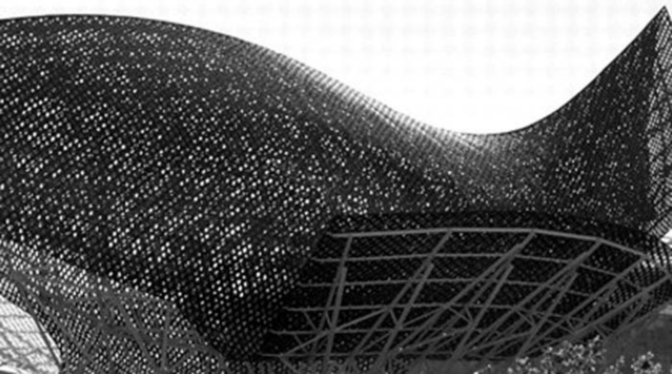
David Chisnall

ESSENTIAL GO CODE AND IDIOMS FOR ALL FACETS OF THE DEVELOPMENT PROCESS

The Go Programming Language

PHRASEBOOK



The Go Programming Language

PHRASEBOOK

David Chisnall

Addison-Wesley

DEVELOPER'S LIBRARY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data:

Chisnall, David.
The Go programming language phrasebook / David Chisnall.
p. cm.
Includes index.
ISBN 978-0-321-81714-3 (pbk. : alk. paper) — ISBN 0-321-81714-1 (pbk. : alk. paper)
1. Go (Computer program language) 2. Computer programming. 3. Open source software. I. Title.
QA76.73.G63C45 2012
005.3—dc23

2012000478

ISBN-13: 978- 0-321-81714-3
ISBN-10: 0-321-81714-1
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing: March 2012

Editor-in-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Marketing Manager
Stephane Nakib

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Gayle Johnson

Publishing Coordinator
Andrea Bledsoe

Cover Designer
Gary Adair

Senior Compositor
Gloria Schurick

Table of Contents

This page intentionally left blank

About the Author

David Chisnall is a freelance writer and consultant. While studying for his PhD, he cofounded the Étoilé project, which aims to produce an open-source desktop environment on top of GNUstep, an open-source implementation of the OpenStep and Cocoa APIs. He is an active contributor to GNUstep and is the original author and maintainer of the GNUstep Objective-C 2 runtime library and the associated compiler support in the Clang compiler. He is also a FreeBSD committer working various aspects of the toolchain, including being responsible for the new C++ stack.

After completing his PhD, David hid in academia for a while, studying the history of programming languages. He finally escaped when he realized that there were places off campus with an equally good view of the sea and without the requirement to complete quite so much paperwork. He occasionally returns to collaborate on projects involving modeling the semantics of dynamic languages.

When not writing or programming, David enjoys dancing Argentine tango and Cuban salsa, playing badminton and ultimate frisbee, and cooking.

Acknowledgments

The first person I'd like to thank is Mark Summerfield, author of *Programming in Go: Creating Applications for the 21st Century.* If you finish this book and want to learn more, I'd recommend you pick up a copy. Mark was the person responsible for making me look at Go in the first place.

The next person I need to thank is Yoshiki Shibata. Yoshiki has been working on the Japanese translation of this book and, in doing so, has sent me countless emails highlighting areas that could be improved. If you enjoy reading this book then Yoshiki deserves a lot of the credit.

Finally, I need to thank everyone else who was involved in bringing this book from my text editor to your hands. A lot of people have earned some credit along the way. In particular, Debra Williams-Cauley, who masterminded the project, and Anne Goebel, who shepherded the book from a draft manuscript to the version you now hold.

A Go Primer

One of the goals of Go was a consistent and unambiguous syntax. This makes it easy for tools to examine Go programs, and also makes it easy to learn. Unhelpful compiler errors make it difficult to learn a language, as anyone who has made a typo in C++ code using templates will know.

In C, for example, function and global variable declarations have almost the same syntax. This means that the compiler can't easily tell which one you meant if you make an error. It gives you helpful error messages like "expected ;" on a line where you don't think a semicolon is expected at all.

The Go grammar was designed to make it possible for the compiler to tell you more accurately what you did wrong. It was also designed to avoid the need to state something that can be easily inferred. For example, if you create a variable and set its value to 42, the compiler could probably guess that this variable should be an integer, without it being explicitly stated. If you initialize it with a function call, then the compiler can definitely tell that the type should be whatever the function returned. This was the same problem that C++ 2011 solves with the `auto` type.

Go adopts JavaScript's idea of *semicolon insertion*, and takes it a step further. Any line that can be interpreted as a complete statement has a semicolon implicitly inserted at the end by the parser.[1] This means that Go programs can freely omit semicolons as statement terminators. This adds some constraints, for example enforcing a brace style where open braces are at the end of the line at the start of flow-control statements, rather than on their own. If you happen to be a human, this is unfortunate, because it means that you can't use the highly optimized symmetry recognition paths, which evolution has spent the last million or so years optimizing in your visual cortex, for recognizing code blocks.

This chapter contains an overview of Go syntax. This is not a complete reference. Some aspects are covered in later chapters. In particular, all of the concurrency-related aspects of Go are covered in Chapter 9, *Goroutines*.

[1]This is an oversimplification. The exact rules for semicolon insertion are more complicated, but this rule of thumb works in most cases.

The Structure of a Go Source File

```go
package main
import "fmt"

func main() {
  fmt.Printf("Hello World!\n")
}
```

From: hello.go

A Go source file consists of three parts. The first is a **package** statement. Go code is arranged in packages, which fill the rôles of both libraries and header files in C. The package in this example is called `main`, which is special. Every program must contain a `main` package, which contains a `main()` function, which is the program entry point.

The next section specifies the packages that this file uses and how they should be imported. In this example, we're importing the `fmt` package.

Once the `fmt` package has been imported, any of its exported types, variables, constants, and functions can be used, prefixed by the name of the package. In this simple example, we're calling `Printf()`, a function similar to C's `printf`, to print "Hello World!" in the terminal.

Although Go uses static compilation, it's important to realize that **`import`** statements are much closer to Java or Python import directives than to C inclusions. They do not include source code in the current compilation unit. Unlike Java and Python packages, Go packages are imported when the code is linked, rather than when it is run. This ensures that a Go application will not fail because of a missing package on the deployment system, at the cost of increasing the size of the executable. Packages in Go are more important than in languages like Java, because Go only provides access control at the package level, while Java provides it at the class level.

When you compile a package (from one or more `.go` files) with the `Gc` compiler, you get an object code file for the package. This includes a metadata section that describes the types and functions that the package exports. It also contains a list of the packages that this package imports.

The input to the `6l` linker is always a `.6` file for the `main` package. This file contains references to every package that the `main` package imports, which may in turn reference further packages. The linker then combines them all.

This eliminates one of the most irritating problems with building complex C programs: you include a header, and then have to work out which library provided it and add the relevant linker flags. With Go, if a package compiles, it will link. You don't have to provide any extra flags to the linker to tell it to link things that you've referenced via **`import`** directives.

The remainder of a Go file contains declarations of types, variables, and functions. We'll explore that for the rest of this chapter.

You may find that you have two packages that you want to import that have the same name. This would cause problems in Go. The badStyleImport.go example is functionally equivalent to the example at the start of this section but renames the `fmt` package, calling it `format`. Renaming a package when you import it is usually a bad idea, because it makes your code harder for people to read. You should only ever use it when you explicitly need to disambiguate two packages with the same name.

```go
package main
import format "fmt"

func main() {
  format.Printf("Hello World!\n")
}
```

From: badStyleImport.go

Declaring Variables

```
var i int
var Θ float32
var explicitly, typed, pointers *complex128
int_pointer := &i
another_int_pointer := new(int)
generic_channel := make(chan interface{})
```

From: variables.go

Variables are declared with the **var** keyword, followed by the variable name, and finally by the type. The existence of a specific keyword for variable declarations makes it easy to differentiate them from other types of statements.

Writing the type at the end looks weird to people familiar with C-family languages, but it makes sense when you read the code. A (typed) variable declaration is an instruction saying, for example, "declare the variable foo to have the type int."

One of the variables declared at the start of this section uses θ (theta) as a variable name. Go permits identifiers to start with any symbols that Unicode classes as letters. This can sometimes be very useful, such as if variable names are mathematical quantities. Don't abuse it, though: the person maintaining your code will not thank you if you use characters that he can't type on his keyboard for frequently used variables.

A declaration statement may declare multiple variables, but they all have the same type. In C, some may have the type that is written at the start of the declaration, some may be pointers to that type, some may be pointers to pointers to that type, and so on. The form used by Go is far less prone to ambiguity.

You will rarely use the long form of declarations. One of the key ideas in writing good code is the *principle of minimum scope*. This means that the scope of a variable—the lexical region where it is valid—should be as small as possible for the variable's lifetime. One corollary of this is that variables should be declared immediately before their first use and initialized as part of their declaration.

Go provides a shorthand syntax, the `:=` initialization operator, which does this. Using this notation, you can declare and initialize a variable in a single statement. More importantly, you avoid the need to declare a type for the variable: the type of the variable is the type of the expression used to initialize it.

The example at the start of this section shows both kinds of declaration. It also introduces Go's syntax for pointers. The variable `int_pointer` is initialized using the *address-of operator (&)*. This should be familiar to C programmers: it returns the address in memory of an object. The returned value, however, is more similar to a Java reference than a C pointer. You can't perform arithmetic using Go pointers, nor use them interchangeably with arrays. As with Java references, you can pass Go pointers around without having to worry about when the underlying object will be deallocated. It will automatically be freed when the last reference is destroyed. Unlike Java references, you can make pointers to primitive types, not just to structures (Go's equivalent of objects).

In this example, you could return `int_pointer` from this function without any problems. This may seem strange to C programmers, because it points to a variable declared locally. The Go compiler will try to allocate `i` on the stack, but that's just an implementation detail. If its address is taken and it is returned from the function then it will be allocated on the heap instead.

This example creates another integer pointer, in a different way. The `new()` built-in function creates a new integer and returns a pointer to it. This is semantically equivalent to declaring an integer variable and then taking its address. Neither guarantees how the underlying storage will be allocated. You can pass any type to `new()`, but it is not the standard way of allocating everything.

Go includes three special types, which we'll look at in a lot more detail later in this book: *slices*, *maps*, and *channels*. These are *reference types*, meaning that you always access them via a reference. If you assign one map-typed variable to another, then you will have two variables referring to the same map. In contrast, if you assign one integer-typed variable to another, then you will have two variables with the same value, but modifying one will not affect the other.

Instances of reference types in Go are created with the `make()` built-in function. This is similar to `new()`, but also performs initialization of the built-in types. Values returned by `new()` are simply zeroed. They are not guaranteed to be immediately useful, although good style suggests that they should be.

Declaring Functions

```go
func printf(str string, args ...interface{}) (int
    , error) {
  _, err := fmt.Printf(str, args...)
  return len(args), err
}

func main() {
  count := 1
  closure := func(msg string) {
    printf("%d %s\n", count, msg)
    count++
  }
  closure("A Message")
  closure("Another Message")
}
```

From: functions.go

Functions in Go are declared using the **`func`** keyword. As with variable declarations, the return type goes at the end. This can be a single value, or a list of values. The `printf()` function in the example shows several important features of Go. This is a *variadic function*, which returns multiple values: an integer and an error. The integer is the number of variadic arguments passed to it, and the error code is one of the values returned from the `Printf()` function from the *fmt package*.

Note the syntax for calling functions that return multiple values. The return values must either all be ignored, or all assigned to variables. The *blank identifier*, `_`, can be used for values that you wish to discard.

Variadic functions in Go are particularly interesting. In C, a variadic function call just pushes extra parameters onto the stack, and the callee has to know how to pop them off. In Go, all variadic parameters are delivered as a *slice* (see Chapter 5, *Arrays and Slices*; for now you can think of a slice as being like an array). The variadic parameters must all be of the same type, although you can use the *empty interface type* (**`interface`**`{}`) to allow variables of any type and then use type introspection to find out what they really are.

The `main()` function in the example is the program entry point. Unlike many other languages, this takes no arguments. Commandline arguments and environment variables are stored globally in Go, making it easy to access them from any function, not just one near the program entry point.

Inside this function, you'll see a closure defined. Closures in Go are declared as anonymous functions, inside other functions. The closure can refer to any variables in the scope where it is declared. In this example, it refers to the `count` variable from the outer function's scope. It would continue to do so even after the outer function returned. In Go, there is no distinction between heap and stack allocated variables, except at the implementation level. If a local variable is referenced after the function that contains it, then it is not freed when the function returns. If `closure` were stored in a global variable, for example, then `count` would not be deallocated, even after the function returned.

Looping in Go

```go
package main
import "fmt"

func main() {
  loops := 1
  // while loop:
  for loops > 0 {
    fmt.Printf("\nNumber of loops?\n")
    fmt.Scanf("%d", &loops)
    // for loop
    for i := 0 ; i < loops ; i++ {
      fmt.Printf("%d ", i)
    }
  }
  // Infinite loop
  for {
    // Explicitly terminated
    break
  }
}
```

From: loop.go

In C, you have three kinds of loops, all with different syntax and overlapping semantics. Go manages to have more expressive loop semantics, but simple and uniform syntax.

Every loop in Go is a **for** statement. We'll only look at the forms that mirror C loop constructs here. The form that iterates over a collection is explained in Chapter 5, *Arrays and Slices*.

The loop.go example shows the three types of general **for** loops in Go. The last one is the simplest. This is an infinite loop, with an explicit **break** statement to terminate it. You'd most commonly use this form for an event loop that would not terminate in normal use. Like C, Go also has a **continue** statement that immediately jumps to the start of the next loop iteration, or exits the loop if the loop condition no longer holds.

Both the **break** and **continue** statements support an optional label for jumping out of nested loops. Note that the label is not a jump target; it is just used to identify the loop.

```go
  for i := 0 ; i<10 ; i++ {
L:
    for {
      for {
        break L
      }
    }
    fmt.Printf("%d\n", i)
  }
```

From: break.go

You can see this in the break.go example. The **break** statement jumps out of the two inner loops, but does not prevent the `Printf` call from running. It jumps to the end of the loop immediately after `L:`, not to the start.

Most of the time, you won't use infinite loops and explicit escapes. The other two types of **for** loops in Go are analogous to `while` and **for** loops in C and the older form of **for** loops in

Java. The outer loop in the example at the start of this section will test its condition and loop as long as it is true. The inner loop first performs the initialization (`i := 0`) then tests the loop condition (`i < loops`), and runs the loop clause as long as it's true. Between each loop iteration, it runs the increment clause (`i++`). If you've used any vaguely C-like language, then this will be very familiar to you. The only difference between a Go **`for`** loop and a C **`for`** loop is that the Go version does not require brackets.

There are a couple of interesting things in this loop. The first is the creation of the loop variable (`i`) at the loop scope. This is similar to C99 or C++. The variable that is declared in the loop initialization clause is only in scope for the duration of the loop.

The second is the *increment statement*. Note that I did not call it a *postincrement expression*. The designers of Go decided to eliminate the confusion between preincrement and postincrement expressions in C. In Go, the increment statement is not an expression, and only the suffix syntax is allowed. This line increments the variable, but it does not evaluate to anything. Writing something like `a := b++` is not valid Go. Writing `++b` is invalid in all contexts in Go: there is no prefix form of the operator.

Creating Enumerations

```
const (
  Red                = (1<<iota)
  Green              = (1<<iota)
  Blue, ColorMask    = (1<<iota), (1<<(iota+1))-1
)
```

From: enum.go

There are several places in Go where it is obvious that someone has spent a lot of thought designing exactly the right syntax for most common uses of a language feature. Enumeration constants are the most obvious example of this attention to detail.

There is no divide between constants and enumerations in Go. This mirrors their implementation in C, where enumerated types can be used interchangeably with integers. Groups of constants within the same declaration in Go are used for enumerations.

There are two common uses for enumerated types. The first is defining a set of mutually-exclusive options. The second is defining a set of overlapping flags. Typically, you'll use a sequence of numbers for the first and a sequence of powers of 2 for the second. You can then create a bitfield by bitwise-oring a combination of enumeration values together.

In C, and most other languages with enumerated types, you need to explicitly provide the numerical values for the second type of enumeration. The first will be automatically numbered by the compiler.

Go provides a much more flexible mechanism for defining enumerations. The `iota` predeclared identifier is similar to the GNU C `__COUNTER__` preprocessor macro, but it's more powerful. It is an integer constant expression. In a normal program scope, it evaluates to zero, but in the scope of a constant declaration it is initially zero but then incremented on each line where it is used.

Unlike `__COUNTER__`, `iota` is scoped. It is zero on the first line of the group in this example, and will always be zero in the first line of this group, irrespective of how it is used elsewhere. If you have multiple **`const`** groups in a single source file, then `iota` will be zero at the start of each of them.

The example at the start of this section shows how to declare a group of constants for use as an enumerated type. This simple example shows the low 3 bits of a bitfield being used to store three flags indicating the presence of three color values. The `ColorMask` constant is defined to provide the value that must be bitwise-and'd with an integer to give the three color flags.

It's possible to reference constants from other constant declarations, so you can combine this kind of declaration easily. For example, you could provide another constant declaration describing another group of flags within a set, and then extend this declaration to use them in the next few bits of the bitfield.

Similarly, you can extend existing constant declarations by inserting another `iota` expression earlier. This will then renumber all subsequent values, so it's important to be careful when the constants are part of a binary interface.

Constants—and therefore enumerations—in Go are not limited to integers. Other types can be specified in the same way. The enum.go example also shows the declaration of the complex constant `i`, with the same definition as in mathematics.

```go
const (
  i complex128 = complex(0, 1)
)
```

From: enum.go

Declaring Structures

```go
type Example struct {
  Val string
  count int
}
```

From: struct.go

Structures in Go are somewhat richer than

C structures. One of the most important differences is that Go structures automatically support *data hiding*.

Any top-level type, method, or variable name that starts with a capital letter is visible outside of the package in which it is declared. This extends to structure fields. In C, if you only put some fields from a structure in a header, then you will encounter problems when someone tries to allocate an instance of it on the stack: his compiler won't allocate enough space for it. Go packages export the offsets of the public fields. This allows them to be created and their public fields accessed from other compilation units.

The example at the start of this section defines a structure with two fields. The first, a string, is public and can be accessed from anywhere. The second, an integer, is private and is only visible to code in the same package as this definition. A structure doesn't have to declare any public fields. You can create *opaque types* by defining a structure where all of the fields are private.

If you're coming from a class-based language like C++ or Java, then you may be wondering why there are public and private fields, but not protected ones. The answer is quite simple: there is no inheritance in Go, so protected would have no meaning. Public and private also have slightly different meanings in Go and a language like Java. A private field in a Go structure can be accessed by any code in the same package, not just by methods of that structure. If you come from Objective-C, then you can think of private fields in Go structures like `@package` instance variables in Objective-C. If you come from C++, then think of all Go functions in a package as implicitly being friends of all structures declared in the same package.

Defining Methods

```go
type integer int
func (i integer) log() {
  fmt.Printf("%d\n", i);
}
func (e *Example) Log() {
  e.count++
  fmt.Printf("%d %s\n", e.count, e.Val)
}
```

From: methods.go

If you've used a class-based language, then you are probably wondering why the last example didn't define any methods defined on the structure. In Go, you may define methods on any concrete type that you define, not just on structures. The example at the start of this section defines two `Log()` methods—recall that the uppercase start letter makes them public—one on the structure defined in the last section and one on a named integer type.

The Go type system lets you assign any `int` to this named type without an explicit cast, but not vice versa. It also prevents you from assigning between two named types. This can be very useful for variables representing quantities. You could, for example, define kilometer and mile types and have the compiler reject any code where you attempted to assign one to the other.

You cannot add methods to existing types—Go does not have an equivalent of Objective-C categories—but you can define new named types and add methods to them.

Methods are declared just like functions, except that there is one extra parameter—the *receiver*—declared before the function name. One of the interesting syntactic quirks of Go is that there is no `this` or `self` keyword. You can give the receiver any name that you want, and this name does not have to be consistent between methods. This idea comes from Oberon-2 and should be popular with people who like the “no magic” philosophy of languages like Objective-C: the receiver is not an implicit hidden parameter that the compiler inserts; it is an explicit parameter just like any other.

The method on the structure in the example at the start of this section takes a pointer as the receiver. This means that it can modify fields of the receiver and these changes will be shared. Methods do not have to take pointers: the other method in the example takes a value. If a method takes a value type, then it can still be called with either a value or a pointer, but it will receive a copy of the structure, so changes that it makes will not be visible from the caller.

When talking about expressions with an explicit type, methods are just functions. You call a method on a structure by using the dot notation, and you declare the parameter that declares how the structure is passed to the method in a special way, but this is just some syntactic sugar. Methods called in this way are semantically equivalent to functions that just take the receiver as an argument: they are statically resolved and are just function calls.

That's not the real power of methods, though. When you call a method via an interface (described in the next section), you get late-bound dynamic lookup. This dual nature of Go methods means that you have a single abstraction that can be used in the same way as either C types or Smalltalk objects. If you require performance, then you can use statically typed definitions and avoid the dynamic lookup. If you require flexibility, then you can use the late binding mechanism of interfaces.

Implementing Interfaces

```go
type cartesianPoint struct {
  x, y float64
}
type polarPoint struct {
  r, θ float64
}

func (p cartesianPoint) X() float64 {return p.x }
func (p cartesianPoint) Y() float64 {return p.y }
func (p polarPoint) X() float64 {
  return p.r*math.Cos(p.θ)
}
func (p polarPoint) Y() float64 {
  return p.r*math.Sin(p.θ)
}
func (self cartesianPoint) Print() {
  fmt.Printf("(%f, %f)\n", self.x, self.y)
}
func (self polarPoint) Print() {
  fmt.Printf("(%f, %f°)\n", self.r, self.θ)
}
type Point interface {
  Printer
  X() float64
  Y() float64
}
type Printer interface {
  Print()
}
```

From: interface.go

The dynamic dispatch mechanism in Go is reminiscent of StrongTalk, a strongly typed Smalltalk dialect. Interfaces describe a set of methods that a type understands. Unlike Java interfaces or Objective-C protocols, they do not need to be explicitly adopted.

Any type can be assigned to a variable with an interface type, as long as it implements all of the required methods. In some cases, this can be checked at compile time. For example, if one interface is a superset of another, then casting from the superset to the subset is always valid, as is casting from a structure type to an interface when the compiler sees that the structure implements the interface.

In other cases, it is not. These cases require a *type assertion*, detailed in the next section, which will generate a runtime panic if they fail. This means that any variable with an interface type is guaranteed to either be `nil`, or hold a valid value of a concrete type that implements the interface.

The example from the start of this section shows the creation of new structure types, and interfaces that they implement. Note that the structure can be defined before the interface. In fact, structures can be defined in entirely different packages to interfaces. This is especially useful if various third-party structures all implement the same method or group of methods: you can define a new interface that can be any one of them.

There are two interfaces declared in this example, both following Go naming conventions. The `Printer` interface defines a single method,

Note: If you're coming from C++, then you should use interfaces in most of the places where you'd use templates in C++. Rather than defining template functions or classes (which Go doesn't support), define an interface that specifies the set of methods that you need, and use it where you would use the template parameter in C++.

so it follows the convention of appending the -er suffix to the method name to give the interface name.

The other interface uses *interface composition* to extend the `Printer` interface. This one defines an abstract data type. It provides methods for accessing the horizontal and vertical coordinates of a two-dimensional point. Interface composition is effectively equivalent to interface inheritance in Java. You can use it in some places where you would consider using single or multiple inheritance in other languages.

This example provides two structures that implement this interface, one using Cartesian and the other using polar coordinates. This is a simple example of how an interface can be used to hide the implementation. The two structures both start with lowercase letters, so they will not be exported from this package, while the interfaces will. You could extend this example by providing functions to construct a `Point` from polar and Cartesian coordinates, each returning one of a different kind of structure.

When you are dealing with interfaces, the distinction between methods that take pointers and ones that take values becomes more important. If you tried to assign an instance of this example structure to an interface that required the `Log()` method, then the assignment would be rejected. Assigning a pointer to an instance of this structure would work.

This seems counterintuitive. If you have a value, then you can always take its address to get a pointer, so why are the two method sets distinct? The answer is very simple: it helps avoid bugs. When you pass a value, you create a copy of a structure. When you pass a pointer, you alias the structure. If you pass a value and then implicitly, via method invocation on an interface, pass a pointer, then any changes that the method made would be made to the temporary copy, not to the original structure. This is probably not what you want, and if it is then you can just pass the pointer originally, rather than the copy.

The Go FAQ gives an example of a case where this could be problematic:

```
var buf bytes.Buffer
io.Copy(buf, os.Stdin)
```

The `io.Copy()` function copies data from something that implements the `io.Reader` interface to something that implements the `io.Writer` interface. When you call this function, it will pass a copy of `buf` as the first argument, because Go always passes by value, not by reference. It will then try to copy data from the standard input into the new copy of `buf`. When the function returns, the copy of `buf` will no longer be referenced, so the garbage collector will free it.

What the person writing this code probably wanted to do was copy data from the standard input into `buf`. The Go type system will reject this, because `buf` does not implement the `io.Writer` interface: the method for writing bytes to a buffer modifies the buffer and therefore requires a pointer receiver. By disallowing this, Go lets you get an error at compile time and trivially fix it by writing this instead:

```
var buf bytes.Buffer
io.Copy(&buf, os.Stdin)
```

If Go allowed values to use methods that are declared as requiring a pointer, then you would instead spend ages wondering why this line appeared to be reading the correct amount of data, but wasn't storing any of it in the buffer that you declared. This is part of the Go philosophy of avoiding ambiguity. It just takes one extra character to make a pointer when you need one. That small amount of extra effort is a lot less than the time you'd spend debugging code where you meant one thing and Go assumed that you meant something else.

Casting Types

```go
type empty interface {}
type example interface {
  notImplemented()
}

func main() {
  one := 1
  var i empty = one
  var float float32
  float = float32(one)
  switch i.(type) {
    default:
      fmt.Printf("Type error!\n")
    case int:
      fmt.Printf("%d\n", i)
  }
  fmt.Printf("%f\n", float)
  // This will panic  at run time
  var e example = i.(example)
  fmt.Printf("%d\n", e.(empty).(int))
}
```

From: cast.go

Unlike C, Go does not allow implicit casting. This is not laziness on the part of the implementors: implicit casting makes it easy for very subtle bugs to slip into code. I recently had to find a bug in some code that had gone undetected for several years, where an implicit cast meant that a value was incorrectly initialized. The code looked correct, until you checked the type declarations of everything involved, which were spread over multiple files.

This is another example of the Go philosophy. You should never need to state the obvious to the compiler, but you should always have to explicitly specify things that are otherwise ambiguous.

The example at the start of this section shows several casts. The concept of casting in other languages is embodied by two concepts in Go. The first is *type conversion*; the second is *type assertion*.

A type conversion is similar to a cast in C. It reinterprets the value as a new type. The conversion from `int` to `float32` is an example of this. The resulting value is a new floating-point value with the same value as the integer. In some cases, the conversion is only approximate. For example, a conversion in the other direction will result in truncation. A type conversion from an integer to a string type will return a single-character string interpreting the integer as a unicode value.

Type assertions are more interesting. They do not convert between types; they simply state to the compiler that the underlying value has the specified type. This assertion is checked at run time. If you try running this example, you will see that it aborts with a runtime panic.

This is because of the type assertion telling the compiler that the type of `i` is something that implements the `example` interface. In fact, the underlying type is `int`, which does

```
1
1.000000
panic: interface conversion: int is not main.
     example: missing method notImplemented

goroutine 1 [running]:
main.main()
  /Users/theraven/Documents/Books/GoPhrasebook/
       startsnippets/cast.go:22 +0x20d

goroutine 2 [syscall]:
created by runtime.main
  /Users/theraven/go/src/pkg/runtime/proc.c:219
exit status 2
```

Output from: cast.go

not implement the `notImplemented()` method that this interface specifies. The type check fails on the type assertion. If you come from C++, you can think of type assertions as roughly equivalent to a `dynamic_cast` that throws an exception[2] on failure.

The final cast-like construct in Go is the *type switch statement*. This is written like a normal **`switch`** statement, but the switch expression is a type assertion to **`type`** and the cases have type names, rather than values.

The type switch in the example is used as a simple type check, like a C++ `dynamic_cast`.

[2]Runtime panics are not quite analogous to C++ exceptions. The differences are covered in Chapter 8, *Handling Errors*.

It is more common to use type switches when defining generic data structures (see Chapter 4, *Common Go Patterns*) to allow special cases for various types.

Index