Nim in Action

by Dominik Picheta

Chapter 1

brief contents

1 Why Nim?

This chapter covers

- What Nim is
- Why you should learn about it
- Comparing Nim to other programming languages
- Use cases
- Strengths and weaknesses

Nim is still a relatively new programming language. In fact, you're holding one of the very first books about it. The language is still not fully complete, but core aspects, like its syntax, the semantics of procedures, methods, iterators, generics, templates, and more, are all set in stone. Despite its newness, there has been significant interest in Nim from the programming community because of the unique set of features that it implements and offers its users.

This chapter answers questions that you may ask before learning Nim, such as why you might want to use it. In this chapter, I outline some of the common practical uses of Nim, compare it to other programming languages, and discuss some of its strengths and weaknesses.

1.1 What is Nim?

Nim is a general-purpose programming language designed to be efficient, expressive, and elegant. These three goals are difficult to achieve at the same time, so Nim's designers gave each of them different priorities, with efficiency being the most important and elegance being the least.

But despite the fact that elegance is relatively unimportant to Nim's design, it's still considered during the design process. Because of this, the language remains elegant in its own right. It's only when trade-offs between efficiency and elegance need to be made that efficiency wins.

On the surface, Nim shares many of Python's characteristics. In particular, many aspects of Nim's syntax are similar to Python's, including the use of indentation to delimit scope as well as the tendency to use words instead of symbols for certain operators. Nim also shares other aspects with Python that aren't related to syntax, such as the highly user-friendly exception tracebacks, shown here:

```
Traceback (most recent call last)
request.nim(74)          request
request.nim(25)          getUsers
json.nim(837)            []
tables.nim(147)          []
Error: unhandled exception: key not found: totalsForAllResults [KeyError]
```

You'll also see many differences, especially when it comes to the semantics of the language. The major differences lie within the type system and execution model, which you'll learn about in the next sections.

> **A little bit about Nim's history**
>
> Andreas Rumpf started developing Nim in 2005. The project soon gained support and many contributions from the open source community, with many volunteers around the world contributing code via pull requests on GitHub. You can see the current open Nim pull requests at https://github.com/nim-lang/Nim/pulls.

CONTRIBUTING TO NIM The compiler, standard library, and related tools are all open source and written in Nim. The project is available on GitHub, and everyone is encouraged to contribute. Contributing to Nim is a good way to learn how it works and to help with its development. See Nim's GitHub page for more information: https://github.com/nim-lang/Nim#contributing.

1.1.1 Use cases

Nim was designed to be a general-purpose programming language from the outset. As such, it consists of a wide range of features that make it usable for just about any software project. This makes it a good candidate for writing software in a wide variety of

application domains, ranging from web applications to kernels. In this section, I'll discuss how Nim's features and programming support apply in several use cases.

Although Nim may support practically any application domain, this doesn't make it the right choice for everything. Certain aspects of the language make it more suitable for some categories of applications than others. This doesn't mean that some applications can't be written using Nim; it just means that Nim may not support the code styles that are best suited for writing some kinds of applications.

Nim is a compiled language, but the way in which it's compiled is special. When the Nim compiler compiles source code, it first translates the code into C code. C is an old but well supported systems programming language that allows easier and more direct access to the physical hardware of the machine. This makes Nim well suited to systems programming, allowing projects such as operating systems (OSs), compilers, device drivers, and embedded system software to be written.

Internet of Things (IoT) devices, which are physical devices with embedded electronics that are connected to the internet, are good targets for Nim, primarily thanks to the power offered by Nim's ease of use and its systems programming capabilities.

A good example of a project making use of Nim's systems programming features is a very simple OS called NimKernel available on GitHub: https://github.com/dom96/nimkernel.

> **HOW DOES NIM COMPILE SOURCE CODE?** I describe Nim's unusual compilation model and its benefits in detail in section 1.1.3.

Applications written in Nim are very fast; in many cases, just as fast as applications written in C, and more than thirteen times faster than applications written in Python. Efficiency is the highest priority, and some features make optimizing code easy. This goes hand in hand with a soft real-time garbage collector, which allows you to specify the amount of time that should be spent collecting memory. This feature becomes important during game development, where an ordinary garbage collector may slow down the rendering of frames on the screen if it uses too much time collecting memory. It's also useful in real-time systems that need to run in very strict time frames.

Nim can be used alongside other much slower languages to speed up certain performance-critical components. For example, an application written in Ruby that requires certain CPU-intensive calculations can be partially written in Nim to gain a considerable speed advantage. Such speed-ups are important in areas such as scientific computing and high-speed trading.

Applications that perform I/O operations, such as reading files or sending data over a network, are also well supported by Nim. Web applications, for example, can be written easily using a number of web frameworks like Jester (https://github.com/dom96/jester). Nim's script-like syntax, together with its powerful, asynchronous I/O support, makes it easy to develop these applications rapidly.

Command-line applications can benefit greatly from Nim's efficiency. Also, because Nim applications are compiled, they're standalone and so don't require any

bulky runtime dependencies. This makes their distribution incredibly easy. One such application written in Nim is Nimble; it's a package manager for Nim that allows users to install Nim libraries and applications.

These are just a few use cases that Nim fits well; it's certainly not an exhaustive list.

Another thing to keep in mind is that, at the time of writing, Nim is still in development, not having yet reached version 1.0. Certain features haven't been implemented yet, making Nim less suited for some applications. For example, Nim includes a backend that allows you to write JavaScript applications for your web pages in Nim. This backend works, but it's not yet as mature as the rest of the language. This will improve with time.

Of course, Nim's ability to compile to JavaScript makes it suitable for full-stack applications that need components that run on a server and in a browser. This is a huge advantage, because code can easily be reused for both the browser and server components of the application.

Now that you know a little bit about what Nim is, its history, and some of the applications that it's particularly well suited for, let's look at some of Nim's features and talk about how it works.

1.1.2 Core features

In many ways, Nim is very innovative. Many of Nim's features can't be found in any other programming language. If you enjoy learning new programming languages, especially those with interesting and unique features, then Nim is definitely the language for you.

In this section, we'll look at some of the core features of Nim—in particular, the features that make Nim stand out from other programming languages:

- A facility called *metaprogramming*, used for, among many things, molding the language to your needs.
- Style-insensitive variable, function, and type names. By using this feature, which is slightly controversial, you can treat identifiers in whatever style you wish, no matter if they were defined using `camelCase` or `snake_case`.
- A type system that's rich in features such as generics, which make code easier to write and maintain.
- Compilation to C, which allows Nim programs to be efficient and portable. The compilation itself is also very fast.
- A number of different types of garbage collectors that can be freely selected or removed altogether.

METAPROGRAMMING

The most practical, and in some senses unique, feature of Nim is its extensive metaprogramming support. Metaprogramming allows you to read, generate, analyze, and transform source code. It was by no means a Nim invention, but there's no other programming language with metaprogramming that's so extensive and at the same

time easy to pick up as Nim's. If you're familiar with Lisp, then you might have some experience with metaprogramming already.

With metaprogramming, you treat code as data in the form of an *abstract syntax tree*. This allows you to manipulate existing code as well as generate brand new code while your application is being compiled.

Metaprogramming in Nim is special because languages with good metaprogramming features typically belong to the Lisp family of languages. If you're already familiar with the likes of Java or Python, you'll find it easier to start using Nim than Lisp. You'll also find it more natural to learn how to use Nim's metaprogramming features than Lisp's.

Although it's generally an advanced topic, metaprogramming is a very powerful feature that you'll get to know in far more detail in chapter 9 of this book. One of the main benefits that metaprogramming offers is the ability to remove boilerplate code. Metaprogramming also allows the creation of domain-specific languages (DSLs); for example,

```
html:
  body:
    p: "Hello World"
```

This DSL specifies a bit of HTML code. Depending on how it's implemented, the DSL will likely be translated into Nim code resembling the following:

```
echo("<html>")
echo("  <body>")
echo("    <p>Hello World</p>")
echo("  </body>")
echo("</html>")
```

That Nim code will result in the following output:

```
<html>
  <body>
    <p>Hello World</p>
  </body>
</html>
```

With Nim's metaprogramming, you can define DSLs and mix them freely with your ordinary Nim code. Such languages have many use cases; for example, the preceding one can be used to create HTML templates for your web apps.

Metaprogramming is at the center of Nim's design. Nim's designer wants to encourage users to use metaprogramming in order to accommodate their style of programming. For example, although Nim does offer some object-oriented programming (OOP) features, it doesn't have a class definition construct. Instead, anyone wishing to use OOP in Nim in a style similar to that of other languages should use metaprogramming to create such a construct.

STYLE INSENSITIVITY

Another of Nim's interesting and likely unique features is style insensitivity. One of the hardest things a programmer has to do is come up with names for all sorts of identifiers like variables, functions, and modules. In many programming languages, these names can't contain whitespace, so programmers have been forced to adopt other ways of separating multiple words in a single name. Multiple differing methods were devised, the most popular being `snake_case` and `camelCase`. With Nim, you can use `snake_case` even if the identifier has been defined using `camelCase`, and vice versa. So you can write code in your preferred style even if the library you're using adopted a different style for its identifiers.

Listing 1.1 Style insensitivity

```
import strutils          <-- The strutils module defines a procedure called toUpper.
echo("hello".to_upper()) <-- You can call it using snake_case.
echo("world".toUpper())  <-- As it was originally defined, you can call it using camelCase.
```

This works because Nim considers the identifiers `to_upper` and `toUpper` to be equal.

When comparing identifiers, Nim considers the case of the *first character*, but it doesn't bother with the case of the rest of the identifier's characters, ignoring the underscores as well. As a result, the identifiers `toUpper` and `ToUpper` aren't equal because the case of the first character differs. This allows type names to be distinguished from variable names, because, by convention, type names should start with an uppercase letter and variable names should start with a lowercase letter.

The following listing shows one scenario where this convention is useful.

Listing 1.2 Style insensitivity and type identifiers

```
type
  Dog = object          <-- The Dog type is defined with an uppercase first letter.
    age: int            <-- Only primitive types such as int start with a lowercase letter.

let dog = Dog(age: 3)   <-- A dog variable can be safely defined because it won't clash with the Dog type.
```

POWERFUL TYPE SYSTEM

One of the many characteristics that differentiate programming languages from one another is their type system. The main purpose of a type system is to reduce the opportunities for bugs in your programs. Other benefits that a good type system provides are certain compiler optimizations and better documentation of code.

The main categories used to classify type systems are *static* and *dynamic*. Most programming languages fall somewhere between the two extremes and incorporate ideas from both. This is because both static and dynamic type systems require certain tradeoffs. Static typing finds more errors at compile time, but it also decreases the speed at which programs can be written. Dynamic typing is the opposite.

Nim is statically typed, but unlike some statically typed programming languages, it also incorporates many features that make development fast. Type inference is a good example of that: types can be resolved by the compiler without the need for you to write the types out yourself (though you can choose to). Because of that, your program can be bug-free and yet your development speed isn't hindered. Nim also incorporates some dynamic type-checking features, such as runtime type information, which allows for the dynamic dispatch of functions.

One way that a type system ensures that your program is free of bugs is by verifying memory safety. Some programming languages, like C, aren't memory safe because they allow programs to access memory that hasn't been assigned for their use. Other programming languages are memory safe at the expense of not allowing programs to access low-level details of memory, which in some cases is necessary. Nim combines both: it's memory safe as long as you don't use any of the unsafe types, such as `ptr`, in your program, but the `ptr` type is necessary when interfacing with C libraries. Supporting these unsafe features makes Nim a powerful systems programming language.

By default, Nim protects you against every type of memory error:

- Arrays are bounds-checked at compile time, or at runtime when compile-time checks aren't possible, preventing both buffer overflows and buffer overreads.
- Pointer arithmetic isn't possible for reference types as they're entirely managed by Nim's garbage collector; this prevents issues such as dangling pointers and other memory issues related to managing memory manually.
- Variables are always initialized by Nim to their default values, which prevents variables containing unexpected and corrupt data.

Finally, one of the most important features of Nim's type system is the ability to use generic programming. Generics in Nim allow for a great deal of code reuse without sacrificing type safety. Among other things, they allow you to specify that a single function can accept multiple different types. For example, you may have a `showNumber` procedure that displays both integers and floats on the screen:

```
proc showNumber(num: int | float) =
  echo(num)

showNumber(3.14)
showNumber(42)
```

Here, the `showNumber` procedure accepts either an `int` type or a `float` type. The `|` operator specifies that both `int` and `float` can be passed to the procedure.

This is a simple demonstration of Nim's generics. You'll learn a lot more about Nim's type system, as well as its generics, in later chapters.

Compilation

I mentioned in the previous section that the Nim compiler compiles source code into C first, and then feeds that source code into a C compiler. You'll learn a lot more about how this works in section 1.1.3, but right now I'll talk about some of the many practical advantages of this compilation model.

The C programming language is very well established as a systems programming language and has been in use for over 40 years. C is one of the most portable programming languages, with multiple implementations for Windows, Linux, Mac OS, x86, AMD64, ARM, and many other, more obscure OSs and platforms. C compilers support everything from supercomputers to microcontrollers. They're also very mature and implement many powerful optimizations, which makes C very efficient.

Nim takes advantage of these aspects of C, including its portability, widespread use, and efficiency.

Compiling to C also makes it easy to use existing C and C++ libraries—all you need to do is write some simple wrapper code. You can write this code much faster by using a tool called `c2nim`. This tool converts C and C++ header files to Nim code, which wraps those files. This is of great benefit because many popular libraries are written in C and C++.

Nim also offers you the ability to build libraries that are compatible with C and C++. This is handy if you want your library to be used from other programming languages. You'll learn all about wrapping C and C++ libraries in chapter 8.

Nim source code can also be compiled into Objective C and JavaScript. The Objective C language is mainly used for iOS software development; by compiling to it, you can write iOS applications natively in Nim. You can also use Nim to develop Android applications by using the C++ compilation backend. JavaScript is the client-side language used by billions of websites; it's sometimes called the "assembly language of the web" because it's the only programming language that's supported by all the major web browsers. By compiling to JavaScript, you can write client-side applications for web browsers in Nim. Figure 1.1 shows the available Nim backends.

You may now be wondering just how fast Nim is at compiling software. Perhaps you're thinking that it's very slow; after all, Nim needs to translate source code to an intermediate language first. But in fact it's fairly fast. As an example, the Nim compiler, which consists of around 100,000 lines of Nim code, takes about 12 seconds to

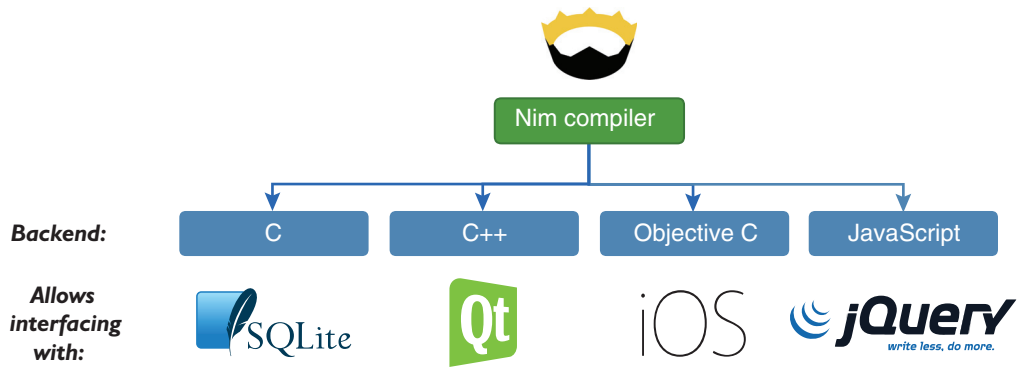


Figure 1.1 Compilation backends

compile on a MacBook Pro with a 2.7 GHz Intel Core i5 CPU. Each compilation is cached, so the time drops to 5 seconds after the initial compilation.

MEMORY MANAGEMENT

C and C++ both require you to manually manage memory, carefully ensuring that what you allocate is deallocated once it's no longer needed. Nim, on the other hand, manages memory for you using a garbage collector. But there are situations when you may want to avoid garbage collectors; they're considered by many to be inadequate for certain application domains, like embedded systems and games. For this reason, Nim supports a number of different garbage collectors with different applications in mind. The garbage collector can also be removed completely, giving you the ability to manage memory yourself.

> **GARBAGE COLLECTORS** Switching between garbage collectors is easy. You just need to specify the `--gc:<gc_name>` flag during compilation and replace `<gc_name>` with `markandsweep`, `boehm`, or `none`.

This was just a small taste of Nim's most prominent features. There's a lot more to it: not just the unique and innovative features, but also the unique composition of features from existing programming languages that makes Nim as a whole very unique indeed.

1.1.3 How does Nim work?

One of the things that makes Nim unique is its implementation. Every programming language has an implementation in the form of an application, which either interprets the source code or compiles the source code into an executable. These implementations are called an *interpreter* and a *compiler*, respectively. Some languages may have multiple implementations, but Nim's only implementation is a compiler. The compiler compiles Nim source code by first translating the code to another programming language, C, and then passing that C source code to a C compiler, which then compiles it into a binary executable. The executable file contains instructions that indicate the specific tasks that the computer should perform, including the ones specified in the original Nim source code. Figure 1.2 shows how a piece of Nim code is compiled into an executable.

The compilers for most programming languages don't have this extra step; they compile the source code into a binary executable themselves. There are also others that don't compile code at all. Figure 1.3 shows how different programming languages transform source code into something that can be executed.

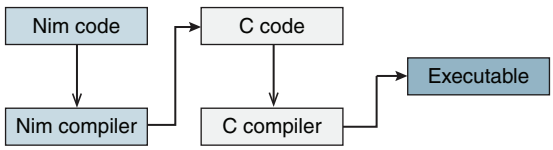


Figure 1.2 How Nim compiles source code

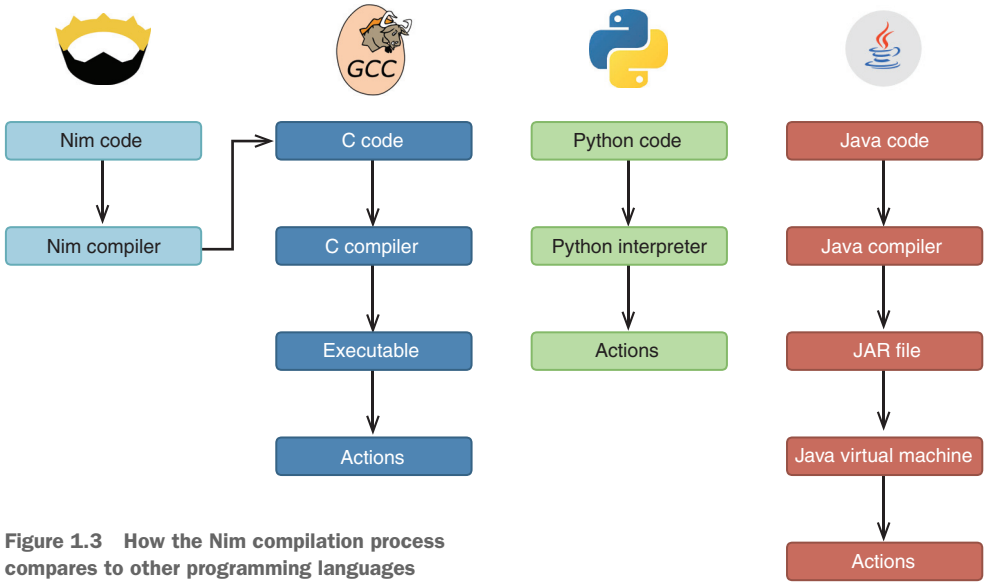


Figure 1.3 How the Nim compilation process compares to other programming languages

Nim connects to the C compilation process in order to compile the C source code that was generated by it. This means that the Nim compiler depends on an external C compiler, such as GCC or Clang. The result of the compilation is an executable that's specific to the CPU architecture and OS it was compiled on.

This should give you a good idea of how Nim source code is transformed into a working application, and how this process is different from the one used in other programming languages. Every time you make a change to your Nim source code, you'll need to recompile it.

Now let's look at Nim's positive and negative aspects.

1.2 Nim's benefits and shortcomings

It's important to understand why you might want to use a language, but it's just as important to learn why that language may not be correct for your particular use case.

In this section, I'll compare Nim to a number of other programming languages, focusing on a variety of characteristics and factors that are typically used in such comparisons. After that, I'll discuss some of the areas where Nim still needs to catch up with other languages.

1.2.1 Benefits

As you read this book, you may wonder how Nim compares to the programming languages that you're familiar with. There are many ways to draw a comparison and multiple factors that can be considered, including the language's execution speed, expressiveness, development speed, readability, ecosystem, and more. This section looks at some of these factors to give you a better idea of the benefits of Nim.

NIM IS EFFICIENT

The speed at which applications written in a programming language execute is often used in comparisons. One of Nim's goals is efficiency, so it should be no surprise that it's a very efficient programming language.

C is one of the most efficient programming languages, so you may be wondering how Nim compares. In the previous section, you learned that the Nim compiler first translates Nim code into an intermediate language. By default, the intermediate language is C, which suggests that Nim's performance is similar to C's, and that's true.

Because of this feature, you can use Nim as a complete replacement for C, with a few bonuses:

- Nim has performance similar to C.
- Nim results in software that's more reliable than software written in C.
- Nim features an improved type system.
- Nim supports generics.
- Nim implements an advanced form of metaprogramming.

In comparison to C, metaprogramming in Nim is unique, as it doesn't use a preprocessor but is instead a part of the main compilation process. In general, you can expect to find many modern features in Nim that you won't find in C, so picking Nim as a C replacement makes a lot of sense.

Table 1.1 shows the results of a small benchmark test.[1] Nim matches C's speed and is significantly faster than Python.

Table 1.1 Time taken to find which numbers from 0 to 100 million are prime

Programming language	Time (seconds)
C	2.6
Nim	2.6
Python (CPython)	35.1

In this benchmark, the Nim application's runtime matches the speed of the C app and is significantly faster than the app implemented in Python. Micro benchmarks such as this are often unreliable, but there aren't many alternatives. Nim's performance matches that of C, which is already one of the most efficient programming languages out there.

NIM IS READABLE

Nim is a very expressive language, which means that it's easy to write Nim code that's clear to both the compiler and the human reader. Nim code isn't cluttered with the curly brackets and semicolons of C-like programming languages, such as JavaScript

[1] You can read more about this benchmark test on Dennis Felsing's HookRace blog: http://hookrace.net/blog/what-is-special-about-nim/#good-performance.

and C++, nor does it require the `do` and `end` keywords that are present in languages such as Ruby.

Compare this expressive Nim code with the less-expressive C++ code

Listing 1.3 Iterating from 0 to 9 in Nim

```
for i in 0 .. <10:
  echo(i)
```

Listing 1.4 Iterating from 0 to 9 in C++

```
#include <iostream>
using namespace std;

int main()
{
   for (int i = 0; i < 10; i++)
   {
       cout << i << endl;
   }

   return 0;
}
```

The Nim code is more readable and far more compact. The C++ code contains many elements that are optional in Nim, such as the `main` function declaration, which is entirely implicit in Nim.

Nim is easy to write but, more importantly, it's also easy to read. Good code readability goes a long way. For example, it makes debugging easier, allowing you to spend more time writing beautiful Nim code, cutting down your development time.

NIM STANDS ON ITS OWN

This has been mentioned already, but it's worth revisiting to describe how other languages compare, and in particular why some require a runtime.

Compiled programming languages such as Nim, C, Go, D, and Rust produce an executable that's native to the OS on which the compiler is running. Compiling a Nim application on Windows results in an executable that can only be executed on Windows. Similarly, compiling it on Mac OS results in an executable that can only be executed on Mac OS. The CPU architecture also comes into play: compilation on ARM results in an executable that's only compatible with ARM CPUs. This is how things work by default, but it's possible to instruct Nim to compile an executable for a different OS and CPU combination through a process known as *cross-compilation*.

Cross-compilation is usually used when a computer with the desired architecture or OS is unavailable, or the compilation takes too long. One common use case would be compiling for ARM devices such as the Raspberry Pi, where the CPU is typically slow. More information about cross-compilation can be found in the Nim Compiler User Guide: http://nim-lang.org/docs/nimc.html#cross-compilation.

Among other things, the JVM was created to remove the need for cross-compilation. You may have heard the phrase "write once, run anywhere." Sun Microsystems created

this slogan to illustrate Java's cross-platform benefits. A Java application only needs to be compiled once, and the result of this compilation is a JAR file that holds all the compiled Java classes. The JAR file can then be executed by the JVM to perform the programmed actions on any platform and architecture. This makes the JAR file a platform- and architecture-agnostic executable. The downside to this is that in order to run these JAR files, the JVM must be installed on the user's system. The JVM is a very big dependency that may contain bugs and security issues. But on the other hand, it does allow the Java application to be compiled only once.

Python, Ruby, and Perl are similar. They also use a virtual machine (VM) to execute code. In Python's case, a VM is used to optimize the execution of Python code, but it's mostly hidden away as an implementation detail of the Python interpreter. The Python interpreter parses the code, determines what actions that code is describing, and immediately executes those actions. There's no compilation step like with Java, C, or Nim. But the advantages and disadvantages are mostly the same as the JVM's; there's no need for cross-compilation, but in order to execute a Python application, the system needs to have a Python interpreter installed.

Write once, run anywhere

Similar to the "write once, run anywhere" slogan, other programming languages adopted the "write once, compile anywhere" philosophy, giving a computer program the ability to be compiled on all platforms without the need to modify its source code. This applies to languages such as C, Pascal, and Ada. But these languages still require platform-specific code when dealing with more-specialized features of the OS, such as when creating new threads or downloading the contents of a web page. Nim goes a step further; its standard library abstracts away the differences between OSs so you can use a lot of the features that modern OSs offer.

Unfortunately, in many cases, virtual machines and interpreters cause more problems than they solve. The number of common CPU architectures and the most popular OSs is not that large, so compiling for each of them isn't that difficult. In contrast, the source code for applications written in interpreted languages is often distributed to the user, and they're expected to install the correct version of the interpreter or virtual machine. This can result in a lot of problems.

One example of the difficulty associated with distributing such applications is the recent introduction of Python 3. Because it's not backward compatible with the previous version, it has caused many issues for software written originally in Python 2. Python 3 was released in 2008, and as of this writing, there are still libraries written for Python 2 that don't work with the Python 3 interpreter.[2] This wouldn't be a problem with a compiled language because the binaries would still continue to work.

The lightweight nature of Nim should make it particularly appealing, especially in contrast to some of the languages mentioned in this section.

[2] See the Python 3 Readiness page for a list of Python 3–ready packages: http://py3readiness.org/.

NIM IS FLEXIBLE

There are many different styles that software can be written in. A programming paradigm is a fundamental style of writing software, and each programming language supports a different set of paradigms. You're probably already familiar with one or more of them, and at the very least you know what object-oriented programming (OOP) is because it's taught as part of many computer science courses.

Nim is a multi-paradigm programming language. Unlike some popular programming languages, Nim doesn't focus on the OOP paradigm. It's mainly a procedural programming language, with varying support for OOP, functional, declarative, concurrent, and other programming styles.

That's not to say that OOP isn't well supported. OOP as a programming style is simply not forced on you. Nim supports common OOP features, including inheritance, polymorphism, and dynamic dispatch.

To give you a better idea of what Nim's primary paradigm looks like, let's look at the one big difference between the OOP paradigm and the procedural paradigm. In the OOP paradigm, methods and attributes are bound to objects, and the methods operate on their own data structure. In the procedural paradigm, procedures are standalone entities that operate on data structures. This may be hard for you to visualize, so let's look at some code examples to illustrate it.

> **SUBROUTINE TERMINOLOGY** In this subsection I mention *methods* and *procedures*. These are simply different names for *subroutines* or *functions*. *Method* is the term used in the context of OOP, *procedure* is used in procedural programming, and *function* is used in functional programming.

The following code listings show the same application. The first is written in Python using the OOP style. The second is written in Nim using the procedural style.

Listing 1.5 Barking dog modeled using OOP in Python

```
class Dog:
  def bark(self):
    print("Woof!")

dog = Dog()

dog.bark()
```

The bark method is associated with the Dog class by being defined within it.

The bark method can be directly invoked on the dog object by accessing the method via the dot.

Listing 1.6 Barking dog modeled using procedural programming in Nim

```
type
  Dog = object

proc bark(self: Dog) =
  echo("Woof!")

let dog = Dog()
dog.bark()
```

The bark procedure isn't directly associated with the Dog type by being defined within it. This procedure could also easily be defined outside this module.

The bark procedure can still be directly invoked on the dog object, despite the fact that the procedure isn't associated with the Dog type as it is in the Python version.

In the Python code, the `bark` method is placed under the `class` definition. In the Nim code, the `bark` method (called a *procedure* in Nim) isn't bound to the `Dog` type in the same way as it is in the Python code; it's independent of the definition of the `Dog` type. Instead, its first argument specifies the type it's associated with.

You could also implement something similar in Python, but it wouldn't allow you to call the `bark` method in the same manner. You'd be forced to call it like so: `bark(dog)`, explicitly passing the `dog` variable to the method as its first argument. The reason this is not the case with Nim is because Nim rewrites `dog.bark()` to `bark(dog)`, making it possible for you to call methods using the traditional OOP style without having to explicitly bind them to a class.

This ability, which is referred to as Uniform Function Call Syntax (UFCS), has multiple advantages. It allows you to create new procedures on existing objects externally and allows procedure calls to be chained.

> **CLASSES IN NIM** Defining classes and methods in Nim in a manner similar to Python is also possible. Metaprogramming can be used to do this, and the community has already created numerous libraries that emulate the syntax. See, for example, the Nim OOP macro: https://nim-by-example.github.io/oop_macro/.

Another paradigm that Nim supports is the functional programming (FP) paradigm. FP is not as popular as OOP, though in recent years it has seen a surge in popularity. FP is a style of programming that primarily avoids the changing of state and the use of mutable data. It uses certain features such as first-class functions, anonymous functions, and closures, all of which Nim supports.

Let's look at an example to see the differences between programming in a procedural style and a functional one. The following code listings show code that separates people's full names into first and last names. Listing 1.7 shows this done in a functional style and listing 1.8 in a procedural style.

Listing 1.7 Iterating over a sequence using functional programming in Nim

Imports the sequtils, future, and strutils modules. These modules define the map, ->, and split procedures respectively.

Defines new list variable containing a list of names

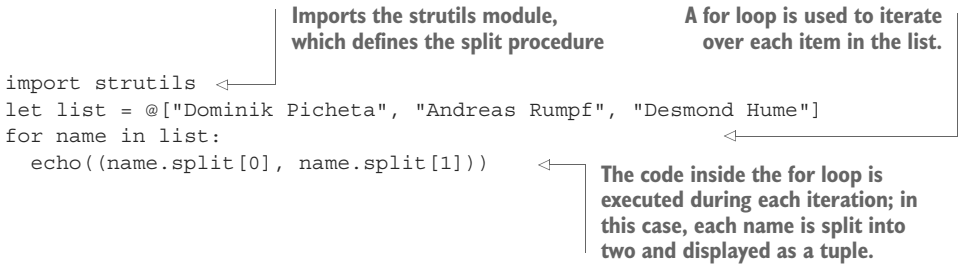
```
import sequtils, future, strutils
let list = @["Dominik Picheta", "Andreas Rumpf", "Desmond Hume"]
list.map(
  (x: string) -> (string, string) => (x.split[0], x.split[1])
).echo
```

The map procedure takes a closure that specifies how to modify each item in the list.

The map procedure is used to iterate over the list.

The modified list is then displayed on the screen.

Listing 1.8 Iterating over a sequence using a procedural style in Nim



The functional version uses the `map` procedure to iterate over the `list` variable, which contains a list of names. The procedural version uses a `for` loop. Both versions split the name into a first and last name. They then display the result in a tuple. (I'm throwing a lot of new terms at you here. Don't worry if you aren't familiar with them; I'll introduce you to them in chapter 2.) The output of the code listings will look similar to this:

```
(Field0: Dominik, Field1: Picheta)
(Field0: Andreas, Field1: Rumpf)
(Field0: Desmond, Field1: Hume)
```

THE MEANING OF FIELD0 AND FIELD1 `Field0` and `Field1` are just default field names given to tuples when a field name isn't specified.

Nim is incredibly flexible and allows you to write software in many different styles. This was just a small taste of the most popular paradigms supported by Nim and of how they compare to Nim's main paradigm. Nim also supports more-obscure paradigms, and support for others can be introduced easily using metaprogramming.

NIM CATCHES ERRORS AHEAD OF TIME

Throughout this chapter, I've been comparing Python to Nim. While Nim does take a lot of inspiration from Python, the two languages differ in one important way: Python is dynamically typed and Nim is statically typed. As a statically typed language, Nim provides a certain level of type safety that dynamically typed programming languages don't provide.

Although Nim is statically typed, it feels very dynamic because it supports type inference and generics. You'll learn more about these features later in the book. For now, think of it as a way to retain the high development speed that dynamically typed programming languages allow, while also providing extra type safety at compile time.

In addition to being statically typed, Nim implements an exception-tracking mechanism that is entirely opt-in. With exception tracking, you can ensure that a procedure won't raise any exceptions, or that it will only raise exceptions from a predefined list. This prevents unexpected crashes by ensuring that you handle exceptions.

COMPARING DIFFERENT PROGRAMMING LANGUAGE FEATURES

Throughout this section, I've compared Nim to various other programming languages. I've discussed efficiency, the dependencies of the resulting software, the flexibility of the language, and the language's ability to catch errors before the software is deployed. Based on these characteristics alone, Nim is an excellent candidate for replacing some of the most popular programming languages out there, including Python, Java, C, and more.

For reference, table 1.2 lists different programming languages and shows some of the features that they do and don't support.

Table 1.2 Common programming language features

Programming language	Type system	Generics	Modules	GC	Syntax	Metaprogramming	Execution
Nim	Static and strong	Yes	Yes	Yes, multiple and optional[a]	Python-like	Yes	Compiled binary
C	Static and weak	No	No	No	C	Very limited[b]	Compiled binary
C++	Static and weak	Yes	No	No	C-like	Limited[c]	Compiled binary
D	Static and strong	Yes	Yes	Yes, optional	C-like	Yes	Compiled binary
Go	Static and strong	No	Yes	Yes	C-like	No	Compiled binary
Rust	Static and strong	Yes	Yes	No	C-like	Limited[d]	Compiled binary
Java	Static and strong	Yes	Yes	Yes, multiple[e]	C-like	No	Executed via the JVM
Python	Dynamic and strong	N/A	Yes	Yes	Python	Yes[f]	Executed via the Python interpreter
Lua	Dynamic and weak	N/A	Yes	Yes	Modula-like[g]	Yes via Metalua	Executed via the Lua interpreter or Lua JIT compiler

[a] Nim supports ref counting, a custom GC, and Boehm. Nim also allows the GC to be switched off altogether.
[b] Some very limited metaprogramming can be achieved via C's preprocessor.
[c] C++ only offers metaprogramming through templates, limited CTFE (compile-time function execution), and no AST macros.
[d] Rust has some support for declarative macros through its `macro_rules!` directive, but no built-in procedural macros that allow you to transform the AST except for compiler plugins, and no CTFE.
[e] See the "Oracle JVM Garbage Collectors Available From JDK 1.7.0_04 And After" article on Fasterj: www.fasterj.com/articles/oraclecollectors1.shtml.
[f] You can modify the behavior of functions, including manipulating their AST, using the `ast` module, but only at runtime.
[g] Lua uses `do` and `end` keywords to delimit scope.

1.2.2 Areas where Nim still needs to improve

Nothing in this world is perfect, and programming languages are no exception. There's no programming language that can solve every problem in the most reliable and rapid manner. Each programming language has its own strengths and weaknesses, and Nim is no exception.

So far, I've been focusing on Nim's strengths. Nim has many more fine aspects that I haven't yet mentioned, and you'll discover them throughout this book. But it would be unfair to only talk about Nim's strengths. Nim is still a young programming language, so of course it can still improve.

NIM IS STILL YOUNG AND IMMATURE

All programming languages go through a period of immaturity. Some of Nim's newer and more-advanced features are still unstable. Using them can result in buggy behavior in the compiler, such as crashes, though crashes don't happen very often. Importantly, Nim's unstable features are opt-in, which means that you can't accidentally use them.

Nim has a package manager called Nimble. Where other programming languages may have thousands of packages available, Nim only has about 500. This means that you may need to write libraries for certain tasks yourself. This situation is, of course, improving, with new packages being created by the Nim community every day. In chapter 5, I'll show you how to create your own Nimble packages.

NIM'S USER BASE AND COMMUNITY IS STILL QUITE SMALL

Nim has a small number of users compared to the mainstream programming languages. The result is that few Nim jobs exist. Finding a company that uses Nim in production is rare, but when it does happen, the demand for good Nim programmers can make the salaries quite high.

On the other hand, one of the most unique things about Nim is that its development is exceptionally open. Andreas Rumpf (Nim's creator) and many other Nim developers (including me) openly discuss Nim's future development plans on GitHub and on IRC. Anyone is free to challenge these plans and, because the community is still quite small, it's easy to do so. IRC is also a great place for newcomers to ask questions about Nim and to meet fellow Nim programmers.

> **IRC** Take a look at appendix A for details on how to connect to Nim's IRC channel.

These problems are temporary. Nim has a bright future ahead of it, and you can help shape it. This book teaches you how.

1.3 Summary

- Created by Andreas Rumpf in 2005, Nim is still a very new programming language; it hasn't yet reached version 1.0. Because Nim is so new, it's a bit immature and its user base is relatively small.

- Nim is efficient, expressive, and elegant (in that order).
- Nim is an open source project that's developed entirely by the Nim community of volunteers.
- Nim is general-purpose programming language and can be used to develop anything from web applications to kernels.
- Nim is a compiled programming language that compiles to C and takes advantage of C's speed and portability.
- Nim supports multiple programming paradigms, including OOP, procedural programming, and functional programming.