# A Go Developer's Notebook

Eleanor McHugh

```go
package main
import (
  . "fmt"
  . "net/http"
  "sync"
)

const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"

func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })

  var servers sync.WaitGroup
  servers.Add(1)
  go func() {
    defer servers.Done()
    ListenAndServe(ADDRESS, nil)
  }()

  servers.Add(1)
  go func() {
    defer servers.Done()
    ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
  }()
  servers.Wait()
}
```

# A Go Developer's Notebook

## or, What I Did During My Holidays

Eleanor McHugh

This book is for sale at http://leanpub.com/GoNotebook

This version was published on 2014-10-06


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# Tweet This Book!

Please help Eleanor McHugh by spreading the word about this book on Twitter!

The suggested hashtag for this book is #GoNotebook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#GoNotebook

*For Uriel*

*Fearless pioneer and homesteader of the electronic frontier.*

*[1982 - 2012]*

# Contents

# Preface

It was a cold, dark night at the tail-end of 1980 and I was attending an open evening for a local secondary school when I first met a digital computer, one of a series of steel-framed rack-mount monstrosities with trailing ribbon cables and large clunky floppy drives that would have looked perfectly at home in the Soviet space programme. These were clustered in a long, narrow, beige room lit by anaemic fluorescent tubes and dominated by an ageing teletype machine, in the midst of one of those prefab 1960s science blocks so popular with the British educational establishment.

This certainly wasn't the first time I'd seen a computer. That honour probably belongs to "Tomorrow's World" or some science fiction show on the TV. My first clear memory though is from a couple of years earlier when I found a book on technology in my local library with photos of the LEO 1, describing how these electronic machines could 'think' and 'solve problems'. This was a common theme of the 1970s with sentient computers on TV shows like "Blake's 7" and in movies such as "Futureworld" and "Logan's Run".

Experience has made me all too painfully aware that it's actually programmers who do the thinking and problem solving but to an unsullied eight year old boffinette the idea that a machine could 'think' was fascinating. A few months later a family friend with an interest in technology gave me a recent issue of Personal Computer World which made me realise that computers were something real people might one day play with at home, and not just the heroes of my favourite sci-fi shows. From then on anything I could find which mentioned computers was read cover to cover, time and time again.

However it was that night in 1980 which was the defining moment in my early life as it fixed my choice of secondary school, and with it my future circle of friends. The following year I joined the first year intake and by the time I received my first computer for Christmas 1983 I'd taught myself rudimentary spaghetti-code programming in out-of-hours sessions on a Research Machines 380Z, typing in program listings from home micro magazines and hobbyist-oriented books on BASIC, then hacking at them until they mostly worked.

Whilst I was dependent on the school machines I was lucky if I'd get two hours of keyboard time in a week, and that had to be juggled with my growing interest in pen-and-paper RPGs. Having a computer of my own changed all that and my relationship with code became much more nuanced, programming emerging as the steady back-beat to my teenage years, something to do when not buried in homework or hanging with friends. By the time I moved on to polytechnic I'd become a reasonably competent hacker, skilled in BASIC and latterly obsessed with FORTH. I'd even dabbled a little in Z80 assembler which back then was seen as a fairly natural thing for a kid to mess with.

You'd probably expect me to have gone on to a Comp Sci degree but that didn't seem to involve much coding, and who wants to be yawned at at parties? Instead I spent the next four years studying applied physics during the day and sitting up half the night with hacker friends, writing stupid little programs in odd languages - much to the despair of my long-suffering tutors. This was the pre-Web era when words like **spam** and **worm** were first being coined and most of the fun to be had online involved MUDs or telnet.

This book is sort of a homage to those hobbyist years. As such it's not a professional book in any usual sense of the term, although I certainly hope the material within will be of interest to professionals and useful to them in solving real world problems using Go.

To reassure the more sober amongst you that this book does indeed have some intrinsic technical merit, I should probably mention that much of the material is drawn from a series of presentations and workshops

I delivered at well-respected software development conferences between 2009 and 2014.

The main advantage of reworking this material in a book format is that I get to explore things at a more leisurely pace than with a live audience, and I hope in doing so to also recapture some of the anarchic can-do feel of those early books and magazine articles which first inspired me as a teenager. So what I'm really looking to create here is the kind of book about programming in Go that if my untutored younger self had found a copy, it would have kept her enthralled for months or even years. In some sense it's therefore an attempt at a "write your own adventure" programming course which takes it for granted that even the casual reader can learn the full measure of Go by tinkering with code.

I mention this because there are many clever, talented people in this world who'd really enjoy programming if only they thought they were able. But somehow they've been convinced that programming is difficult. Well yes, it often is. In the same way that writing a story or building with LEGO kit can be difficult. But not for the bogus reasons propagated in media depictions of typical programmers. You don't need to be good at maths (I flunked calculus), a straight-A student (I certainly wasn't) or look anything like a stereotypical nerd (I'm an ex-goth). What matters is being interested in making things, and having the perseverence to keep going when things aren't quite working how you imagine. So even if you're looking at this book somewhat nervously after thumbing through the sample chapter, and worrying that perhaps you're not the kind of person it's aimed at, please do yourself the favour of setting aside all your worries. You're most decidedly who I'm writing for, even if I am too close to the machine to have much clue **how** to write for you. I guarantee that if you're willing to invest time in to playing with the code examples and reading around the various topics I introduce that you'll get at least as much from this book as those with CS degrees.

So why have I decided to write this book about Go, and not one of the conventionally recognised beginner-friendly languages? That's a good question. To start with I refute the proposition that Go is a difficult language. It's probably the smallest practical language I know of, and the design is very clean. Despite that it features a number of powerful concepts like first-class functions and type inference which are often associated with academic languages, without the often mind-numbing theory needed to place them in context. So using Go to teach programming seems like a pretty reasonable choice.

But I've also made this decision for personal reasons.

Until a couple of years ago I was mostly known amongst my hacker friends for my deep knowledge of Ruby, a dynamic language which is both a great joy to work with and also commercially much in demand. If I were writing a conventional programming book to address an established audience then that's where I'd be most comfortable claiming expertise, but frankly there are enough good books on Ruby already (and bad ones for that matter) so I've nothing to add to that genre. And whilst I love Ruby to bits, that affection is tempered by the grim realities of commercial practice.

Go on the other hand has been an immense source of untarnished joy since I first encoutered it, fresh off a flight from Poland in November 2009. At the time it was still a little ropey around the edges with makefiles and all kinds of odd limitations, but it was already the easiest way to write concurrent programs in a C-like language and I admit I fell immediately in love.

It's that love that's sustained me through almost half a decade of unfunded Go research and that's led me to talk about it publicly at every opportunity, blagging my way onto the bill of some of the most prestigious software development conferences and ultimately leading me to write this book.

For this reason I've started with a large body of example code drawn from those conference sessions and very little text beyond this preface. In the coming months I'll be adding additional code organically either as it occurs to me or in response to readers' queries, and adding to the text iteratively on a weekly cycle.

This dear reader is where you come in. I want this book to be interesting and quirky and packed with useful content so please query anything which doesn't make sense to you and offer any ideas you have

for new material. I can't promise that every new idea will be used in quite the way you envisage but I'll do my best to turn each into an interesting code example so we can all learn more about Go. And trust me, I'll be learning just as much as you will, because programming isn't one of those subjects where we ingest a fixed set of facts and then we're done. Nor do I claim expertise in any of the esoteric subjects I'll be covering.

For those on tight budgets or who fancy an ironman challenge, I've designed the tutorial chapter **Hello World** so that it should be sufficient to learn Go without any outside assistance. It will always be available free of charge.

All source code is of course available online[1] though I highly recommend manually typing it in as you work your way through this book. It's easy to forget in an age of source control and GUI tools that coding is first and foremost a text-based process and there's value in developing muscle memory whenever we get the chance.

Now give me your hand and together lets enjoy some amazing adventures in the land of Go.

Ellie

London, 2014

[1] http://github.com/feyeleanor/GoNotebook

# Hello World

It's a tradition in programming books to start with a canonical "Hello World" example and whilst I've never felt the usual presentation is particularly enlightening, I know we can spice things up a little to provide useful insights into how we write Go programs.

Let's begin with the simplest Go program that will output text to the console.

**Example 1.1**

```
1  package main
2
3  func main() {
4    println("hello world")
5  }
```

The first thing to note is that every Go source file belongs to a package, with the **main** package defining an executable program whilst all other packages represent libraries.

```
1  package main
```

For the **main** package to be executable it needs to include a **main()** function, which will be called following program initialisation.

```
3  func main() {
```

Notice that unlike C/C++ the **main()** function neither takes parameters nor has a return value. Whenever a program should interact with command-line parameters or return a value on termination these tasks are handled using functions in the standard package library. We'll examine command-line parameters when developing **Echo**.

Finally let's look at our payload.

```
4  println("hello world")
```

The **println()** function is one of a small set of builtin generic functions defined in the language specification and which in this case is usually used to assist debugging, whilst "**hello world**" is a value comprising an immutable string of characters in utf-8 format.

We can now run our program from the command-line (Terminal on MacOS X or Command Prompt on Windows) with the command

```
$ go run 01.go
hello world
```

## Packages

Now we're going to apply a technique which I plan to use throughout this book by taking this simple task and developing increasingly complex ways of expressing it in Go. This runs counter to how experienced programmers usually develop code but I feel this makes for a very effective way to introduce features of Go in rapid succession and have used it with some success during presentations and workshops.

There are a number of ways we can artificially complicate our **hello world** example and by the time we've finished I hope to have demonstrated all the features you can expect to see in the global scope of a Go package. Our first change is to remove the builtin **println()** function and replace it with something intended for production code.

**Example 1.2**

```
1  package main
2  import "fmt"
3
4  func main() {
5    fmt.Println("hello world")
6  }
```

The structure of our program remains essentially the same, but we've introduced two new features.

```
2  import "fmt"
```

The **import** statement is a reference to the fmt package, one of many packages defined in Go's standard runtime library. A **package** is a library which provides a group of related functions and data types we can use in our programs. In this case **fmt** provides functions and types associated with formatting text for printing and displaying it on a console or in the command shell.

```
5  fmt.Println("hello world")
```

One of the functions provided by **fmt** is **Println()** which takes one or more parameters and prints them to the console with a carriage return appended. Go assumes that any identifier starting with a capital letter is part of the public interface of a package whilst identifiers starting with any other letter or symbol are private to the package.

In production code we might choose to simplify matters a little by importing the **fmt** namespace into the namespace of the current source file, which requires we change our import statement.

```
2  import . "fmt"
```

And this consequently allows the explicit package reference to be removed from the **Println()** function call.

```
5  Println("hello world")
```

In this case we notice little gain however in later examples we'll use this feature extensively to keep our code legible.

**Example 1.3**

```
1  package main
2  import . "fmt"
3
4  func main() {
5    Println("hello world")
6  }
```

One aspect of imports that we've not yet looked at is Go's builtin support for code hosted on a variety of popular social code-sharing sites such as GitHub and Google Code. Don't worry, we'll get to this in later chapters.

## Constants

A significant proportion of Go codebases feature identifiers whose values will not change during the runtime execution of a program and our **hello world** example is no different, so we're going to factor these out.

**Example 1.4**

```
1  package main
2  import . "fmt"
3
4  const Hello = "hello"
5  const world = "world"
6
7  func main() {
8    Println(Hello, world)
9  }
```

Here we've introduced two constants: **Hello** and **world**. Each identifier is assigned its value during compilation, and that value cannot be changed at runtime. As the identifier **Hello** starts with a capital letter the associated constant is visible to other packages - though this isn't relevant in the context of a **main** package - whilst the identifier **world** starts with a lowercase letter and is only accessible within the **main** package.

We don't need to specify the type of these constants as the Go compiler identifies them both as strings.

Another neat trick in Go's armoury is multiple assignment so let's see how this looks.

**Example 1.5**

```
1  package main
2  import . "fmt"
3
4  const Hello, world = "hello", "world"
5
6  func main() {
7    Println(Hello, world)
8  }
```

This is compact, but I personally find it too cluttered and prefer the more general form.

**Example 1.6**

```
1  package main
2  import . "fmt"
3
4  const (
5    Hello = "hello"
6    world =  "world"
7  )
8
9  func main() {
10   Println(Hello, world)
11 }
```

Because the **Println()** function is **variadic** (i.e. can take a varible number of parameters) we can pass it both constants and it will print them on the same line, separate by whitespace. **fmt** also provides the **Printf()** function which gives precise control over how its parameters are displayed using a format specifier which will be familiar to seasoned C/C++ programmers.

```
10 Printf("%v %v\n", Hello, world)
```

**fmt** defines a number of % replacement terms which can be used to determine how a particular parameter will be displayed. Of these **%v** is the most generally used as it allows the formatting to be specified by the type of the parameter. We'll discuss this in depth when we look at user-defined types, but in this case it will simply replace a **%v** with the corresponding string.

When parsing strings the Go compiler recognises a number of **escape sequences** which are available to mark tabs, new lines and specific unicode characters. In this case we use **\n** to mark a new line.

**Example 1.7**

```
1  package main
2  import . "fmt"
3
4  const (
5    Hello = "hello"
6    world =  "world"
7  )
8
9  func main() {
10   Printf("%v %v\n", Hello, world)
11 }
```

## Variables

Constants are useful for referring to values which shouldn't change at runtime, however most of the time when we're referencing values in an imperative language like Go we need the freedom to change these values. We associate values which will change with variables. What follows is a simple variation of our **Hello World** program which allows the value of **world** to be changed at runtime by creating a new value and assigning it to the **world** variable.

**Example 1.8**

```
1  package main
2  import . "fmt"
3
4  const Hello = "hello"
5  var world = "world"
6
7  func main() {
8    world += "!"
9    Println(Hello, world)
10 }
```

There are two important changes here. Firstly we've introduced syntax for declaring a variable and assigning a value to it. Once more Go's ability to infer type allows us assign a **string** value to the variable **world** without explicitly specifying the type.

```
5  var world = "world"
```

However if we wish to be more explicit we can be.

```
5  var world string = "world"
```

Having defined **world** as a variable in the global scope we can modify its value in **main()**, and in this case we choose to append an exclamation mark. Strings in Go are immutable values so following the assignment **world** will reference a new value.

```
8  world += "!"
```

To add some extra interest I've chosen to use an **augmented assignment** operator. These are a syntactic convenience popular in many languages which allow the value contained in a variable to be modified and the resulting value then assigned to the same variable.

I don't intend to expend much effort discussing scope in Go. The point of this book is to experiment and learn by playing with code, referring to the comprehensive language specification[2] available from Google when you need to know the technicalities of a given point. However to illustrate the difference between **global** and **local** scope we'll modify this program further.

**Example 1.9**

```
1  package main
2  import . "fmt"
3
4  const Hello = "hello"
5  var world = "world"
6
7  func main() {
8    world := world + "!"
9    Println(Hello, world)
10 }
```

Here we've introduced a new *local* variable **world** within **main()** which takes its value from an operation concatenating the value of the *global* **world** variable with an exclamation mark. Within **main()** any subsequent reference to **world** will always access the *local* version of the variable without affecting the *global* **world** variable. The is known as **shadowing**.

The := operator marks an assignment declaration in which the type of the expression is inferred from the type of the value being assigned. If we chose to declare the local variable separately from the assignment we'd have to give it a different name to avoid a compilation error.

**Example 1.10**

```
1  package main
2  import . "fmt"
3
4  const Hello = "hello"
5  var world = "world"
6
7  func main() {
8    var w string
9    w = world + "!"
10   Println(Hello, w)
11 }
```

Another thing to note in this example is that when *w* is declared it's also initialised to the zero value, which in the case of *string* happens to be *""*. This is a *string* containing no characters.

[2] http://http://golang.org/ref/spec

In fact all variables in Go are initialised to the zero value for their type when they're declared and this eliminates an entire category of initialisation bugs which could otherwise be difficult to identify.

## Functions

Having looked at how to reference values in Go and how to use the **Println()** function to display them, it's only natural to wonder how we can implement our own functions. Obviously we've already implemented **main()** which hints at what's involved, but **main()** is something of a special case as it exist to allow a Go program to execute and it neither requires any parameters nor produces any values to be used elsewhere in the program.

**Example 1.11**

```
1  package main
2  import . "fmt"
3
4  const Hello = "hello"
5
6  func main() {
7    Println(Hello, world())
8  }
9
10 func world() string {
11   return "world"
12 }
```

In this example we've introduced **world()**, a function which to the outside world has the same operational purpose as the variable of the same name that we used in the previous section.

The empty brackets **()** indicate that there are no parameters passed into the function when it's called, whilst **string** tells us that a single value is returned and it's of type **string**. Anywhere that a valid Go program would expect a **string** value we can instead place a call to **world()** and the value returned will satisfy the compiler. The use of **return** is required by the language specification whenever a function specifies return values, and in this case it tells the compiler that the value of **world()** is the string "**world**".

Go is unusual in that its syntax allows a function to return more than one value and as such each function takes two sets of **()**, the first for parameters and the second for results. We could therefore write our function in long form as

```
10 func world() (string) {
11   return "world"
12 }
```

In this next example we use a somewhat richer function signature, passing the parameter **name** which is a string value into the function **message()**, and assigning the function's return value to **message** which is a variable declared and available throughout the function.

**Example 1.12**

```
package main
import "fmt"

func main() {
  fmt.Println(message("world"))
}

func message(name string) (message string) {
  message = fmt.Sprintf("hello %v", name)
  return message
}
```

As with **world()** the **message()** function can be used anywhere that the Go compiler expects to find a string value. However where **world()** simply returned a predetermined value, **message()** performs a calculation using the **Sprintf()** function and returns its result.

**Sprintf()** is similar to **Printf()** which we met when discussing constants, only rather than create a string according to a format and displaying it in the terminal it instead returns this string as a value which we can assign to a variable or use as a parameter in another function call such as **Println()**.

Because we've explicitly named the return value we don't need to reference it in the return statement as each of the named return values is implied.

**Example 1.13**

```
package main
import . "fmt"

func main() {
  Println(message("world"))
}

func message(name string) (message string) {
  message = Sprintf("hello %v", name)
  return
}
```

If we compare the **main()** and **message()** functions, we notice that **main()** doesn't have a **return** statement. Likewise if we define our own functions without return values we can omit the **return** statement though later we'll meet examples where we'd still use a **return** statement to prematurely exit a function.

**Example 1.14**

```
1  package main
2  import . "fmt"
3
4  func main() {
5    greet("world")
6  }
7
8  func greet(name string) {
9    Println("hello", name)
10 }
```

In the next example we'll see what a function which uses multiple return values looks like.

**Example 1.15**

```
1  package main
2  import . "fmt"
3
4  func main() {
5    Println(message())
6  }
7
8  func message() (string, string) {
9    return "hello", "world"
10 }
```

Because **message()** returns two values we can use it in any context where at least two parameters can be consumed. **Println()** happens to be a **variadic** function, which we'll explain in a moment, and takes zero or more parameters so it happily consumes both of the values **message()** returns.

For our final example we're going to implement our own **variadic** function.

**Example 1.16**

```
1  package main
2  import . "fmt"
3
4  func main() {
5    print("Hello", "world")
6  }
7
8  func print(v ...interface{}) {
9    Println(v...)
10 }
```

We have three interesting things going on here which need explaining. Firstly I've introduced a new type, **interface{}**, which acts as a proxy for any other type in a Go program. We'll discuss the details of this shortly but for now it's enough to know that anywhere an **interface{}** is accepted we can provide a string.

In the function signature we use **v ...interface{}** to declare a parameter **v** which takes any number of values. These are received by **print()** as a sequence of values and the subsequent call to **Println(v...)** uses this same sequence as this is the sequence expected by **Println()**.

So why did we use **...interface{}** in defining our parameters instead of the more obvious **...string**? The **Println()** function is itself defined as **Println(...interface{})** so to provide a sequence of values en masse we likewise need to use **...interface{}** in the type signature of our function. Otherwise we'd have to create a **[]interface{}** (a **slice** of **interface{}** values, a concept we'll cover in detail in a later chanpter) and copy each individual element into it before passing it into **Println()**.

## Encapsulation

In this chapter we'll for the most part be using **Go**'s primitive types and types defined in various standard packages without any comment on their structure, however a key aspect of modern programming languages is the encapsulation of related data into structured types and **Go** supports this via the **struct** type. A **struct** describes an area of allocated memory which is subdivided into slots for holding named values, where each named value has its own type. A typical example of a **struct** in action would be:

**Example 1.17**

```
1  package main
2
3  import "fmt"
4
5  type Message struct {
6    X string
7    y *string
8  }
9
10 func (v Message) Print() {
11   if v.y != nil {
12     fmt.Println(v.X, *v.y)
13   } else {
14     fmt.Println(v.X)
15   }
16 }
17
18 func (v *Message) Store(x, y string) {
19   v.X = x
20   v.y = &y
21 }
22
23 func main() {
24   m := &Message{}
25   m.Print()
26   m.Store("Hello", "world")
27   m.Print()
28 }
```

```
$ go run 17.go

Hello world
```

Here we've defined a struct **Message** which contains two values: X and y. **Go** uses a very simple rule for deciding if an identifier is visible outside of the package in which it's defined which applies to both package-level constants and variables, and **type** names, methods and fields. If the identifier starts with a capital letter it's visible outside the package otherwise it's private to the package.

The **Go** language spec guarantees that all variables will be initialised to the zero value for their type. For a **struct** type this means that every field will be initialised to an appropriate zero value. Therefore when we declare a value of type **Message** the **Go** runtime will initialise all of its elements to their zero value (in this case a zero-length string and a nil pointer respectively), and likewise if we create a **Message** value using a literal:

```
24    m := &Message{}
```

Having declared a **struct** type we can declare any number of **method** functions which will operate on this type. In this case we've introduced **Print()** which is called on a **Message** value to display it in the terminal, and **Store()** which is called on a pointer to a **Message** value to change its contents. The reason **Store()** applies to a pointer is that we want to be able to change the contents of the **Message** and have these changes persist. If we define the method to work directly on the value these changes won't be propagated outside the method's scope. To test this for yourself, make the following change to the program:

**Example 1.18**

```
18  func (v Message) Store(x, y string) {
```

If you're familiar with functional programming then the ability to use values immutably this way will doubtless spark all kinds of interesting ideas.

There's another **struct** trick I want to show off before we move on and that's **type embedding** using an anonymous field. **Go**'s design has upset quite a few people with an inheritance-based view of object orientation because it lacks inheritance, however thanks to **type embedding** we're able to compose types which act as proxies to the __method__s provided by anonymous fields. As with most things, an example will make this much clearer:

**Example 1.19 Type Embedding**

```
1   package main
2
3   import "fmt"
4
5   type HelloWorld struct {}
6
7   func (h HelloWorld) String() string {
8     return "Hello world"
9   }
10
11  type Message struct {
```

```
12    HelloWorld
13  }
14
15  func main() {
16    m := &Message{}
17    fmt.Println(m.HelloWorld.String())
18    fmt.Println(m.String())
19    fmt.Println(m)
20  }
```

```
$ go run 19.go
Hello world
Hello world
Hello world
```

Here we're declaring a type **HelloWorld** which in this case is just an empty struct, but which in reality could be any declared type. **HelloWorld** defines a **String()** method which can be called on any **HelloWorld** value. We then declare a type **Message** which *embeds* the **HelloWorld** type by defining an anonymous field of the anmer type. Wherever we encounter a value of type **Message** and wish to call **String()** on its embedded **HelloWorld** value we can do so by calling **String()** directly on the value (), **calling __String()** on the **Message** value, or in this case by allowing **fmt.Println()** to match it with the **fmt.Stringer** interface.

Any declared type can be embedded, so in our next example we're going to base **HelloWorld** on the primitive **bool** boolean type to prove the point:

**Example 1.20 Type Embedding**

```
1   package main
2
3   import "fmt"
4
5   type HelloWorld bool
6
7   func (h HelloWorld) String() (r string) {
8     if h {
9       r = "Hello world"
10    }
11    return
12  }
13
14  type Message struct {
15    HelloWorld
16  }
17
18  func main() {
19    m := &Message{ HelloWorld: true }
20    fmt.Println(m)
21    m.HelloWorld = false
```

```
22    fmt.Println(m)
23    m.HelloWorld = true
24    fmt.Println(m)
25  }
```

In our final example we've declared the **Hello** type and embedded it in **Message**, then we've implemented a new **String()** method which allows a **Message** value more control over how it's printed:

**Example 1.21 Type Embedding**

```
1   package main
2
3   import "fmt"
4
5   type Hello struct {}
6
7   func (h Hello) String() string {
8     return "Hello"
9   }
10
11  type Message struct {
12    *Hello
13    World string
14  }
15
16  func (v Message) String() (r string) {
17    if v.Hello == nil {
18      r = v.World
19    } else {
20      r = fmt.Sprintf("%v %v", v.Hello, v.World)
21    }
22    return
23  }
24
25  func main() {
26    m := &Message{}
27    fmt.Println(m)
28    m.Hello = new(Hello)
29    fmt.Println(m)
30    m.World = "world"
31    fmt.Println(m)
32  }
```

```
$ go run 21.go

Hello
Hello world
```

In all these examples we've made liberal use of the * and **&** operators. An explanation is in order.

**Go** is a systems programming language, and this means that a **Go** program has direct access to the memory of the platform it's running on. This requires that **Go** has a means of refering to specific addresses in memory and of accessing their contents indirectly. The **&** operator is prepended to the name of a variable or to a value literal when we wish to discover its address in memory, which we refer to as a **pointer**. To do anything with the **pointer** returned by the **&** operator we need to be able to declare a **pointer variable** which we do by prepending a **type name** with the * operator. An example will probably make this description somewhat clearer:

**Pointers and Addresses**

```
1  package main
2  import . "fmt"
3
4  type Text string
5
6  func main() {
7    var name Text = "Ellie"
8    var pointer_to_name *Text
9
10   pointer_to_name = &name
11   Printf("name = %v stored at %v\n", name, pointer_to_name)
12   Printf("pointer_to_name references %v\n", *pointer_to_name)
13 }
```

```
$ go run aside_01.go
name = Ellie stored at 0x208178170
pointer_to_name references Ellie
```

**Go** allows user-defined types to declare methods on either a **value type** or a **pointer** to a **value type**. When methods operate on a **value type** the **value** manipulated remains immutable to the rest of the program (essentially the method operates on a copy of the value) whilst with a **pointer** to a **value type** any changes to the **value** are apparent throughout the program. This has far-reaching implications which we'll explore in later chapters.

## Generalisation

Encapsulation is of huge benefit when writing complex programs and it also enables one of the more powerful features of **Go**'s type system, the **interface**. An **interface** is similar to a **struct** in that it combines one or more elements but rather than defining a type in terms of the data items it contains, an **interface** defines it in terms of a set of **method** signatures which it must implement.

As none of the primitive types (**int**, **string**, etc.) have methods they match the empty **interface** (**interface{}**) as do all other types, a property used frequently in **Go** programs to create generic containers.

Once declared an **interface** can be used just like any other declared type, allowing functions and variables to operate with unknown types based solely on their required behaviour. **Go**'s type inference system will then recognise compliant values as instances of the interface, allowing us to write generalised code with little fuss.

In the next example we're going to introduce a simple **interface** (by far the most common kind) which matches any type with a **func String() string** method signature.

**Example 1.22**

```
1  package main
2
3  import "fmt"
4
5  type Stringer interface {
6    String() string
7  }
8
9  type Hello struct {}
10
11 func (h Hello) String() string {
12   return "Hello"
13 }
14
15 type World struct {}
16
17 func (w *World) String() string {
18   return "world"
19 }
20
21 type Message struct {
22   X Stringer
23   Y Stringer
24 }
25
26 func (v Message) String() (r string) {
27   switch {
28   case v.X == nil && v.Y == nil:
29   case v.X == nil:
30     r = v.Y.String()
31   case v.Y == nil:
32     r = v.X.String()
33   default:
34     r = fmt.Sprintf("%v %v", v.X, v.Y)
35   }
36   return
```

```
37  }
38
39  func main() {
40    m := &Message{}
41    fmt.Println(m)
42    m.X = new(Hello)
43    fmt.Println(m)
44    m.Y = new(World)
45    fmt.Println(m)
46    m.Y = m.X
47    fmt.Println(m)
48    m = &Message{ X: new(World), Y: new(Hello) }
49    fmt.Println(m)
50    m.X, m.Y = m.Y, m.X
51    fmt.Println(m)
52  }
```

```
$ go run 22.go

Hello
Hello world
Hello Hello
world Hello
Hello world
```

This **interface** is copied directly from **fmt.Stringer**, so we can simplify our code a little by using that interface instead:

**Example 1.23**

```
17  type Message struct {
18    X fmt.Stringer
19    Y fmt.Stringer
20  }
```

As **Go** is strongly typed **interface** values contain both a pointer to the value contained in the **interface**, and the **concrete** type of the stored value. This allows us to perform **type assertions** to confirm that the value inside an **interface** matches a particular concrete type:

**Example 1.24**

```
package main

import "fmt"

type Hello struct {}

func (h Hello) String() string {
  return "Hello"
}

type World struct {}

func (w *World) String() string {
  return "world"
}

type Message struct {
  X fmt.Stringer
  Y fmt.Stringer
}

func (v Message) IsGreeting() (ok bool) {
  if _, ok = v.X.(*Hello); !ok {
    _, ok = v.Y.(*Hello)
  }
  return
}

func main() {
  m := &Message{}
  fmt.Println(m.IsGreeting())
  m.X = new(Hello)
  fmt.Println(m.IsGreeting())
  m.Y = new(World)
  fmt.Println(m.IsGreeting())
  m.Y = m.X
  fmt.Println(m.IsGreeting())
  m = &Message{ X: new(World), Y: new(Hello) }
  fmt.Println(m.IsGreeting())
  m.X, m.Y = m.Y, m.X
  fmt.Println(m.IsGreeting())
}
```

```
go run 24.go
false
true
true
true
true
true
```

Here we've replaced **Message**'s **String()** method with **IsGreeting()**, a predicate which uses a pair of **type assertion__s to tell us whether or not one of __Message**'s data fields contains a value of concrete type **Hello**.

So far in these examples we've been using pointers to **Hello** and **World** so the **interface** variables are storing pointers to pointers to these values (i.e. ****Hello** and ****World**) rather than pointers to the values themselves (i.e. ***Hello__ and __World***). In the case of **World** we have to do this to comply with the **fmt.Stringer** interface because **String()** is defined for ***World** and if we modify main to assign a **World** value to either field we'll get a compile-time error:

**Example 1.25**

```
29 func main() {
30   m := &Message{}
31   fmt.Println(m.IsGreeting())
32   m.X = Hello{}
33   fmt.Println(m.IsGreeting())
34   m.X = new(Hello)
35   fmt.Println(m.IsGreeting())
36   m.X = World{}
37 }
```

```
$ go run 25.go
# command-line-arguments
./25.go:36: cannot use World literal (type World) as type fmt.Stringer in assignment:
        World does not implement fmt.Stringer (String method has pointer receiver)
```

The final thing to mention about **interface__s is that they support embedding of other __interface__s. This allows us to compose a new, more restrictive __interface** based on one or more existing **interface__s. Rather than demonstrate this with an example we're going to look at code lifted directly from the standard __io** package which does this:

**Extract from io**

```
67 type Reader interface {
68   Read(p []byte) (n int, err error)
69 }
```

```
78  type Writer interface {
79    Write(p []byte) (n int, err error)
80  }
```

```
106 type ReadWriter interface {
107   Reader
108   Writer
109 }
```

Here **io** is declaring three **interfaces**, the **Reader** and **Writer** which are independent of each other, and the **ReadWriter** which combines both. Any time we declare a variable, field or function parameter in terms of a **ReaderWriter** we know we can use both the **Read()** and **Write()** methods to manipulate it.

## Startup

One of the less-discussed aspects of computer programs is the need to initialise many of them to a pre-determined state before they begin executing. Whilst this is probably the worst place to start discussing what to many people may appear to be advanced topics, one of my goals in this chapter is to cover all of the structural elements that we'll meet when we examine more complex programs.

Every Go package may contain one or more **init()** functions specifying actions that should be taken during program initialisation. This is the one case I'm aware of where multiple declarations of the same identifier can occur without either resulting in a compilation error or the shadowing of a variable. In the following example we use the **init()** function to assign a value to our **world** variable

**Example 1.26**

```
1  package main
2  import . "fmt"
3
4  const Hello = "hello"
5  var world  string
6
7  func init() {
8    world = "world"
9  }
10
11 func main() {
12   Println(Hello, world)
13 }
```

However the **init()** function can contain any valid Go code, allowing us to place the whole of our program in **init()** and leaving **main()** as a stub to convince the compiler that this is indeed a valid Go program.

**Example 1.27**

```
package main
import . "fmt"

const Hello = "hello"
var world  string

func init() {
  world = "world"
  Println(Hello, world)
}

func main() {}
```

When there are multiple **init()** functions the order in which they're executed is indeterminate so in general it's best not to do this unless you can be certain the **init()** functions don't interact in any way. The following happens to work as expected on my development computer but an implementation of Go could just as easily arrange it to run in reverse order or even leave deciding the order of execution until runtime.

**Example 1.28**

```
package main
import . "fmt"

const Hello = "hello"
var world  string

func init() {
  Print(Hello, " ")
  world = "world"
}

func init() {
  Printf("%v\n", world)
}

func main() {}
```

## HTTP

So far our treatment of **Hello World** has followed the traditional route of printing a preset message to the console. Anyone would think we were living in the fuddy-duddy mainframe era of the 1970s instead of the shiny 21st Century, when web and mobile applications rule the world.

Turning **Hello World** into a web application is surprisingly simple, as the following example demonstrates.

**Example 1.29**

```
package main
import (
  . "fmt"
  "net/http"
)

const MESSAGE = "hello world"
const ADDRESS = ":1024"

func main() {
  http.HandleFunc("/hello", Hello)
  if e := http.ListenAndServe(ADDRESS, nil); e != nil {
    Println(e)
  }
}

func Hello(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/plain")
  Fprintf(w, MESSAGE)
}
```

```
$ go run 29.go
```

Our web server is now listening on localhost port 1024 (usually the first non-privileged port on most Unix-like operating systems) and if we visit the url **http://localhost:1024/hello** with a web browser our server will return **Hello World** in the response body.

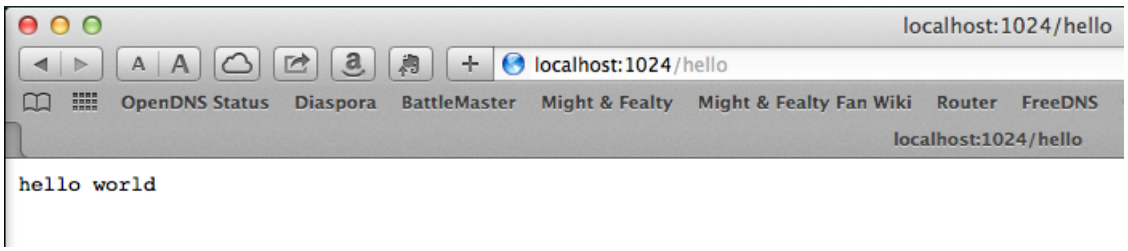


**Image 1.29 http://localhost:1024/hello**

The first thing to note is that the **net/http** package provides a fully-functional web server which requires very little configuration. All we have to do to get our content to the browser is define a **handler**, which in this case is a function to call whenever an **http.Request** is received, and then launch a server to listen on the desired address with **http.ListenAndServe()**. **http.ListenAndServe** returns an error if it's unable to launch the server for some reason, which in this case we print to the console.

We're going to import the **net/http** package into the current namespace and assume our code won't encounter any runtime errors to make the simplicity even more apparent. If you run into any problems whilst trying the examples which follow, reinserting the if statement will allow you to figure out what's going on.

**Example 1.30**

```
package main
import (
  . "fmt"
  . "net/http"
)

const MESSAGE = "hello world"
const ADDRESS = ":1024"

func main() {
  HandleFunc("/hello", Hello)
  ListenAndServe(ADDRESS, nil)
}

func Hello(w ResponseWriter, r *Request) {
  w.Header().Set("Content-Type", "text/plain")
  Fprintf(w, MESSAGE)
}
```

**HandleFunc()** registers a URL in the web server as the trigger for a function, so when a web request targets the URL the associated function will be executed to generate the result. The specified handler function is passed both a **ResponseWriter** to send output to the web client and the **Request** which is being replied to. The **ResponseWriter** is a file handle so we can use the **fmt.Fprint()** family of file-writing functions to create the response body.

Finally we launch the server using **ListenAndServe()** which will block for as long as the server is active, returning an error if there is one to report.

In this example I've declared a function **Hello** and by referring to this in the call to **HandleFunc()** this becomes the function which is registered. However Go also allows us to define functions anonymously where we wish to use a function value, as demonstrated in the following variation on our theme.

**Example 1.31**

```
package main
import (
  . "fmt"
  . "net/http"
)

const MESSAGE = "hello world"
const ADDRESS = ":1024"

func main() {
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, MESSAGE)
  })
```

```
15    ListenAndServe(ADDRESS, nil)
16  }
```

Functions are first-class values in Go and here **HandleFunc()** is passed an anonymous function value which is created at runtime. This value is a closure so it can also access variables in the lexical scope in which it's defined. We'll treat closures in greater depth later in the book, but for now here's an example which demonstrates their basic premise by defining a variable **messages** in **main()** and then accessing it from within the anonymous function.

**Example 1.32**

```
1   package main
2   import (
3     . "fmt"
4     . "net/http"
5   )
6
7   const ADDRESS = ":1024"
8
9   func main() {
10    message := "hello world"
11    HandleFunc("/hello", func(w ResponseWriter, r *Request) {
12      w.Header().Set("Content-Type", "text/plain")
13      Fprintf(w, message)
14    })
15    ListenAndServe(ADDRESS, nil)
16  }
```

This is only a very brief taster of what's possible using **net/http** so we'll conclude by serving our **hello world** web application over an SSL connection.

**Example 1.33**

```
1   package main
2   import (
3     . "fmt"
4     . "net/http"
5   )
6
7   const SECURE_ADDRESS = ":1025"
8
9   func main() {
10    message := "hello world"
11    HandleFunc("/hello", func(w ResponseWriter, r *Request) {
12      w.Header().Set("Content-Type", "text/plain")
13      Fprintf(w, message)
14    })
15    ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
16  }
```

Before we run this program we first need to generate a certificate and a public key, which we can do using **crypto/tls/generate_cert.go** in the standard package library.

```
$ go run $GOROOT/src/pkg/crypto/tls/generate_cert.go -ca=true -host="localhost"
2014/05/16 20:41:53 written cert.pem
2014/05/16 20:41:53 written key.pem
$ go run 33.go
```

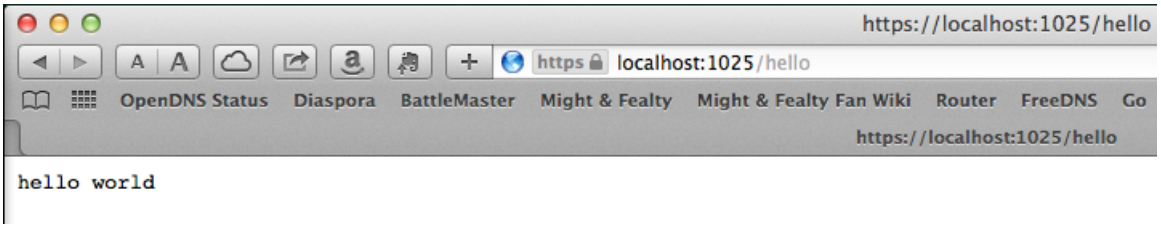


**Image 1.33 https://localhost:1025/hello**

This is a self-signed certificate, and not all modern web browsers like these. Firefox will refuse to connect on the grounds the certificate is inadequate and not being a Firefox user I've not devoted much effort to solving this. Meanwhile both Chrome and Safari will prompt the user to confirm the certificate is trusted. I have no idea how Internet Explorer behaves.

If you're anything like me (and you have my sympathy if you are) then the next thought to idle through your mind will be a fairly obvious question: given that we can serve our content over both HTTP and HTTPS connections, how do we do both from the same program?

To answer this we have to know a little - but not a lot - about how to model concurrency in a Go program. The **go** keyword marks a **goroutine** which is a lightweight thread scheduled by the Go runtime. How this is implemented under the hood doesn't matter, all we need to know is that when a **goroutine** is launched it takes a function call and creates a separate thread of execution for it. Here we're going to launch a **goroutine** to run the **HTTP** server then run the **HTTPS** server in the main flow of execution.

**Example 1.34**

```
1  package main
2  import (
3    . "fmt"
4    . "net/http"
5  )
6
7  const ADDRESS = ":1024"
8  const SECURE_ADDRESS = ":1025"
9
10 func main() {
11   message := "hello world"
12   HandleFunc("/hello", func(w ResponseWriter, r *Request) {
13     w.Header().Set("Content-Type", "text/plain")
14     Fprintf(w, message)
15   })
16
```

```
17    go func() {
18      ListenAndServe(ADDRESS, nil)
19    }()
20
21    ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
22  }
```

When I first wrote this code it actually used two **goroutine__s, one for each server. Unfortunately no matter how busy any particular __goroutine** is, when the **main()** function returns our program will exit and our web servers will terminate. So I tried the primitive approach we all know and love from C:

```
10  func main() {
11    message := "hello world"
12    HandleFunc("/hello", func(w ResponseWriter, r *Request) {
13      w.Header().Set("Content-Type", "text/plain")
14      Fprintf(w, message)
15    })
16
17    go func() {
18      ListenAndServe(ADDRESS, nil)
19    }()
20
21    go func() {
22      ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
23    }()
24
25    for {}
26  }
```

Here we're using an infinite **for** loop to prevent program termination: it's inelegant, but this is a small program and dirty hacks have their appeal. Whilst semantically correct this unfortunately doesn't work either because of the way __goroutine__s are scheduled: the infinite loop starves the thread scheduler and prevents the other __goroutine__s from running.

```
$ go version
go version go1.3 darwin/amd64
```

In any event an **infinite loop** is a nasty, unnecessary hack as Go allows concurrent elements of a program to communicate with each other via **channels**, allowing us to rewrite our code as:

**Example 1.35**

```
package main
import (
  . "fmt"
  . "net/http"
)

const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"

func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })

  done := make(chan bool)
  go func() {
    ListenAndServe(ADDRESS, nil)
    done <- true
  }()

  ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
  <- done
}
```

The next pair of examples we're going to use two separate **goroutine__s to run our __HTTP** and **HTTPS** servers, yet again coordinating program termination with a shared channel. In the first example we'll launch both of the **goroutine__s from the __main()** function, which is a fairly typical code pattern:

**Example 1.36**

```
package main
import (
  . "fmt"
  . "net/http"
)

const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"

func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })
```

```
16
17    done := make(chan bool)
18    go func() {
19      ListenAndServe(ADDRESS, nil)
20      done <- true
21    }()
22
23    go func () {
24      ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
25      done <- true
26    }()
27    <- done
28    <- done
29  }
```

For our second deviation we're going to launch a **goroutine** from **main()** which will run our **HTTPS** server and this will launch the second **goroutine** which manages our **HTTP** server:

**Example 1.37**

```
1   package main
2   import (
3     . "fmt"
4     . "net/http"
5   )
6
7   const ADDRESS = ":1024"
8   const SECURE_ADDRESS = ":1025"
9
10  func main() {
11    message := "hello world"
12    HandleFunc("/hello", func(w ResponseWriter, r *Request) {
13      w.Header().Set("Content-Type", "text/plain")
14      Fprintf(w, message)
15    })
16
17    done := make(chan bool)
18    go func () {
19      go func() {
20        ListenAndServe(ADDRESS, nil)
21        done <- true
22      }()
23
24      ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
25      done <- true
26    }()
27    <- done
28    <- done
29  }
```

There's a certain amount of fragile repetition in this code as we have to remember to explicitly create a channel, and then to send and receive on it multiple times to coordinate execution. As **Go** provides first-order functions (i.e. allows us to refer to functions the same way we refer to data) we can refactor the server launch code as follows:

**Example 1.38**

```
package main
import (
  . "fmt"
  . "net/http"
)

const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"

func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })

  Spawn(
    func() { ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil) },
        func() { ListenAndServe(ADDRESS, nil) },
  )
}

func Spawn(f ...func()) {
  done := make(chan bool)

  for _, s := range f {
    go func() {
      s()
      done <- true
    }()
  }

  for l := len(f); l > 0; l-- {
    <- done
  }
}
```

However this doesn't work as expected, so let's see if we can get any further insight:

```
$ go vet 38.go
38.go:28: range variable s enclosed by function
```

Running **go** with the **vet** command runs a set of heuristics against our source code to check for common errors which wouldn't be caught during compilation. In this case we're being warned about this code:

```
26  for _, s := range f {
27    go func() {
28      s()
29      done <- true
30    }()
31  }
```

Here we're using a closure so it refers to the variable **s** in the **for**..**range** statement, and as the value of **s** changes on each successive iteration, so this is reflected in the call **s()**.

To demonstrate this we'll try a variant where we introduce a delay on each loop iteration much greater than the time taken to launch the **goroutine**.

**Example 1.39**

```
1   package main
2   import (
3     . "fmt"
4     . "net/http"
5     "time"
6   )
7
8   const ADDRESS = ":1024"
9   const SECURE_ADDRESS = ":1025"
10
11  func main() {
12    message := "hello world"
13    HandleFunc("/hello", func(w ResponseWriter, r *Request) {
14      w.Header().Set("Content-Type", "text/plain")
15      Fprintf(w, message)
16    })
17
18    Spawn(
19      func() { ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil) },
20          func() { ListenAndServe(ADDRESS, nil) },
21    )
22  }
23
24  func Spawn(f ...func()) {
25    done := make(chan bool)
26
27    for _, s := range f {
28      go func() {
29        s()
30        done <- true
31      }()
32      time.Sleep(time.Second)
```

```
33    }
34
35    for l := len(f); l > 0; l-- {
36      <- done
37    }
38  }
```

When we run this we get the behaviour we expect with both **HTTP** and **HTTPS** servers running on their respective ports and responding to browser traffic. However this is hardly an elegant or practical solution.

**Example 1.40**

```
26    for _, s := range f {
27      go func(server func()) {
28        server()
29        done <- true
30      }(s)
31    }
```

By accepting the parameter **server** to the **goroutine**'s closure we can pass in the value of **s** and capture it so that on successive iterations of the **range** our __goroutine__s use the correct value.

**Spawn()** is an example of how powerful **Go**'s support for first-class functions can be, allowing us to run any arbitrary piece of code and wait for it to signal completion. It's also a **variadic** function, taking as many or as few functions as desired and setting each of them up correctly.

If we now reach for the standard library we discover that another alternative is to use a **sync.WaitGroup** to keep track of how many active **goroutines** we have in our program and only terminate the program when they've all completed their work. Yet again this allows us to run both servers in separate **goroutines** and manage termination correctly.

**Example 1.41**

```
1   package main
2   import (
3     . "fmt"
4     . "net/http"
5     "sync"
6   )
7
8   const ADDRESS = ":1024"
9   const SECURE_ADDRESS = ":1025"
10
11  func main() {
12    message := "hello world"
13    HandleFunc("/hello", func(w ResponseWriter, r *Request) {
14      w.Header().Set("Content-Type", "text/plain")
15      Fprintf(w, message)
16    })
17
```

```
18    var servers sync.WaitGroup
19    servers.Add(1)
20    go func() {
21      defer servers.Done()
22      ListenAndServe(ADDRESS, nil)
23    }()
24
25    servers.Add(1)
26    go func() {
27      defer servers.Done()
28      ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
29    }()
30    servers.Wait()
31  }
```

As there's a certain amount of redundancy in this, let's refactor a little by packaging server initiation into a new **Launch()** function. **Launch()** takes a parameter-less function and wraps this in a **closure** which will be launched as a **goroutine** in a separate thread of execution. Our **sync.WaitGroup** variable **servers** has been turned into a global variable to simplify the function signature of **Launch()**. When we call **Launch()** we're freed from the need to manually increment **servers** prior to **goroutine** startup, and we use a **defer** statement to automatically call **servers.Done()** when the **goroutine** terminates.

**Example 1.42**

```
1   package main
2   import (
3     . "fmt"
4     . "net/http"
5     "sync"
6   )
7
8   const ADDRESS = ":1024"
9   const SECURE_ADDRESS = ":1025"
10
11  var servers sync.WaitGroup
12
13  func main() {
14    message := "hello world"
15    HandleFunc("/hello", func(w ResponseWriter, r *Request) {
16      w.Header().Set("Content-Type", "text/plain")
17      Fprintf(w, message)
18    })
19
20    Launch(func() {
21      ListenAndServe(ADDRESS, nil)
22    })
23
24    Launch(func() {
25      ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
```

```
  })
  servers.Wait()
}

func Launch(f func()) {
  servers.Add(1)
  go func() {
    defer servers.Done()
    f()
  }()
}
```

## The Environment

The main shells used with modern operating systems (Linux, OSX, FreeBSD, Windows, etc.) provide a persistent environment which can be queried by running programs, allowing a user to store configuration values in named variables. Go supports reading and writing these variables using the **os** package functions **Getenv()** and **Setenv()**.

In our next example we're going to query the environment for the variable **SERVE_HTTP** which we'll assume contains the default address on which to serve web content.

**Example 1.43**

```
package main
import (
  . "fmt"
  . "net/http"
  "os"
  "sync"
)

const SECURE_ADDRESS = ":1025"

var address string
var servers sync.WaitGroup

func init() {
  if address = os.Getenv("SERVE_HTTP"); address == "" {
    address = ":1024"
  }
}

func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })
```

```
26
27    Launch(func() {
28      ListenAndServe(address, nil)
29    })
30
31    Launch(func() {
32      ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
33    })
34    servers.Wait()
35  }
36
37  func Launch(f func()) {
38    servers.Add(1)
39    go func() {
40      defer servers.Done()
41      f()
42    }()
43  }
```

Here we've defined a global variable **address** which we set in **init()** to either the value provided in **SERVE_HTTP** or a default value "**:1024**".

```
$ go run 43.go
```

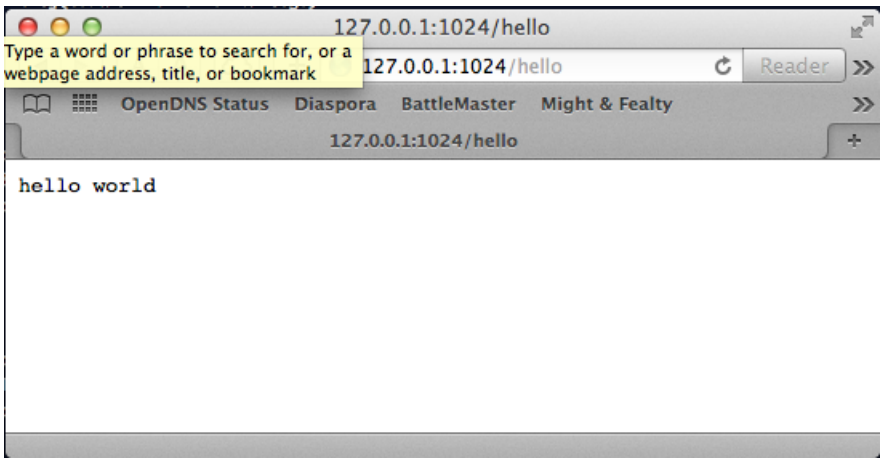


**Image 1.43a http://localhost:1024/hello**

```
$ SERVE_HTTP=":3030" go run 43.go
```

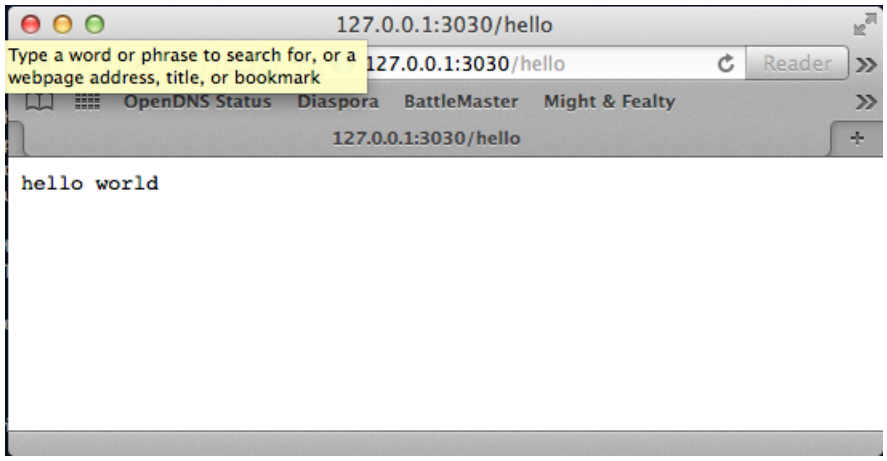


**Image 1.43b http://localhost:3030/hello**

If we now extend this further to make the program fully configurable from the environment we arrive at

**Example 1.44**

```
1  package main
2  import (
3    . "fmt"
4    . "net/http"
5    "os"
6    "sync"
7  )
8
9  var (
10   address string
11   secure_address string
12   certificate string
13   key string
14 )
15 var servers sync.WaitGroup
16
17 func init() {
18   if address = os.Getenv("SERVE_HTTP"); address == "" {
19     address = ":1024"
20   }
21
22   if secure_address = os.Getenv("SERVE_HTTPS"); secure_address == "" {
23     secure_address = ":1025"
24   }
25
26   if certificate = os.Getenv("SERVE_CERT"); certificate == "" {
27     certificate = "cert.pem"
28   }
29
30   if key = os.Getenv("SERVE_KEY"); key == "" {
31     key = "key.pem"
```

```
32    }
33  }
34
35  func main() {
36    message := "hello world"
37    HandleFunc("/hello", func(w ResponseWriter, r *Request) {
38      w.Header().Set("Content-Type", "text/plain")
39      Fprintf(w, message)
40    })
41
42    Launch(func() {
43      ListenAndServe(address, nil)
44    })
45
46    Launch(func() {
47      ListenAndServeTLS(secure_address, certificate, key, nil)
48    })
49    servers.Wait()
50  }
51
52  func Launch(f func()) {
53    servers.Add(1)
54    go func() {
55      defer servers.Done()
56      f()
57    }()
58  }
```

## Handling Signals

If you've been running our example in the terminal and wondering how to terminate it without exiting the shell then you probably come from a GUI background and haven't met control-C and its relatives (or rather you have, but most likely as cut'n'paste shortcuts).

Both Windows and Unix-style operating systems have the concept of a **signal** which can be sent from one process to another, and for historic reasons many of these can be manually entered using a control-key combination. This is a useful convenience but shutting down a production server this way can result in data loss or corruption. However that's not the case with our **Hello World** server, so we have an excellent excuse to examine how to catch a **termination signal** and do something of our own choosing.

To listen for signals in **Go** we use the **os/signal** package in the standard library, which allows us to register a **channel** (an atomic **queue** for transferring messages between **goroutines** at runtime) on which notifications are to be received using the **signal.Notify()** function. Which signals will be made available depends largely on which operating system you're working with and **Go** provides only two as standard across Windows and Unix: **os.Interrupt** and **os.Kill**. Of these **os.Interrupt** can be sent with **control-C** whilst **os.Kill** equates to **SIGKILL** on *nixen and is usually a non-maskable interrupt, meaning that it terminates execution and will never be received by **Notify()**.

In the following example we're initialising a **signal handler** at program startup. This consists of a goroutine containing an infinite loop and blocking on a channel of fixed size (in this case able to hold

only one element at a time). The **signal handler** should be trapping the **Interrupt** and **Kill** signals and in the case of an **Interrupt** (which will be generated by the control-C keypress on a Unix system). In both cases we print a message to the console before exiting, however as previously mentioned the **Kill** signal (which can be sent from another shell session using the **kill** command) will never be received by our **Go** code.

**Example 1.45**

```
package main
import (
  . "fmt"
  . "net/http"
  "os"
  "os/signal"
  . "sync"
)

const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"

var servers WaitGroup

func init() {
  go SignalHandler(make(chan os.Signal, 1))
}

func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })

  Launch(func() {
    ListenAndServe(ADDRESS, nil)
  })

  Launch(func() {
    ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
  })
  servers.Wait()
}

func Launch(f func()) {
  servers.Add(1)
  go func() {
    defer servers.Done()
    f()
  }()
}
```

```
43
44 func SignalHandler(c chan os.Signal) {
45   signal.Notify(c, os.Interrupt)
46   for s := <- c; ; s = <- c {
47     switch s {
48     case os.Interrupt:
49       Println("^C received")
50       os.Exit(0)
51     case os.Kill:
52       Println("SIGKILL received")
53       os.Exit(1)
54     }
55   }
56 }
```

When we run this in the terminal on a Mac we'll see something like:

```
$ go run 45.go
^C^C received
```

The key point of signals is that they allow our program to apply its own logic to events. In the following example we're going to override the **Interrupt** signal sent by **control-C** so that the program continues execution. We're then going to scan for other signals and use these to terminate the program. The **syscall** package defines a number of **os.Signal** values which can be detected by **Notify()** and of these I've chosen **SIGABRT**, **SIGTERM** and **SIGQUIT** as plausible termination signals. We'll treat **SIGABRT** as an error condition and the other two as clean terminations.

Something else to note is that our **signal handler** is using a standard **for** loop statement to poll for input from the signal **channel** and then compare it to the **cases** of a **switch** statement. As the **signal handler** is designed to run for as long as **Notify()** is receiving signals we can simplify this a little:

**Example 1.46**

```
45 func SignalHandler(c chan os.Signal) {
46   signal.Notify(c, os.Interrupt, syscall.SIGABRT, syscall.SIGTERM, syscall.SIGQUIT)
47   for s := <- c; ; s = <- c {
48     switch s {
49     case os.Interrupt:
50       Println("interrupt - continue running")
51     case syscall.SIGABRT:
52       Println("abnormal exit")
53       os.Exit(1)
54     case syscall.SIGTERM, syscall.SIGQUIT:
55       Println("clean shutdown")
56       os.Exit(0)
57     }
58   }
59 }
```

```
$ go build 46.go
$ ./46
^Cinterrupt - continue running
^Cinterrupt - continue running
^\clean shutdown
```

So far we've looked at how our program receive signals, however it's also possible to send signals. For now we're going to focus on sending a **SIGABRT** signal from our program to itself when there's an error launching one of the servers, in this case by setting ADDRESS and SECURE_ADDRESS to the same value.

**Example 1.47**

```
1  package main
2  import (
3    . "fmt"
4    . "net/http"
5    "os"
6    "os/signal"
7    . "sync"
8    "syscall"
9  )
10
11 const ADDRESS = ":1024"
12 const SECURE_ADDRESS = ":1024"
13
14 var servers WaitGroup
15
16 func init() {
17   go SignalHandler(make(chan os.Signal, 1))
18 }
19
20 func main() {
21   message := "hello world"
22   HandleFunc("/hello", func(w ResponseWriter, r *Request) {
23     w.Header().Set("Content-Type", "text/plain")
24     Fprintf(w, message)
25   })
26
27   Launch("HTTP", func() error {
28     return ListenAndServe(ADDRESS, nil)
29   })
30
31   Launch("HTTPS", func() error {
32     return ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
33   })
34   servers.Wait()
35 }
36
37 func Launch(name string, f func() error) {
38   servers.Add(1)
```

```
39    go func() {
40      defer servers.Done()
41      if e := f(); e != nil {
42        Println(name, "->", e)
43        syscall.Kill(syscall.Getpid(), syscall.SIGABRT)
44      }
45    }()
46  }
47
48  func SignalHandler(c chan os.Signal) {
49    signal.Notify(c, os.Interrupt, syscall.SIGABRT, syscall.SIGTERM, syscall.SIGQUIT)
50    for s := <- c; ; s = <- c {
51      switch s {
52      case syscall.SIGABRT:
53        Println("abnormal exit")
54        os.Exit(1)
55      case os.Interrupt, syscall.SIGTERM, syscall.SIGQUIT:
56        Println("clean shutdown")
57        os.Exit(0)
58      }
59    }
60  }
```

We've modified our **Launch()** function to take a name which can be displayed as part of an error message, and its function parameter now has the signature **func() error** which specifies that it must return an **error** value, which is what's returned by both **ListenAndServe()** and **ListenAndServeTLS()**. In the event the **error** (which is a predeclared interface) contains a value then we know an error condition's occurred and can send a **SIGABRT** signal with **syscall.Kill()**. As **Kill()** is able to send signals to any running process we need to specify the ID of the current process, which we find using **syscall.Getpid()**.

```
$ go run 47.go
2014/06/25 14:42:25 HTTPS -> listen tcp :1024: bind: address already in use
abnormal exit
exit status 1
```

## TCP/IP

Printing text in a web browser is a cool trick, but what of **real** network programming? You know, the kind that bearded sandle-wearing ***nix** hackers go in for? It turns out this is surprisingly simple:

**Example 1.48 TCP/IP server**

```
package main

import (
  . "fmt"
  "net"
)

func main() {
  if listener, e := net.Listen("tcp", ":1024"); e == nil {
    for {
      if connection, e := listener.Accept(); e == nil {
        go func(c net.Conn) {
          defer c.Close()
          Fprintln(c, "hello world")
        }(connection)
      }
    }
  }
}
```

The **net** package revolves around server **Listener__s and client __Connection__s. A __Listener** is an **interface** which allows any type implementing its specified methods - **Accept()**, **Close()** and **Addr()** - to be used interchangeably and is a key tool in **Go** for generalising program design. Writing a server then becomes a simple process of:

- **Listen()** on a specified protocol and port number
- **Accept()** incoming connections
- for each **net.Conn**, run a handler in a separate **goroutine**
- read from and write to the connection whilst performing work
- close each connection when it finishes its work

Here we start to see some of the power of **interfaces** as a **net.Conn** implements the **Writer** interface defined in the **io** package, and **fmt.Fprintf()** takes any type which integrates **io.Writer** as its target.

Moving away from **HTTP** means abandoning the browser for testing but both ***nix** and **Windows** have a handy command-line utility called **telnet** which allows us to connect directly to a **TCP/IP** server and interact with it. We'll get into the interaction side of things later in the book, for now here's an example run of our program.

```
$ go run 48.go &
[1] 17415
$ telnet localhost 1024
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello world
Connection closed by foreign host.
```

Telnet's a useful tool, but it'd be nice if we could connect our own client to the server as this could then be built for any platform supported by **Go**. For stream-oriented protocols like TCP/IP we do this using the **net**.**Dial()** function to open a **net**.**Conn** connection to a server and we can then interact with this using the **io**.**Reader** and **io**.**Writer** interfaces. These interfaces are supported throughout the **Go** standard package library, allowing files and streaming connections to be used interchangeably.

**Example 1.49 TCP/IP client**

```
1  package main
2
3  import (
4    "bufio"
5    . "fmt"
6    "net"
7  )
8
9  func main() {
10   if connection, e := net.Dial("tcp", ":1024"); e == nil {
11     defer connection.Close()
12     if text, e := bufio.NewReader(connection).ReadString('\n'); e == nil {
13       Printf(text)
14     }
15   }
16 }
```

Because a **net**.**Conn** represents streams of data flowing between client and server we've introduced the **bufio** package to our client so that the data it's receiving is buffered. This avoids our having to write our own code for buffering incoming data and is another example of the flexibility **Go**'s interfaces provide.

```
$ go run 48.go &
[1] 6102
$ go run 49.go
hello world
$ go run 49.go
hello world
```

Most books on network programming tend to stop at vanilla TCP/IP and leave figuring out how to establish a secure connection between client and server as an exercise for the reader. However we're not likely to get another chance to look at this problem with the same lack of clutter that **Hello World**

provides, and anyway we know how to generate a key and a certificate from our **HTTPS** adventure so we might as well reuse the knowledge. This time we're going to need two sets of keys so let's take care of that first.

```
$ cp cert.pem server.cert.pem
$ cp key.pem server.key.pem
$ go run $GOROOT/src/pkg/crypto/tls/generate_cert.go -ca=true -host="localhost"
2014/05/16 20:41:53 written cert.pem
2014/05/16 20:41:53 written key.pem
$ cp cert.pem client.cert.pem
$ cp key.pem client.key.pem
```

Now we have our keys sorted, let's take a look at what a TCP/IP server looks like in **Go**.

**Example 1.50 TCP/IP server with tls**

```
1  package main
2
3  import (
4    "crypto/rand"
5    "crypto/tls"
6    . "fmt"
7  )
8
9  func main() {
10   if certificate, e := tls.LoadX509KeyPair("server.cert.pem", "server.key.pem"); e == n\
11 il {
12     config := tls.Config{
13       Certificates: []tls.Certificate{ certificate },
14       Rand: rand.Reader,
15     }
16
17     if listener, e := tls.Listen("tcp", ":1025", &config); e == nil {
18       for {
19         if connection, e := listener.Accept(); e == nil {
20           go func(c *tls.Conn) {
21             defer c.Close()
22             Fprintln(c, "hello world")
23           }(connection.(*tls.Conn))
24         }
25       }
26     }
27   }
28 }
```

Importing **crypto/tls** provides us with an equivalent API to that defined in **net** and this means that as **tls.Listen()** fulfils the **net.Listener** interface our connections will be of type **net.Conn**. As a result if we want to pass the connection around inside our code we either have to import **net** so we can use **net.Conn** or perform a type assertion to use the connection as a ***tls.Conn**. We've made the latter choice here.

```
18    if connection, e := listener.Accept(); e == nil {
19      go func(c *tls.Conn) {
20        defer c.Close()
21        Fprintln(c, "hello world")
22      }(connection.(*tls.Conn))
23    }
```

For a server we import **crypto/rand** to access **rand.Reader**, a cryptographically secure pseudo-random number generator which we'll be using as a source of randomness in the TLS connection. We then create a certificate using **tls.LoadX509KeyPair()** to load the server key pair and if this is successful then we set up a listener to accept incoming connections and write "**Hello World**" to a client.

As we're using **TLS** we can't test this version of **Hello World** using **telnet** so instead we need to write a client. Yet again this requires a keypair and where in our previous client we called **net.Dial()** we now use **tls.Dial()**, resulting in a very similar program.

**Example 1.51 TCP/IP client with tls**

```
1   package main
2
3   import (
4     "bufio"
5     "crypto/tls"
6     . "fmt"
7   )
8
9   func main() {
10    if certificate, e := tls.LoadX509KeyPair("client.cert.pem", "client.key.pem"); e == n\
11  il {
12      config := tls.Config{
13        Certificates: []tls.Certificate{ certificate },
14        InsecureSkipVerify: true,
15      }
16
17      if connection, e := tls.Dial("tcp", ":1025", &config); e == nil {
18        defer connection.Close()
19        if text, e := bufio.NewReader(connection).ReadString('\n'); e == nil {
20          Printf(text)
21        }
22      }
23    }
24  }
```

```
$ go run 50.go &
[1] 6107
$ go run 51.go
hello world
$ go run 51.go
hello world
```

Looking back at our **HTTP** experiments, we were able to write a program which served over both **HTTP** and **HTTPS** connections. It'd be nice to do something similar with **TCP/IP**, if only to compare the two code-paths.

**Example 1.52 TCP/IP dual-mode server**

```
1  package main
2
3  import (
4    "crypto/rand"
5    "crypto/tls"
6    . "fmt"
7    "net"
8    "sync"
9  )
10
11 var servers sync.WaitGroup
12
13 func main() {
14   if listener, e := net.Listen("tcp", ":1024"); e == nil {
15     Serve(listener)
16   }
17
18   Serve(TLSListener("server.cert.pem", "server.key.pem", ":1025"))
19   servers.Wait()
20 }
21
22 func TLSListener(cert, key, address string) (r net.Listener) {
23   if certificate, e := tls.LoadX509KeyPair(cert, key); e == nil {
24     config := tls.Config{
25       Certificates: []tls.Certificate{ certificate },
26       Rand: rand.Reader,
27     }
28     if listener, e := tls.Listen("tcp", address, &config); e == nil {
29       r = listener
30     }
31   }
32   return
33 }
34
35 func Serve(listener net.Listener) {
36   if listener != nil {
37     Launch(func() {
```

```
38        for {
39          if connection, e := listener.Accept(); e == nil {
40            go func(c net.Conn) {
41              defer c.Close()
42              Fprintln(c, "hello world")
43            }(connection)
44          }
45        }
46      })
47    }
48  }
49
50  func Launch(f func()) {
51    servers.Add(1)
52    go func() {
53      defer servers.Done()
54      f()
55    }()
56  }
```

We've reused **Launch()** from **Example 1.33** to manage the lifecycle of our two server **goroutines** and introduced **Serve()** to phrase the server behaviour in terms of the **net.Listener** interface. We then move all the setup code for creating a **tls.Listener** into a separate function **TLSListener()** which returns a **net.Listener** value as **tls.Listener** complies with its interface, or a nil value if __tls.Listen() returns an error.

If we now run this server we can connect to it with both of our client programs.

```
$ go run 52.go &
[1] 6278
$ go run 49.go
hello world
$ go run 51.go
hello world
$ go run 51.go
hello world
$ go run 49.go
hello world
```

## UDP

Both TCP/IP and HTTP communications are connection-oriented and this involves a reasonable amount of handshaking and error-correction to assemble data packets in the correct order. For most applications this is exactly how we want to build our network applications but sometimes the size of our messages is sufficiently small that we can fit them into individual packets, and when this is the case the UDP protocol is an ideal candidate.

As with our previous examples we still need both server and client applications.

**Example 1.53 UDP server**

```
package main

import (
  . "fmt"
  "net"
)

var HELLO_WORLD = ([]byte)("Hello World\n")

func main() {
  if address, e := net.ResolveUDPAddr("udp", ":1024"); e == nil {
    if server, e := net.ListenUDP("udp", address); e == nil {
      for buffer := MakeBuffer(); ; buffer = MakeBuffer() {
        if n, client, e := server.ReadFromUDP(buffer); e == nil {
          go func(c *net.UDPAddr, packet []byte) {
            if n, e := server.WriteToUDP(HELLO_WORLD, c); e == nil {
              Printf("%v bytes written to: %v\n", n, c)
            }
          }(client, buffer[:n])
        }
      }
    }
  }
}

func MakeBuffer() (r []byte) {
  return make([]byte, 1024)
}
```

We have a somewhat more complex code pattern here than with TCP/IP to take account of the difference in underlying semantics: UDP is an unreliable transport dealing in individual packets (datagrams) which are independent of each other, therefore a server doesn't maintain streams to any of its clients and these are responsible for any error-correction or packet ordering which may be necessary to coordinate successive signals. Because of these differences from TCP/IP we end up with the following generic workflow:

- **net.ResolveUDPAddr()** to resolve the address
- **net.ListenUDP()** opens a UDP port and listens for traffic
- **net.ReadFromUDP()** copies incoming data into a buffer and provides the remote client's address
- **net.WriteToUDP()** writes data back to the remote client's address

For trivial uses of UDP we could probably forego the use of a separate **goroutine** to process each received packet, and indeed we may also have an application architecture where instead of processing the packet we'd hand it off to a message queue elsewhere. However many real-world examples such as serving DNS requests may introduce appreciable delays for processing and by using __goroutine__s we ensure the server itself isn't stalled.

Here our main overhead is the cost of buffer allocation as we use a different data buffer for each request. In a real-world example we'd very likely introduce a buffer pool which would expand and contract with demand, and reuse individual buffers once their associated request has completed. This is surprisingly simple to implement in **Go** and we'll look at this in detail in a later chapter.

Our client has the same basic boilerplate as the server, only we use **net.DialUDP()** to set up a connection. We could use **net.ReadFromUDP()** and **net.WriteToUDP()** however as a **net.UDPConn** connection implements the **io.ReadWriter** interface we can use **bufio.Reader** to manage reading, and for writes the connection already knows the server address. As the server only knows about clients by receiving data from them we start our interaction with a **UDPConn.Write()** and then perform the buffered ReadString() to get a response.

**Example 1.54 UDP client**

```
1  package main
2
3  import (
4    "bufio"
5    . "fmt"
6    "net"
7  )
8
9  var CRLF = ([]byte)("\n")
10
11 func main() {
12   if address, e := net.ResolveUDPAddr("udp", ":1024"); e == nil {
13     if server, e := net.DialUDP("udp", nil, address); e == nil {
14       defer server.Close()
15       if _, e = server.Write(CRLF); e == nil {
16         if text, e := bufio.NewReader(server).ReadString('\n'); e == nil {
17           Printf("%v", text)
18         }
19       }
20     }
21   }
22 }
```

Let's give this a test run in the shell.

```
$ go run 53.go &
[2] 12777
$ go run 54.go
12 bytes written to: 127.0.0.1:58015
Hello World
$ go run 54.go
12 bytes written to: 127.0.0.1:50159
Hello World
$ go run 54.go
12 bytes written to: 127.0.0.1:51813
Hello World
```

Note how each time we run the client program it's assigned a different network port by the operating system each time **net.DialUDP** is called.

In the case of OSX this port will usually be at the upper end of the non-privileged port range.

We can make this apparent by performing multiple sequences of Write() and Read() operations:

**Example 1.55 UDP client**

```
1  package main
2
3  import (
4    "bufio"
5    . "fmt"
6    "net"
7  )
8
9  var CRLF = ([]byte)("\n")
10
11 func main() {
12   if address, e := net.ResolveUDPAddr("udp", ":1024"); e == nil {
13     if server, e := net.DialUDP("udp", nil, address); e == nil {
14       defer server.Close()
15       for i := 0; i < 3; i++ {
16         if _, e = server.Write(CRLF); e == nil {
17           if text, e := bufio.NewReader(server).ReadString('\n'); e == nil {
18             Printf("%v: %v", i, text)
19           }
20         }
21       }
22     }
23   }
24 }
```

```
$ go run 53.go &
[1] 12883
$ go run 55.go
12 bytes written to: 127.0.0.1:51732
0: Hello World
12 bytes written to: 127.0.0.1:51732
1: Hello World
12 bytes written to: 127.0.0.1:51732
2: Hello World
$ go run 55.go
12 bytes written to: 127.0.0.1:55504
0: Hello World
```

```
12 bytes written to: 127.0.0.1:55504
1: Hello World
12 bytes written to: 127.0.0.1:55504
2: Hello World
```

## RSA obfuscated UDP

With all of our network examples to date we've included a secure transport option, but UDP doesn't have a secured mode so we appear stuck with sending our message unencrypted for all the world to see. This is fine for a message such as **Hello World** which we're happy for intervening network nodes to observe, but what if we want to send confidential data in our UDP packet?

In the following example we're going to use our existing client RSA key-pair by sending the public key to our server which will then encrypt the message with this key and send it back to the client. The client already possesses the RSA private key so it's a simple task to decrypt the message and display it. When we send the public key we could do so in a number of different formats: as an RSA pem file, as a raw binary buffer, or serialised in some form. As both client and server are written in **Go** we'll opt for the serialisation format provided in package **gob**. This is a pragmatic choice as if we were to send a **pem** file then that'd make it obvious that we're using an encrypted format, and if we use a raw binary buffer we'd have to include a discussion of **Go**'s **unsafe** and **reflection** packages which are covered later in this book.

**Example 1.56 RSA-enabled UDP server**

```
1  package main
2
3  import (
4    "bytes"
5    "crypto/rand"
6    "crypto/rsa"
7    "crypto/sha1"
8    "encoding/gob"
9    . "fmt"
10   . "net"
11 )
12
13 var HELLO_WORLD = []byte("Hello World")
14 var RSA_LABEL = []byte("served")
15
16 func main() {
17   Serve(":1025", func(connection *UDPConn, c *UDPAddr, packet *bytes.Buffer) (n int) {
18     var key rsa.PublicKey
19     if e := gob.NewDecoder(packet).Decode(&key); e == nil {
20       if response, e := rsa.EncryptOAEP(sha1.New(), rand.Reader, &key, HELLO_WORLD, RSA\
21 _LABEL); e == nil {
22         n, _ = connection.WriteToUDP(response, c)
23       }
24     }
25     return
26   })
27 }
```

```
28
29 func Serve(address string, f func(*UDPConn, *UDPAddr, *bytes.Buffer) int) {
30   Launch(address, func(connection *UDPConn) {
31     for {
32       buffer := make([]byte, 1024)
33       if n, client, e := connection.ReadFromUDP(buffer); e == nil {
34         go func(c *UDPAddr, b []byte) {
35           if n := f(connection, c, bytes.NewBuffer(b)); n != 0 {
36             Println(n, "bytes written to", c)
37           }
38         }(client, buffer[:n])
39       }
40     }
41   })
42 }
43
44 func Launch(address string, f func(*UDPConn)) {
45   if a, e := ResolveUDPAddr("udp", address); e == nil {
46     if server, e := ListenUDP("udp", a); e == nil {
47       f(server)
48     }
49   }
50 }
```

So, the first thing to note is that we've refactored connection management into **Serve()** to make the server code easier to follow, and then we're passing a **function literal** into this with the tasks to be performed each time a client connects. For now this is a quick hack so we're not launching **Serve()** in its own **goroutine** with all the extra boilerplate for **sync.WaitGroup** which we've seen in previous examples. However we are spawning a separate **goroutine** for each packet received so that the server doesn't block, and as an added bonus each time data is written to a client the number of bytes transferred is logged.

For each connection we read the client's message which we know should be a valid public key in **gob** format. To decode this we create a **gob.Decoder** with the message as its base, then **Decode()** this to get a valid **rsa.PublicKey** which we then use to encrypt our message with **rsa.EncryptOAEP()**. The main thing to note here is that **RSA_LABEL** is a parameter which must be set the same for both **rsa.EncryptOAEP()** and **rsa.DecryptOAEP()** for the message to be correctly read by the latter. There's no reason why this couldn't be configured on a per-connection basis.

Now let's take a look at our client application.

**Example 1.57 RSA-enabled UDP client**

```
1 package main
2
3 import (
4   "bytes"
5   "crypto/rand"
6   "crypto/rsa"
7   "crypto/sha1"
8   "crypto/x509"
```

```
9     "encoding/gob"
10    "encoding/pem"
11    "io/ioutil"
12    . "fmt"
13    . "net"
14  )
15
16  var RSA_LABEL = []byte("served")
17
18  func main() {
19    Connect(":1025", func(server *UDPConn, private_key *rsa.PrivateKey) {
20      cipher_text := MakeBuffer()
21      if n, e := server.Read(cipher_text); e == nil {
22        if plain_text, e := rsa.DecryptOAEP(sha1.New(), rand.Reader, private_key, cipher_\
23  text[:n], RSA_LABEL); e == nil {
24          Println((string)(plain_text))
25        }
26      }
27    })
28  }
29
30  func Connect(address string, f func(*UDPConn, *rsa.PrivateKey)) {
31    LoadPrivateKey("client.key.pem", func(private_key *rsa.PrivateKey) {
32      if address, e := ResolveUDPAddr("udp", ":1025"); e == nil {
33        if server, e := DialUDP("udp", nil, address); e == nil {
34          defer server.Close()
35          SendKey(server, private_key.PublicKey, func() {
36            f(server, private_key)
37          })
38        }
39      }
40    })
41  }
42
43  func LoadPrivateKey(file string, f func(*rsa.PrivateKey)) {
44    if file, e := ioutil.ReadFile(file); e == nil {
45      if block, _ := pem.Decode(file); block != nil {
46        if block.Type == "RSA PRIVATE KEY" {
47          if key, _ := x509.ParsePKCS1PrivateKey(block.Bytes); key != nil {
48            f(key)
49          }
50        }
51      }
52    }
53    return
54  }
55
56  func SendKey(server *UDPConn, public_key rsa.PublicKey, f func()) {
57    var encoded_key bytes.Buffer
```

```
58    if e := gob.NewEncoder(&encoded_key).Encode(public_key); e == nil {
59      if _, e = server.Write(encoded_key.Bytes()); e == nil {
60        f()
61      }
62    }
63  }
64
65  func MakeBuffer() (r []byte) {
66    return make([]byte, 1024)
67  }
```

The most obvious thing about this code is the heavy use of **function literals**, giving it a clean compositional feel. This is an aesthetic I picked up working with Ruby and which I always missed when dipping back into C or other low-level languages, so expect to see many more examples of this style in later chapters.

**Connect()** is the client version of **Serve()**, abstracting away the details of contacting a UDP server, and the meat of our program's interaction is a simple **Read()** of an encrypted message from the server which is then decrypted using **rsa.DecryptOAEP()** and displayed. Before our code initiates the connection though we need it to load an RSA key-pair so we can transmit our public key to the server. We do this in **LoadPrivateKey()** which uses **ioutil.ReadFile()** to load a pem-encoded file into memory and ensure it contains a private key before invoking a function passed to it as a parameter. In this case the passed function sets up the connection, sends the public key to the server and then invokes the function passed to **Connect()** in **SendKey()**.

To keep **SendKey()** as generic as possible it takes a parameterless function which is basically just a closure into the caller's environment. In the case of **Connect()** the closure we pass to **SendKey()** binds to the **server** and **private_key** variables.

## Error Handling

The examples in this chapter are for the most part designed to follow the *happy* path as our interest is in seeing some simple **Go** code that we can later build upon. The one obvious exception was when we explored signal handling and used the presence of an error as an excuse to send a **SIGABRT** to terminate the server. However error-handling is a large part of most real-world programming - especially in a system level language where.

**Go** takes a typically pragmatic approach to error handling, the language specification defining **error** type as a predeclared interface:

```
type error interface {
        Error() string
}
```

In the following example we're going to rewrite our encrypted UDP server from example 1.56 so that start-up errors cause the server to terminate and signal an error to the shell whilst client errors will log an appropriate message for later analysis using the **log** package. Whilst our program is still trivial in purpose, we now have all the basic conveniences for running a scalable server and integrating it with third-party monitoring and logging tools.

**Example 1.58**

```
package main

import (
  "bytes"
  "crypto/rand"
  "crypto/rsa"
  "crypto/sha1"
  "encoding/gob"
  "log"
  . "net"
)

var HELLO_WORLD = []byte("Hello World")
var RSA_LABEL = []byte("served")

func main() {
  Serve(":1025", func(connection *UDPConn, c *UDPAddr, packet *bytes.Buffer) (n int) {
    var e error
    var key rsa.PublicKey
    var response []byte

    if e = gob.NewDecoder(packet).Decode(&key); e != nil {
      log.Println("unable to decode wrapper:", c)
    }

    if response, e = rsa.EncryptOAEP(sha1.New(), rand.Reader, &key, HELLO_WORLD, RSA_LA\
BEL); e != nil {
      log.Println("unable to encrypt server response")
    }

    if n, e = connection.WriteToUDP(response, c); e != nil {
      log.Println("unable to write response to client:", c)
    }
    return
  })
}

func Serve(address string, f func(*UDPConn, *UDPAddr, *bytes.Buffer) int) {
  Launch(address, func(connection *UDPConn) {
    for {
      buffer := make([]byte, 1024)
      if n, client, e := connection.ReadFromUDP(buffer); e == nil {
        go func(c *UDPAddr, b []byte) {
          if n := f(connection, c, bytes.NewBuffer(b)); n != 0 {
            log.Println(n, "bytes written to", c)
          }
        }(client, buffer[:n])
```

```
48        } else {
49          log.Println(address, e.Error())
50        }
51      }
52    })
53  }
54
55  func Launch(address string, f func(*UDPConn)) {
56    var e error
57    var a *UDPAddr
58    var server *UDPConn
59
60    if a, e = ResolveUDPAddr("udp", address); e != nil {
61      log.Fatalln("unable to resolve UDP address:", e.Error())
62    }
63
64    if server, e = ListenUDP("udp", a); e != nil {
65      log.Fatalln("can't open socket for listening:", e.Error())
66    }
67
68    f(server)
69  }
```

One of the cool things about interfaces is that they're reference types, something we'll routinely use to decide whether an error has occured or not. If it has the interface will contain an **error** value, and if not the interface itself will be a **nil** value. This leads to the common code pattern

```
if _, e := SomeCall(); e != nil {
  log.Println("some error", e)
  // this is our sad path
} else {
  // performed desired actions
}
```

Our *sad* path will generally either return it's own error to the calling function or terminate the program with a call to **log.Fatalln()** or **os.Exit()**. Because **Go** functions allow for multiple return values, their use is accompanied by the convention that the last value returned will be of type **error**. This convention encourages us to handle errors where they occur rather than bubbling **exceptions** up the call stack, as would be the case in many languages. We'll look at how we can achieve a similar outcome in the next section.

For now we're going to tidy this code up a little by using the **if...{}...else if {}... construct which thanks to __if**'s ability to combine an assignment with a test leads to very succinct code.

**Example 1.59**

```
package main

import (
  "bytes"
  "crypto/rand"
  "crypto/rsa"
  "crypto/sha1"
  "encoding/gob"
  "log"
  . "net"
)

var HELLO_WORLD = []byte("Hello World")
var RSA_LABEL = []byte("served")

func main() {
  Serve(":1025", func(connection *UDPConn, c *UDPAddr, packet *bytes.Buffer) (n int) {
    var key rsa.PublicKey
    var response []byte

    if e := gob.NewDecoder(packet).Decode(&key); e != nil {
      log.Println("unable to decode wrapper:", c)
    } else if response, e = rsa.EncryptOAEP(sha1.New(), rand.Reader, &key, HELLO_WORLD,\
 RSA_LABEL); e != nil {
      log.Println("unable to encrypt server response")
    } else if n, e = connection.WriteToUDP(response, c); e != nil {
      log.Println("unable to write response to client:", c)
    }
    return
  })
}

func Serve(address string, f func(*UDPConn, *UDPAddr, *bytes.Buffer) int) {
  Launch(address, func(connection *UDPConn) {
    for {
      buffer := make([]byte, 1024)
      if n, client, e := connection.ReadFromUDP(buffer); e == nil {
        go func(c *UDPAddr, b []byte) {
          if n := f(connection, c, bytes.NewBuffer(b)); n != 0 {
            log.Println(n, "bytes written to", c)
          }
        }(client, buffer[:n])
      } else {
        log.Println(address, e.Error())
      }
    }
  })
```

```
48  }
49
50  func Launch(address string, f func(*UDPConn)) {
51    var connection *UDPConn
52
53    if a, e := ResolveUDPAddr("udp", address); e != nil {
54      log.Fatalln("unable to resolve UDP address:", e.Error())
55    } else if connection, e = ListenUDP("udp", a); e != nil {
56      log.Fatalln("can't open socket for listening:", e.Error())
57    }
58    f(connection)
59  }
```

Whilst I personally find this easier on the eye in many cases, it's a less common idiom than successive **if** statements and because **Go**'s scoping rules allow each assignment to introduce new variables local to that **if** statement's scope some care should be taken in variable naming to avoid accidental **shadowing**.

Because **error** is defined as an interface rather than a concrete type we can declare our own types for error handling and then check for them to specialise error handling behaviour. In the next example we introduce the **LaunchError** type which complies with the predeclared **error** interface.

**Example 1.60**

```
1   package main
2
3   import (
4     "bytes"
5     "crypto/rand"
6     "crypto/rsa"
7     "crypto/sha1"
8     "encoding/gob"
9     "fmt"
10    "log"
11    . "net"
12  )
13
14  var HELLO_WORLD = []byte("Hello World")
15  var RSA_LABEL = []byte("served")
16
17  type LaunchError []interface{}
18
19  func (l LaunchError) Error() (r string) {
20    if len(l) > 0 {
21      r = fmt.Sprintf(l[0].(string), l[1:]...)
22    }
23    return
24  }
25
26  func NewLaunchError(format string, v ...interface{}) (l LaunchError) {
27    return LaunchError(append([]interface{}{ format }, v))
```

```
28  }
29
30  func main() {
31    Serve(":1025", func(connection *UDPConn, c *UDPAddr, packet *bytes.Buffer) (n int) {
32      var key rsa.PublicKey
33      var response []byte
34
35      if e := gob.NewDecoder(packet).Decode(&key); e != nil {
36        log.Println("unable to decode wrapper:", c)
37      } else if response, e = rsa.EncryptOAEP(sha1.New(), rand.Reader, &key, HELLO_WORLD,\
38   RSA_LABEL); e != nil {
39        log.Println("unable to encrypt server response")
40      } else if n, e = connection.WriteToUDP(response, c); e != nil {
41        log.Println("unable to write response to client:", c)
42      }
43      return
44    })
45  }
46
47  func Serve(address string, f func(*UDPConn, *UDPAddr, *bytes.Buffer) int) {
48    e := Launch(address, func(connection *UDPConn) (e error) {
49      defer func() {
50        if x := recover(); x != nil {
51          e = LaunchError{ "serve failure %v", x }
52        }
53      }()
54      for {
55        buffer := make([]byte, 1024)
56        if n, client, e := connection.ReadFromUDP(buffer); e == nil {
57          go func(c *UDPAddr, b []byte) {
58            if n := f(connection, c, bytes.NewBuffer(b)); n != 0 {
59              log.Println(n, "bytes written to", c)
60            }
61          }(client, buffer[:n])
62        } else {
63          log.Println(address, e.Error())
64        }
65      }
66      return
67    })
68
69    if e, ok:= e.(LaunchError); ok {
70      log.Fatalln(e.Error())
71    }
72  }
73
74  func Launch(address string, f func(*UDPConn) error) error {
75    var connection *UDPConn
76
```

```
77    if a, e := ResolveUDPAddr("udp", address); e != nil {
78      return NewLaunchError("unable to resolve UDP address:", e.Error())
79    } else if connection, e = ListenUDP("udp", a); e != nil {
80      return LaunchError{ "can't open socket for listening:", e.Error() }
81    }
82    return f(connection)
83  }
```

This is quite a complicated example, so let's take a look at the various changes we've made in detail.

```
17  type LaunchError []interface{}
18
19  func (l LaunchError) Error() (r string) {
20    if len(l) > 0 {
21      r = fmt.Sprintf(l[0].(string), l[1:]...)
22    }
23    return
24  }
25
26  func NewLaunchError(format string, v ...interface{}) (l LaunchError) {
27    return LaunchError(append([]interface{}{ format }, v))
28  }
```

For simplicity we've made **LaunchError** a slice of **interface{}** values and declared an **Error()** method which uses the first element in the slice as a format string and successive elements as parameters for **fmt.Sprintf()**. The **NewLaunchError()** function is a typical example of a value constructor which we'd export from a package, and later in the code we show both construction this way and using a **LaunchError{}** literal.

```
1   func Launch(address string, f func(*UDPConn) error) error {
2     var connection *UDPConn
3
4     if a, e := ResolveUDPAddr("udp", address); e != nil {
5       return NewLaunchError("unable to resolve UDP address:", e.Error())
6     } else if connection, e = ListenUDP("udp", a); e != nil {
7       return LaunchError{ "can't open socket for listening:", e.Error() }
8     }
9     return f(connection)
10  }
```

We've made another change to **Launch()** related to our new approach to error handling, which is to propogate these errors back to the caller via a return value - and just for completeness we're allowing errors to bubble up from the function parameter **f** as well. This leads to changes in **Serve** as well.

```
func Serve(address string, f func(*UDPConn, *UDPAddr, *bytes.Buffer) int) {
  e := Launch(address, func(connection *UDPConn) (e error) {
    defer func() {
      if x := recover(); x != nil {
        e = LaunchError{ "serve failure %v", e }
      }
    }()
```

Here we set a value for **e** from the **error** returned by **Launch()** as well as intercepting any **panic** raised by the associated closure with **defer** and instead returning a **LaunchError** rather than crashing the program. As **defer** takes a closure the **e** referenced inside it is the same **e** as that declared by the function literal **func(connection *UDPConn) (e error)**, a pattern encountered in many available **Go** codebases. We then use the returned error value before exiting **Serve()**.

```
  if e, ok := e.(LaunchError); ok {
    log.Fatalln(e.Error())
  }
```

It should come as no surprise that **Go** provides support for creating **error** values without our having to define **error** types, in the form of **errors.New** and **fmt.Errorf**. Using these we can remove the **LaunchError** type and rewrite our program.

**Example 1.61**

```
package main

import (
  "bytes"
  "crypto/rand"
  "crypto/rsa"
  "crypto/sha1"
  "encoding/gob"
  "errors"
  "fmt"
  "log"
  . "net"
)

var HELLO_WORLD = []byte("Hello World")
var RSA_LABEL = []byte("served")

func main() {
  Serve(":1025", func(connection *UDPConn, c *UDPAddr, packet *bytes.Buffer) (n int) {
    var key rsa.PublicKey
    var response []byte

    if e := gob.NewDecoder(packet).Decode(&key); e != nil {
      log.Println("unable to decode wrapper:", c)
    } else if response, e = rsa.EncryptOAEP(sha1.New(), rand.Reader, &key, HELLO_WORLD,\
```

```
 RSA_LABEL); e != nil {
      log.Println("unable to encrypt server response")
    } else if n, e = connection.WriteToUDP(response, c); e != nil {
      log.Println("unable to write response to client:", c)
    }
    return
  })
}

func Serve(address string, f func(*UDPConn, *UDPAddr, *bytes.Buffer) int) {
  e := Launch(address, func(connection *UDPConn) (e error) {
    defer func() {
      if x := recover(); x != nil {
        e = fmt.Errorf("serve failure %v", x)
      }
    }()
    for {
      buffer := make([]byte, 1024)
      if n, client, e := connection.ReadFromUDP(buffer); e == nil {
        go func(c *UDPAddr, b []byte) {
          if n := f(connection, c, bytes.NewBuffer(b)); n != 0 {
            log.Println(n, "bytes written to", c)
          }
        }(client, buffer[:n])
      } else {
        log.Println(address, e.Error())
      }
    }
    return
  })

  if e != nil {
    log.Fatalln(e.Error())
  }
}

func Launch(address string, f func(*UDPConn) error) error {
  var connection *UDPConn

  if a, e := ResolveUDPAddr("udp", address); e != nil {
    return fmt.Errorf("unable to resolve UDP address: %v", e)
  } else if connection, e = ListenUDP("udp", a); e != nil {
    return errors.New(fmt.Sprintf("can't open socket for listening: %v", e.Error()))
  }
  return f(connection)
}
```

## Exceptions

If we consider these programs for a minute or two it becomes apparent that propagating our errors this way works very well when we wish to deal with an error immediately, which is usually the case. However there are occasions when an error will need to be propagated through several layers of function calls, and when this is the case there's a lot of boilerplate involved.

We can do away with much of this by rolling our own lightweight equivalent of **exceptions** using **defer** and the **panic()** and **recover()** calls. In the final example of this chapter we'll do just this, introducing the **Exception** type which is an **interface** with **error** embedded within it. This means that any **error** value will also be an **Exception** value.

**Example 1.62**

```
1  package main
2
3  import (
4    "bytes"
5    "crypto/rand"
6    "crypto/rsa"
7    "crypto/sha1"
8    "encoding/gob"
9    "fmt"
10   "log"
11   . "net"
12 )
13
14 var HELLO_WORLD = []byte("Hello World")
15 var RSA_LABEL = []byte("served")
16
17 type Exception interface {
18   error
19 }
20
21 type LaunchException []interface{}
22
23 func (l LaunchException) Error() (r string) {
24   if len(l) > 0 {
25     r = fmt.Sprintf(l[0].(string), l[1:]...)
26   }
27   return
28 }
29
30 func RaiseLaunchException(format string, v ...interface{}) {
31   panic(LaunchException(append([]interface{}{ format }, v)))
32 }
33
34 func main() {
35   Serve(":1025", func(connection *UDPConn, c *UDPAddr, packet *bytes.Buffer) (n int) {
36     var key rsa.PublicKey
```

```
37      var response []byte
38
39      if e := gob.NewDecoder(packet).Decode(&key); e != nil {
40        log.Println("unable to decode wrapper:", c)
41      } else if response, e = rsa.EncryptOAEP(sha1.New(), rand.Reader, &key, HELLO_WORLD,\
42   RSA_LABEL); e != nil {
43        log.Println("unable to encrypt server response")
44      } else if n, e = connection.WriteToUDP(response, c); e != nil {
45        log.Println("unable to write response to client:", c)
46      }
47      return
48    })
49  }
50
51  func Serve(address string, f func(*UDPConn, *UDPAddr, *bytes.Buffer) int) {
52    defer func() {
53      if e := recover(); e != nil {
54        switch e := e.(type) {
55        case LaunchException:
56          log.Fatalln("Launch Error:", e.Error())
57        case Exception:
58          log.Fatalln("Exception:", e.Error())
59        default:
60          panic(e)
61        }
62      }
63    }()
64
65    Launch(address, func(connection *UDPConn) {
66      for {
67        defer func() {
68          if e := recover(); e != nil {
69            if _, ok := e.(Exception); ok {
70              panic(e)
71            }
72            RaiseLaunchException("serve failure %v", e)
73          }
74        }()
75
76        buffer := make([]byte, 1024)
77        if n, client, e := connection.ReadFromUDP(buffer); e == nil {
78          go func(c *UDPAddr, b []byte) {
79            if n := f(connection, c, bytes.NewBuffer(b)); n != 0 {
80              log.Println(n, "bytes written to", c)
81            }
82          }(client, buffer[:n])
83        } else {
84          log.Println(address, e.Error())
85        }
```

```
86      }
87      return
88    })
89  }
90
91  func Launch(address string, f func(*UDPConn)) {
92    var connection *UDPConn
93
94    if a, e := ResolveUDPAddr("udp", address); e != nil {
95      RaiseLaunchException("unable to resolve UDP address:", e.Error())
96    } else if connection, e = ListenUDP("udp", a); e != nil {
97      panic(LaunchException{ "can't open socket for listening:", e.Error() })
98    }
99    f(connection)
100 }
```

Our updated code looks surprisingly similar to that of **Example 1.60** with a set of type declarations but **NewLaunchException()** is replaced by **RaiseLaunchException()** which as its name suggests generates a **panic** to propagate the **LaunchException** value back up the calling stack.

```
1  func RaiseLaunchException(format string, v ...interface{}) {
2    panic(LaunchException(append([]interface{}{ format }, v)))
3  }
```

To intercept this **panic** we need a **defer** statement somewhere up the call stack which can handle it, otherwise it will bubble up through **main()** and the program will terminate with a stack trace. In this case we check for an **Exception** in **Serve()**.

```
1  func Serve(address string, f func(*UDPConn, *UDPAddr, *bytes.Buffer) int) {
2    defer func() {
3      if e := recover(); e != nil {
4        switch e := e.(type) {
5        case LaunchException:
6          log.Fatalln("Launch Error:", e.Error())
7        case Exception:
8          log.Fatalln("Exception:", e.Error())
9        default:
10         panic(e)
11       }
12     }
13   }()
```

Note how we're using a **type** switch to select different courses of action depending on the **type** of the value intercepted by **recover()**, and we have a **default** case which causes any unrecognised **panic** to continue bubbling up the call stack.

We use an **if** statement in the function passed to **Launch()** to achieve the same effect, bubbling any **Exception** up through the call stack and converting any other value into a **LaunchException** with **RaiseLaunchException()**.

```
defer func() {
  if e := recover(); e != nil {
    if _, ok := e.(Exception); ok {
      panic(e)
    }
    RaiseLaunchException("serve failure %v", e)
  }
}()
```

Similar code patterns can be used to pass any value up the call stack and it's possible to implement a number of non-standard flow-of-control constructs in a very similar way to **Exception**.