coding practices. However, in keeping with the advantages of dynamic languages, you need not specify what type of data a container represents at design-time. Instead, haXe uses C++-style templates and type inference, so that you can benefit from the flexibility of dynamic types.

Another benefit of haXe is that it supports both functional programming and object-oriented programming principles, while still maintaining solid best programming practices. On the functional programming side, haXe supports type inference as already mentioned, as well as nested functions and recursion. The object-oriented capabilities of haXe, however, allow for classes, interfaces, enumerators, and getters and setters.

> *Throughout the course of this book, the main focus will be that of object-oriented programming techniques in haXe, though the functional programming features will be outlined for reference.*

The haXe Libraries

The haXe language comes complete with various libraries that one would expect from any mature language. As standard, haXe provides support for XML, regular expressions, sockets support, and database connectivity. Unlike many languages, the haXe core library also provides a template system, a persistent database objects framework, and a remoting framework that allows for scripted communication between Flash, JavaScript, and Neko.

If a particular feature is not present in the current haXe distribution, you will be happy to know that the active community is releasing new modules all the time. Many new and exciting modules are in development that seem destined to far exceed any other language in terms of feature, speed, and ease of use. While creating your own libraries is so simple, you'll find yourself building all sorts of creations for the shear fun of it.

All of the core libraries are covered later in the book.

How Does haXe Work?

With the exception of any Neko modules you may be using in your project, all haXe files are merely text documents with the extension `.hx`. Each file contains classes that make up your application, regardless of what platform you will be exporting to.

Your application will also make use of a number of standard classes in the form of packages, which are also in `.hx` files and provide reusable functionality for use in everyday applications. In all likelihood, you will probably create your own packages in order to speed up applications development as you gain experience with haXe.

As you write your applications, you will likely write code that is specific to certain platforms and some code that is relevant to all platforms. Later in the book, you'll learn how you can separate this code for each platform so that only relevant code will compile. This helps maintain your code in a way that reduces the need to duplicate a lot of application logic.

Once you are happy with your code, you simply compile the `.hx` files to the requested platform, which will then produce a file readable by the target platform's interpreter or player. This will be an SWF file for Flash players, a JS file for JavaScript interpreters, and an N file for the Neko virtual machine. Compiling is explained in Chapter 2, "Installing and Using haXe and Neko."

So What Is Neko?

If haXe is the aesthetic syntactical sugar coating of programming, Neko is the almighty creator of functionality. Since discovering and falling in love with the haXe language and compiler, it didn't take too long to discover the true power of Neko and realize that, although haXe is a breakthrough scripting language, Neko is by far Nicolas Cannasse's greatest creation.

Unlike haXe, the standard Neko language is dynamically typed. However, also unlike haXe, the language was created to be more easily interfaced by language generators rather than directly by a programmer. The reason for this is that the Neko compiler was never truly meant to exist by itself, but to provide a powerful tool and virtual machine for existing languages. This is why haXe and Neko are considered a package or toolkit.

The scripting capabilities of Neko come in three flavors:

- The Standard Neko language
- NXML
- NekoML

The standard Neko script is a procedural language definition, which you use later in this book while you prototype your own custom modules. The Neko language makes testing new modules very quick and easy, as no object-oriented structure or methods are required.

The other two Neko languages include an XML style language called NXML and a functional language called NekoML. NXML, like the procedural Neko language, is aimed at language compilers and generators. The reason for this choice is that, although standard languages are more easily read by people, XML trees are far easier for machines to generate and navigate. NXML also provides a more simple way to include file and line information for the compiled `.hx` files so that more accurate debugging information can be provided to the developer. By adopting NXML, any language that supports XML is able to build scripts that can then be compiled by the Neko compiler for execution by vthe virtual machine.

NekoML, on the other hand, is a different kettle of fish as it is styled around functional languages from the ML family, with a specific similarity to Objective Caml. ML languages provide a very organic method of processing data using nested functions and recursion that reduces the amount of used resources and is very well suited to symbolic processing and pattern matching as used by compilers. This makes NekoML the perfect language for creating your own compilers.

> *Although the Neko compiler was originally written in Objective Caml, it is now written in NekoML and compiled by the Neko compiler. This round-robin technique is known as bootstrapping.*

As time goes on, other languages will be written for compilation against the Neko virtual machine. At the time of writing this book, there is a small group of individuals working toward building a compiler to allow Ruby scripts to run under Neko. If, after reading this book, you find yourself wanting to create a compiler for your own favourite language, drop an e-mail to Nicolas Cannasse on the haXe mailing list and he'll steer you in the right direction.

haXe and Neko Requirements

While using numerous other scripting languages in the past, we have often had issues with complex installations, numerous dependent software installations, complex file structures, and all sorts of yoga-like key combinations that practically dislocate your fingers, just to get the things running. We have since given up these annoyances and left any references to the game of Twister to Lee's six-year-old daughter.

haXe and Neko are different. They are very minimal programming platforms. All you need is the haXe and/or Neko distributions from `www.haxe.org` and `www.nekovm.org` respectively and a computer that has either Windows 95 or above, a flavor of Linux, or Mac OSX as the running operating system.

Summary

In this chapter, you looked at:

- The issues with developing for the Internet, today
- What haXe and Neko are
- Why haXe and Neko are beneficial to developers
- What requirements are needed to run haXe and Neko

You install your copies of haXe and Neko in the next chapter.