Introduction to Programming in Go

Student Manual

Terms of Usage

Document Information

Course Title:	Introduction to Programming in Go
Author:	Rod Davison
e-mail:	rod@exgnosis.com
Version:	1.0
Release Date:	2016-10-01
ProTech ID:	PT10491

"The primary objective of copyright is not 'to reward the labor of authors, but to promote the Progress of Science and useful Arts.' To this end, copyright assures authors the right to their original expression, but encourages others to build freely upon the ideas and information conveyed by a work. This result is neither unfair nor unfortunate. It is the means by which copyright advances the progress of science and art."

Justice Sandra Day O'Connor
Feist Publications, Inc. v. Rural Telephone Service Co.
499 US 340, 349(1991)

Student Preface

This is the student manual for the course *Introduction to Programming in Go* offered at Capital One and presented by Protech Training. This manual is intended to serve a couple of functions:

Provide you with a more complete record of the class content

There is only so much you can cram into a slide or powerpoint presentation. This manual covers the same topics as in the slides but provides more depth on each of the topics. This means that you can go back and revisit the material later without having to remember (and you won't) what the instructor actually said about that slide.

Ensure that you get all the intended content

Sometimes in a class not all of the topics are covered equally. This may be due to the skill level of the class, or the fact the class wants to go into more depth on certain topics and less on others, or any of a number of other legitimate reasons for the content as delivered in class to not match up the with the content that was planned. If this happens in your class, and hopefully it won't, at least you have all the planned content in the manual to take away with you.

Add more interesting content beyond what is presented in class.

At some points in the manual I have refrenced extra content, often from the blogs and presentations by the developers of the Go language which, I think, gives some interesting insights and background to what is being discussed in the lectures. This extra material is generally not included in the presentation simply because of the time restrictions we have on the class time available.

What this manual is not

This manual is not a textbook or a brilliant and incisive exploration of Go. It is merely intended to be a useful resource for the course. There is a substantial amount of material floating around about Go – in fact, there is so much that it would be very difficult for me to actually come up with any really novel content.

Go has a fantastic amount of reference material that is easily available including the Go language specification, Go blogs, the Go tour and many other presentations and articles, many of which are written by Rob Pike and the other inventors of the Go language.

This manual is not intended to be a restatement of those available materials. I will reference them and provide instructions on how to access them, but getting them and actually using them will be your responsibility

Prerequisite Expectations

This course is intended for experienced programmers who have spent time programming in a C-type language, by which we mean any language that uses a syntax and structure derived from C such as C#, Java, C++ or even JavaScript. If your programming background is in another language like Python, you may have to work a bit harder to understand the examples but if you have a good solid understanding of programming constructs, then you should be okay.

This is not an introductory programming course. There will be no remedial work done in class to get people up to speed on programming concepts – there is just no time to do that. Consider this a fair warning that if you are not comfortable with writing code then you will find the class hard going. One possible strategy that you might explore if you find yourself out of your depth is to partner up with another student who is a good programmer to at least be able to see how a programmer works with the exercises.

Expected Technical Setup

This section only applies if you are not using one of the supplied virtual machines.

It is assumed that you have the Go programming environment installed and tested on your computer before class starts. There will be no time during the class to go through any installation procedures although one is described in the first module of this manual. The procedure described here is the installation onto a generic machine from the Go project website and other public websites, however Capital One may have their own internal sites from which Go should be installed instead if you are installing it on a Capital One computer.

Course Objectives

The purpose of this course is to get you familiar enough with Go, both in terms of how the language works, and the Go ecosystem that you can go back and start working with Go in your operational environment. The course will have a series of traditional modules covering a number of Go topics, each of which is followed by a lab session where you can work with the concepts from the presentation. The latter part of the course will be a selection of programming projects that should allow you to pull together the concepts and techniques of the course in some hands on programming challenges.

Keep in mind that this is an introductory survey course. That means that we will touch on a wide range of Go related topics without going into a great deal of depth on many of them. For example, we are going to learn about concurrency in Go but we will not learn everything about the topic. Some realistic expectations are that after the class is that you should be able to:

1. Write clean Go code and be able to read Go code and understand the Go code design and techniques used.
2. Be able to use the Go programming environment and utilities (the Go ecosystem) and understand how they work.

3. Get a feel for what the “Go way” of doing things is, especially some of the Go idioms and practices.

This course will not make you into an expert Go programmer – to reach that goal will take a lot more study and practice beyond what we will be able to do in these three days. Programming expertise in any programming language is made up of two parts: first, understanding how the language works and then developing your skill at designing solutions and writing code to implement those solutions in that language. Three days is not enough time for you to develop that depth of Go programming skill, that will be something that you do yourself through practice after this session is done. Think of this course as an orientation into the Go programming paradigm and the starting point for your Go skill development in the months ahead.

You also have to keep in mind that this course is not comprehensive or intensive. There are many topics we just don’t have the time to get into like the cryptography libraries and working with integrating Go with legacy C code, as interesting as these topics might be. There are also topics we can only touch on superficially because of the same time constraints for the course – we could probably spend two days alone exploring concurrency in Go.

I often use the analogy that learning a programming language is like learning to speak a foreign language like French. This course is akin a one semester college course that introduces you to French grammar, pronunciation and vocabulary (and assumes that you already speak Spanish). But learning about how the French the language works, while a critical step in becoming fluent, is not the same as learning to speak French, which is a skill that only comes by going out and speaking French.

Hands On Work

Hands on work in a course like this is critical for learning. In the first module, along with the instructions for installing Go, is a recommendation for a couple of IDEs that Go programmers often use. I strongly recommend that you get one and use it during the class since tools like colored syntax highlighting, automatic formatting and expression completion make the process of learning the basic of the language a lot smoother and easier

During the instructor’s presentation, there will be a number of code samples used to illustrate concepts. The code for all of these examples is available to you in the appropriate directories in the provided github repository (or some other place will be provided to you at the start of the class), as well as being available on the virtual machine you will be using. Very often, we learn programming concepts most effectively by writing code, playing with code and trying to break code. I would encourage you to load the example code, run it and play with it during the presentation.

The code examples are not intended to show off some sort of nifty programming expertise – they are kept lean and sparse so that the concept being demonstrated is easily seen. It may make the code seem a bit simplistic but that is the whole point of an example. Personally I find it frustrating when I’m learning a new programming language and the author illustrates a simple idea by throwing out a 500 line cargo scheduling application. Yes the application is cool, but it is so complex that I have a hard time finding exactly the point in all the code that is supposed to be exemplified. The more complex code samples will be part of the labs.

Lab Work

I have tried to get away from the usual, step-by-step, follow-the-instructions-by-rote, get-the-answer-in-the-back-of-the-book labs. I've tried to mix it up a bit with some discussion questions, some labs that are mini-projects and some that are where you experiment with code and then try and explain the results. The last part of the course has labs that are fairly significant in size and complexity where we build a couple of interesting applications. Most of these applications variations on standard Go mini-projects so you will be able to find other implementations of them just by a bit of Googling.

We are not looking for "right answers" in the lab. The labs are for experimentation, practicing and getting that hands on component that is often the most critical part in learning a programming language. You might think you "got it" during the lecture but the only way to be sure is to work with it.

The labs are not intended to be a solitary activity. I would encourage you to collaborate and confer with your classmates. Often a lot of learning can take place just in the process of trying to explain how something works to one of your colleagues. It's also enlightening to see how other people are approaching a problem and compare that to how you are doing it. Think of the labs as a team or class activity if you want, but if it is more your style to work on your own, just do you own thing.

Above all, try to have fun with the labs. You learn best when you are enjoying yourself.

If you are not an experienced programmer, you may find some of the more challenging problems too advanced in which case I would suggest again looking at the provided solutions or work with someone who is more experienced. I would suggest the latter as the preferred approach

> *"The mediocre teacher tells. The good teacher explains. The superior teacher demonstrates. The great teacher inspires."*
>
> William Arthur Ward

Module One

Getting Started with Go

The most important thing in a programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language.
Donald E. Knuth

Go is not meant to innovate programming theory. It's meant to innovate programming practice.
Samuel Tesla

Why would you have a language that is not theoretically exciting? Because it's very useful.
Rob Pike

1.1 Installing Go

You may not need this section for class if you are using a Protech virtual machine, however you may want to install Go on one of your own computers, in which case this section may be useful to you.

This section contains a copy of the instructions from the Go language project on how to download and install go. While these instructions use the public Go project website, Capital One may have their own internal URL that you should install from.

The public URL for getting Go is:

https://golang.org/

The current version of Go at the time this manual was prepared was 1.8.1

1.1.1 Installing the Go Tools

There can only be one Go installation at a time on a machine so any previous versions of Go must be removed prior to a new installation.

The following are the installation instructions for Go as they appear on the golang.org website.

1.1.1.1 Linux, Mac OS X, and FreeBSD tarballs

Download the archive and extract it into /usr/local, creating a Go tree in /usr/local/go. For example:

```
tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

Choose the archive file appropriate for your installation. For instance, if you are installing Go version 1.2.1 for 64-bit x86 on Linux, the archive you want is called go1.2.1.linux-amd64.tar.gz. (Typically these commands must be run as root or through sudo.)

Add /usr/local/go/bin to the PATH environment variable. You can do this by adding this line to your /etc/profile (for a system-wide installation) or $HOME/.profile:

```
export PATH=$PATH:/usr/local/go/bin
```

Installing to a custom location

The Go binary distributions assume they will be installed in /usr/local/go (or c:\Go under Windows), but it is possible to install the Go tools to a different location. In this case you must set the GOROOT environment variable to point to the directory in which it was installed.

For example, if you installed Go to your home directory you should add the following commands to $HOME/.profile:

```
export GOROOT=$HOME/go
export PATH=$PATH:$GOROOT/bin
```

Note: GOROOT must be set only when installing to a custom location.

1.1.1.2 Mac OS X package installer

Download the package file, open it, and follow the prompts to install the Go tools. The package installs the Go distribution to /usr/local/go.

The package should put the /usr/local/go/bin directory in your PATH environment variable. You may need to restart any open Terminal sessions for the change to take effect.

1.1.1.3 Windows

The Go project provides two installation options for Windows users (besides installing from source): a zip archive that requires you to set some environment variables and an MSI installer that configures your installation automatically.

MSI installer

Open the MSI file and follow the prompts to install the Go tools. By default, the installer puts the Go distribution in c:\Go.

The installer should put the c:\Go\bin directory in your PATH environment variable. You may need to restart any open command prompts for the change to take effect.

Zip archive

Download the zip file and extract it into the directory of your choice (we suggest c:\Go).

If you chose a directory other than c:\Go, you must set the GOROOT environment variable to your chosen path.

Add the bin subdirectory of your Go root (for example, c:\Go\bin) to your PATH environment variable.

Setting environment variables under Windows

Under Windows, you may set environment variables through the "Environment Variables" button on the "Advanced" tab of the "System" control panel. Some versions of Windows provide this control panel through the "Advanced System Settings" option inside the "System" control panel.

1.2 Go IDEs

The following is a partial list of IDEs that support Go. The one that was used in the preparation of the course materials is the LiteIDE since it is specifically targeted for Go development. Some others:

1. Go plugin for Eclipse. Found at https://github.com/GoClipse/goclipse.
2. Go plugin for IntelliJ IDE. Found at http://go-ide.com.
3. GOSublime Found at https://github.com/DisposaBoy/GoSublime
4. Atom. Found at https://atom.io/packages/go-plus

The LiteIDE can be found at

https://sourceforge.net/projects/liteide/

My reason for using LiteIDE that it seems to be the easiest one to install, use and configure, but then that might just be a matter of personal preference.

However, if you already have a preferred Go IDE please use that but be aware that we will not spend class time on learning any particular IDE and most of the class work can be done with a standard editor (VIM or gedit or notepad++) that supports Go syntax highlighting.

If you do choose to use the LiteIDE, installation of the LiteIDE is quite simple.

1. Unpack the tar file or archive at the location you want to install the IDE. Call this location IDEHOME.
2. Add $IDEHOME/bin to your path.
3. Either run from the command line or create a shortcut to the executable.

1.3 The Go Playground

If you have access to the golang.org site, you can run basic Go programs in an interactive Go environment called the Go playground located at:

https://play.golang.org/

Not all of the functionality of GO is available in the GO playground but it can be used for executing self contained GO code. For example, the only IO features available in the playground are writing to stdout and stderr. As well, the time in the playground never changes but is always 11pm November 10, 2009 UTC.

Using the playground is only a temporary measure and, while you will be able to do some of the things in the course with it, you will not be able to do everything with it that you need to do in the course.

1.4 Hello World

We start with the standard “Hello World” program.

We are assuming that you already have experience programming in a C-style programming language like C, C++, C#, Java and to a certain extent, JavaScript and PERL. Because Go is also a C-Style language, that means that the "hello world" Go program looks quite similar to programs written in one of those languages, which we sort of expect since C is the grandparent of all C-style programming languages, so there is a family resemblance in how code looks in all of those languages.

However, Go is not Java or C – we have to be careful of what we call *faux amis* or false friends. This term is used by French instructors to warn English speakers learning French that while some French words look exactly like English words, the meanings in the two languages are quite different. For example the French word “location” actually means “rental” and not "place."

There are constructs in Go that look like constructs in these other C-style languages, but they don't always do exactly the same thing in Go that they do in those other languages. Don't assume that because something in Go looks like something in Java or C that it works the same way as those languages.

Example 01-01 is our Hello World program in Go.

```go
// Example 01-01  Hello World

package main

import (
  "fmt"
)

func main() {
  fmt.Println("Hello World")
}
```

```
[Module01]$ go run hello.go
Hello World
```

The following is a line by line walk-through of the Hello World program.

1.4.1 Comments

```
// Example 01-01  Hello World
```

The first line in the file is a comment.

Comments in Go work just like comments in other C-style languages.

1.4.2 The package Statement

```
package main
```

The first line in every Go source file must be a package statement.

1. Packages in Go can be thought of code modules or libraries.
2. Each source file begins with a package declaration. Every code construct in Go ***must*** belong to a package.
3. The main package is a special package where Go looks for the entry point to a stand-alone application.

The designers of Go are quite insistent that packages do not form namespaces in the same sense they do in C++. The package model used in Go is not derived from Java but instead is derived from Modula-2, along with the Module-2 mechanism for imports and syntax for declarations.

```
import (
   "fmt"
)
```

1.4.3 The import Statement

To use symbols in other packages, the packages must be imported. Importing doesn't actually move anything around, it just identifies where another package is located so that symbols in that package can be accessed. In this case, we are accessing symbols, which can be data or functions, in the package "fmt."

It is a compile time error to import a package that is not used in order to simplify dependency resolution. This is a perfect example of the Go design approach. Since we are concerned about build times, we don't want to go out and start pulling in or resolving all kinds of imports when they are never used, so the compiler checks to see if they are used and generates an error if they are not.

1.4.4 The main Function

```
func main() {
```

The main() function in Go is the entry point for a Go standalone application – execution of the application starts with a call to the main function. The concept of a standalone main function should be familiar to C and C++ programmers since it serves the same purpose in those languages as it does in Go.

The main function must be in the main package. If there is not exactly one main function in the main package, compilation is aborted.

1.4.5 The fmt.Println("Hello World") function call

```
fmt.Println("Hello World!")
```

The Println() function is located in the fmt package, which is why we have the import statement – so that we can actually find the function.

1. Symbols used from other packages are prefixed with the package name.
2. Only symbols that begin with an uppercase letter can be imported.
3. Symbols that begin with a lowercase letter are private to the package.

The import and visibility mechanism in Go was apparently a point of some debate. As Rob Pike says

> *Go takes an unusual approach to defining the visibility of an identifier, the ability for a client of a package to use the item named by the identifier. Unlike, for instance, private and public keywords, in Go the name itself carries the information: the case of the initial letter of the identifier determines the visibility. If the initial character is an upper case letter, the identifier is exported (public); otherwise it is not:*
>
> *upper case initial letter: Name is visible to clients of package*
>
> *otherwise: name (or _Name) is not visible to clients of package*

This rule applies to variables, types, functions, methods, constants, fields... everything. That's all there is to it.This was not an easy design decision. We spent over a year struggling to define the notation to specify an identifier's visibility. Once we settled on using the case of the name, we soon realized it had become one of the most important properties about the language. The name is, after all, what clients of the package use; putting the visibility in the name rather than its type means that it's always clear when looking at an identifier whether it is part of the public API. After using Go for a while, it feels burdensome when going back to other languages that require looking up the declaration to discover this information.

1.5 The Go Tool

The go tool is a bundle of functionality that we will be accessing throughout the course. The idea behind the tool is that Go is not just a programming language but a software development system. We will look at the motivation for this view in the next section. The go tool is part of the standard Go installation.

Typing "go" at the command prompt should produce the following output.

```
Usage:

        go command [arguments]

The commands are:

        build       compile packages and dependencies
        clean       remove object files
        doc         show documentation for package or symbol
        env         print Go environment information
        fix         run go tool fix on packages
        fmt         run gofmt on package sources
        generate    generate Go files by processing source
        get         download and install packages and
                    dependencies
        install     compile and install packages and
                    dependencies
        list        list packages
        run         compile and run Go program
        test        test packages
        tool        run specified go tool
        version     print Go version
        vet         run go tool vet on packages
```

One of the problems that Pike and the other Go designers had noticed is that to create and deploy software, a programming language usually has to be supported by a number of third party build and deployment tools. The proliferation of tools and environments in large scale software development environments, like that found at Google, starts to put limitations on the productivity of software teams in terms of both working together and in maintaining a standard development environment. The go tool addresses those issues by making the build tools part an integral part of the Go language environment.

We will be looking at a number of these uses of the tool during the course but only in enough detail so that you get a good sense of what these various options do. Each of the options has a rich set of features and going into detail about them would take more time than we have available in class.

The go tool is fully documented at:

https://golang.org/cmd/go/

There is also an introductory article on using the go tool at:

https://golang.org/doc/articles/go_command.html

1.5.1 Compiling and Running "Hello World"

A couple of points about compiling and building Go code in general:

1. Any source file to be compiled has to have a .go extension.
2. The go tool can run/compile/build the file in three different ways.
3. The “run” option compiles, links and runs the resulting executable file and then deletes the executable and any intermediate files that were created. This is the option we will be using in the first few modules.
4. The "build" option compiles a source file and all of its dependencies but does not update any installed binaries. Instead it just leaves an executable in the directory the tool is called from. The "go build" command deletes all of the intermediate files it created during the build.
5. We will look at the “install” option later in this course, but as you might guess it does the same thing as build but then installs the binaries in their appropriate places. What we mean by appropriate places will be a topic for a late module.
6. You may not be able to use "build" or "run" if more that one file in the same directory has a main function.

We will explore this topic more in the lab.

1.6 Design Philosophy of Go

In one his lectures on Go, Rob Pike, one of the lead developers of the Go language, described the motivation for developing Go.

> *The Go programming language was conceived in late 2007 as an answer to some of the problems we were seeing developing software infrastructure at Google.The problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model were being worked around rather than addressed head-on. Moreover, the scale has changed: today's server programs comprise tens of millions of lines of code, are worked on by hundreds or even thousands of programmers, and are updated literally every day. To make matters worse, build times, even on large compilation clusters, have stretched to many minutes, even hours.Rob Pike*

The motivation and design ideas behind Go are not something we want to spend a lot of time on, but it is important to understand something about the reasons why Go was designed to work in the specific ways that it does. Understanding the "why" to a certain extent helps understand "how," which I believe makes learning the language easier.

Go was not intended to be a new kind of programming language that was supposed to go head to head against other languages like Java in the same way that Java was intended to be a more modern and improved evolutionary step up from C++.

In terms of execution performance on standard benchmark tasks, Go seems to benchmark better than Java but not as good as C or C++. Most gurus agree this result is partly due to the maturity of decades of development of the C/C++ compilers. But Go was not intended to compete on computing tasks with C/C++ or any other programming language, instead the focus of Go is on making big software projects get into production fast and efficiently.

We also want to lay to rest the misconception that Go is like Ruby or some of the more exotic programming languages introduced to provide new ways to write code or implement new programming paradigms or ideas. Go is not a new kind of programming language or a new style of programming, but is an attempt to develop a programming language that is more efficient for software production in modern computing architectures. In fact some of Go's features can be thought of as a reversion to C programming.

Go is not about programming, Go is about software production.

The root cause of the problem that motivated Google to develop Go was that the existing languages that Google was using were ancient – dating back to the 1990s and 1980s when the the scale of computing systems was smaller, when hardware was a lot simpler and when there were no large massive computing environments like those in modern enterprise IT environments such as those found at Amazon, Facebook or Google.

These older languages were great for programming conceptually, and still are, but they can not keep up with the production demands of modern computing in large sale software production environments. Those historical languages just do not scale well into modern software development environments and practices.

From https://golang.org/doc/articles/go_command.html:

> *You might have seen early Go talks in which Rob Pike jokes that the idea for Go arose while waiting for a large Google server to compile. That really was the motivation for Go: to build a language that worked well for building the large software that Google writes and runs. It was clear from the start that such a language must provide a way to express dependencies between code libraries clearly, hence the package grouping and the explicit import blocks. It was also clear from the start that you might want arbitrary syntax for describing the code being imported; this is why import paths are string literals.*
>
> *An explicit goal for Go from the beginning was to be able to build Go code using only the information found in the source itself, not needing to write a makefile or one of the many modern replacements for makefiles. If Go needed a configuration file to explain how to build your program, then Go would have failed.*

The one point that will become a theme in the course is that Go simplifies the build process by tossing out features that do not provide critical value for software engineering but can have a downside effect by either increasing the complexity of the language implementation, which in turn can result in a more lengthy build process, or by producing code that is overly complex either at the source level or compiled binary level.

1.6.1 The Focus of Go

1. Go was not intended to be a "better" programming language or new way of programming (C and C++ produce faster executing code in general).
2. Go addresses issues of build and delivery for complex and large scale software projects.
3. Go utilizes modern technology for efficiency which older languages can not do because of their original design.
4. Go address issues of scale that cannot be addressed with other languages' using ad hoc compile and build technologies.
5. Go provides a consistent tool set to avoid third party tool mahem.

As Pike again notes

> *When Go launched, some claimed it was missing particular features or methodologies that were regarded as de rigueur for a modern language. How could Go be worthwhile in the absence of these facilities? Our answer to that is that the properties Go does have address the issues that make large-scale software development difficult. These issues include:*
>
> - *slow builds*
> - *uncontrolled dependencies*
> - *each programmer using a different subset of the language*
> - *poor program understanding (code hard to read, poorly documented, and so on)*
> - *duplication of effort*
> - *cost of updates*
> - *version skew*
> - *difficulty of writing automatic tools*
> - *cross-language builds*

> *The primary considerations for any language to succeed in this context are:*
>
> *(1) It must work at scale, for large programs with large numbers of dependencies, with large teams of programmers working on them.*
>
> *(2) It must be familiar, roughly C-like. Programmers working at Google are early in their careers and are most familiar with procedural languages, particularly from the C family. The need to get programmers productive quickly in a new language means that the language cannot be too radical.*
>
> *(3) It must be modern. C, C++, and to some extent Java are quite old, designed before the advent of multicore machines, networking, and web application development. There are features of the modern world that are better met by newer approaches, such as built-in concurrency.*

1.6.2 Simplification in Go

Go simplifies inorder to improve build efficiency and support clean code design by eliminating or revising language features that add complexity to a language but may not offer enough benefit to justify the increase in complexity. For example:

1. Features that make other languages overly complex are omitted
2. Features producing overly complex compiled code are eliminated
3. Features that slow compilation are eliminated
4. Concurrency is a fundamental and core part of the language
5. Incorporation of remote importing of packages for internet type environments is supported as a fundamental part of the Go ecosystem
6. Modern environments and practices such as unit testing are part of the Go ecosystem

If you are a programmer who is used to another language, then Go may seem to be missing a lot of modern programming language features, and you would be right.

Go does throw away:

1. Function Overloading
2. Operator Overloading
3. Inheritance
4. Implicit type conversion
5. Exceptions and Exception handling

The reasons Pike gave for dumping these features is that they are not critical to code development and either add to the complexity of the language implementation, including the compilation process, or increase the likelihood they will contribute to poorly designed code.

That doesn't mean, for example, that Go doesn't implement object orientation, it just does it differently to try to get the same sort of capabilities these features offer in other languages but without the associated costs.

1.7 Go Formatting

One of the innovative and welcome features of Go is that formatting of source code is both standardized and automated. If you are used to working in languages like C or Java with free form formatting, then Go may seem a bit more rigid and unyielding.

Certain formatting features are not optional, such as the opening brace “{“ for a function body appearing on the same line as the function declaration. Some of these rules are not in place for stylistic reasons but rather to speed up the lexer, which is a part of the compiler that parses the string of characters in a source file into Go tokens and symbols.

Go uses semi-colons to terminate statements but does not want them around except when separating more than one statement on a single line. Instead, the lexer inserts semicolons at the ends of lines that terminate Go statements. Again, this is to speed up the lexer portion of the compiler.

The formatting that is not critical to the syntax of the language is standardized to ensure a consistent Go style across large software projects, shared libraries and packages irrespective of the authors.

In order to support a standard formatting, the Go tool has a fmt option that will reformat any Go source file into the standard Go style code format. We will explore this feature in the lab. Of course if your formatting is so bad it produces compile errors, then your code is beyond the ability of go fmt to save.

If you are using an IDE, you might notice that whenever you save a file the IDE calls the go fmt utility and automatically formats your file. At least the LiteIDE does.

1.8 Go Standard Libraries

Go, like other programming languages, comes with a large number of standard libraries or packages, for example we have already used the “fmt” package to use the Println function for output. The fmt library is described this way in the standard documentation:

> *Package fmt implements formatted I/O with functions analogous to C's printf and scanf. The format 'verbs' are derived from C's but are simpler.*

Aside from cases where we are interested in exploring some specific functionality, we will not be delving deeply into all of the functionality of the standard packages for three reasons:

1. The documentation for the libraries is extensive, well written, easy to use and very accessible on the Go language site. In fact the URL is https://golang.org/pkg/
2. The purpose of the course is not to exhaustively explore all of the contents of the libraries, which would get very boring after a while, but to to help you understand the concepts of Go programming.
3. As an experienced programmer, you know that you don't really remember all the functionality in a standard library, just what you tend to use the most. When you need to see if some functionality exists, you go look for it. The course has the same expectation.

Many of these libraries or core Go packages are very similar to other libraries in other languages, such as the class libraries in Java. There are a number of these that we will not deal with during the course since they are intended for very specific applications or are not as central to the theme of the course. These are:

1. archive: provides support for using zip and tar archives
2. bufio: provides support for buffered I/o operations.
3. compress: provides support for various compression algorithms like bzip2, zlib, lwz and others.
4. context: defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.
5. crypto: used to implement various crypto related standards like DES, elliptic cruves, MD5 hashes and others.
6. database: implementation of various drivers and a generic SQL interface for databases.
7. debug: provides access to symbolic debugging information for various binary formats.
8. hash: interfaces to hash function like adler checksum and crc checksums.

9. image: a basic 2-D image library
10. mime: implements mime multi-part parsing.
11. path: utility routines for manipulating slash-separated paths.
12. regexp: implements regular expression search
13. text: scanner and tokenizer and data driven output templates
14. unsafe: contains operations that step around type safety in Go

We will be using a number of the other libraries in the course. A lot of these packages, like the net package, have sub-packages and can be quite large and comprehensive. We won't be exploring any of these packages in depth but we will be using some of the functionality from them, mostly in the labs.

1. bytes: support for byte slice operations, similar to operations in strings
2. container: support for heaps, double linked lists and circular lists.
3. encoding: supports different encodings like Json, base 64 etc
4. errors: implementation of Go error mechanism.
5. html: functions for escaping and unescaping HTML text.
6. index: fast substring searching
7. math: basic math library
8. net: a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets.
9. os: a platform-independent interface to operating system functionality.
10. reflect: run-time reflection, allowing a program to manipulate objects with arbitrary types.
11. runtime: operations that interact with Go's runtime system
12. sort: provides primitives for sorting slices and user-defined collections.
13. strconv: implements conversions to and from string representations of basic data types.
14. strings: implements simple functions to manipulate UTF-8 encoded strings.
15. sync: provides basic synchronization primitives such as mutual exclusion locks.
16. testing: provides support for automated testing of Go packages.
17: time: functionality for measuring and displaying time.
18: unicode: data and functions to test some properties of Unicode code points.

Module Two

Variables and Data Types

In Go, the code does exactly what it says on the page.
Andrew Gerrand

Go doesn't implicitly anything.
Steven in golang-nuts

2.1 Introduction

Programming languages generally tend to fall into one of two categories. On one hand we have the strictly typed compiled languages like C, C++ and Java, while on the other we have the untyped interpreted languages like Python, Ruby and JavaScript. Generally the languages within each category tend to resemble each other in how they manage variables – if you know C++ you can generally translate that knowledge into Java, for example, without too much problem.

Go is a strictly typed compiled language which handles variables similar to how Java and C++ does... sort of. Go generally is not as casual as other languages about casting and converting from one data type to another, a quirk that which can feel somewhat restrictive initially if this is something you commonly do while writing code.

The whole idea of implicit conversion of types is forbidden in Go. Other strictly typed languages permit implicit widening conversions (copying the value from a smaller size variable into a larger size variable such as a short to a long) without any extra syntax. Go does not even allow that relatively safe implicit conversion.

In this module, we will briefly introduce the basic data types. We are not going to go into detail about what they are since the Go data types are pretty standard and should be familiar to you from other C-style languages. We will only focus on the few things that Go does that are somewhat unique to the language.

The section on variables will assume you know how to use variables in other C-style languages and understand concepts like scope, what a local variable is and other fundamental programming concepts related to variables.

The last section deals with pointers since they are used extensively in Go. If you are a C or C++ programmer, this material will be quite familiar to you. If you are a Java programmer, you may have to adjust from thinking in terms of references to thinking in pointers. If you are unfamiliar with pointers, then welcome to what C programmers call “real programming.”.

2.2 Basic Data types

Data Types

Numeric				Non-numeric	
integer		floats	complex	string	bool
signed	unsigned	float32	complex64		
		float64	complex128		
int8	uint8				
int16	uint16				
int32	uint32				
int64	uint64				

int - either int32 or int64
uint - either uint32 or uint64

rune alias for int32
byte alias for uint8

Most of the data types are the same as in other programming languages. The suffix on the numeric types indicates the size of that type. The float types are defined as the IEEE-754 32 bit and 64 bit floating point numbers complete with NaN and the infinity values. The integer forms are either signed twos-complement forms or unsigned integers, much the same as in C. The main difference is that Go has dumped the traditional type names like double, short and long and replaced them with a rather utilitarian naming of the numeric types.

The one addition that most other languages do not have is the complex number type. A complex number is an ordered pair of two floats (a,b) that represent the complex number a + bi. Complex numbers don't occur that often outside of scientific work so we will not be delving into them in this course. As a mathematician, it grieves me to say that.

The bool values are as you would expect, true and false, with no numerical equivalent like that which exists in C and C++.

2.2.1 Strings

Strings are handled quite differently in Go than most other C-style languages. In C and C++ strings are pointers to arrays of integers while Java also treats strings more like an array or object than a basic built-in data type. Strings in Go are basic data types just like the numeric types.

Like Java, strings in Go are immutable and for the same reason: when a string is immutable then it can be allocated into contiguous memory without worrying about ever having to support re-sizing operations. This results in faster and more efficient string processing.

However strings are not quite as simple as in other C-style languages. Strings use UTF-8 encoding in Go and are actually arrays of what we call code points, which we can think of as characters in some alphabet, where each code point can be represented by 1 to 4 bytes of data. The aliases for rune and byte are specifically implemented to help us use UTF-8 strings: runes represent code points while bytes are exactly that.

2.2.2 The int and uint types

The uint and int types are very C-like since they parallel the notion of what an int is in C. The size of these two types is either 32 or 64 bits depending on the underlying architecture of the target machine. The programmer cannot override these and force these types be the size they prefer, that decision is made by the compiler and is out of the control of the programmer. If a specific size is wanted for int, then the corresponding int32 or int64 should be used, and similarly for the uint.

2.2.3 No implicit data conversions

Most programming languages, like Java, allow some sort of implicit conversion of data types. For example, the assignment of a smaller int variable into a larger one is generally allowed because it is considered safe. Similarly, when we divide an integer by a float, generally called a mixed mode operation where the integer is implicitly changed into a float before the division takes place.

So why are implicit conversions and mixed mode arithmetic not allowed in Go? Simply because this is a common source of errors in code and requires the implementation of a lot of behind the scene logic to resolve what sort of promotions or conversions should be done to produce a "correct" result for these sorts of expressions. The kinds of errors that result from these sorts of conversions are usually not syntactic errors that cause some compile time problems but are semantic errors that are not caught until unusual results start showing up during testing, or even worse, during production.

Most programmers have had experience with this sort of problem when data types are converted implicitly in some computation and wind up producing a different result than the one the programmer was expecting when writing the code.

Many programmers, myself included, always explicitly cast data types when we do conversions or mixed mode arithmetic because we know that this is a source of those “duh” errors we have a habit of making. Go just makes that explicit conversion mandatory to eliminate a whole class of problems.

From the Go Blog

> *In the early days of thinking about Go, we talked about a number of problems caused by the way C and its descendants let you mix and match numeric types. Many mysterious bugs, crashes, and portability problems are caused by expressions that combine integers of different sizes and "signedness".*

Nasty bugs lurk here. C has a set of rules called "the usual arithmetic conversions" and it is an indicator of their subtlety that they have changed over the years (introducing yet more bugs, retroactively). When designing Go, we decided to avoid this minefield by mandating that there is no mixing of numeric types. If you want to add i and u, you must be explicit about what you want the result to be.

This strictness eliminates a common cause of bugs and other failures. It is a vital property of Go. But it has a cost: it sometimes requires programmers to decorate their code with clumsy numeric conversions to express their meaning clearly.

2.3 Variable Declaration

Example 02-01 shows what is called the basic long form for initializing variables. The var keyword is required in this form. This is the only form of variable declaration that can be used for package level variable declarations. Why? The same basic reason that we use the func keyword to declare function definitions – it simplifies the lexer component of the compilation process, which will be necessary when we start using functions as first class object. Just like needing that opening "{" on the same line as the function declaration – it allows a much more efficient parsing process. We don't need this efficiency quite as much at the local level because the lexer has more contextual information to work with which means we can use a more compact form for defining local variables.

```
// Example 02-01  Declaring Variables
package main

import "fmt"

var x float32
var c complex64
var b bool
var str string

func main() {
  var message string = "x=%f c=%f b=%t str=|%s| i=%d \n"
  var i int
  fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-01.go
x=0.000000 c=(0.000000+0.000000i) b=false str=|| i=0
```

The example uses the fmt.Printf() function which is from the standard fmt package. The function works a lot like the fprint() or sprintf() functions in other C-style languages so we won't discuss it any further here. If you are unfamiliar with printf() types, check the documentation for the fmt package in the Go documentation.

2.3.1 Default Zero Value

The output of example 02-01 demonstrates the Go principle that there can never be an uninitialized variable. If a variable is not explicitly initialized, then it is set to the zero value for that type. The zero value is the appropriate numeric zero for all the numeric types, false for Boolean values and the empty string for string values. In most C-style languages string variables are actually pointers or references so the undefined value of a string is the null pointer or nil or null or something similar. Since strings are basic data types, the zero value for strings is the empty string.

2.3.2 Explicit Initialization with Literals

Example 02-02 demonstrates how variables are declared and explicitly initialized using literals. One important thing to notice is that we may have to convert the literal to the specific type we need in order to do the initial assignment.

The reason for this is that while literals themselves don't have a type associated with them, they have a default type that is used if one is not explicitly provided. For example, the default type for the literal “0” is int, which means that when we use “0” to initialize a complex variable, we have to convert it first to “0.0 + 0.0i” which is the 0 value for a complex number.

```
// Example 02-02  Initializing Variables
package main

import "fmt"

var x float32 = 32.0
var c complex64 = complex64(0)
var b bool = true
var str string = "Hi"

func main() {
  var message string = "x=%f c=%f b=%t str=|%s| i=%d \n"
  var i int = 3
  fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-02.go
x=32.000000 c=(0.000000+0.000000i) b=true str=|Hi| i=3
```

2.3.3 Implicit Initialization with Literals

If the literal being used to initialize a variable has a known type, then we don't need a type identifier for the variable, it is implied from the type of the initializing literal.

Consider the literal 0 – it could be a int8 or a uint16 or an int128. Because in Go a literal has a default type, the literal 0 defaults to an int in the absence of any other information. If we want the literal to take on any other possible type then we use the conversion operation to change the type.

However we cannot do impossible conversions, for example int64(true) will not work since we can't convert a non-numeric into a numeric.

Example 02-03 shows implicit initialization. Just as a note the “%T” in the fmt.Printf() statement prints out the type of the variable.

```
// Example 02-03  Implicit Initialization
package main

import "fmt"

var x float32 = 32.0
var c = 5.0 + 3.1i
var b = true
var str = "Hi"

func main() {
  var message = "x=%T c=%T b=%T str=%T i=%T \n"
  var i = 3
  fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-03.go
x=float32 c=complex128 b=bool str=string i=int
```

Examples 02-04 and 02-05 show some variations on the syntax that can be used in declaring both package level and local variables.

If we are initializing a number of variables in a single statement as we do in the second "var" statement in the example, then we have to initialize all or none. We cannot have a mismatch between the number of variables and the number of initializers.

Another conservative decision Go has made is to make it a compile time error to declare or define a variable and then not use it. In other languages this is usually a warning usually but Go takes a more code safe approach and makes it an error.

```
// Example 02-04  Implicit Initialization
package main

import "fmt"

var x = float32(0)
var c = complex64(0)
var b = true
var str = "Hi"

func main() {
  var message = "x=%T c=%T b=%T str=%T i=%T \n"
  var i = uint8(0)
  fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-04.go
x=float32 c=complex64 b=bool str=string i=uint8
```

```go
// Example 02-05  Variations
package main

import "fmt"

var i, j int8
var b, str, x = true, "hi", float32(45)

func main() {
  fmt.Printf("i=%T j=%T b=%T str=%T x=%T \n",
       i, j, b, str, x)
}
```

```
[Module02]$ go run ex02-05.go
i=int8 j=int8 b=bool str=string x=float32
```

2.4 Short Form Variable Declaration

When we are declaring variables inside a local scope (ie. not at the package level) we can use the short form of a declaration which is demonstrated in example 02-06

The differences between the long form and the short form are:

1. The var is omitted in the short form and the assignment operator ":=" is used instead of "=".
2. The values on the left hand side of the ":=" may be existing variables like “k” that are being assigned a new value or variables that are being created at that point in the code like “i”.
3. At least one of the variables on the left hand side of the ":=" must be a new declaration otherwise the “=” must be used instead of the “:=” operator.
4. There is no explicit type declaration, the type is always inferred from the initializing literal.
5. This form can only be used with local variables, never with variables at the package level.

Why the short form? The short form will be necessary for doing in-line variable declarations in constructs like for loops where the longer syntax using the "var" keyword just can't work syntactically. The short form can be thought of as more of an "in-line" sort of construct.

```
// Example 02-06  Short Form
package main

import "fmt"

var (
  k int = 1
  m int
)

func main() {

  i, j, k := 3, 4, 5
  x := 1.2

  fmt.Printf("i=%d j=%d x=%f k=%d \n", i, j, x, k)
}
```

```
[Module02]$ go run ex02-06.go
i=3 j=4 x=1.200000 k=5
```

2.5 Scope

There are five kinds of scope in Go, two of which (file and universal) are not really that important to us as programmers. The scopes are:

1. *Universal Scope:* This refers to symbols (like “int32”) that are accessible everywhere in any package in a Go program. We don't write code for this package.
2. *File Scope:* This usually refers to symbols that have the current file as their scope. The package and import statements have file scope since they are only valid in the current file. Aside from these two cases, we don't use file scope in our code in this course.
3. *Package scope:* Symbols which have package scope can be referenced by any code in that package, even if the code is in a different file. These are roughly the global variables of Go, or at least global to the package in which they are defined. Symbols with package scope are always defined with some initial keyword such as func, var or type.
4. *Function scope:* This scope should be familiar to all programmers. These are the variables that are defined in the body of a function or passed as parameters or, in the case of Go, are named return values (more about them later).
5. *Block scope:* Like C and other C-style languages, Go allows the definition of variables that are local to a block, which is a set of code statements delimited by braces { }. Blocks usually occur as the bodies of if statement, for loops or other similar constructs, although we can define a block almost anywhere we want to in a function body.

Example 02-07 on the next page demonstrates this concept.

1. The variable k has package scope indicated by the outermost green box.
2. The variable I has function scope which is indicated by the red box
3. The variable x has block scope indicated by the innermost blue box.

```go
// Example 02-07  Variable Scope
package main

import "fmt"

var k int = 1

func main() {

  i := 1

  {
      x := 1.2
      fmt.Printf("i=%d x=%f k=%d \n", i, x, k)
  }
}
```

2.5.1 Shadowing

We are assuming that everyone is familiar with the idea of variable shadowing since it occurs in all C-style programming languages in one form or another. Go allows variable shadowing without any warning or error from the compiler. This is illustrated in example 02-08

```
// Example 02-08  Shadowing
package main

import "fmt"

var k int = 1

func main() {

  fmt.Printf("Package Scope k=%d \n", k)
  k := 2
  fmt.Printf("Function Scope k=%d \n", k)
  {
       k := 3
       fmt.Printf("Block Scope k=%d \n", k)
  }
  fmt.Printf("Function Scope k=%d \n", k)
}
```

```
[Module02]$ go run ex02-08.go
Package Scope k=1
Function Scope k=2
Block Scope k=3
Function Scope k=2
```

One reason that shadowing of package variables is allowed without comment is that a package variable may be defined in a different file in the same package – not allowing it to be shadowed could cause some very cryptic appearing errors to the programmer who is trying to define a second variable with the same name as an existing variable at a lower level of scope.

2.6 Constants

Constants work the same way as variables both syntactically and semantically with the following differences.

1. Once a constant's value has been initialized, it may not be changed.
2. Defining a constant uses the keyword const instead of var.
3. Because of (2), the short form with ":=" cannot be used to define a constant
4. The value of a constant can be computed by a constant expression, which is a computation that can be done at compile time by the compiler.

Example 02-09 demonstrates the use of constants.

```
// Example 02-09  Constants
package main

import "fmt"

const a float32 = 1
const b = 4 / 3

func main() {
  const c string = "Hello Word"
  fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
  fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
}
```

```
[Module02]$ go run ex02-09.go
a=float32 b=int c=string
a=1.000000 b=1 c=Hello Word
```

2.6.1 iota

iota is an enum generator which was borrowed from a rather innovative programming language called APL. It is used to generate sets of related constants which represent some sort of integral enumeration.

Within a set of constants (which means they are all defined in the same const statement) the first time iota is used, it assigns the first constant the value 0. Then each constant following is assigned the next integer – 1, 2, 3 … etc.

This is illustrated in example 02-10.

```
// Example 02-10 iota and enums
package main

import "fmt"

const (
  a = iota
  b
  c
)

func main() {
  fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
  fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
}
```

```
[Module02]$ go run ex02-10.go
a=int b=int c=int
a=0 b=1 c=2
```

2.6.1 The Blank Variable

If we want the enum series to start at 1 instead of 0 we can use the blank variable "_" for the first enum constant. The blank variable is used extensively in Go when we need to assign a value to something, but want to ignore the value after the assignment. The blank variable is a placeholder used instead of a real variable, which we would then have to use in a statement to avoid a compiler error. The blank variable has no type and cannot be used in any context except as the target in a variable assignment statement. That means that the following statement is not allowed:

```
_ := 42
```

This is illustrated in example 02-11

```
// Example 02-11 iota and enums
package main

import "fmt"

const (
  _ = iota
  a
  b
  c
)

func main() {
  fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
  fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
}
```

```
[Module02]$ go run ex02-11.go
a=int b=int c=int
a=1 b=2 c=3
```

2.6.2 Generator Functions

We don't have to have just a series of sequential integers as an enum series, we can use iota in a generator expression to compute a sequence of enum values. In example 02-12 a simple expression is used to generate the even integers as an enum series. However, the generator function used must be a constant function which can be computed by the compiler at compile time.

```
// Example 02-12 Enums with Generator
package main

import "fmt"

const (
  a = iota * 2
  b
  c
)

func main() {
  fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
  fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
```

```
[Module02]$ go run ex02-12.go
a=int b=int c=int
a=0 b=2 c=4
```

2.7 Pointers

Go uses pointers in a manner quite similar to C. Go has pointer variables which can contain the memory address of a particular type of value. The notation for the addressing operators and de-referencing operators is the same as for C and C++.

For example if we declare the integer variable "i"

```
var i int
```

then we can declare p to be a pointer to an integer with the declaration

```
var p *int
```

They "type" of a pointer is the type of the variable it points to. If p is a pointer to an integer, then it cannot point to a string or a float.

We can assign the address of i to p by using the address-of operator &

```
p = &i
```

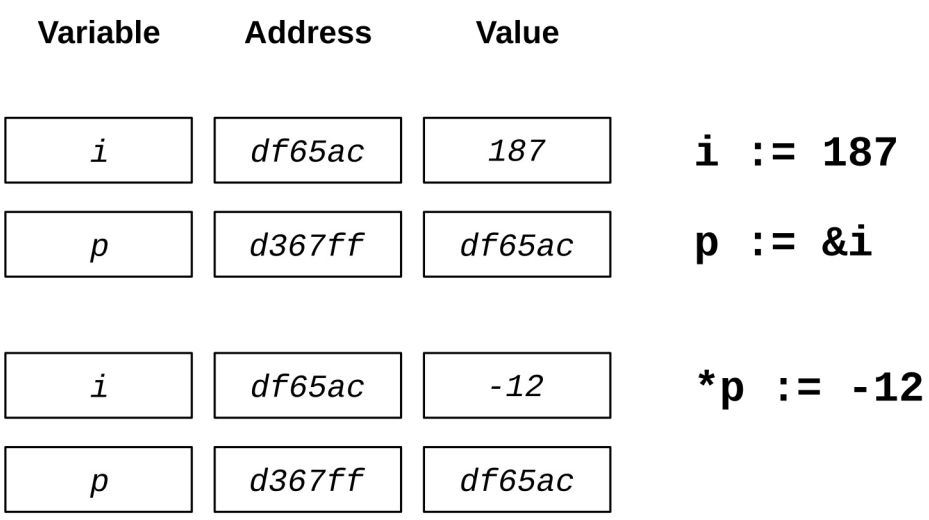
To access the contents of what p points to, we use the de-referencing operator “*”.
If we wanted to assign the value that p points to to another variable, then we would say

```
int j = *p
```

which is often read a “j is assigned the contents of p.”

Pointers are illustrated in example 02-13.

Go allows us to use assignment to assign one pointer to another, and allows the use of the == operator to see if two pointers point to the same memory location. However Go does not support the sort of pointer arithmetic that C and C++ do because it was decided that this provided minimal value to the programmer while permitting operations that historically have been a major source of problems in C and C++ programs. We will play with pointers more in the lab as well as seeing them later in the course quite at bit.



The diagram on the previous page is a graphical illustration of the example. The variable “i” is located at a memory address, which we are calling “df65ac” and the contents of that memory location are the value of “i” which ls the integer 187.

When “p” is assigned the address of i, it is implicitly takes on the type “pointer to int” and itself is a variable with a memory location. However the value of p is the address of the variable “i.” We can do a lot with p, we can assign it to another variable of type “pointer to int” for example, or we can assign it the address of a different int.

However we can also access the value of what “p” points to indirectly by using the “*p” notation, which can be read as “the value of the variable that p points to.” This is often called the indirection operator or the de-referencing operator. We can use this to change the value of “i” to -12, but notice that the value of “p” itself does not change.

```
// Example 02-13 Pointers

package main

import "fmt"

func main() {
  var i int = 187
  var p *int

  p = &i
  fmt.Println("i=", i, " &i or p =", p, " *p =", *p)
  *p = -12
  fmt.Println("i=", i, " &i or p =", p, " *p =", *p)
}
```

```
[Module02]$ go run ex02-13.go
i= 187  &i or p = 0xc82000a398  *p = 187
i= -12  &i or p = 0xc82000a398  *p = -12
```

Module Three

Control Stuctures

Most of the appeal for me is not the features that Go has, but rather the features that have been intentionally left out
txxxxd in Hacker News

I like that Go forces you to clean up little messes, like unused variables, for example.
Jamu Kakar

3.1 Introduction

This module is not deep in terms of introducing new concepts, but it does introduce a lot of detail at the nuts and bolts level of writing Go code. This module is really about getting comfortable with the general appearance, flow and feel of writing Go code in the places where it differs from other C-style languages.

I don't know if my personal experience is commonly shared by other programmers, but there are two things I find frustrating when starting to learn a new programming new language.

The first is that my code doesn't doesn't compile or execute but to my programmer's eye it should because I don't see anything obviously wrong with it. This is because I have missed some critical difference between how I normally write code and how code should be written in the new language. Of course the difference is usually so minor that it does not warrant mention in a lot of the material on the language, but it is not obvious enough to me to pick it up right away – especially since I am conditioned to look at the code in terms of other programming languages I already know.

I have been writing C, Java and C++code for decades and will often write a variable declaration and “int i” in Go because that is what I automatically do, and I then wonder for just a moment what the compiler is complaining about until I remember..oh yeah this is Go and that is not how we do it in Go.

The second thing I find frustrating is when low level details are just not mentioned in most of the material on a language because the people who write in the language regularly see the details as either blindingly obvious or too trivial to mention. That may be true if you know the language, and after I learn the language I realize it is in fact trivial or obvious, but if you don't know the language then it's neither of those initially but almost impossible to find any clarification on that specific topic or point.

This section just focuses on the aspects of Go control structures that are important to point out in terms of transitioning from writing code in another language to writing code in Go. There is no exposition in this module on what conditionals are, or how they work, or what a for loop does – the focus is only how a Go conditional is written differently from conditionals in C, C++ and Java.

3.2 Mixed Mode Operations

The last thing we want to do in this course is to start going over all the operators. Given that you are an experienced programmers, then only teaching points we have to make are about the differences between operators in Go and other C-style languages – there are only a few but, based on my own personal experience, they are significant enough that they really confused me the few times I encountered them.

The most minor difference, in my opinion, is that there is no exponentiation operator. This is not a real loss since there is an exponentiation function in the math package that provides exactly the same functionality.

3.2.1 Increment Operator

The main difference that takes getting used to, or at least it was for me because I was so used to using it in other languages, was that the increment operator "i++" is actually a statement and not an expression. That means that it cannot appear on the right hand side of any sort of assignment operation.

An expression is something that can be evaluated to produce a result which can then use in some way, such as assigning it to a variable or as a value in another expression.

In Go `x++` is a short form of the statement `x = x +1` which does not return a result. A common construct I would use in C++ is j=i++ would be interpreted in Go as `j = i = i+1` which is an illegal statement in Go. In fact Go does not support this chaining of operations where more than one “=” appears in a line.

The operation i++ does not produce a result, it just has a side effect which is to change the value of the variable i. Since i++ is not an expression, the need to distinguish between incrementing before or after we use it is no longer an issue. These means that ++i and i++ are semantically identical which makes the prefix operator is redundant so Go drops its usage and only supports the postfix form.

3.2.2 No Mixed Mode Arithmetic

The mixed mode arithmetic operations rule is something that just takes a little getting used to but more because of the strictness of the rule than because of any conceptual complexity. It's easy to remember that you will get a compiler error when you add a int32 and a float32 but it's more difficult to remember that you can't add a float32 and a float64 either.

The rationale for the no mixed mode rule actually makes a lot of sense. In the section in Module 2 on variable type conversions, I presented a quote by Rob Pike about why this decision was made. It is relevant here as well.

This is illustrated in example 03-01.

```go
// Example 03-01 Operators

package main

import "fmt"

func main() {

  a, b, c := uint8(1), uint16(1), int8(1)
  //fmt.Println("1. a + b =", a+b)
  //fmt.Println("2. a + c =", a+c)
  fmt.Println("3. a + uint8(b)=", a+uint8(b))
  fmt.Println("4. uint8(a) + c=", int8(a)+c))
}
```

```
[Module03]$ go run ex03-01.go
3. a + uint8(b)= 2
4. uint8(a) + c= 2
```

3.2.3 Parallel Assignment

Parallel multiple assignment is not supported in other C-style languages so those languages tend use the the more cumbersome C-style way of using a temporary variable to swap two values. Go does all assignments in parallel as shown in example 03-02

```
// Example 03-02 parallel Assignment

package main

import "fmt"

func main() {

  first, last := "York", "New"
  fmt.Println(first, last)
  first, last = last, first
  fmt.Println(first, last)
}
```

```
[Module03]$ go run ex03-02.go
York New
New York
```

3.3 Conditionals – the if Statement

Conditionals in Go work for the most part like conditionals in other C-style languages except for the following differences:

1. No parentheses "(..)" are allowed around the test condition
2. Local variables can be defined in the if statement itself
3. Braces "{ .. }" are mandatory for all then and else blocks
4. The opening "{" for each block cannot start on a new line
5. The else keyword cannot appear at the start of a new line

Most of these differences are syntactic which is not a conceptually difficult stretch to make. The major difference between Go and other C-style conditionals is the way variables that are local to the conditional are defined.

```
// Example 03-03 Basic If Statement

package main

import "fmt"

func main() {
  x := 22
  if x == 0 {
       fmt.Printf("%d is zero\n", x)
  } else if x%2 == 0 {
       fmt.Printf("%d is even\n", x)
  } else {
       fmt.Printf("%d is odd\n", x)
  }
}
```

```
[Module03]$ go run ex03-03.go
22 is even
```

3.2.1 Local Variables in Conditionals

We can do the same in Go as in other C-style languages and define local variables in each of the blocks that make up the clauses of the if statement – but doing it this way means that variable is defined only for that block and so has to be redefined for each block that corresponds to a clause in the conditional.

However Go allows the definition of local variables at the start of the if statement which are then local to *all* of the clauses of the if statement. Since only one of the clauses will ever execute (the whole point of a conditional statement) then any interaction between these local variables across the clauses is impossible.

We can only use the short form of variable declarations here. Earlier in the class we said that we would see a reason for having the two forms of variable declaration – well here it is.

```
// Example 03-04 Local Variables

package main

import "fmt"

func main() {

  if x, y := 22, "hi"; x == 0 {
       fmt.Println("Value of x=", x, " y=", y)
  } else if x % 2 == 0 {
       fmt.Println("Value of x=", x, " y=", y)
  } else {
       fmt.Println("Value of x=", x, " y=", y)
  }
}
```

```
[Module03]$ go run ex03-04.go
Value of x= 22  y= hi
```

3.2.2 Using Non-local variables

We can also assign values to existing variables using the “=” operator instead of the “:=” operator.

However we cannot mix the local and non-local or the local definition will shadow the the non-local definition. This will be explored as one of the lab exercises.

```
// Example 03-05 Local and Non-Local Variables

package main

import "fmt"

var x int = 10

func main() {

  if x, y := 22, "hi"; x == 0 {
       fmt.Println("Value of x=", x, " y=", y)
  } else if x % 2 == 0 {
       fmt.Println("Value of x=", x, " y=", y)
  } else {
       fmt.Println("Value of x=", x, " y=", y)
  }
  fmt.Println("Value of x=", x)
}
```

```
[Module03]$ go run ex03-05.go
Value of x= 22  y= hi
Value of x= 10
```

3.4 Loops – Differences in Go

The only loop construct in Go is the for loop. The major differences between Go for loops and other C-style language are similar to those for conditionals.

1. No parentheses "(..)" allowed in the for clause
2. Braces "{..}" are mandatory for the loop body
3. The pre and post terms in the for clause can be empty

The last point means that a for loop can look like:

```
for  ; text==true ; {...}
```

which can be also written as if it were a while loop

```
for text==true { ...}
```

Like the if statement in Go, the opening brace "{" of the loop body must appear on the same line as the for clause.

Also notice in the example we cannot define "total" in the for clause because that would make it local to the loop body.

```go
// Example 03-06 Basic for loop

package main

import "fmt"

func main() {
  var total = 0
  for count := 0; count < 100; count++ {
       total += count
  }
  fmt.Println("total = ", total)
}
```

```
[Module03]$ go run ex03-06.go
total =  4950
```

3.4.1 Multiple Loop Variables

Example 03-07 shows how we can define multiple loop variables in the for loop. The example also illustrates the use of the break statement – the break and continue statements work the same way in Go as they do other C-style languages.

```
// Example 03-07 Multiple Declarations

package main

import "fmt"

func main() {
  var total = 0
  for i, m := 0, "abort"; i < 100; i++ {
       total += i
       if total > 100 {
            fmt.Println(m)
            break
       }
  }
  fmt.Println("total = ", total)
}
```

```
[Module03]$ go run ex03-07.go
abort
total =  105
```

3.4.2 Non-local Variables

Just a reworking of the previous example to show how non-local variables are used in the loop. The dangers of mixing the two types – local and non-local – are explored in the lab. The example also shows the use of the continue statement, used the same as in other C-style languages.

```
// Example 03-08 Non Local Variables

package main

import "fmt"

func main() {
  var total, i int = 1000, 1000
  for i, total = 0, 0; i < 100; i++ {
       if total > 200 {
            continue
       } else {
            total += i
       }
  }
  fmt.Println("total = ", total)
}
```

```
[Module03]$ go run ex03-08.go
total =  210
```

3.4.3 For Loop as a While Loop

A while loop in C-style languages is equivalent to a for loop with an empty clause usually written and "for(;;)" In Go we can do that as well, and by dropping off the extraneous semicolons we get something that looks just like a while loop.

```
// Example 03-09 For loop as a while loop

package main

import "fmt"

func main() {

  count := 0

  for count < 2 {
       count++
  }

  fmt.Println("count = ", count)
}
```

```
[Module03]$ go run ex03-09.go
count =  2
```

3.4.4 Looping Using Ranges

This loop form is functionally equivalent to a for-each loop. In the example, we are using the range function, which we will see a lot in this course, to iterate over a string.

The example also demonstrates the multiple return value feature of Go functions. During each iteration, the range function returns both the current index as an integer and the current contents at that index, which in this example is the letter at that position.

```
// Example 03-10 Looping with range

package main

import "fmt"

func main() {

  test := "Hi!"

  for index, letter := range test {
       fmt.Printf("Letter %d is %#U\n", index, letter)
  }
}
```

```
[Module03]$ go run ex03-10.go
Letter 0 is U+0048 'H'
Letter 1 is U+0069 'i'
Letter 2 is U+0021 '!'
```

3.5 Switch Statements

One small change which I think makes the Go version of the switch statement cleaner than in other languages is revising the default behavior of the case construct so that the flow of control breaks by default at the end of the case rather than falling through to the next case. In Go, the fallthrough statement is now required to fall through to the next case rather than needing to use the break statement to prevent a fall through. The break statement is still available but now it is used only when you want to break prematurely out of a case instead of executing the whole case.

In C-style languages, a test value must be provided and it usually must be some sort of integral type. In Go both of these requirements are dropped which, again in my opinion, make the switch in Go more powerful. One of the more useful roles of the switch statement that we will explore in a later module is the ability to switch on the basis of the type of variable.

In the example things to note are the lack of break statements and the use of the fallthrough statement to get the default clause to execute right after the '1' case when the switch test value is 1.

```
// Example 03-11 Simple Switch

package main

import "fmt"

func main() {

  for i := 0; i < 3; i++ {
       switch i {
       case 0:
            fmt.Println("Case 0")
       case 1:
            fmt.Println("Case 1")
            fallthrough
       default:
            fmt.Println("Default")
       }
  }
}
```

```
[Module03]$ go run ex03-11.go
Case 0
Case 1
Default
Default
```

3.5.1 Break Statements

A sort of contrived example to show the use of the break statement.

```
// Example 03-12 Switch Statement Break

package main

import "fmt"

func main() {

  for i := 0; i < 2; i++ {
       switch i {
       case 0:
            fmt.Println("Case 0")
       case 1:
            fmt.Println("Case 1")
            break
            fmt.Println("After break")
       default:
            fmt.Println("Default")
       }
  }
}
```

```
[Module03]$ go run ex03-12.go
Case 0
Case 1
```

3.5.2 Non-integral Test Value

Simple example where the test value is a string rather than a numeric value. Obviously the restriction is that whatever type the switch test value is, it must be something that has the notion of equality defined. In this example using more than one matching value for a test case, a feature of other C-style switch statements, is also demonstrated.

```
// Example 03-13 Non-integral test

package main

import "fmt"

func main() {

  os := "fedora"
  switch os {
  case "fedora", "redhat":
       fmt.Println("Open Source")
  case "Windows":
       fmt.Println("Proprietary")
  default:
       fmt.Println("unknown")
  }
}
```

```
[Module03]$ go run ex03-12.go
Open Source
```

3.5.3 Switch – No Test Value

In this variant, the cases are evaluated as usual from start to finish. However each test case now has to be a predicate (ie. an expression that evaluates to true or false). The first test case that evaluates to true is the case that is executed.

The developers of Go note that the switch statement in this form is equivalent to a series of else-if statements. Aside from the fact it might be an easier way to write complex switch logic, this would seem to be useful when the test value is not a variable but a condition or combination of conditions that cannot be represented as a single variable of a specific type.

```
// Example 03-14 Switch with no test value

package main

import "fmt"

func main() {
  x := 22

  switch {
  case x == 0:
       fmt.Println("zero")
  case x % 2 == 0:
       fmt.Println("even")
  default:
       fmt.Println("odd")
  }
}
```

```
[Module03]$ go run ex03-14.go
a=int b=int c=int
a=0 b=2 c=4
```

Module Four

Function Basics

I have reimplemented a networking project from Scala to Go. Scala code is 6000 lines. Go is about 3000. Even though Go does not have the power of abbreviation, the flexible type system seems to out-run Scala when the programs start getting longer. Hence, Go produces much shorter code asymptotically.
Petar Maymounko

Go isn't functional, it's pragmatical. Why pure paradigms like FP or OOP are always a must? (sigh)
Frank Mueller

4.1 Introduction

Functions are one of the fundamental building blocks of any programming language whether they are defined as C style procedures, or methods that belong to a class as in Java, or as program objects like in functional programming languages like Scala.

The initial syntax of functions in Go can easily mislead you into thinking that as far as functions are concerned, Go is just a 21st century rewrite of how C "does functions" – which it most definitely is not. Go is not a programming language that implements a particular programming paradigm like Ruby implements object oriented programming or Scala implements functional programming. Instead it has taken ideas from a variety of programming models and created a very powerful implementation of the function concept.

This is the first of four modules on functions. The focus of this module is on the basics of Go functions with an emphasis on how Go functions are different that those in other C-style programming languages.

Module seven on advanced functions looks at functions as first class objects, which is programming jargon that means we can do almost anything with a function that we can do with a variable. We also look at using anonymous functions in Go, which is the way that Go implements lambda functions, as well as examining function literals and closures.

Module nine introduces Go methods which are a mechanism for associating Go functions with user defined types to deliver the same sort of capabilities we find in object oriented languages.

Module eleven introduces one of the more unique features of Go functions which is using functions as concurrent goroutines. This is Go's very powerful and elegant model for implementing concurrency.

4.2 Function Definitions

Functions in Go are defined using the following syntax:

```
func fname(var1 type1,..varn typen) type { ...}
```

However functions in Go can also have named return values, in which case the syntax looks like this

```
func fname(var1 type1,..varn typen) (var type) { ... }
```

Named return values are something that other C-style languages do not have. A named return value is a local variable defined in the function declaration which can be used like any other local variable but which is returned when the function is finished executing.

The func keyword is mandatory for a couple of reasons. The first is that it conforms to the rule that all package level constructs begin with an identifying token to improve the efficiency of parsing of Go code. The other reason is that as we get into more advanced use of functions, it will be critical to know if a chunk of code is just an ordinary chunk of code or if it is in fact a function literal. We will return to that topic in module seven.

The order of the parameters and types, and the return values and their types is the same as for Go variable declarations, which is reversed from most of the other languages. This takes a bit of getting used to in you are a long time programmer in C or Java.

Go, like most other C-style languages, passes parameters and returns by value, which means that the function is working with a local copy of the arguments that were passed by the calling function. I am assuming that since you are experienced programmers, that the concepts of call by value and call by reference are quite familiar to you so we won't go into the mechanics of it in this material.

4.2.1 Basic function definition

Example 4.1 shows the most C-style function definition we can have in Go. If you are a C-style programmer, aside from the order of the parameters and types, this looks like pretty standard code.

Example 4.2 demonstrates the same function but using a named return value. A couple of points to note:

1. If a named return value is used, then the parentheses around the return values are not optional. Similarly, if multiple returns are used, the parentheses around the list of return values are not optional.
2. When the function is called, the named return variables are created and initialized to their zero values. After that, they are treated just like any other local variable in the function body.

```go
// Example 04-01 Basic function sytnax

package main

import "fmt"

func square(x int) int {
  y := x * x
  return y

}
func main() {
  x := 4
  retval := square(x)
  fmt.Println("x = ", x, "square = ", retval)
}
```

```
[Module04]$ go run ex04-01.go
x =  4 square =  16
```

```go
// Example 04-02 Named return value

package main

import "fmt"

func square(x int) (result int) {
  result = x * x
  return
}
func main() {
  x := 4
  retval := square(x)
  fmt.Println("x = ", x, "square = ", retval)
}
```

```
[Module04]$ go run ex04-02.go
x =  4 square =  16
```

3. When using named return values, we don't need to specify what is returned and can use a naked return statement (just the word "return" alone) since Go knows which variables to return.
4. Mixing the two styles is not allowed, you either use a naked return with named return values or return the value explicitly like in C or Java.

Just for completeness, example 04-03 shows an alternate way of writing a parameter list which is in keeping with the Go way of declaring variables.

```
// Example 04-03 Multiple parameters

package main

import "fmt"

func divides(x, y int) (div bool) {
  div = (y % x) == 0
  return true
}
func main() {
  x, y := 2, 21
  fmt.Println(x, "|", y, " is ", divides(x, y))
}
```

```
[Module04]$ go run ex04-03.go
2 | 21  is  true
```

4.2.2 Multiple Return Values

One of the more innovative features of Go is that functions can return multiple values. We can emulate this other languages by bundling up a collection of values into some return object, but we still wind up returning a single object. One of the reasons Go allows multiple returns is to accommodate the error handling mechanism it uses instead of exceptions. We will look at errors a bit later in this module.

The syntax for returning multiple values is a logical extension of the single return value case is shown below. Multiple return values can be named or unnamed but may not be a mixture of the two.

```
func fname(v1 t1...) (type1, type2, ..) { ... }
func fname(v1 t1...) (var1 type1, var2 type2) { ... }
```

Example 04-04 shows the use of multiple unnamed return values.

```
// Example 04-04 Multiple return values

package main

import "fmt"

func divides(x, y int) (int, int) {
  return (y / x), (y % x)
}
func main() {
  x, y := 3, 23
  quot, rem := divides(x, y)
  fmt.Println(x, "/", y, " is ", quot, "R", rem)
}
```

```
[Module04]$ go run ex04-04.go
3 / 23  is  7 R 2
```

Example 04-05 shows the use of multiple named return values.

```
// Example 04-05 Multiple named return values

package main

import "fmt"

func divides(x, y int) (q int, r int) {
  q = y / x
  r = y % x
  return
}
func main() {
  x, y := 3, 23
  quot, rem := divides(x, y)
  fmt.Println(x, "/", y, " is ", quot, "R", rem)
}
```

```
[Module04]$ go run ex04-05.go
3 / 23  is  7 R 2
```

There are some basic rules about multiple return values:

1. The return values either all named or all unnamed – you cannot mix the two types.
2. Unnamed return values have to be returned in a list following the keyword "return."
3. The named return value form uses a naked return. You don't have to tell Go what to return, when it encounters a return, it will return the current values of the return variables at the moment the return is executed.
4. The function passes the return values to the calling function in the same order they listed in the function declaration.
5. It's all or nothing, all of the values have to be returned, you can't return some of them. Notice though for named return values this is automatic.

4.3 Deferred Execution

This concept illustrates the first of several Go operators that change the behavior of a function when it is called, sort of an operator on a function. What Go does with the defer operator is allow you to call a function and have it run later rather than at the time you called actually called it.

We defer a function's execution by using the word “defer” in front of it when we call it. This has the effect of waiting until the calling function is about to exit before actually running the deferred function. The result is that we are calling the function, evaluating the arguments to the call, then waiting until the calling function executes before running the deferred function.

There is an important aspect about deferring a function which makes them very useful – the arguments to the deferred function are evaluated when the function is called, not when it is executed.

Example 4.6 shows that the deferred function f() clearly runs after the main is finished executing and just before the main function exits because the message printed by f() occurs after the message printed by main.

Also notice that the value of the message is what the value of “m” was when f() was called, not when f started to execute.

```
 // Example 04-06 Deferred execution

package main

import "fmt"

func f(message string) {
  fmt.Println("Value of param =", message)
  return
}
func main() {
  m := "before defer"
  defer f(m)
  m = "after defer"
  fmt.Println("Value of m when main() exits =", m)
}
```

```
[Module04]$ go run ex04-06.go
Value of m when main() exits = after defer
Value of param = before defer
```

A deferred function is almost like a closure in the sense that it is closed over its arguments when it is called. If you don't know what a closure is, don't worry, we will be seeing them in module seven.

Two of the uses of defer are to recover from panics, which we will look at later, and to do some sort of cleanup before a function exits. For example, suppose we have a program that opens some sort of connection or has a lock on some resource; if the code that closes the connection is put into a deferred function, then no matter when the original function executes a return, the deferred function will execute and the cleanup will be done.

4.3.1 Stacked Defers

We can defer as many functions calls as we want. Each time we make a deferred call, the function arguments and evaluated and the function put into a stack of deferred function calls. When the calling function exits, the deferred functions are then executed in a first in last out order (like a stack).

```
// Example 04-07 Stacked defer

package main

import "fmt"

func f(k int) {
  fmt.Printf("executed f(%d)\n", k)
  return
}

func main() {
  for i := 1; i < 4; i++ {
       fmt.Printf("called f(%d)\n", i)
       defer f(i)
  }
  fmt.Println("end main")
}
```

```
[Module04]$ go run ex04-07.go
called f(1)
called f(2)
called f(3)
end main
executed f(3)
executed f(2)
executed f(1)
```

Example 04-07 demonstrates a stacked defer. Notice that each function call remembers the value of “i” when it was called, not when it executed. Also notice that the function calls execute in the reverse order they were called.

4.4 Recursion

Go supports recursion in the same way most programming languages do. I am assuming that you know what recursion is and how it works so we will not be going into the mechanics of recursion in this class. Example 04-08 just demonstrates what recursion looks like in Go. The function sum() recursively adds up the a series of integers from 1 to k for a given k.

```
// Example 04-08 Recursion

package main

import "fmt"

// sums numbers from 1 - k

func sum(k int) int {
  fmt.Printf("executed sum(%d)\n", k)
  if k > 0 {
       return (k + sum(k-1))
  } else {
       return 0
  }
}

func main() {
  fmt.Println(sum(3))
}
```

```
[Module04]$ go run ex04-08.go
executed sum(3)
executed sum(2)
executed sum(1)
executed sum(0)
6
```

4.5 Varadic Functions

A Varadic function is one that can be called with a varadic parameter which is one that can be passed a variable number of arguments by the calling function. Varadic functions are incredibly useful – for example the fmt.Println() function is varadic since we can pass it a variable number of arguments when we call it.

Go does not allow optional arguments or keyword named parameters like some other programming languages, and the rules for varadic functions tend to be a little bit on the strict side.

The format for a varadic function is:

```
func fname (p1 t1, v ... type)
```

for example:

```
func v(message string, x ... int)
```

In the example above, message is a non varadic parameter since there is only one, but x is a varadic parameter so there may be any number of ints passed as arguments to x. All of the following would be valid calls to v()

```
v(“hi”)
v(“hi,1)
v(“hi”,1,2,3,4)
```

Varadic functions must obey the following two rules:

1. There may be only one varadic parameter although there may be other non-varadic parameters. The varadic parameter must be the last parameter in the parameter list.
2. Ellipsis “...” is used to identify the parameter as varadic.

4.5.1 Processing varadic parameters

In the next module we will be looking at arrays in Go but for now all you have to know is what an array is, which you is something that should be quite familiar to you, and also from the lab in the last module.

The arguments passed to the varadic parameter are made available as an array of the parameter type.

This is demonstrated in example 04-09.

```go
// Example 04-09 Varadic functions

package main

import "fmt"

func addup(nums ...int) (sum int) {
  for _, val := range nums {
       sum += val
  }
  return
}

func main() {
  fmt.Println(addup(1, 2, 3, 4, 5))
}
```

```
[Module04]$ go run ex04-09.go
15
```

4.6 Error Handling

Almost everyone who programs in an object oriented language uses exceptions. Go does not use exceptions for a variety or reasons.

First, is the fact that exceptions are cumbersome to compile and produce a lot of code bloat in binaries which has an impact on build time. In fact, Bjarne Stroustrup recommends that if we are looking to improve the performance of C++, then not using exceptions makes the resulting code smaller and faster by a significant amount, but at the cost of being somewhat riskier.

Second, throwing exceptions that might be caught literally anywhere, or even worse they might never be caught, can produce problems in managing large code bases. While exceptions have some nice properties for certain kinds of designs, the developers of Go felt that it was a effective optimization to not have them in the language for the kinds of large scale code bases Go is intended to be used with.

Third, exceptions are often used to get around the issue of how we handle error conditions. The usual explanation is that because we were confined to looking at a single return value, trying to determine if the value returned is an error code or valid value, the code in the calling function can become very complex. This is very true unless we can return multiple values, and because Go supports multiple return values we can return an additional error object that we can test locally to see whether or not a function call succeeded. We do want to avoid a discussion on which is better theoretically – exceptions or the Go error system – but there are certainly good arguments made by the language developers for this way of handling errors.

Not every function needs to deal with errors. The only time that we normally use a Go error is when there is a possibility that a function may fail for the same types of conditions that would raise an exception in a language like Java. For example, a simple computation would probably never need an error but a read from a file would because it is possible that a number of exceptional conditions could occur that would prevent the read from happening.

Some basics about using errors:

1. The error is always the last return value in the return list.
2. The error is created using the New operator – this construct will be covered a later module so we defer the discussion of what that one line of code means until then.
3. A nil can be returned as the error if no error occurs. We discuss nil in a future module but for now just think of this as being like what nil means in other C-Style languages, or a null pointer.
4. If the error is nil, then we can proceed with execution normally
5. If the error is non-nil, we should ignore the other results returned because they are now suspect, although in some cases we may be able to fix things up.

```go
// Example 04-10 Error Handling

package main

import "fmt"
import "errors"

func division(num, denom int) (int, error) {
  if denom == 0 {
       return 0, errors.New("Divide by zero")
  }
  return (num / denom), nil
}

func main() {
  res, e := division(56, 0)
  if e != nil {
       fmt.Println(e)
  } else {
       fmt.Println(res)
  }
}
```

```
[Module04]$ go run ex04-10.go
Divide by zero
```

6. The error is printable. We will look more into the error construct in a later module, but for now just note that there is a string output we can get by printing the error.

Example 4-11 and 4-10 show variations on the handling of an error where we don't care about what error occurred but are testing to see if we should proceed with processing or should abort processing because of an error.

```go
// Example 04-11 Error Handling

package main

import "fmt"
import "errors"

func division(num, denom int) (int, error) {
  if denom == 0 {
       return 0, errors.New("Divide by zero")
  }
  return (num / denom), nil
}

func main() {
  res, e := division(56, 2)
  if e != nil {
       fmt.Println(e)
  } else {
       fmt.Println(res)
  }
}
```

```
[Module04]$ go run ex04-11.go
28
```

4.6.1 Comma ok Idiom

In some cases, such as in example 4-11, all we want to know is whether an error occurred or not. In this case we often return a bool instead of an error object to provide a test value which by convention we assign the to the variable "ok". Generally we use a bool when all we care about is if an error occurred or not, but we really don't care what kind of error nor are we interested in any details about the error.

We will see this idiom later when we look at maps and in other places where we just want to see if some data value exists or some processing took place. We tend to use this idiom when the condition is not so much an error as some condition that could happen that we have to respond to or take into account.

For example, we may want to delete a file but the file might not be there. Is it really an error if the file is not there or just a possibility we have to handle? In this case the ok variable may just tell us whether or not the file is there so that we know if we should proceed with the delete operation.

```
// Example 04-12 Comma OK Idiom

package main

import "fmt"

func division(num, denom int) (int, bool) {
  if denom == 0 {
       return 0, false
  }
  return (num / denom), true
}

func main() {
  res, ok := division(56, 2)
  if ok {
       fmt.Println(res)
  } else {
       fmt.Println("division failed")
  }
}
```

```
[Module04]$ go run ex04-12.go
28
```

4.7 Panics and Recoveries

Those who are used to UNIX environments will know what a panic is, but if you don't, then think of a panic as being like a run time error or run time exception that is thrown in Java.

Generally, the best practice if a panic occurs is to terminate the application gracefully.

However is some cases we may be able to recover from the panic. It is advisable to only recover from a panic into some fail-safe state where the application can clean up its environment and then perform a controlled shutdown. The problem is that we can get sloppy and use panics to deal with situations that are not really serious enough to be panics but should be handled instead with the use of errors in the program logic. This is the same sort of trap that OO programmers fall into with using exceptions for things that are not really exceptional.

Summarizing panics and recoveries

1. A panic is generated by Go when a runtime error occurs
2. During a panic, all deferred functions are called, then execution halts
3. Panics can be generated by calling the panic() function
4. Panics can be recovered from by executing the recover() function
5. The recover() function only works in deferred function
6. Panics should be used only for critical problems, use errors otherwise

```
// Example 04-13 Runtime panic

package main

import "fmt"

func main() {
  x, y := 1, 0
  fmt.Println(x / y)
}
```

```
[Module04]$ go run ex04-13.go
panic: runtime error: integer divide by zero
[signal 0x8 code=0x1 addr=0x40102c pc=0x40102c]
```

```go
// Example 04-14 Runtime panic

package main

import "fmt"

func division(num, denom int) int {
  if denom == 0 {
       panic("Dividing by zero?!?")
  }
  return (num / denom)
}

func main() {
  res := division(56, 0)
  fmt.Println(res)
}
```

```
[Module04]$ go run ex04-14.go
panic: Dividing by zero?!?
goroutine 1 [running]:
panic(0x4b8e00, 0xc82000a3c0)
```

Example 4-15 shows how we can generate a panic by a call to the panic() function. Generally this should be one of those programming strategies of last resort, meaning the situation you have encountered cannot be reasonably resolved and some sort of system shutdown is necessary. I wouldn't recommend it to be used the way it is in the example.

Example 4-16 shows the use of the recover() function to try and recover from a panic.

Since a panic causes an immediate exit from the function where the panic occurred, any recovery code has to go into a deferred function that has already been called. No more function calls can take place after a panic so only deferred functions can execute.

If there is no panic, executing the recover() function does nothing, there are no effects from its execution and it returns a nil. However, if a panic has occurred, then recover() prevents the program from shutting down and returns the panic item that caused the panic so that any actual recovery code can be executed. In this example, the recovery code just prints out a message before exiting.

The best practical use of recovery is to move the system in to a fail-safe state before shutting down so that we do not leave the environment or other connected resources in an unstable or invalid state.

```
// Example 04-15 Runtime panic and recovery

package main

import "fmt"

func rec() {
  r := recover()
  if r != nil {
       fmt.Println("recovered value = ", r)
  }
}
func main() {
  x, y := 1, 0
  defer rec()
  fmt.Println(x / y)
}
```

```
[Module04]$ go run ex04-15.go
recovered value =  runtime error: integer divide by zero
```

Module Five

Arrays and Slices

Go is like a very delicious trifle - the further into it you go, the more delicious things you find. The quality is clear throughout. Go is very unassuming. You start by wondering what the big deal is + getting annoyed with the minor differences from other C-syntax languages before slowly progressing towards quite liking it, then eventually once you grok how simple + elegant and well engineered it is you come to love it.

Lorenzo Stoakes

A fool with a tool is still a fool.

Martin Fowler

5.1 Introduction

Arrays in Go work much like arrays do conceptually in other C-style programming languages and use the same syntax that most C-style programmers will recognize and are comfortable with. However Go arrays are quite different how they are implemented which has the dual effect of making them safer and more efficient to use from an implementation perspective, but at the cost of reduced functionality and convenience from the programming perspective.

Arrays in most C-style programming language are pointers or references to blocks of memory. Arrays can be passed efficiently in function argument lists – which we call pass by reference or pass by pointer – and can be used in other ways that are quite elegant because they are pointers dressed up with array notation.

However this elegance comes with a cost in terms of the behind the scenes overhead like memory management (every programmer has spent at least on all nighter looking at code muttering “What null pointer? Why does the program say there is a null pointer?”), memory leaks, problems with defining equality, shallow versus deep copies, and program safe code issues (C is quite happy to modify the 10th element of a four element array for example).

What Go has done is split the normal C-style concept of an array into two parts. A safe value type (that means it is managed in memory like an int or a float) which Go calls the array type, and a second part called a slice which is a pointer that can reference parts of the array in a way that C-style programmers are used to.

By using both arrays and slices, we get the advantages of arrays that we expect from other C-style languages but we mitigate the risks of using treating an array as a pointer to a chunk of heap memory.

5.2 Arrays

Arrays in Go are *value* data types. This means that they are managed the same way in memory as an int or float. This is different than most of the other C-style languages where an array is usually a *reference* data type, which means that arrays are implemented as pointers or references to chunks of memory that have to be allocated in the code.

An array in Go is a sequence of elements where:

1. Each element is of the same type, including user defined types (which will be covered in module eight).
2. The length of the array is fixed.

The size of an array, which can be referenced with the function len(array) is part of the array type. So and [5]int and [6]int array are of different types because they are of different lengths.

The value that specifies the length of the array must either be a constant or the result of a constant expression. Remember that a constant expression is one that can be evaluated at compile time. Of course it goes without saying that the size must also be an integer and non-negative.

The length of an array is immutable, there is no way to grow or shrink the size of an array once it has been created.

5.2.1. Declaring Arrays

Arrays are declared with the following syntax

```
var name [size]type
var ar   [3]int
```

The above line declares a length 3 array of ints referenced by the variable ar. When arrays are declared without an explicit initialization, the array elements are all initialized to their zero values.

5.2.2 Accessing arrays

Arrays are accessed using the standard array [..] notation common to most C-style languages

Like Java, Go checks for array indexes that are out of bounds. Because this cannot be checked for at compile time, if an array index is out of bounds a runtime panic is generated when the illegal access occurs, just like Java.

```go
// Example 05-01 Basic array syntax

package main

import "fmt"

func main() {
  var a [3]int
  fmt.Println("a =", a)
  a[0] = 1
  a[1] = a[0] + 1
  fmt.Println("a =", a)
}
```

```
[Module05]$ go run ex05-01.go
a = [0 0 0]
a = [1 2 0]
```

5.2.3 Initializing arrays

Arrays can be initialized using what are called array literals. These are of the form

```
var arrayname = [size]type{list of element literals}
arrayname := [size]type{list of element literals}
```

Arrays can declared using the size [...] when initializing with literals. In this case the compiler will count the number of literals and insert the correct size.

It is possible to also initialize some elements of the array and let the others take their zero values. In case below the 3rd and 4th elements are initialized explicitly while the others are initialized to their zero values (remember we count from 0 for indices).

```
var ar = [5]int{3: 3, 4: 4}
```

or

```
var ar = [...]int{3: 3, 4: 4}
```

In the last case, in order to be able to count the right number of literals for initialization, the last element of the array must be explicitly initialized. Alternatively we could say that the last explict intitalization determines the length of the array.

```go
// Example 05-02 Explicit array initialization

package main

import "fmt"

func main() {
  var ar1 = [5]int{0, 1, 2, 3, 4}
  var ar2 = [...]string{"Hello", "World"}
  ar3 := [2]bool{true, false}
  ar4 := [...]int{3: -1, 4: -1}
  fmt.Println("ar1=", ar1, "length=", len(ar1))
  fmt.Println("ar2=", ar2, "length=", len(ar2))
  fmt.Println("ar3=", ar3, "length=", len(ar3))
  fmt.Println("ar4=", ar4, "length=", len(ar4))
}
```

```
[Module05]$ go run ex05-02.go
ar1= [0 1 2 3 4] length= 5
ar2= [Hello World] length= 2
ar3= [true false] length= 2
ar4= [0 0 0 -1 -1] length= 5
```

5.2.4 Array Operations

Comparison

Since array are values, they can be compared using the comparison operators == and != . However in order to be comparable, they must both have elements of the same type and be the same size.

Equality is defined to mean that the two arrays are equivalent if they are of the same type and that the corresponding elements in the two arrays are equivalent. Other operators like < are not defined because there is no meaningful or useful interpretation of those sorts of comparisons for arrays.

```
// Example 05-03 Array comparison

package main

import "fmt"

func main() {
  var ar1 = [5]int{0, 1, 2, 3, 4}
  var ar2 = [5]int{0, 1, 2, 3, 4}
  fmt.Println("ar1 == ar2 is ", ar1 == ar2)
  fmt.Println("ar1 != ar2 is ", ar1 != ar2)

}
```

```
[Module05]$ go run ex05-03.go
ar1 == ar2 is  true
ar1 != ar2 is  false
```

Arguments to Functions

Arrays can be passed as arguments to functions. However because functions pass by value, a copy of the array is used in the called function. This means a function cannot change the contents of the original array passed as an argument from the calling function.

This is inconvenient and uses up a lot of memory making copies, especially if the array being passed is a large array. But this also avoids a lot of unwanted side effects because the original array cannot be affected by anything done to the passed copy.

We can go the route of creating a pointer to an array and then passing the pointer instead of the array, but this is considered unseemly in Go – that is a C idiom, not a Go idiom. In the next section, we will see that we can do the same thing with slices that we could with pointers, but using slices is much more the Go way to do this.

```
// Example 05-04 Array as parameter

package main

import "fmt"

func delta(prm [3]int) {
  prm[0] = -1
  fmt.Println("prm = ", prm)
}

func main() {
  var arg = [3]int{99, 98, 97}
  fmt.Println("arg = ", arg)
  delta(arg)
  fmt.Println("arg = ", arg)
}
```

```
[Module05]$ go run ex05-04.go
arg =  [99 98 97]
prm =  [-1 98 97]
arg =  [99 98 97]
```

Assignment

Since array are values, they can be compared using the comparison operators == and != . However in order to be comparable, they must both have elements of the same type and be the same size.

Equality is defined to mean that the two arrays are of the same type and that the corresponding elements in the two arrays are equivalent. Other operators like < are not defined because there is no meaningful or useful interpretation of those sorts of comparisons for arrays.

```
// Example 05-05 Array assignment

package main

import "fmt"

func main() {
  var ar1 = [3]int{99, 98, 97}
  var ar2 [3]int
  ar2 = ar1
  ar2[0] = 0
  fmt.Println("ar1 =", ar1)
  fmt.Println("ar2 =", ar2)
}
```

```
[Module05]$ go run ex05-05.go
ar1 = [99 98 97]
ar2 = [0 98 97]
```

Iterating over arrays

The range function was introduced in module two as a way to iterate over sequences. Just as would be expected, we can iterate over an array the same as shown in example 05-06 on the next page.

The range function returns two values for each iteration. The first is the current index of the position being processed. The second is the element in the array at that index. A common idiom is to use the blank variable “_” to receive the index when we are not interested in where in the array we are, we just want to iterate sequentially through the entire array and do something to each value in turn .

```go
// Example 05-06 Iteration with range

package main

import "fmt"

func main() {
  words := []string{"the", "best", "of", "times"}
  for index, value := range words {
       fmt.Println(index, " ", value)
  }
}
```

```
[Module05]$ go run ex05-06.go
0   the
1   best
2   of
3   times
```

5.2.5 Multidimensional Arrays

As mentioned before arrays are one dimensional, however they can be "layered" to form multi-dimensional arrays similar to how we build up multidimensional arrays in other C-style languages. The following snippets both define the same multi-dimensional array.

```
var matrix [2][3]int
var matrix [2]([3]int)
```

These can be read to mean that matrix is either a 2x3 array of ints, or it is an array of length 2 where each element of the array is itself an array of length 3 where each element of each of those arrays is an int. These two definitions are just two different ways of describing the same underlying structure. Either one is acceptable in Go and it is often a matter of choosing the one that makes the programmer intent clear in a given context. Higher dimensions work in a similar way.

This means that all multidimensional arrays must be square. In the example, matrix is a two dimensional array of one dimensional arrays of type [3]int. Remember that the size is part of the array type. If we wanted a ragged two dimensional array, we would have to have matrix be an array where the first element is an array of type [2]int and the second an array of say [3]int. But then the rule that all of the elements of an array must be of the same type would be violated, which means there is no way that we can make matrix into a ragged array.

```go
// Example 05-07 Multidimensional array

package main

import "fmt"

func main() {
  var matrix [2][3]int
  value := 10
  for row, col := range matrix {
       for index, _ := range col {
            matrix[row][index] = value
            value++
       }
  }
  fmt.Println("Matrix: ", matrix)
}
```

```
[Module05]$ go run ex05-07.go
Matrix:  [[10 11 12] [13 14 15]]
```

Example 05-07 above demonstrates iterating over a two dimensional array. A simple for loop with defined loop indices could have been used but in this example nested ranges are used to accomplish the same effect. Notice that we are not interested in the value returned by range in the inner for loop so we throw it away by using the blank variable "_"

5.2.6 Multidimensional Initialization

A multi-dimensional array can be initialized using literals in several different formats. In the first form in example 05-08, the syntax emphasizes the layers that the array is built up from.

```
// Example 05-08 Multidimensional initialization

package main

import "fmt"

func main() {
  var matrix = [4][4]int{
       [4]int{1, 2, 3, 4},
       [4]int{2, 4, 8, 16},
       [4]int{3, 9, 27, 81},
       [4]int{4, 16, 64, 256},
  }
  fmt.Println("Matrix: ", matrix)
}
```

```
[Module05]$ go run ex05-08.go
Matrix:  [[1 2 3 4] [2 4 8 16] [3 9 27 81] [4 16 64 256]]
```

This form helps void possible ambiguity errors. However we can also leave out the extra info when the context is unambiguous, which is which is demonstrated in example 05-09, which is logically equivalent to the 05-08. Both initialize the array and one might be preferred over the other by some because they might feel that it depicts more clearly exactly what the conceptual structure of the array is.

```go
// Example 05-09 Multidimensional initialization

package main

import "fmt"

func main() {
  var matrix = [4][4]int{
       {1, 2, 3, 4},
       {2, 4, 8, 16},
       {3, 9, 27, 81},
       {4, 16, 64, 256},
  }
  fmt.Println("Matrix: ", matrix)
}
```

```
[Module05]$ go run ex05-08.go
Matrix:  [[1 2 3 4] [2 4 8 16] [3 9 27 81] [4 16 64 256]]
```

Because this code gets really dense fast, either of these are probably not a good way to initialize higher dimensional or large multi-dimensional arrays. It would not take long before increases in size or dimension would produce an unreadable wall of text.

5.3 Slices

Slices are references to contiguous segments of an underlying array. A slice can be thought of as consisting of a pointer to a starting position somewhere in the array and a count of the number of elements from the start which make up the rest of the slice. From a programming point of view, slices look and behave syntactically just like arrays.

For C programmers, slices can be thought of as something analogous to how pointers into an array work. Given a chunk of memory, we can define multiple pointers into that chunk of memory where each pointer defines an overlay that slices up the chunk ol memory into logical arrays starting at different points.

Slices can be created from existing arrays by taking a sub-sequence of the array as illustrated in the 05-10. Trying to take sub-sequence where the ranges would be outside the bounds of the underlying array is an error.

```
// Example 05-10 Basic Slice

package main

import "fmt"

func main() {

  var a = [6]int{0, 1, 2, 3, 4, 5}

  s := a[2:] // s is a slice of the array a
  fmt.Println("s= ", s)

  a[4] = -20 // changing underlying array
  fmt.Println("s= ", s)

  s[0] = 999 // change the array via the slice
  fmt.Println("a= ", a)
}
```

```
[Module05]$ go run ex05-10.go
s=  [2 3 4 5]
s=  [2 3 -20 5]
a=  [0 1 999 3 -20 5]
```

5.3.1 Length and Capacity

Each slice has two values associated with it.

1. *Length*: Just like an array, the length of a slice is the number of elements contained in the slice. This value is dynamic and may change during execution. The minimum length of a slice is 0 and the maximum size of a slice is the length of the underlying array.
2. *Capacity*: The capacity of a slice is how long the slice can become. Capacity is calculated by adding the current length of the slice and the number of elements from the end of the slice to the end of the underlying array.

Trying to make a slice larger than its capacity is an error when the slice is taken from an existing array. For example

```
a := [...]int{1,2,3}
s := a[0:8]
```

will be an error since have exceeded the slice capacity.

5.3.2 Working with elements in a Slice.

Slices do not contain their own data. That means that multiple slices can overlay or reference the same element in the underlying array, and this can lead to problems if we are not careful. Syntactically, we access elements of a slice exactly like we access elements in an array.

5.3.3 Slices of slices

We can also take slices of other slices, not just of underlying arrays. The code in example 05-11 illustrates this and how the slices can overlap. Notice that the third slice s2 seems to be defined with more elements that exist in what it is a slice of, namely s1. But the elements of a1 are not really "in" the s1 slice but are in the underlying array, taking a slice of a slice is just another way of taking a slice of the underlying array.

In example 05-12, two overlapping slices s0 and s1 are defined on the same underlying array. Changing an element in one slice is actually a change to the element in the underlying array which then changes the value of the second slice.

```
// Example 05-11 Slices

package main

import "fmt"

func main() {

  var a = [...]int{0, 1, 2, 3, 4, 5}

  s0 := a[1:4]
  s1 := s0[1:3]
  s2 := s1[0:4] // ???

  fmt.Println("s0 length=", len(s0), " s1 length=", len(s1),
       " s2 length=", len(s2))
  fmt.Println("s0=", s0, " s1=", s1, " s2=", s2)
}
```

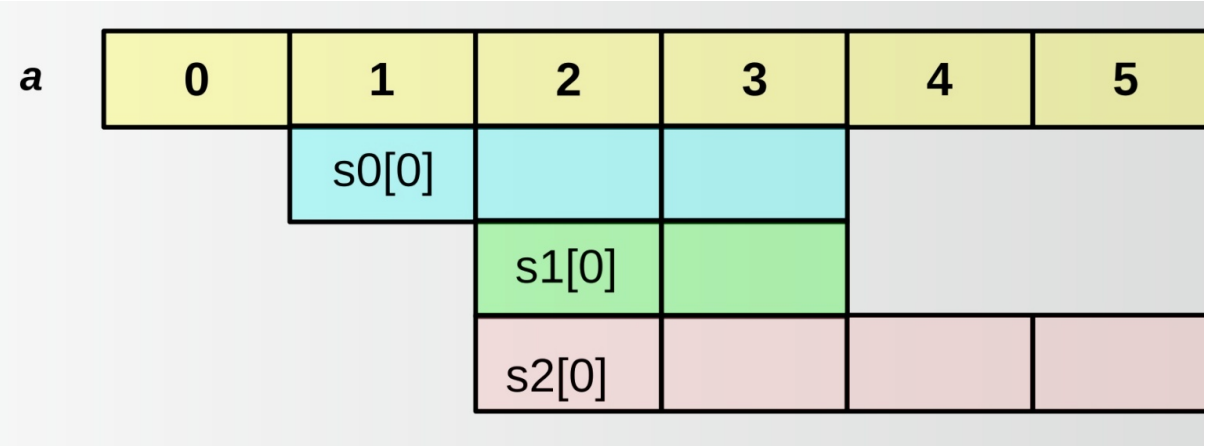
```
[Module05]$ go run ex05-11.go
s0 length= 3  s1 length= 2  s2 length= 4
s0= [1 2 3]  s1= [2 3]  s2= [2 3 4 5]
```

The actual situation in the above code is depicted graphically below.

Remember that we are still working with the underlying array "a". This means that the slices are like sliding windows on top of the array. When we said

```
s2 := s1[0:4]
```

what we were really saying was: "give me four elements of the underlying array starting at the position pointed to by slice s1"



5.4 Creating Slices Directly

So far we have looked at created slices from an existing array. However, since we tend to use slices more in Go than arrays, we can also create a slice directly in one of two ways.

5.4.1 Slice initialization

Slices can be initialized with an array literal using the following syntax

```
s1 := []int{1,2,3}
```

which looks very similar to the array initialization syntax except for the missing ellipsis between the square brackets. Since slices themselves cannot hold data because they are pointers into an underlying array, Go creates an array large enough to hold the initializing literal and then assigns s1 the pointer to that newly created array.

This underlying array is anonymous, there is no way to access it without going through the slice so for all intents and purposes, we can think of the slice as being the actual object itself which, while not technically accurate, is a useful conceptualization.

5.4.2 Using Make

We can also create a slice directly without having to initialize it explicitly by using the make() function like this

```
slice := make([]type,length)
```

The make() function does essentially the same thing as creating a slice using an initialization. An underlying anonymous array is created of [length]type and a reference to this array is returned and assigned to the slice variable.

The underlying array, just like in the case with initialization, cannot be accessed directly but can only be accessed through the slice, and any other slices we define from that original slice. Example 05-12 demonstrates these two way s of creating slices.

```go
// Example 05-12 Slice creation

package main

import "fmt"

func main() {

  s1 := []int{1, 2, 3}

  s2 := make([]int, 3)

  fmt.Println(s1, s2)
}
```

```
[Module05]$ go run ex05-12.go
[1 2 3] [0 0 0]
```

The only sign that we are creating a slice and not an array when we use an array literal is that a slice does not have the ellipsis between the [ ].

This statement creates an array

```
a := [...]int{1,2,3}
```

but this statement creates a slice

```
s := []int{1,2,3}
```

5.4.3 Appending Elements

The append function allows us to add elements to the end of a slice. However if we don't understand what is going on with slices, then we can get what appear to be counter intuitive results such as the results of example 05-13.

```go
// Example 05-13 Odd behavior

package main

import "fmt"

func main() {

  a := [...]int{100, 200, 300}
  s := a[:2]
  fmt.Println("Initially a=", a, "s= ", s)
  s = append(s, -1) //result is a[2] == -1
  s[0] = 0          // result a[0] == 0
  fmt.Println("After first op a=", a, "s=", s)
  s = append(s, -2) // this would go in a[3]?
  fmt.Println("After second op a=", a, "s=", s)
  s[0] = 999 // a now remains unchanged
  fmt.Println("After third op a=", a, "s=", s)
}
```

```
[Module05]$ go run ex05-13.go
Initially a= [100 200 300] s=  [100 200]
After first op a= [0 200 -1] s= [0 200 -1]
After second op a= [0 200 -1] s= [0 200 -1 -2]
After third op a= [0 200 -1] s= [999 200 -1 -2]
```

In the example, the first two operations work as we expect, we make modifications to the slice “s” and they show up in the underlying array “a”. The length of "s" is 2 but has capacity of 3 since it doesn't extend to the end of the underlying array. When we append -1 to "s", that has the effect of extending "s" to include the last position "a". So far, so good – everything works exactly as we would expect it to.

Then we append one more element to "s" which does not generate an error but now "a" and "s" no longer appear to be linked. Changes to "s" do not affect "a" and vice-versa.

When we create a slice directly, Go creates the underlying array with the length specified either by the length argument in make() function or by the number of items provided in the literal used to initialize the slice. But slices can grow dynamically so when we try to add an element once the array is full, Go creates a new array double the size as the old one then copies the old array contents into the new array. The slice is now reassigned so that it references this new array, and the old array is garbage collected.

The problem is that because there is separate reference to the original array, the variable “a”, Go cannot garbage collect the original array and are still able to access it. If we had defined the slice without using an previously defined array, we would not see this behavior.

5.4.4 Specifying Slice Capacity

As stated before, the capacity of a slice can be thought of as how many times we can append something to that slice before Go has to re-size the underlying array. For slices with anonymous underlying arrays, the initial length of the slice determines the initial size of the underlying array; by default the capacity is the same as the length. We can provide a third argument to make() to specify the initial capacity of the slice.

```
Slice := make([]type,length, capacity)
```

Every time try we append something to the slice that would exceed the capacity, Go re-sizes the array. This is demonstrated in example 05-14.

```
// Example 05-14 Re-sizing Slices

package main

import "fmt"

func main() {

  s1 := make([]int, 1, 3)
  for i := 0; i < 10; i++ {
       s1 = append(s1, i)
       fmt.Println("s1=", s1, "len=", len(s1),
                                      "Cap=", cap(s1))
  }
}
```

```
[Module05]$ go run ex05-14.go
s1= [0 0] len= 2 Cap= 3
s1= [0 0 1] len= 3 Cap= 3
s1= [0 0 1 2] len= 4 Cap= 6
s1= [0 0 1 2 3] len= 5 Cap= 6
s1= [0 0 1 2 3 4] len= 6 Cap= 6
s1= [0 0 1 2 3 4 5] len= 7 Cap= 12
s1= [0 0 1 2 3 4 5 6] len= 8 Cap= 12
s1= [0 0 1 2 3 4 5 6 7] len= 9 Cap= 12
s1= [0 0 1 2 3 4 5 6 7 8] len= 10 Cap= 12
s1= [0 0 1 2 3 4 5 6 7 8 9] len= 11 Cap= 12
```

In the example, we created a slice with length 1 and capacity 3. When we tried to add a fourth element, then the slice doubled in capacity.

Deciding on an initial capacity is really a matter of what is important to the programmer – whether you want to avoid the costs of constantly re-sizing the array or want to avoid having arrays that are not being re-sized very often but have big chunks of unused space.

5.5 Slices as function arguments

Earlier in this module we looked at the issues of trying to pass an array as an argument to a function as a result of Go passing the array by value. However, since a slice is a reference to an array, passing a slice as a parameter has the effect of passing the array by reference.

Example 5-15 demonstrates this.

```
// Example 05-15 Slices as parameters

package main

import "fmt"

func f(p []int) {
  p[0] = -1
}

func main() {

  s := []int{1, 2, 3, 4}
  fmt.Println("Before call", s)
  f(s)
  fmt.Println("After call", s)
}
```

```
[Module05]$ go run ex05-15.go
Before call [1 2 3 4]
After call [-1 2 3 4]
```

Module Six

Maps

C++ is about objects. Go is about algorithms.

Unknown

Perfection [in design] is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Antoine de Saint-Exupéry

6.1 Introduction

Maps in Go are are implemented as references to underlying hash tables. Hash tables tend to perform in close to constant time, depending on how collisions are handled, which makes the one of the most efficient data structures available to us. We are not going to explore how maps work at the level of the hash table since going off on that tangent gets us away from talking about maps in Go and the focus of this module which is using Go maps in our code.

Because maps are references, the cost of passing them around as arguments to functions is minimal because we we are just moving pointers around and not the underlying hash table. However direct array access is generally faster in Go than map access, some authors have cited 100x faster, so this suggests that if performance is an issue then slices may be a better choice for managing data. I cannot verify or disprove these claims about performance.

6.2 Declaring, Creating and Initializing Maps

The map data structure can be thought of as a set of key value pairs (key, value) where the keys are unique and are used to reference or find their corresponding values. The underlying low-level details of how this is done is beyond the scope of this course.

6.2.1 Declaring Maps

Maps are a reference type, just like slices, and are declared using the general syntax

```
var mapname[keytype]valuetype
```

This declaration does not create the underlying hash table any more than [ ]int creates the underlying array. What has been created is a variable that can be used to point to or reference some underlying hash table. Since that table does not yet exist, we call this the nil map since it doesn't reference anything... yet.

Maps, like arrays, have types. For example the following code snippet declares severity to be a reference to a map that has keys of type string and values of type string.

```
var severity[string]string
```

If we tried to use severity to reference a map that had int as the key type and string as the value type, then we would get a compile time error because map[string]string and map[int]string are two different map types.

6.2.2 Choice of Key Data Types

The actual choice of what data type the key ought to be should be dictated by what type makes sense naturally for the real world data we are working with. As far as Go is concerned, the only requirement for a type to be usable as a key is that the comparison operators `==` and `!=` be defined for that type. Since `==` is not defined for slices, for example, they could not be used as a key type.

However, just because a data type can be used as a key does not mean it should. Several discussion questions in the lab explore why some types are not good choices for keys.

6.2.3 Declaring versus Creating Maps

Declaring a map does not create the hash table. There are two ways that we can create the underlying hash table, which, as far as we are concerned, are the same conceptually as creating the map object.

Our map reference severity declared above is the nil map because it doesn't point to anything. If we try to add data to severity, then Go will generate a panic because there is no place yet to put anything. The nil map which severity references can be thought of as being like a zero pointer in C or a null pointer in Java.

There are two ways we can create a map structure to go along with our declaration.

6.2.4 Map Literals

By supplying a map literal as an initializer, the Go compiler automatically creates the underlying hash table to hold the literal in the same way it created an underlying array when we declared a slice initialized with an array literal.

```
// Example 06-01 Creating maps with literals

package main

import "fmt"

var errs map[int]string

func main() {
  severity := map[string]string{
       "Blue":   "normal",
       "Orange": "moderate",
       "Red":    "severe"}

  severity["Black"] = "apocalyptic"

  fmt.Println(severity, " size =", len(severity))
  fmt.Println(errs, " size =", len(errs))
}
```

```
[Module06]$ go run ex06-01.go
map[Blue:normal Orange:moderate Red:severe Black:apocalyptic]
size = 4
map[]  size = 0
```

In the example above, a literal, which is a list of key value pairs, has been provided to initialize the map severity. We know the map exists because we can add an element to it.

6.2.5 Using make()

The other way to create the underlying hash table is to use make() which operates analogously to new in Java and C++ and malloc() in C. The make() function creates a hash table and returns a pointer or reference to the newly created map. Once the map has been created, then it can be accessed dynamically.

```go
// Example 06-02 Creating maps with make()

package main

import "fmt"

var errs map[int]string

func main() {
  errs = make(map[int]string)
  errs[0] = "Hardware"
  errs[1] = "Segmentation fault"
  fmt.Println(errs, " size =", len(errs))
  errs[0] = "Firmware fault"
  fmt.Println(errs, " size =", len(errs))
  fmt.Println("The errorcode '0' is a ", errs[0])
}
```

```
[Module06]$ go run ex06-02.go
map[1:Segmentation fault 0:Hardware]  size = 2
map[0:Firmware fault 1:Segmentation fault]  size = 2
The errorcode '0' is a  Firmware fault
```

Just as a point of clarification: there are two kinds of maps that are hard to tell apart because both have length 0. The first is the nil map which, as we saw earlier, is when we have a map variable declared but which does not reference an underlying hash table. The other type is the empty map where the map variable references an existing hash table with no elements. Both have length 0 and both print out the same way – with the empty string.

The difference is illustrated in example 06-03 on the next page.

```
// Example 06-03 Empty versus nil maps

package main

import "fmt"

var errs map[string]string

func main() {
  fmt.Println("Check for nil before make() ", errs == nil)
  fmt.Println("Length of errs ", len(errs))

  errs = make(map[string]string)

  fmt.Println("Check for nil after make() ", errs == nil)
  fmt.Println("Length of errs ", len(errs))
}
```

```
[Module06]$ go run ex06-03.go
Check for nil before make()  true
Length of errs  0
Check for nil after make()  false
Length of errs  0
```

6.2.6 Map Capacity

Maps grow dynamically. Every time an element is added to a map, the size of the map increases by one. In terms of efficiency, this may mean a lot of behind the scenes memory management as the underlying data is potentially reorganized every time the map grows.

This process can be made more efficient if we know in advance roughly how many entries the map will have. For example, suppose I want to automate looking up the error codes for a legacy application. Looking at my administration guide I can see that there are 2000 error codes that will eventually have to be entered into the map, however only 200 of the error codes account for 99% of all the errors that actually occur but I don't know which ones those 200 are.

I've decided I have better things to do with my time than write out a 2000 item map literal. The alternative approach I decide on is to create an initial map of 200 items and as each error occurs, it will get added to the map. The initial capacity of the map ensures that for most of the 200 additions I make to the map, there will no need for the map to re-size. If there are more than 200, then I can live with a couple of re-sizings or, if I can't, I'll just set the initial capacity to 250.

In that contrived example, the problem is small enough that there probably will not be any real savings in setting up an initial capacity, but when we scale up the possible numbers of entries or the size of the entries or frequencies of additions by several orders of magnitude – a situation we might find at a Google or Facebook – then these savings start to become significant.

The syntax for specifying an initial capacity is:

```
make(map[keytype]valuetype, capacity)
```

```
// Example 06-04 Map capacity

package main

import "fmt"

var errs map[string]string

func main() {
  errs = make(map[string]string, 200)
  fmt.Println("Length of errs ", len(errs))
}
```

```
[Module06]$ go run ex06-4.go
Length of errs  0
```

Example 06-04 shows that the capacity of a map is not the same as the length. Because there are no elements in the map, its length is 0 but it has been allocated with enough memory to store up to 200 elements without needing to re-size.

6.3 Key Existence

In the examples so far in this module, we have seen the basic syntax for adding, accessing and updating elements in a map. We have not spent a lot of time on this since presumably you are familiar with this sort of associative array syntax already from other programming languages.

However these basic operations alone are not enough for us to do all the things we would like to do in terms of processing the elements in a map. There are a couple of scenarios when we want to ensure that a specific key value either exists or doesn't exist before we perform some operation on the map. We could just test if we can fetch the item from the map first to see if it exists, but aside from being a rather cumbersome way to code, that still doesn't solve all of our problems

Fetching the element is not a solution because when we try to fetch an element that does not exist in the map, ie. we cannot find an element with the requested key, the [ ] operator returns the default zero value for the map's value type. Which means we don't really know if the entry exists and has a zero value or doesn't exist and Go is just returning a zero value.

Consider the following cases:

6.3.1 Avoiding Overwrites

Using the syntax map[key] = value, the operation actually performs two different actions.

1. If the key does not exist, then a new element is added to the map.
2. If the key does exist, the existing value associated with the key is overwritten

However I may only want the first effect to take place – if the key is already in the map, I don't want its existing value to be overwritten. How do I ensure this doesn't happen? Getting a default zero back does not give me a definitive answer.

6.3.2 Deleting Elements

Elements are deleted from maps with the following function

```
delete(mapname, keyvalue)
```

The problem is that if the key value does not exist, then a panic occurs. We would like to know if the value actually exists before we try to delete it. Even trying to fetch it doesn't help because of the default zero value again.

6.3.3 Selective Processing

Suppose that we are given a list of keys for elements that might be in our map. We want to go through our map and only process the elements whose keys that are in the list we have been given. The problem we encounter is that when we execute the statement

```
v := mapname[keyval]
```

we always get a value back. If keyval is in the map, we get the associated value back, but if the keyval is not in the map, I get back the zero value for the valuetype. Now the problem is trying to figure out if the element {keyval, value} actually exists and value just happens to be zero value, or is it that the element {keyval, value} is not in the map.

6.3.4 Comma ok form

We can test for existence by using this form:

```
value, ok = mapname[keyval]
```

ok is a Boolean value that is true if the element exists in the map and false if the element does not exist.

In example 06-05, the comma ok form is used to ensure that we only update existing entries in a map and do not add new entries.

```
// Example 06-05  comma ok update
package main

import "fmt"

func update(m map[int]int, key int, val int) {
  _, ok := m[key]
  if ok {
       m[key] = val
  }
}
func main() {
  data := map[int]int{1: 100, 3: 300}
  update(data, 1, -1)
  update(data, 2, 200)
  fmt.Println(data)
}
```

```
[Module06]$ go run ex06-05.go
map[1:-1 3:300]
```

6.4 Iteration

Just like with arrays and slices, we can iterate through a map. Unlike an array, the order of the keys is arbitrary from our perspective because the map organizes them internally in order to provide the most efficient access path.

The range operation iterates through the map in the internal order of the keys and each time through provides two items, the key and the corresponding value.

Example 6-6 demonstrates this. Notice that the elements were not printed out in the order in which they were defined nor in any recognizable lexicographic order.

```
// Example 06-06 Iteration

package main

import "fmt"

func main() {

  dotw := map[string]int{
       "Sun": 1, "Mon": 2, "Tue": 3, "Wed": 4,
       "Thu": 5, "Fri": 6, "Sat": 7}
  for day, num := range dotw {
       fmt.Printf("(%s : %d) ", day, num)
  }
}
```

```
[Module06]$ go run ex06-06.go
(Thu : 5) (Fri : 6) (Sat : 7) (Sun : 1) (Mon : 2) (Tue : 3)
(Wed : 4)
```

Module Seven

Advanced Functions in Go

As someone who has written a fair bit of code in functional languages and a fair bit of Go, I find that the more Go I write the less I care about the language features (or lack thereof) that I was horrified by at first, and the more I see most other languages as overcomplicated. Go isn't a very good language in theory, but it's a great language in practice, and practice is all I care about, so I like it quite a bit
supersillyus in hackernews

Go is like a better C, from the guys that didn't bring you C++
Ikai Lan

7.1 Introduction

In module four, we looked the basics properties and uses of functions in Go but most of our discussion was essentially confined to how Go does certain things, like handle errors, in ways that are different than other programming languages while still more or less treating Go functions as a different flavor of the same sort of functions we find typically in C-style languages.

However, we can do a lot more with functions in Go than just defining and calling them. In this module we look at some more advanced concepts involving using functions which may be new to you if you have been programming in a language that does not allow this sort of functionality, or does allow it but in very abstruse and convoluted ways. If you have been programming in a functional language like Scala or Haskell, many of these concepts will be quite familiar to you.

7.2 Functions as First Class Objects

Functions in Go are “first class citizens” which in programming language jargon means that you can do the same things to and with functions that you can do to or with anything else in the language. This means assigning a function to variable, passing it as an argument to another function and using a function as a return value from other functions.

7.2.1 Function variables

In example 07-01, two functions f1 and f2 are defined. The variable f is a variable of type 'func() string' which means that is able to be assigned as a value "a function of no arguments that has a string return value". In the lab, you will change f1 and f2 in different ways to verify this.

The “type” of a function then is made up to two parts:

1. The number and type of arguments to the function.
2. The return value type.

```
// Example 07-01 Function variables

package main

import "fmt"

func f1() string {
  return "I'm f1"
}
func f2() string {
  return "and I'm f2"
}

func main() {
  f := f1
  fmt.Printf("f is of type '%T' \n", f)
  fmt.Println(f())
  f = f2
  fmt.Println(f())
}
```

```
[Module07]$ go run ex07-01.go
f is of type 'func() string'
I'm f1
and I'm f2
```

Example 07-01 has a number of points that need to be understood.

The variable f is a function variable which means that it holds a reference to a function. However because we have defined f with a short form assignment, the type of function f can hold is determined when the variable is initialized. Since f is a variable, it can be used in the same way as any other variable.

However because f holds a function, we can execute whatever function is currently referenced as the value of f. If you look at the listing carefully you will see that f is initially set to f1 which means that the functional call f() is in fact the function call to f1(). Then when we assign the function f2 to f, the function call f() is now a function call to f2().

7.2.2 Passing function as parameters

Functions can also be passed as parameters, but we need to be able to specify what type of function we are passing as a parameter in the same way that we have to specify the data type of something we are passing.

Example 07-02 illustrates this

```
// Example 07-02  Functions as parameters

package main

import "fmt"

func f1() string {
  return "I'm f1"
}
func f2(fparam func() string) {
  fmt.Println(fparam())
}

func main() {
  f := f1
  f2(f)
}
```

```
[Module07]$ go run ex07-02.go
I'm f1
```

The function f1 is the function that is going to be passed as a parameter. We already know from the last example that the function type of the function f1 is 'func() string'

The function f2 is the one we will be passing to f1 as a parameter. You should have noticed by now that when we use the syntax f() we are making a function call but when we use the function name alone it means that we are not calling the function but are using the function definition as a variable value.

The parameter fparam is a function variable and has the appropriate type defined.

This example now also answers one of the points we deferred on in the module on basic functions –why do we need the func keyword? One of the reasons is that it makes this sort of syntax using functions as first class object possible.

The main function calls f2() and passes the function f1 as a parameter. The executable line in f2 now uses the () notation to execute the function in the variable fparam.

7.2.3 Functions as return arguments

If we can use functions as parameters, then we should be able to use them as return values as well, which is exactly what we can do.

```
// Example 07-03  Function as return value

package main

import "fmt"

func f1() string {
  return "I'm f1"
}
func f2() func() string {
  return f1
}

func main() {
  f := f2()
  fmt.Println(f())
}
```

```
[Module07]$ go run ex07-03.go
I'm f1
```

Example 07-03 Demonstrates this although you may have to read through the code a few times to follow what is happening.

The function f2 is going to return the function f1 as a return value. Since we declare return types in the same way that we declare parameters, we use the same type expression we did when we used f1 as a parameter.

The part that may be difficult to get on the first reading is decoding what is happening in the main function. We are calling f2 and get back a function that we assign to f and then we execute the function that now the value of function variable f. There is another way to do this without using f by just doing f2()() which means execute the result of executing f2. We will be working with this syntax in one of the labs.

7.3. Function literals

The way we have been working with functions so far is a bit clunky and not something we normally do, but it has illustrated the idea of functions as first class objects.

Continuing the analogy with variables, the use of the function definitions that we have used in previous modules are the equivalent to the var form of defining a variable – we are defining the function and then using the name of the function to pass it around. As we have seen,with variables, this form is a little more formal and awkward so we tend to use the short form definition of variables with the := operator. That leads to the obvious question as to whether there is some way that we could use the := operator for functions the same way we do for variables.

The answer is yes and is demonstrated in example 07-04 where we create a function variable f and initialize it with what is called a function literal. A function literal is a function definition without a name that we can then use as a regular function once it is assigned to a variable.

```
// Example 07-04  Function literal

package main

import "fmt"

func main() {
  f := func(i int) bool { return i == 0 }
  fmt.Println("2 == 0 is", f(2))
  fmt.Println("0 == 0 is", f(0))
}
```

```
[Module07]$ go run ex07-04.go
2 == 0 is false
0 == 0 is true
```

However the scoping rules still apply to f – it is a still a local variable so we can't call it directly from outside the scope in which it was defined. This sort of dynamic defining of a function using function literals is usually not found in compiled languages but is more characteristic of interpreted languages like Ruby and JavaScript.

But what if we wanted to use our function outside of the scope in which it is defined? We can pass it as a parameter into that other scope. This is demonstrated in example 07-05 where we pass the function literal directly to f2 and execute it there.

Until you get used to it, the code is a bit harder to read but after you work with it for a while, it becomes second nature.

```go
 // Example 07-05  Function parameter

package main

import "fmt"

func f2(p func(int) bool) {
  fmt.Println("2 == 0 is", p(2))
  fmt.Println("0 == 0 is", p(0))
}

func main() {
  f2(func(i int) bool { return i == 0 })
}
```

```
[Module07]$ go run ex07-05.go
2 == 0 is false
0 == 0 is true
```

7.4 Anonymous functions.

So finally we ask the question of whether we even need to use function variables at all given that we can use function literals. The answer is no we don't, we can use these function literals on their own in a form we call anonymous inner functions.

In example 07-06 the function literal is executed directly without needing to assign it to a variable. There is an important notational point here that needs to be clarified

```
func() { ...}  is a function literal
func() {...}()      executes the function literal
```

this means that

```
f: = func() {...}
```

assigns the function literal to the variable f while

```
g : = func() {...} ()
```

assigns the result of evaluating the function literal to g.

```go
// Example 07-06  Executing a literal directly

package main

import "fmt"

func main() {
  z := func(x int) (y int) {
       y = x + 1
       return
  }(0)
  fmt.Println(z)
}
```

```
[Module07]$ go run ex07-06.go
1
```

Another very simplistic form of an anonymous inner function is depicted in example 07-07.

Unlike example 07-06 where we execute the anonymous function and assign the result to Z, in example 07-07 we don't have to assign the function or the result to anything. The function is inner since it is defined inside another function and it is anonymous since there is no way to reference the function: it just executes. The only real difference between 07-07 and 07-06 is that we don't care about a return value in 07-07: all we are doing is encapsulating a task or some functionality in a function literal.

While this all seems a bit complex and unnecessary, in a future module, we will find that these anonymous functions are very useful in implementing concurrency.

```
// Example 07-07  Anonymous inner function

package main

import "fmt"

func main() {
  defer func(name string) {
       fmt.Println("Hello ", name)
       return
  }("World")
fmt.Println("Main Function")
}
```

```
[Module07]$ go run ex07-07.go
Main Function
Hello  World
```

7.5 Closures

Closures are often confused with anonymous functions, but while they are related to them, closures are not the same thing. For example, the anonymous function in example 07-07 is not a closure.

Closures occur when a function has what are called free variables in the function body. A free variable is a variable used in a function which is not a local variable defined in the function body or one that is passed as a parameter, although in Go we should also add variables that are used as named return values are not free variables either.

In example 07-08, the variable “a” is a free variable. It is defined in the same scope as the anonymous function but is not part of the function. When we assign the anonymous function to z, we create an instance of the function, but because there is a free variable, the instance also makes a copy of the free variable. The function instance along with the copy of the free variable it makes is called the closure of the function.

We know this is a closure because when we pass the anonymous function as a parameter to f, we create an instance of the function including a copy of the variable “a”. Since f is out of the scope of the original definition of “a” it can't access the original variable so instead it uses the copy that was sent along with the function instance in the closure.

A closure can be thought of as an instance of the function that also has copies of all the necessary information from the scope in which it was defined (ie. free variables) so that it can execute in another scope.

```
// Example 07-08  Simple Closure

package main

import "fmt"

func f(p func(string)) {
  p("Mars") // "a" is out of scope here
}

func main() {
  a := "again "
  z := func(name string) {
       fmt.Println("Hello ", a, name)
       return
  }
  f(z)
}
```

```
[Module07]$ go run ex07-08.go
Hello  again  Mars
```

A more complex example of a closure is shown in example 07-09 which is taken from the Go Tour on the Golang website.

```
// Example 07-09 Closure

package main

import "fmt"

func intSeq() func() int {
  i := 0
  return func() int {
       i += 1
       return i
  }
}

func main() {
  nextInt := intSeq()
  fmt.Println(nextInt())
  fmt.Println(nextInt())
  newInts := intSeq()
  fmt.Println(newInts())
}
```

```
[Module07]$ go run ex07-09.go
1
2
1
```

In this case the closure is taking place in the function intSeq. When the line

```
nextInt := intSeq()
```

is executed, an instance of the inner function is created with a closure that contains a copy of the variable i. Every time this instance is executed, the copy of "i" in the closure is incremented.

The statement

```
newInts := intSeq()
```

creates a new instance of the inner function and a new closure copy of the variable i which is independent of the other copy in the other closure. This can be seen in the output where each time a closure is created, the value of “i” is set to 0.

Module Eight

User Defined Types and Structs

[the Go authors] designed a language that met the needs of the problems they were facing, rather than fulfilling a feature checklist

ywgdana in reddit

I like a lot of the design decisions they made in the [Go] language. Basically, I like all of them

Martin Odersky, creator of Scala

8. 1 User Defined Types

All programming languages have some mechanism for creating data types. Generally a data type is a bundle of data items that conceptually describes some entity in the real world encapsulated (packaged up) so that we can manipulate that bundle as a single object in our code. For example, COBOL has record types, C has struct types (which is a direct ancestor to Go structs) and object oriented languages have class definitions.

For this module, I am assuming that as an experienced programmer, you have worked with user defined types before in other programming languages so we can focus on the Go specific way that user defined data types are used.

User defined types in Go are created by using the keyword type. There are two kinds of new types that we can define in Go, aliases and structs.

8.1.1 Aliases

When we use the type keyword to create an alias, we are not really defining a new type but rather a new name for an existing type. For example, Go has two aliases defined in its list of basic data types

```
type rune int32
type byte uint8
```

There are a number of reasons why we might want to alias a type, often it is just to be able to use a more natural name for a type, as in the case of runes and bytes. In some cases we may not be able to do certain things with the existing type that we could do with an alias. We will see an example of needing to do exactly that in the next module.

8.1.2 Defining structs

Before we can use a struct we have to define its structure and its type. A struct in Go is a user defined type that has a type name and a set of fields, each of which is (usually) named and has a type. The order the fields appear in is significant in determining the struct type. That means that if we have two structs with the identical fields but the order of the fields is different, then the two structs define two different types.

Structs are defined using the following syntax:

```
type name struct {
  field1 type1
  field2 type2
...
}
```

There is also a more compact form of the definition for simple structs:

```
type name struct { field1 type1, field2 type2..}
```

An example of two struct definitions appear on the next page.

```
 type employee struct {
  firstname, lastname string
  id  int
  job string
  salary int }

type point struct {x,y float }
```

There are very few restrictions on what type a field can be, the primary one of interest to us is that the structure definition cannot be recursive, meaning the struct cannot have itself as the type of one of its fields.

For example, the following is illegal:

```
type employee struct {
  fname, lname string
  id  int
  job string
  salary int
  supervisor employee  // not allowed
}
```

But we can build up structs using other structs, a technique which is referred to as composition. For example

```
  type circle {center point, radius float}
```

Structs are a lot like arrays in terms of how they behave, in fact an array is a struct in which all the the fields are of the same type and can be referenced by position instead of name. A good guess for how a struct would act in a program is to think about how an array would act in the same situation.

Like arrays, all of the memory to hold the fields of a struct is allocated contiguously which means that structs are very efficient in terms of performance.

8.2 Creating structs

Given a struct definition, there are two ways to create objects of that type, which we also call instances of the struct or type,

8.2.1 Variable declaration

Just like arrays, we can define variables of our type like any other type

```
var bob employee
var origin point
```

In these cases, the appropriate object is created and initialized by initializing each field to its default zero value. In the above example, origin is initialized to {0,0}

```
// Example 08-01 Basic struct definiton

package main

import "fmt"

type point struct{ x, y int }
type employee struct {
  fname, lname string
  id           int
  job          string
  salary       int           }

func main() {

  var anil employee
  var p point
  fmt.Println("anil=", anil, "p=", p)
}
```

```
[Module058$ go run ex08-01.go
anil= {  0  0} p= {0 0}
```

8.2.2 Initialization

Just like other variables, we can use literals to initialize structs

```
var anil = employee {“Anil”, “Patel”, 8971,
                     “Developer”, 100000}

p := point{4,2}
```

Selected fields may be initialized

```
var anil = employee{id: 8971, fname: “Anil”, lname: “Patel”}
```

In this last form, because field names are provided, the order does not matter. Any field not named in the initial list is set to its default zero value.

```go
// Example 08-02 Struct initialization
package main

import "fmt"

type point struct{ x, y int }
type employee struct {
  fname, lname string
  id           int
  job          string
  salary       int
}

func main() {

  var p = point{2, 3}
  fmt.Println("p =", p)

  anil := employee{"Anil", "Patel", 9891,
                   "Developer", 100000}
  greta := employee{id: 8897, fname: "Greta", lname: "Smith"}

  fmt.Println("anil =", anil)
  fmt.Println("greta =", greta)
}
```

```
[Module08]$ go run ex08-02.go
p = {2 3}
anil = {Anil Patel 9891 Developer 100000}
grea = {Greta Smith 8897  0}
```

8.2.3 Using new

We an also use the new() function to allocate memory for a struct object and return a pointer to the newly allocated object

```
var greta *employee = new(employee)
```

The usual sort of notation for pointers applies. We assign the value returned by new to a pointer, then we can access the underlying object with the de-referencing operator *.

The notation

```
p := &point{4,5}
```

is a short way for calling the new operator and then initializing the fields of the new point object. This is a very common idiom in Go. We will return to pointers shortly since they are incredibly useful when working with structs.

```go
// Example 08-03 Using new

package main

import "fmt"

type point struct{ x, y int }

func main() {

  var pp1 *point = new(point) // pp1 is a pointer
  fmt.Println("pp1 =", pp1, "*pp=", *pp1)

  p := point{3, 4} // p is a variable
  pp := &p         // pp is a pointer
  fmt.Println("pp =", pp, "*pp=", *pp, "p=", p)

  pp2 := &point{5, 6} //pp2 is a pointer
  fmt.Println("pp2 =", pp2, "*pp2=", *pp2)
}
```

```
[Module08]$ go run ex08-03.go
pp1 = &{0 0} *pp= {0 0}
pp = &{3 4} *pp= {3 4} p= {3 4}
pp2 = &{5 6} *pp2= {5 6}
```

To keep it straight, the following might help:

```
bob := employee{0,0} bob is an object
ptr_to_bob := &bob   ptr_to_bob is the address of bob

ptr_to_sue := &employee{fname: "Sue"}
*(ptr_to_sue)  is the sue object pointed to by
ptr_to_sue
```

8.3 Working with structs

The fields of a struct are accessed using selectors, which is the usual dot notation used by most C-style programming languages. The dot notation works the same whether or not the variable is a value object or a pointer.

```
// Example 08-04 Field access

package main

import "fmt"

type point struct{ x, y int }

func main() {
    p := point{2, 3}        // object
    pp := &point{4, 5}    // pointer
    fmt.Println("x coord of p", p.x)
    fmt.Println("x coord of pp", pp.x)
    pp.y = -1
    p.y = 0
    fmt.Println("p=", p, " pp=", *pp)
}
```

```
[Module08]$ go run ex08-04.go
x coord of p 2
x coord of pp 4
p= {2 0}  pp= {4 -1}
```

8.3.1 Comparing structs

Like arrays, the operator == is defined on a struct if all the fields in the struct have the == defined for their type.

Two structs are equivalent if they are the same type and the each corresponding field in the two structs is also equivalent.

Warning! When working with pointers be sure you are comparing the objects they point to and not the pointers themselves. In example 08-05 pp1 and pp2 point to two different point objects, but the point objects themselves are equivalent.

pp1 == pp2 means "Are we both pointing to the same object?"

*pp1 == *pp2 means "Are the objects we are pointing to equivalent?"

```go
// Example 08-05 Comparisons

package main

import "fmt"

type point struct{ x, y int }

func main() {

  p1 := point{2, 3}
  p2 := point{4, 5}
  p3 := point{2, 3}

  fmt.Println("p1 == p2? ", p1 == p2)
  fmt.Println("p1 == p3? ", p1 == p3)

  pp1 := &point{1, 1}
  pp2 := &point{1, 1}

  fmt.Println("pp1 == pp2? ", pp1 == pp2)
  fmt.Println("*pp1 == *pp? ", *pp1 == *pp2)
}
```

```
[Module08]$ go run ex08-05.go
p1 == p2?  false
p1 == p3?  true
pp1 == pp2?  false
*pp1 == *pp?  true
```

8.3.2 Struct pointers and functions

What slices do for arrays, pointers do for structs. Because structs are value object, like arrays, when we pass them to a function, the function works with a copy of the struct. This has a couple of disadvantages

1. If our structs start to get large or we pass them around a lot, or a combination of the two, our program efficiency start to take a hit from the constant copying.
2. Functions are working with copies of our structs which means they can't make any changes to the structs they are passed.

Just like we used slices to deal with these issues for arrays, we do the same with pointers for structs

In example 08-06, we have two versions of a function that swaps the x and y co-ordinates of a point. The first one – swap1 – is passed the point as an object. Because it is working on a copy of the point, the changes it made locally are never propagated back to the original argument.

The second version – swap2 – works with a pointer to the original "a", which is why &a (the address of a) is used as an argument. As can be seen by the output, swap2 successfully mutates a.

```
// Example 08-06 Struct pointers

package main

import "fmt"

type point struct{ x, y int }

func swap1(p point) {
  p.x, p.y = p.y, p.x
  fmt.Println("After executing swap1 p=", p)
}
func swap2(p *point) {
  p.x, p.y = p.y, p.x
}

func main() {
  a := point{1, 2}
  fmt.Println("Original a =", a)
  swap1(a)
  fmt.Println("After swap1 a =", a)
  swap2(&a)
  fmt.Println("After swap2 a =", a)
}
```

```
[Module08]$ go run ex08-06.go
Original a = {1 2}
After executing swap1 p= {2 1}
After swap1 a = {1 2}
After swap2 a = {2 1}
```

8.4 Embedded structs

Structs can be embedded in two ways in other structs. The first is where we just use the one struct as a field type in a larger struct.

In example 08-07, we do just that, using a point struct as a field in a circle struct. The selector works as you would expect in setting the x co-ordinate of the center.

```
// Example 08-07 Embedded structs 1

package main

import "fmt"

type point struct{ x, y int }

type circle struct {
  center point
  radius float32
}

func main() {
  c := circle{point{50, 32}, 13.0}
  fmt.Println("c=", c)
  c.center.x = 0
  fmt.Println("c=", c)
}
```

```
[Module08]$ go run ex08-07.go
c= {{50 32} 13}
c= {{0 32} 13}
```

8.4.1 Anonymous fields

An anonymous field is one that only has a type but no name. By using an anonymous field we can access the fields in the inner struct as if they were fields in the outer struct. This is best illustrated in example 08-08 where we have two anonymous fields.

A point structure is used anonymously within a circle struct which means that we can reference the fields of the point struct as if they were fields of circle. There is also a bool which does not have a name which can also be used anonymously but has a bit of an odd selector syntax.

```go
// Example 08-08 Anonymous fields

package main

import "fmt"

type point struct{ x, y int }

type circle struct {
  point
  bool
  radius float32
}

func main() {
  c := circle{point{50, 32}, false, 3.0}
  fmt.Println("c=", c)
  c.x = 0
  c.bool = true
  fmt.Println("c=", c)
}
```

```
[Module08]$ go run ex08-08.go
c= {{50 32} false 3}
c= {{0 32} true 3}
```

There are a couple of rules about anonymous fields

1. There cannot be two anonymous fields of the same type.
2. Names in the outer struct shadow names in the inner struct
3. Two anonymous fields of different types may have fields with the same name but an error will occur if either of the fields with the same are actually accessed

The last one is a bit odd, but basically the conflict in names between fields of two different anonymous structures are not picked up unless one of them is referenced at which time the ambiguity is discovered.

8.5 Pointers as fields

Struct definitions cannot contain themselves recursively, however containing a pointer instead is not recursive. Consider the problem we had before at the start of this module linking an employee to their boss who was also an employee. By converting the boss field to a pointer, the structure works the way we would want it to.

```
// Example 08-09 Pointers as fields

package main

import "fmt"

type employee struct {
  fname, lname string
  id           int
  job          string
  salary       int
  boss         *employee
}

func main() {
  greta := employee{fname: "Greta", lname: "Smith"}
  anil := employee{fname: "Greta", lname: "Smith",
                                          boss: &greta}
  fmt.Println("Anil's boss is", anil.boss.fname)
}
```

```
[Module08]$ go run ex08-09.go
Anil's boss is Greta
```

8.6 Pseudo-Constructors

In most OO languages, one of the roles of the constructors of a class is to ensure that the object instantiated from the class is logically correct. Since we don't have constructors in Go, we can emulate the role of a constructor by applying the factory pattern and define a function that generates well formed structs for us. This is quite straightforward if we use pointers.

Consider for example the point struct. We may, for some reason decide that in our application a point can only be in the upper right quadrant, ie, both x and y are non-negative. If either component is negative, then we reflect the point around the appropriate axes.

```
// Example 08-10 Struct factory

package main

import "fmt"

type point struct{ x, y int }

func makePoint(x, y int) *point {
  if x < 0 {
       x = -x
  }
  if y < 0 {
       y = -y
  }
  return &point{x, y}
}
func main() {
  p1 := makePoint(1, 1)
  fmt.Println(*p1)
  p1 = makePoint(-4, -9)
  fmt.Println(*p1)
}
```

```
[Module08]$ go run ex08-10.go
{1 1}
{4 9}
```

Module Nine

Methods

Go is like a very delicious trifle - the further into it you go, the more delicious things you find. The quality is clear throughout. Go is very unassuming. You start by wondering what the big deal is + getting annoyed with the minor differences from other C-syntax languages before slowly progressing towards quite liking it, then eventually once you grok how simple + elegant and well engineered it is you come to love it

Lorenzo Stoakes

The way errors are handled in Go as opposed to the old exception model is huge. I'm talking the next evolution of programming languages huge. Knowing where nearly all your points of failure are and deciding what to do with them before you role a product out to a client is priceless.

Michael Schneider

9. Methods in Go

A method is a Go function that is defined in a way so that it operates on objects of a particular type. The type that the method works on is called the receiver of the method, and the set of all methods that work on a type T is called the method set of T. We can think of a method for a type as a function that sends a message, in a generic sense, to an object of that type.

In a language like Java or C++, a class definition is conceptually a type, and the class definitions contains the methods that are associated with that class. That is how we know they are methods for that class, they appear inside the class definition. The methods of a class make up the set of messages that we can send to objects of that type.

Where Go differs from these other languages is how this type and method relationship is specified, although it also is a bit more general in its application than in Java or C++. To associate a function with a type in Go, the function just needs to declare that a particular type is a receiver of the method. The effective result is the same – a type and its associated method set are functionally equivalent to a class with its instance variables and methods.

The reason why Go does it this way is the same reason that Go does everything else – it is leaner and faster and makes for a smoother compile and build process. Go throws away a lot of the features used in other OO languages because they make the language and build process overly complex, and the capabilities that they add for the programmer are, in the judgment of tho language designers, minimal.

9.1 Method definition

While a Go method is conceptually a lot like a method in other languages, the concept is more general since the receiver type of a method can be almost anything, not just a struct. There are only a few exceptions to what can have methods: interfaces being the one Go construct that cannot that we will encounter in this course.

The general syntax of a method is

```
func (receiver type) func_name(parameters)( return_values) {}
```

The receiver variable can be used exactly like a parameter in the body of the method.

9.1.1 A non-struct example

In this example, we are going to define a method on the type int32. However, one of the rules of Go methods is that we can only define methods on types in the same package, which means that the int32 type is out of scope.

However, we can define an alias for int32 – call it myint – in the current package that we can define a method on. This is one of the tricky uses of aliases, to get around a scoping rule.

```go
// Example 09-01 Method for int32

package main

import "fmt"

type myint int32

func (x myint) negate() {
  x = - x
}

func main() {
  z := myint(89)
  fmt.Println(z)
  z.negate()
  fmt.Println(z)
}
```

```
[Module09]$ go run ex09-01.go
89
89
```

The receiver variable x works exactly like a parameter. In this case, the argument z has been passed by value so x in the method is a local copy.

The example 09-01, the method could not change the receiver value.

9.1.2 Pointers as receivers

If we want a method to alter something about the receiver, which we often do with structs, then we define the method on a pointer to the receiver.

It would be awkward to have to use addresses like this in the calling code:

```
(&z).negate()
```

When Go sees a method called on a type T and there is a method with a receiver of type *T or pointer to T, then Go handles the messiness of taking the address for you.

If negate() is defined as:

```
func(x *myint) negate() {}
```

then Go does the following conversion automatically

```
z.negate()  –> (&z).negate
```

```go
// Example 09-02 Method for *int32

package main

import "fmt"

type myint int32

func (x *myint) negate() {
  *x = -*x
}

func main() {

  z := myint(89)
  fmt.Println("Before call", z)
  z.negate()
  fmt.Println("After call", z)
}
```

```
[Module09]$ go run ex08-01.go
Before call 89
After call -89
```

The method in example 09-01 was not able to change the value of what was passed into the receiver variable beause "x" was a copy of the receiver variable "z".

In example 09-02 we have changed the receiver to a pointer so that we can mutate the receiver of the method. We have also been careful to negate what the receiver points by de-referencing the points (ie. using *x).

Because of this automatic conversion that takes place, we cannot define two functions with the same name, one of which operates on a type and the other which operates on a pointer to the type. This produces an ambiguity that Go cannot resolve, the price of having Go do that bit of re-writing mentioned on the previous page.

9.2 Combining Method Sets

In Go functions cannot be overloaded as they are in languages like C++ or Java. However methods can be overloaded in a way. Two methods can have identical signatures as long as they operate on different receiver types. Remember that we are treating a variable type T and a pointer *T as the same for the purpose of method sets.

This makes sense since functions are more or less defined at the package level while methods are grouped into method sets based on their receivers. This means that we are refining the rule about function names being unique a bit by saying that a function name has to be unique within a method set.

9.2.1 Method overloading

The use of method sets partitions all our function and method definitions into disjoint sets.

Suppose we have the following three methods and functions

```
1.func (x *myint) negate() {...}
2.func (p * point) negate() {...}
3.func negate(x float32) float32 {..}
```

There will never be any ambiguity about which one of these functions will be called. The first two are distinguished based on the type of the variable that is receiving the message. The third can only be called when there is not a receiving variable.

The rule in Go is actually that the name of a function has to be unique within a method set.

```go
// Example 09-03 Method overloading

package main

import "fmt"

type myint int32
type myfloat float64

func (x *myint) negate() {
  *x = -(*x)
}
func (x *myfloat) negate() {
  *x = 0.0
}

func main() {
  i := myint(89)
  f := myfloat(189.9)
  i.negate()
  f.negate()
  fmt.Println(i, f)
}
```

```
[Module09]$ go run ex09-03.go
-89 0
```

In example 09-03 two functions called negate are defined that are in different methods sets. We are cheating and having the myfloat version just set the receiver variable value to 0 so that we can see which method is called. Notice that the appropriate method is called based on the receiver type.

9.3 Methods on inner structs

Suppose we have a struct which we will call it the outer struct. One of the fields of the outer struct is also a struct which we will call the inner struct. Consider the circle and point for example.

```
type point struct{ x, y int }
type circle struct {
  center point
  radius float32
}
```

If both point and circle have methods defined on them then both method sets apply to circle. Getting this sorted out is a bit confusing the first time you encounter it so an analogy and an example both help.

This is how Go emulates inheritance so we can use that as an analogy. The inner struct is like a super class or parent class in an inhertance hierarchy. The inner point struct has a method set that is now acessible through the circle struct, which is analogous to the subclass or child class in the inheritance heirarchy. Because a circle contains a point, the circle gets to use all of the methods in the method set for point.

Lets start with the first file in our example which contains the two structs and their method sets.

```go
// Example 09-04 Supporting code

package main

import "math"

type point struct{ x, y int }

func (p *point) swap() {
  p.x, p.y = p.y, p.x
}

type circle struct {
  center point
  radius float32
}

func (c *circle) area() float32 {
  return c.radius * c.radius * math.Pi
}
```

Because the point is a named field (center), we have to use the selector that we see in the example to get swap() to be sent to the right receiver. If we just try to send the message to the circle, the circle will reject it because it doesn't recognize it. Of course we can send the circle an area() method because it is in the method set of circle.

This doesn't seem a lot like inheritance until we make the point an anonymous field and then we see a change in behaviour that looks a lot more like inheritance.

```
// Example 09-04 Inner struct methods

package main

import "fmt"

func main() {

  c := circle{point{1, 0}, 3.0}
  fmt.Println(c)
  c.center.swap()
  fmt.Println(c, c.area())
}
```

```
[ex04]$ ./ex09-04
{{1 0} 3}
{{0 1} 3} 28.274334
```

If we change the struct definition of circle to example below, then we can rewrite the main function code as well,

```
// Example 09-05 modifications

type circle struct{
  point
  radius float32
}

...

func main() {
  c := circle{point{1, 0}, 3.0}
  fmt.Println(c)
  c.swap()
  fmt.Println(c, c.area())
}
```

```
[ex05]$ ./ex09-05
{{1 0} 3}
{{0 1} 3} 28.274334
```

Considering how accessing fields worked in the last module, this behavior is quite consistent with what we saw earlier.

But what if there is a swap function defined for the circle itself? We can speculate what happens by analogy to a situation like Java or C++. If the derived class has a method with the same name as the parent class the derived or child class version overrides the parent class version, which is exactly what we see happening here in example 6

```
// Example 09-06  Modifications to circle

type circle struct{
  point
  radius float32
}

func (c *circle) swap() {
}

func (c *circle) area() float32 {
  return c.radius * c.radius * math.Pi
}
```

In the above code, a swap function has been added to circle's method set, but it doesn't actually do anything. If we call swap() on a circle, there are three possibilities:

1. The swap() method for point will be called and we will see the x and y co-ordinates of the point swapped.
2. The swap() method for circle will be called and the x and y co-ordinates will remain unchanged.
3. We will get an error.

```
// Example 09-06 Method overloading

package main

import "fmt"

func main() {

  c := circle{point{1, 0}, 3.0}
  fmt.Println(c)
  c.swap()
  fmt.Println(c, c.area())
}
```

```
[ex06]$ ./ex09-06
{{1 0} 3}
{{1 0} 3} 28.274334
```

9.4 Combining objects

These two ways of nesting structs correspond to two different techniques commonly used in OO programming for combining objects.

1. *Aggregation.* Two objects can be combined by aggregation when one object delegates incoming messages to another object that it holds a reference to. In this case the circle object holds a copy to the point object. When we have a named field (center point), then the swap() method is delegated to the point object.

   The downside to this way of combining functionality is that we have to explicitly delegate using the selector notation c.center.swap(). On the upside, we can have as many point objects as we want associated with the circle because the selector notation ensures the right point object is being swapped.

2. *Inheritance.* Two objects are combined so that the parent objects' methods become available to the child. This is what happened when we embedded point anonymously in circle, the inner struct's (point) method set became part of the outer struct's (circle) method set. This Go form emulates inheritance quite well.

9.4.1 Multiple inheritance

It may seem that the restriction in Go on having only one anonymous field would preclude an emulation of multiple inheritance but in fact it doesn't. In multiple inheritance, a child class inherits from two different parent classes, which would be the same as having two anonymous fields of different types in a struct in Go.

In languages like C++ which allow multiple inheritance, there are usually mechanisms like virtual inheritance to prevent two copies of the same parent class form being present in the inheritance hierarchy. Go enforces the analogous restriction by allowing only one anonymous field of a given type in a struct.

Module Ten

Go Interfaces

The belief that complex systems require armies of designers and programmers is wrong. A system that is not understood in its entirety, or at least to a significant degree of detail by a single individual, should probably not be built.
Niklaus Wirth

The key to performance is elegance, not battalions of special cases.
Jon Bentley and Doug McIlroy

10.1 Interfaces in Go

In the previous two modules, we have looked at types and methods, which allow us to do very typically object oriented kinds of things. In some ways though, the notion of a type is analogous to a concrete class in Java or C++ but without any sort of abstraction mechanism. Composition is supported, as we have seen, but there is nothing that is like a Java style inheritance mechanism in Go.

Instead, Go has opted for a different kind of abstraction model, similar to the notion of a Java Interface, where a Go interface is used to define an abstract type. The idea of an interface in Go defining a type is very much in keeping with the idea of an interface in general in OO programming – it specifies a contract about what an object or type can do, but says nothing about how that contract is implemented or the internal workings of the object that implements the contract.

An interface in Go does not tell you anything more than what an implementing object can do – it defines what might be called a behavioral type. For example, if I say I am a swimmer, then that tells you something about what I do, but nothing about how I do it or anything else about me. We have an good intuitive notion of what a “swimmer” interface is – things that implement a swimmer interface can swim. But how we implement a swimmer interface for a dog is different than how a salmon or squid implements it – the only common thing all those types share is that because they have all implemented the swimmer interface, they can all swim.

Notice that in this abstraction model there is no abstract swimmer type that salmon, squids and Go programmers inherit from, but we are all of type swimmer only because we implement the swimmer interface.

This is exactly how interfaces work in Go. A Go interface defines a set of methods and any type that has those methods defined in its method set is considered to have implemented the interface. There is no need to have any formal declaration that a type implements an interface – if it has the methods defined in the interface, then it implements the interface. After all, you didn’t have to formally declare that you were implementing a swimmer interface, you just do it by being able to swim

10.1.1 Interface Types

To see how interfaces work, we look at a simple example based on the above. In this case we will define a Swimmer interface that defines one method called swim(). Generally, interfaces in Go tend to be quite small, often just one or two methods.

We also define two types, fish and human each of which has a swim method. Without any further coding, both of these types implement the Swimmer interface. However the human type also implements a function not in the Swimmer interface, but this does not affect the fact that human implements the Swimmer interface. There is nothing to prevent a type from having more methods in its method set than are in the interface which means that a type can implement multiple interfaces.

```go
// Example 10-01  Swimmer Interface

package main

import "fmt"

type Swimmer interface {
  swim()
}

type fish struct{ species string }
type human struct{ name string }

func (f *fish) swim() {
  fmt.Println("Underwater")
}

func (f *human) swim() {
  fmt.Println("Dog Paddle")
}

func (f *human) walk() {
  fmt.Println("Strolling around")
}
func main() {
  tuna := new(fish)
  tuna.swim()
  knuth := new(human)
  knuth.swim()
  knuth.walk()
}
```

```
[Module10]$ go run ex10-01.go
Underwater
Dog Paddle
Strolling around
```

10.2 Interface Variables

The example on the previous page seems trivial given that we are working with a couple of concrete types. However, the power of interfaces comes when we use variables whose type is an interface, just like we do in other languages.

In example 10.2, we create a variable "s" of type Swimmer and in turn, we assign it a fish and a human. When we call swim() on s, the interface variable remembers which implementation of swim() is should call based on the type of implementation it references.

The interface variable contains a set of references, one for each method defined in the interface. When an object is assigned to the variable, those references are all updated to point to the correct methods for that concrete type.

Now what we are getting in the example below is a variable of type Swimmer that can hold a reference to any object that implements the swimmer interface.

In the following examples, the code from ex10-01 that defines the concrete types is in the file swimmer.go so we can focus on the code of interest to us.

```
// Example 10-02  Interface Variable

package main

import "fmt"

type Swimmer interface {
   swim()
}

func main() {
  var s Swimmer
   s = new(fish)
   s.swim()
   s= new(human)
   s.swim()
}
```

```
[Module10]$ ./ex10-02
Underwater
Dog Paddle
```

10.2.1 Interface types as parameters.

In example 10-03, we see the use of an interface variable as a parameter which allows us to write much more powerful and generic functions than we would be able to do if we were limited to concrete types.

In this case the submerge() function takes any object of type Swimmer and calls the appropriate swim() method on that object.

```
// Example 10-03  Interface variables as parameters
package main

import "fmt"

type Swimmer interface {
  swim()
}

func submerge(s Swimmer) {
  s.swim()
}

func main() {
    submerge(new(fish))
    submerge(new(human))
}
```

```
[Module10]$ ./ex10-03
Underwater
Dog Paddle
```

10.2.2 Limitation of an interface variable

in example 10.04 we see that if we assign a concrete object to an interface variable, the variable can only “see” the methods in its interface. If we assign a human object to a Swimmer variable s, then for each method in the interface, "s" holds a reference to the concrete implementation of swim() in the human method set.

But if we try to call the walk() method, which is implemented for a human type but is not part of the Swimmer interface, then the method call fails because “s” does not know about the walk() method.

```
// Example 10-04  Interface variable limitiation

package main

import "fmt"

type Swimmer interface {
   swim()
}

func main() {
  var s Swimmer
   s= new(human)
   s.swim()
   s.walk()
}
```

```
[Module10]$ ./ex10-04
./ex10-04.go:8: s.walk undefined (type Swimmer has no field or
method walk)
```

10.3 Combining Interfaces

Go interfaces are typically small, usually one to three methods, but we can combine them in the same way that we can combine structs. In a sense this is exactly what we are doing because an interface is essentially a struct in terms of how it is implemented "under the hood".

In example 10.5 we have added a new interface called Walker and then combined the Walker and Swimmer interfaces into a new interface call Amphib

```
// Example 10-05 Amphib Type

package main

import "fmt"

type Swimmer interface {
   swim()
}

type Walker interface {
   walk()
}

type Amphib interface {
   Walker
   Swimmer
}

func main() {
   var a Amphib
   a = new(human)
   a.walk()
   a.swim()
}
```

```
[Module10]$ ./ex10-05
Strolling around
Dog Paddle
```

Combining the interfaces can be done in a variety of ways, all that matters is the aggregate of methods in the final result. All of the following would be equivalent ways to define the Amphib interface.

```
type Amphib interface {
  Walker
  Swimmer
}
```

```
type Amphib interface {
  Walker
  swim()
}
```

```
type Amphib interface {
  walk()
  Swimmer
}
```

```
type Amphib interface {
  walk()
  swim()
}
```

11.4 Type Testing

At any given time an interface variable can potentially hold any one of a number of types. At points in our program logic, it may be important for us to know exactly what sort of concrete type our interface variable is referencing

We can do this by asking the variable if it contains a particular type using the syntax

```
n, ok := v.(T)
```

If the object referenced by v is of type T, then ok is set to true and n is created as a reference to the object but is a variable of type T. If v is not of type T, then ok is false and v is set to the default zero value for T.

This sounds a little confusing until we walk through a concrete example in 10-06

In example we have created a slice of two swimmers, the first is a human and the second is a fish. In the for loop, we ask if each entry in turn is actually a human. The first time through, ok is true and the human object “Bobby” is assigned to h. Now that we have a reference to the human type, we can call the walk() method which which would not have worked otherwise – specifically swimmer.walk() would have produced an error just like we saw before because the swimmer variable does not know about anything outside the Swimmer interface.

The second time through the loop, testing the fish produces an ok value of false, so the body of the conditional is skipped.

```
// Example 10-06 Type Testing

package main

type Swimmer interface {
  swim()
}


func main() {
  var x = []Swimmer{&human{"bobby"}, &fish{}}

  for _, swimmer := range x {
      if h, ok := swimmer.(*human); ok {
          h.walk()
      }
  }
}
```

```
[Module10]$ ./ex10-06
Strolling around
```

11.4.1 The Type Switch Statement

A more convenient way to deal with processing multiple possible types is to use the switch statement we saw back in module three. In this case, our test value is the type the object and each case corresponds to each different type our object could be. Again a simple example clarifies how this is done.

In example 10-7, we have added a squid and cat types into the swimmer.go file both of which have swim methods as shown below. This means that all of these new structs are of type Swimmer.

```
  // Exmaple 10-07 Swimmer.go

  package main

  import "fmt"

  type fish struct{ species string }
  type human struct{ name string }
  type squid struct{ inkyness int }
  type cat struct{ breed string }

  func (f *fish) swim() {
    fmt.Println("Underwater")
  }
  func (f *squid) swim() {
    fmt.Println("Jetting along")
  }
  func (f *cat) swim() {
    fmt.Println("If I must...")
  }
  func (f *human) swim() {
    fmt.Println("Dog Paddle")
  }
  func (f *human) walk() {
    fmt.Println("Strolling around")
}
```

We can use the switch statement to now select type specific processing as we iterate through a list of Swimmer objects. Back in module three, I mentioned that the generalization of the Go switch statement from that normally seen in C-style languages is quite useful and this is an example of exactly that usefulness.

```
// Example 10-07 Switching on types

package main

import "fmt"

type Swimmer interface {
  swim()
}

var x = []Swimmer{&human{"bobby"}, &fish{}, &cat{}, &squid{}}

func main() {
  for index, swimmer := range x {
       switch t := swimmer.(type) {
       case *human:
            fmt.Printf("Item %d is a human and is ", index)
            t.walk()
       case *squid:
            fmt.Println("Item ", index, "is a squid")
       case *fish:
            fmt.Println("Item ", index, "is a fish")
       default:
            fmt.Printf("Item %d is of type %T\n", index, t)
       }
  }

}
```

```
[Module10]$ ./ex10-07
Item 0 is a human and is Strolling around
Item  1 is a fish
Item 2 is of type *main.cat
Item  3 is a squid
```

10.5 The empty Interface

One of the enduring constructs in various OO languages is the idea of a common universal type at the root of all inheritance hierarchies, whether it is the “vanilla” flavor in ZetaLisp or the "Object" class in Java, because having such a construct simplifies a lot of language design issues.

While Go does not have inheritance hierarchies, it does have a useful construct that emulates this kind of construct called the empty interface. The empty interface has no methods and thus by default, every type implements the empty interface, whether it is a built-in type or a user defined type.

From a practical perspective, what this allows us to do is to create collections of arbitrary types of objects. In the following example, we define an interface called “whatever” which allows us to create a slice made up of different things.

From an implementation point of view, what we are doing is creating an underlying array of “whatever” interface objects. Each whatever object has two kinds of information embedded in it, a record of the actual type it refers to and a reference to that object. Remember earlier that the interface knows dynamically how to call the methods of the interface by essentially maintaining pointers to the methods of the concrete type, or at least that is a good way to think about it conceptually. Essentially the same sort of mechanism is in play here.

There is not a predefined empty interface, we can create as many empty interfaces as we want, all named different things. Functionally, they are all the same but we may want more than one name for clarity in our code.

```
// Example 10-08 The empty Interface

package main

import "fmt"

type Swimmer interface {
   swim()
}
type myint int32
type point struct{ x, y int }

var i myint = 0

type whatever interface{}

// continued on next page...
```

```
// Example 10-08 The empty Interface... continued

func main() {

   mylist := []whatever{i, float64(45), &point{2, 3}, true,
          &fish{"tuna"}}
   for index, object := range mylist {
        switch v := object.(type) {
        case bool:
             fmt.Printf("item %d is a bool = %v\n", index, v)
        case myint:
             fmt.Printf("item %d is a myint = %v\n", index, v)
        case float64:
             fmt.Printf("item %d is a float64 = %v\n",
                                                  index, v)
        case *point:
             fmt.Printf("item %d is a point = %v\n"
                                                 index, *v)
        default:
             fmt.Printf("item %d is a %T = %v\n",
                                                index, v, v)
        }
  }

}
```

```
[Module10]$ ./ex10-08
item 0 is a myint = 0
item 1 is a float64 = 45
item 2 is a point = {2 3}
item 3 is a bool = true
item 4 is a *main.fish = &{tuna}
```

Module Eleven

Concurrency in Go

[Go] is the most elegant imperative language ever (including dynamic ones like Lua, Ruby, and Python)
Quoc Anh Trinh

Four out of five language designers agree: Go sucks. The fifth was too busy [to answer] actually writing code [in Go].
aiju

11.1 Concurrency in Go

As Rob Pike pointed out in one of his lectures:

> *when people hear the word concurrency they often think of parallelism, a related but quite distinct concept. In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.*

The motivation for making concurrency an integral part of the Go language is that we live in a concurrent world. The real world is made up of independent activities that interact with each other and are coordinated to achieve some common goal. As the developers of Go point out, you can't model these real world concurrent activities in an application with just sequential logic. This real world concurrency is exactly what the developers at Google were working with in in their problem domain, so it became quite natural to think of concurrency as being a fundamental part of the language design when trying to look at ways to address these kinds of challenges

Concurrency is defined in Go as t*he composition of independently executing computations so that software can be structured to interact with real world while still adhering to the principles of clean code.* (This is a slightly paraphrased definition from Rob Pike's presentations.)

11.1.2 Parallelism versus Concurrency

Parallelism is about doing many tasks at the same time, usually in some sort of highly coordinated manner. Parallelism requires concurrency but concurrency does not require parallelism. Concurrency is a way to model a solution be breaking it down into independent parts, and a concurrent application may be able to use parallelism (eg. multi-core processors) if the facility if the resources are available to do so.

As an example of a concurrent but non-parallel situation, consider the usual way of handling user input at a standard workstation. The keyboard handler and the mouse handler are concurrent, and occasionally coordinating their activities, but we do not require any sort of parallelism to implement this concurrent solution.

Concurrent applications can be implemented in a single CPU environment but parallel applications generally cannot be. Bottom line is Go is has built in concurrency which is independent of the underlying infrastructure and architecture.

11.1.3 Concurrency In Go

The model that Go uses to implement concurrency is based on an approach called CSP (communicating sequential processes) first proposed by Tony Hoare in 1978 and later implemented in several variants in Occam (1983) and Erlang (1986). The form of concurrency that Go uses is the latest in what Pike calls the Occam lineage.

Go implements concurrency through four mechanisms, three of which are supported by basic language features while the fourth is supported through the use of specific Go idioms or programming patterns.

1. Concurrent function execution implemented with goroutines.
2. Synchronization and communication between goroutines (channels).
3. Multi-way concurrent control (**select** statement).
4. Specific Go concurrency idioms.

The fourth point is a bit different that the other three since it refers to the ways in which Go programmers compose the three language constructs to produce elegant concurrent solutions. Generally we use the term *idiom* instead of pattern when we are speaking about a programming pattern within a specific programming language and reserve the word *pattern* for design solutions that are programming language independent.

The design goals of the implementation of Go concurrency are:

1. Make it very simple to code by having a lot of the low level synchronization handled by Go in the background.
2. Keep goroutines very lightweight, lighter than threads, so they are very efficient when it comes to resource utilization.
3. Keep the garbage collection managed by Go so that memory management does not have to be dealt with at the program level.

11.2 Goroutines

A goroutine is an independently executing function. It is not as heavyweight as a thread, in fact goroutines are multiplexed into threads and Go itself does the thread management. A single thread could possibly have thousands of goroutines multiplexed into it where each goroutine has its own dynamic stack which grows and shrinks dynamically as required. Goroutines can effectively be thought of as really cheap to use threads.

In every Go program, the **main()** function is by default a goroutine.

11.2.1 Creating Goroutines

Any function can be turned into a goroutine by use of the **go** operator. Consider first the non concurrent function in example 11.01 called “service” that does nothing in particular except print some stuff out. If we call this function in the usual synchronous manner, then the main function will wait for the service function to return before proceeding. Since the service function never returns because of the infinite loop, the program runs forever. Or at least in the listing below until the program was manually halted.

```
// Example 11-01 Normal function call
package main

import (
    "fmt"
     "math/rand"
     "time"
)

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    service("Message:")
}
```

```
[Module11]$ go run ex11-01.go
Message: 0
Message: 1
Error: process crashed or was terminated while running.
```

In example 11-02, by using the go operator during the function call, the service() function now executes independently of the main function. What we have essentially done is turned what was a synchronous function call into an asynchronous function call. The side effect though is that the main function continues to execute and exits before the service() function can actually do anything. In Go when the main function exits, any goroutines spawned since the main function started are immediately terminated.

```
// Example 12-02 Goroutine function call
package main

import (
    "fmt"
     "math/rand"
     "time"
)

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    go service("Message:")
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-02.go
Done!
```

In example 11-03, we put the main function to sleep for a few seconds to give the service() goroutine some time to actually execute.

```
// Example 11-03 Goroutine function call
package main

import (
    "fmt"
     "math/rand"
     "time"
)

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    go service("Message:")
    time.Sleep(2 * time.Second)
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-03.go
Message: 0
Message: 1
Message: 2
Message: 3
Message: 4
Message: 5
Done!
```

In example 11-04, we make two go calls to service() and the output shows that they are in fact executing independently of each other with the output of the two running versions interleaved.

```
// Example 11-04 Multiple Goroutine function call
package main

import (
    "fmt"
     "math/rand"
     "time"
)

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    go service("Alpha:")
    go service("Beta:")
    time.Sleep(2 * time.Second)
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-04.go
Alpha: 0
Beta: 0
Alpha: 1
Beta: 1
Alpha: 2
Beta: 2
Alpha: 3
Beta: 3
Alpha: 4
Beta: 4
Alpha: 5
Done!
```

11.2.2 Anonymous Goroutines

Often the task that we set out to run as a goroutine is quite simple and doesn't require a formal function definition. A very common Go idiom is to define the task with an anonymous function literal and set it running as a goroutine. You can think of this as delegating a task off to a goroutine and then forgetting about it.

This is demonstrated in example 11-05 by making the service function into an anonymous function. The final effect is the same although one way may be more natural to code than the other depending on the programming context.

```
// Example 11-05 Anonymous Goroutine function call
package main

import (
    "fmt"
     "math/rand"
     "time"
)

func main() {
    go func() {
        for i := 0; ; i++ {
            fmt.Println("Anon:", i)
            time.Sleep(time.Duration(rand.Intn(1e3)) *
                                              time.Millisecond)
         }
    }()
    time.Sleep(2 * time.Second)
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-05.go
Anon: 0
Anon: 1
Anon: 2
Anon: 3
Anon: 4
Anon: 5
Done!
```

11.3 Channels.

So far we can create goroutines that all run independently, but in order to get true concurrency, we need to be able to compose these different goroutines into a logical structure which means they need to be able to communicate with each other and to coordinate and synchronize their activities.

This is done through something called a *channel*, which is a bit like a socket or a pipe in Unix. Channels in Go are bi-directional by default so they can be thought of as both a sink and a source of data.

A channel is a first class object in Go which means that we can use them in all the ways that we use variables in Go. But because we can pass channels as parameters and use them as return values, it means that like other first class objects such as variables, pointers and functions; channels have types which describes the type of data that can be sent over the channel.

11.3.1 Creating channels

The following are two equivalent ways to create an int channel called “c” which would be used by two goroutines to send integer “messages” back and forth . Recall from using make() that the variable c is actually a pointer to a channel, which means that the cost of passing it as a parameter is very low, like passing references to maps or slices.

```
var c chan int
c = make(chan int)

or

c := make(chan int)
```

11.3.2 Using Channels

The channel operator is the left pointing arrow < - which is used for both read from and write to a channel.

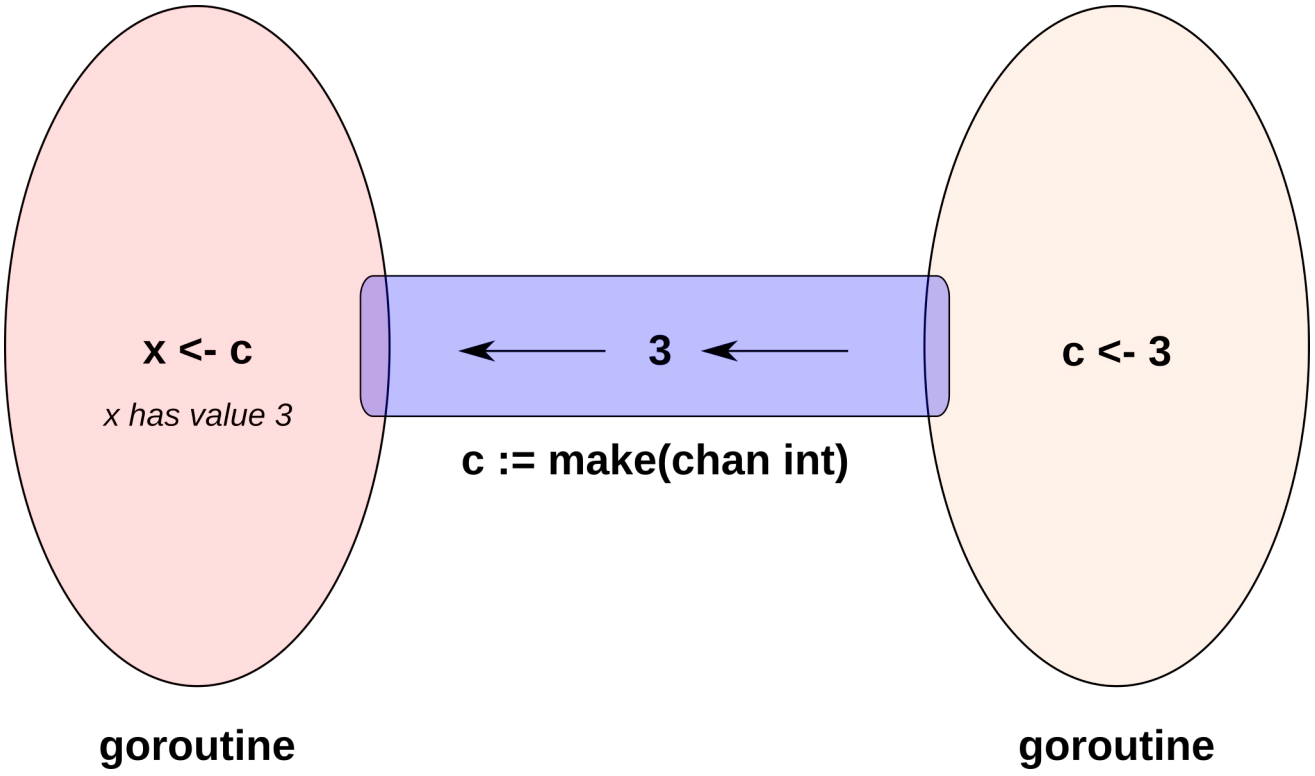
To write to the channel c we use the syntax

```
c <-  2
```

and to read from a channel c we use the syntax

```
myvariable <- c
```

In summary, when the channel appears on the left hand side of the arrow we are writing to the channel, and when it appears on the right hand side of the arrow, we are reading from the channel. This is illustrated in the diagram at the the top of the next page.



In the diagram above, a channel “c” is used to connect two goroutines. The value 3 is put onto the channel by the goroutine on the right then made available to goroutine on the left which then reads it and assigns it to the variable x.

Channels will block on either side if the channel transmission cannot be completed. In other words, if the goroutine on the right in the diagram above tries to write to the channel, it will block at the statement “c <-3 “ until the goroutine on the left executes the statement “x <- c”. And similarly if the goroutine on the left blocks when it tries to read until the go routine on the right writes to the channel. Both goroutines have to be at their corresponding channel i/o statements for the operation to complete, and each goroutine will wait for the other if necessary to get there.

This forces a synchronization to occur between the two goroutines at the point the channel is used.

The basic use of a channel is demonstrated in example 11-06. In this case the channel is created then passed as a parameter to the service() goroutine both the main and service() functions have access to the same channel.

```go
// Example 11-06 Basic channel
package main

import (
    "fmt"
    "time"
)

func service(message string, c chan string) {
    for i := 0; ; i++ {
        fmt.Println("service: ", i)
        c <- fmt.Sprintf("%s %d", message, i)
    }
}

func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-06.go
service:  0
service:  1
main 0 Got back: "Message:  0"
main 1 Got back: "Message:  1"
service:  2
main 2 Got back: "Message:  2"
service:  3
main 3 Got back: "Message:  3"
service:  4
main 4 Got back: "Message:  4"
service:  5
Done!
```

Ignoring the first iteration for now, notice that main function and service() function are now synchronized – the loops are iterating in lockstep with each other. One of the reasons why we are ignoring the first iteration is that the order of print statements may not be a totally accurate picture of the order of execution because of the time lag for the print statements to execute, but by slowing down the iterations with the wait function in the for loop in the main function, we can more or less corrects for the lag.

11.3.3 Deadlocks

Because of the unbuffered and synchronous behavior of channels, it is possible to deadlock a channel. Generally in Go having deadlocked channels is considered to be a symptom of poor program design.

Example 11-07 demonstrates an example of a deadlock can occur. The first action taken by both the main function and the service() function on the channel is to write to it. Both goroutines have written to the channel and are now waiting for the other goroutine to read it. At this point since no read can occur the program is deadlocked.

However Go monitors the status of all the active channels and when it detects a deadlock situation, it generates a panic an ends the program, as can be seen output of the example below.

```
// Example 11-07 Deadlocks

package main

import "fmt"

func service(c chan string) {
    c <- "Hello"     //write
    fmt.Println(<-c) // read
}

func main() {
    c := make(chan string)
    go service(c)
    c <- "ho"        // write
    fmt.Println(<-c) //read
}
```

```
[Module11]$ go run ex11-07.go
fatal error: all goroutines are asleep - deadlock!
```

11.3.4 Buffered Channels

It is possible to attach a buffer to to a channel which has the effect of creating an asynchronous channel. The value of a buffered channel comes when there is a mismatch between the rate of reading and writing of the two goroutines at either end of the channel. The use of a buffer decouples the read write operations of the goroutines when it may, from a program design perspective, be advantageous to reduce the synchronization between the goroutines.

Buffered channels are created by providing a capacity value to the make() operation. For example

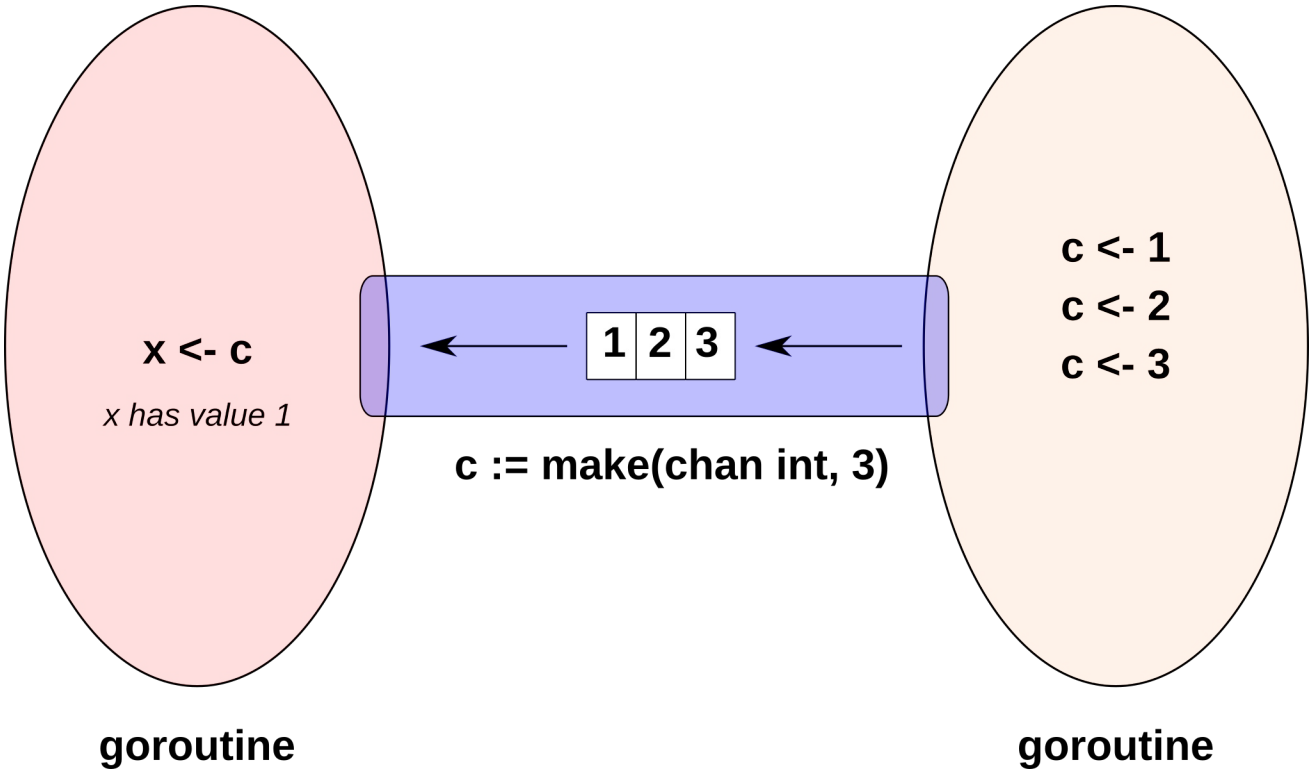
```
c: = make(chan int, 5)
```

makes an int channel with a buffer of size five.

The capacity of the buffer can be accessed with cap(buffer) and the current number of items in the buffer can accessed by using len(buffer).

Each write to a buffered channel pushes the item onto the end of the buffer unless the buffer is full, in which case it blocks until a slot in the buffer is available.

Each read from a buffered channel takes the item from the front of the buffer – the buffer is a FIFO queue. If there are no items in the buffer, then it behaves like an unbuffered channel and the attempt to read blocks until something is in the buffer.



```go
// Example 11-08 Buffered Channels

package main

import (
    "fmt"
    "time"
)

func service(message string, c chan string) {
    for i := 0; ; i++ {
        fmt.Println("service: ", i,"buffered items ",len(c)))
        c <- fmt.Sprintf("%s %d", message, i)
    }
}

func main() {
    c := make(chan string, 3)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-08.go
service:  0 buffered items  0
service:  1 buffered items  0
service:  2 buffered items  1
service:  3 buffered items  2
service:  4 buffered items  3
main 0 Got back: "Message:  0"
main 1 Got back: "Message:  1"
service:  5 buffered items  3
main 2 Got back: "Message:  2"
service:  6 buffered items  3
service:  7 buffered items  3
main 3 Got back: "Message:  3"
main 4 Got back: "Message:  4"
service:  8 buffered items  3
Done!
```

11.3.5 Unidirectional Channels

So far all of the channels we have seen have been bi-directional. However in many cases, the nature of the problem suggests that a goroutine only need to receive or send but not do both. This is very common in a pattern of goroutine composition called the pipeline where a goroutine will take inputs from one or more goroutines, perform some sort of operation on the inputs then send the results of the operation off to another set of one or more goroutines. In this case, only unidirectional message passing is required.

Channels themselves are not inherently unidirectional, obviously they cannot be since my receive channel has to be your send channel or the channel is useless. What I can specify that at your end, the channel is write only when I send the channel reference to you.

In example 11-09 the channel has been passed to the service goroutine as a write only channel The mnemonics chan<- and<-chan are used to specific that the channel is a write only or a read only channel. The only difference between this and the previous example is that the direction of the channel as passed to the goroutine is now specified.

```
// Example 11-09 Unidirectional channel
package main

import (
    "fmt"
    "time"
)

func service(message string, c chan<- string) {
    for i := 0; ; i++ {
        fmt.Println("service: ", i)
        c <- fmt.Sprintf("%s %d", message, i)
    }
}

func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done!")
}
```

11.3.6 Closing a channel

The close operation on a channel is performed by a sender to indicate that no more data will be sent. If an attempt is made to write to a closed channel, then Go generates a panic. It is also a compile error to try and close a read only channel since the closing a channel indicates that there will be no more data written.

In example 11-10 we close the channel after we send on it, then because the loop continues, iteration 2 causes attempts to write on the closed channel and we get a panic.

```
// Example 11-10 Closing a channel
package main

import (
    "fmt"
    "time"
)

func service(message string, c chan<- string) {
    for i := 0; ; i++ {
        fmt.Println("service: ", i)
        c <- fmt.Sprintf("%s %d", message, i)
        close(c)  // will cause a panic next iteration
    }
}

func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-10.go
service:  0
service:  1
panic: send on closed channel
```

In example 11-11 we see these concepts combined in a more elegant concurrency pattern. A channel is created, it is passed to a "sender," a goroutine that generates output to the channel as a write channel, and it is also passed to a "receiver," a goroutine reads input from the same channel that the other goroutine writes to.

When the output is finished, the sender closes the output channel to indicate that they are finished sending. Meanwhile the receiver is using the range function to read continuously from the channel until the channel is closed on the other end. Once that happens, the receiver now reaches the end of the “range” of the channel and exits the for loop.

This is also an excellent example of the use of the range operation. A channel is essentially a sequence so the range operation can read from that sequence until it gets to the end of the sequence (i.e. when the channel is closed) and then we are done.

```
// Example 11-11 Unidirectional Channels
package main

import "fmt"
import "time"

func sender(output chan<- int) {
  for i := 0;    outputi < 3; i++ {
       output <- i
  }
  close(output)
}

func receiver(input <-chan int) {
  for j := range input {
       fmt.Println("Received ", j)

  }
  fmt.Println("I'm done ")
}

func main() {
  c := make(chan int)
  go receiver(c)
  go sender(c)
  time.Sleep(1 * time.Second)
}
```

```
[Module11]$ go run ex11-11.go
Received  0
Received  1
Received  2
I'm done
```

11.3.7 The Blocking Problem

In the previous example, what if the sender doesn't bother closing the channel? Then the receiver blocks. One of the best practices is to always start a goroutine using defer at the start of the goroutine that is responsible for closing a channel . Then if the goroutine exits for any reason, its output channel is closed and the rest of the goroutines that are waiting on input from it can stop waiting and proceed with their own execution.

If the range function is not the right tool for the job of checking to see if a channel is closed, then we can check explicitly to see if a channel is closed by using the comma ok idiom from before as demonstrated in example 11-12 where the receiver function is modified to use the ok comma idiom. If a channel is open, then ok is true but as soon as it is closed, then ok is false and we know not to continue listening on that channel.

```
// Example 11-12 Modification to receiver and sender

func sender(output chan<- int) {
  defer close(output)
  for i := 0; i < 3; i++ {
       output <- i
  }
}

func receiver(input <-chan int) {
  for {
       j, ok := <-input
       fmt.Println(ok)
       if !ok {
            break
       }
       fmt.Println("Received ", j)
  }
  fmt.Println("I'm done ")
}
```

```
[Module11]$ go run ex11-12.go
true
Received  0
true
Received  1
true
Received  2
false
I'm done
```

11.4 The Select Statement

The Go select statement looks like a switch statement, and in fact it is a lot like it, but it is a specialized polling mechanism that monitors a number of channels as a sort of multiplexed listener. This is critically important to developing concurrent applications because without the facility, we can find ourselves getting int all sorts of blocking issues that pretty much subvert the power of being able to use concurrency.

One of the common problems we need to solve in concurrency is the multiplex problem where a goroutine needs to read from several different sources and merge the inputs. As it stands, we cannot do this effectively since even using buffered channels our multiplexer could block if any one of the channels we are reading from blocks.

To allow effective multiplexing, Go uses the select statement which syntactically looks like a switch statement. Each case in the select statement is the input from a channel. Each time through the select statement, input is read from a channel that is ready. If more than one channel is ready, then one of those is picked at random. If none of the channels are ready then the default case, if present, executes.

In the example 11-13 there are two source goroutines to be multiplexed. One produces a stream of odd integers and the other a stream of even integers. Each one generates output on its channel at random intervals, just to make the problem interesting.

The select statement polls the two channels and when one is ready, reads from that channel and executes the case statement associated with it. If neither are ready, then the default case executes

```
// Example 11-13 Select Statement Multiplexer

package main
import (
  "fmt"
  "math/rand"
  "time"
)

func sourceEven(outchan chan<- int) {
  for i := 0; ; i++ {
       outchan <- i * 2
       time.Sleep(time.Duration(rand.Intn(1e3)) *
                                          time.Millisecond)
  }
}
func sourceOdd(outchan chan<- int) {
  for i := 0; ; i++ {
       outchan <- (i * 2) + 1
       time.Sleep(time.Duration(rand.Intn(1e3)) *
                                          time.Millisecond)
  }
}
func counter(ceven <-chan int, codd <-chan int) {
  for {
       select {
       case e := <-ceven:
            fmt.Println("Even:", e)
       case o := <-codd:
            fmt.Println("Odd:", o)
       default:
            fmt.Println("no one is ready")
            time.Sleep(100 * time.Millisecond)
       }
  }
}

func main() {
  codd := make(chan int)
  ceven := make(chan int)
  go sourceOdd(codd)
  go sourceEven(ceven)
  go counter(ceven, codd)
  time.Sleep(2 * time.Second)
}
```

```
[Module11]$ go run ex11-13.go
Even: 0
Odd: 1
no one is ready
Odd: 3
no one is ready
no one is ready
no one is ready
no one is ready
Even: 2
no one is ready
Odd: 5
Even: 4
no one is ready
Even: 6
no one is ready
no one is ready
no one is ready
Odd: 7
no one is ready
no one is ready
Even: 8
no one is ready
no one is ready
no one is ready
no one is ready
```

11.4 Race Conditions

All of the concurrent goroutines that have been presented so far are concurrency safe. However when two concurrent goroutines access a common resource, usually a variable or other kind of data structure, then we can have what is called a race condition. This is a common sort of problem in any concurrent system where accesses to a shared resource by concurrent processes may be interleaved in an unpredictable manner. Concurrency by its nature leads to race conditions since while we know that two concurrent goroutines will access a shared resource, we cannot predict in which order they will access it.

The classic example of a race condition is a shared variable update. The code in example 11-14 shows the sort of code that can lead to a data race condition – the data is read from a variable, updated then written back to the variable. This is a simplified version of something that can occur in a number of ways – access to a shared data structure via pointers, modifying part of a complex data structure and so on. What makes this clearly a problem is that between the original read of the data, other goroutines may change the value of the data before the the original reader does its update.

The problem with a data race condition is that it doesn't occur all of the time, it may take a particular combination of loading or timing for the race condition to occur, which means that it often will not be caught by standard testing.

The example 11-14 is rather artificial since it deliberately introduces a delay from the time the balance variable is accessed until it is updated to allow the data race condition to take place. This is our way of emulating some exceptional condition to create a data race problem. Without putting the first goroutine to sleep, the balance is updated as we expect it be, but when we add in the sleep condition then we have wrong result. If we actually ran the example without the sleep, it is doubtful that we would see an actual data race condition manifest itself.

In the real world, the sort of thing that the sleep condition represents may be anything from the time it takes data to transfer or network latency due to loading, or any other sort of condition that may affect the rate at which the concurrent routines work.

```
// Example 11-14 Race Condition
package main
import "fmt"
import "time"

var balance int

func update(amount int) {
  balance = amount
}
func query() int {
  return balance
}
func main() {
  go func() {
       bal := query()
       time.Sleep(time.Duration(time.Millisecond))
       update(bal + 100)
  }()
  go func() {
       bal := query()
       update(bal + 1)
  }()

  time.Sleep(time.Duration(time.Second))
  fmt.Println("Final Balance=", query())
}
```

```
[Module11]$ go run ex11-14.go
Final Balance= 100
```

There are two solutions to this problem, one is a design solution and the other involves introducing a lock on the variable like in other programming languages. The preferred solution is to think in terms of the concurrent design solution.

The core of the design solution is to let only one goroutine access the resource and have it take requests from other goroutines via channels to update the resource. This solution is more in keeping with the idea of concurrency through shared messaging not shared memory.

This sort of solution is intuitive since we often find that in the non-computer world we often restrict access to resources to “keepers of the resource” to whom we have to make requests. The example that comes to mind is when I was a graduate student working with historical documents. None of us were allowed to get or return the documents from the collection – we had to have the librarian do that for us. We can apply this pattern to rethink the sort of situation in example 11-14.

```go
// Example 11-15 Shared Resource CSP Style
package main

import "fmt"
import "time"

var deposit chan int
var query chan int

func accountUpdater(d <-chan int, q chan<- int) {
  balance := 0
  for {
       select {
       case amt := <-d:
            balance = balance + amt
       case q <- balance:

       }
  }
}

func accountRequester(amt int, d chan<- int) {
  for i := 0; i < 5; i++ {
       time.Sleep(time.Duration(time.Millisecond))
       d <- amt
  }

}

func main() {
  deposit := make(chan int)
  query := make(chan int)
  go accountUpdater(deposit, query)
  go accountRequester(100, deposit)
  go accountRequester(10, deposit)
  time.Sleep(time.Duration(time.Second))
  fmt.Println("Final Balance=", <-query)
}
```

```
[Module11]$ go run ex11-15.go
Final Balance= 550
```

11.4.1 Synchronization

Go also supports a variety of other more traditional methods of locking resources using mutex and semaphore types, as well as other standard concepts. For the purpose of this course, we will look at only one of these as an example of how they are used.

In example 11-16 we use some of the functionality from the sync package to lock the balance variable over a critical section of code. We start by defining a lock on the resource, in this case called bal_lock. When a goroutine locks the shared resource, no other goroutine can access the resource until the lock is released.

This is a more typical approach to synchronization as seen in many other programming languages that support concurrency, but getting into the standard sorts of models is bit out of scope of the course simply because it is a large and complex topic. While it is always preferable to design concurrency into a program like we did before because it usually tends to both produce a more elegant design but also one that does not depend on lower level constructs in the operational environment to support concurrency. However it is not always possible to build a concurrent design, or there may be other sorts of implementation issues that require us to use a more standard concurrency technique.

```go
// Example 11-16 Race Condition and Mutex
package main

import "fmt"
import "time"
import "sync"

var (
  bal_lock sync.Mutex
  balance  int
)

func update(amount int) {
  balance = amount
}
func query() int {
  return balance
}

func main() {
  go func() {
       bal_lock.Lock()
       bal := query()
       time.Sleep(time.Duration(time.Millisecond))
       update(bal + 100)
       bal_lock.Unlock()
  }()
  go func() {
       bal_lock.Lock()
       bal := query()
       update(bal + 1)
       bal_lock.Unlock()
  }()

  time.Sleep(time.Duration(time.Second))
  fmt.Println("Final Balance=", query())
}
```

```
[Module11]$ go run ex11-16.go
Final Balance= 110
```

Module Twelve

Testing and Benchmarking

More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded - indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest.

Boris Beizer

12.1 Testing and Benchmarking in Go

In the first module, we mentioned that Go was intended to be a integrated software production environment which had as one of its design goals the reduction of third party tool dependencies used in the process of building and delivering software.

One of the current best practices in modern software engineering is the use of test driven development and other “test as you go” strategies. The effectiveness of these approaches has led to a wide proliferation of tools like Junit, TestNG, nUnit, RSpec and whole raft of others. Of course these are exactly sort of third party tool dependencies that the developers of Go wanted to eliminate, but not at the cost of reducing the use of the good test supported development practices.

The go tool supports a limited unit test and benchmarking environment implemented as a lightweight automated testing framework. Of course like with any other tool, the Go test utility does not “do testing,” is merely automates your testing activities – if you write bad tests or use testing techniques poorly, automating those does not produce any significant benefit. The actual concepts of how to write unit tests, how to plan a testing strategy and all of the other testing related content is outside the scope of this course. All we will be looking at in this course is how to use the tools.

12.1.1 Tool Capabilities

The go test tool has the following capabilities:

1. *Test functions:* these are special functions that reside in files that are named <filename>_test.go. These correspond to the usual sort of functional unit tests that we would run to see how correct our code is. Functions that implement tests have a specific naming convention and signature we will look at a bit later.
2. *Benchmarks:* These are functions that have names that start with ‘Benchmark’ and which measure performance – more or less our non-functional tests.
3. *Examples;* These are functions that start with ‘Example’ and provides machine checked documentation. Since these are part of the documentation function, we will not be looking at them in this course.

12.1.2 Test Independence

One of the principles of automated testing is that the testing environment should not pollute the production environment with any testing artifacts. Normally the test functions are ignored in the build process, but when "go test" is invoked, an alternative build is run to run the tests, report the results and then remove any traces of the test build.

```
// Example 12-01 Trivial Function To Test
package main

import "fmt"

func count(s string) (vowels, cons int) {

  for _, letter := range s {
       switch letter {
       case 'a', 'e', 'i', 'o', 'u':
            vowels++
       default:
            cons++
       }
  }
  return
}
func main() {

  input := "This is a test"
  v, c := count(input)
  fmt.Printf("vowels=%d, consonants=%d\n", v, c)
}
```

```
[Module12]$ go run ex12-01.go
vowels=4, consonants=10
```

12.1.3 Writing a Basic Test

The function we are going to test is a very badly written and trivial function that counts the number of vowels and consonants in a string. And does it wrong.

Creating the test cases and how we set them up conceptually is outside the scope of this course. However we will assume that there is a correctly constructed set of unit tests that have been developed so that the focus of this module can just be on the automation of the unit testing process.

We will assume that we have two unit tests, one which is the string “a test case” and which should produce 5 consonants and 4 vowels, and the test case consisting of the string “And more stuff” which should produce 8 consonants and 4 vowels.

12.2 Test Automation Framework

There are only a few steps needed to set up the unit tests.

12.2.1 Creating the Test File

For each source file in our directory that contains functions we want to unit test, say a file named “counter.go” for example, we create a corresponding test file named the same but with the suffix “_test” added to the name. For our counter.go file, the file containing the tests would be in counter_test.go.

The test file is in the same package and same directory as the file that it contains test for. There are two different build processes in Go, the normal production build process that ignores all files named like “*_test.go” and a test build process which adds these files and then creates its own main() function entry point to run the tests instead of running the application.

The production build process is called by “go build” and the test build process is called by “go test”.

Each one of the test files must import the “testing” package in order to be able to hook into the automated Go testing framework

12.2.2 Creating the Test Functions

Go relies a lot on the names of things encoding information about what kinds of things they are. In the test files there are test functions that will be automatically called when the test build is executed. These test functions must all be of the form

```
func Testxxx(t *testing.T) {}
```

where the xxx can be any character string you want to make the function name meaningful.

The structure T is a data object that is passed to each test function to maintain test state and the test logs – it is data Go needs to manage the testing process.

It is up to us to write the test logic which usually calls the function under test, gives it the test data and then examines the results to ensure that they are correct. The Go test framework has no idea whether a test has passed or failed unless we tell it. If a test fails, we basically enter that into the test logs maintained in that *testing.T structure, usually by calling the Error() method. The Error() method should be provided with a description of what happened so that when we are reviewing the results, we have enough information to go back and correct the errors that we made.

```
// Example 12-01b Testing the function count

package main

import "testing"

func TestOne(t *testing.T) {
  v, c := count("a test case")
  if c != 5 {
       t.Error("Test One: Expected 5 cons, got ", c)
  }
  if v != 4 {
       t.Error("Test One: Expected 4 vowels, got ", v)
  }
}
func TestTwo(t *testing.T) {
  v, c := count("And more stuff")
  if c != 8 {
       t.Error("Test Two: Expected 8 cons, got ", c)
  }
  if v != 4 {
       t.Error("Test Two: Expected 4 vowels, got ", v)
  }
}
```

Example 12-01b illustrates the implementation of the test functions for the two test cases. Generally we have one test function for each test case.

We can now run the tests by locating to the directory and executing ‘go test’. Doing so produces the following output for out two test cases. Notice that the main() function in our original code is ignored for the test build.

```
[Module12-01]$ go test
--- FAIL: TestOne (0.00s)
     ex12-01_test.go:9: Test One: Expected 5 cons, got  7
--- FAIL: TestTwo (0.00s)
     ex12-01_test.go:18: Test Two: Expected 8 cons, got  11
     ex12-01_test.go:21: Test Two: Expected 4 vowels, got  3
FAIL
exit status 1
FAIL    examples/Module12/ex12-01 0.002s
```

Now corrections are made in the original source code and the test are rerun and the now all the tests pass.

```
// Example 12-02 Corrected Function

package main

import "fmt"

func count(s string) (vowels, cons int) {

  for _, letter := range s {
       switch letter {
       case 'a', 'e', 'i', 'o', 'u':
            vowels++
       case 'A', 'E', 'I', 'O', 'U':
            vowels++
       case ' ':
            break
       default:
            cons++
       }
  }
  return
}
```

```
[Module12-02]$ go test
PASS
ok    examples/Module12/ex12-02 0.002s
```

12.2.3 Test Coverage

Coverage is a term that refers to how well a set of tests cover or exercise our code. Go does provide a basic coverage tool that tell you what percentage of your executable statements actually were exercised during the test execution.

Statement coverage is not a very sophisticated kind of quality measure so it should not be taken as a metric for how good your tests are. However the tool does give you a rough idea of whether or not you are actually running the test you think you are running. For example, if you think you have a full set of test cases and you get 40% coverage, then it suggests that some of the tests are not running or there are some serious problems with the code so resulting in 60% of the code not executing when you may be thinking it is.

One of the more interesting uses of the cover option is to explore code that may be non-essential. If I have a set of unit tests that provide 100% functional coverage (ie. they account for all possible types of input) and I have a cover value of 50%, then I have to wonder what that other 50% of my code is actually doing.

These sorts of metrics are good for getting a general idea but they are not intended to be at the level of a dedicated testing tool. For example, if we compare the coverage of the two test we just ran, we seem to be adding more to the coverage, but we also added more statements so there are a number of factors that will affect the coverage value.

```
[Module12-01]$ go test -cover
--- FAIL: TestOne (0.00s)
     ex12-01_test.go:9: Test One: Expected 5 cons, got  7
--- FAIL: TestTwo (0.00s)
     ex12-01_test.go:18: Test Two: Expected 8 cons, got  11
     ex12-01_test.go:21: Test Two: Expected 4 vowels, got  3
FAIL
coverage: 62.5% of statements
exit status 1
FAIL    examples/Module12/ex12-01 0.002s
```

```
[Module12-02]$ go test -cover
PASS
coverage: 70.0% of statements
ok    examples/Module12/ex12-02 0.002s
```

12.2 Benchmarking

Generally benchmarking is defined as measuring the performance of a function under a fixed workload.

In the Go testing framework, benchmarking functions look a lot like the test functions except that they have the form

```
func BenchmarkXxx(b *testing.B)
```

where B is a struct that is used to track benchmarking information.

By default no benchmarking functions are run by “go test,” they have to be specified on the command line with a regular expression that matches the names of the benchmark functions to be run. Using the pattern /./ will call all the benchmarking functions to be run.

Benchmarks are run in a for loop where the benchmarking program runs the function to be measured b.N times with increasingly large values of N until it seems that the benchmark values have stabilized. How exactly this works is outside the scope for this course.

In example 12-03 we have a standard Fibonacci recursive function which has the interesting property that the larger the value we provide, the length of time to compute the result increases exponentially.

```go
// Example 12-03 Benchmarking a function

package main

import "fmt"

func Fibonacci(n int) int {
  if n < 2 {
       return n
  }
  return Fibonacci(n-1) + Fibonacci(n-2)
}

func main() {
  fmt.Println(Fibonacci(30))
}
```

```
[Module12-03]$ go run ex12-03.go
832040
```

The Benchmarking function below is now run. Notice the use of the b.N as the loop limit so that the metering program can look for a stable extrapolation of the function performance.

```
// Example 12-03 Benchmarking

package main

import "testing"

func BenchmarkFib20(b *testing.B) {
  for n := 0; n < b.N; n++ {
       Fibonacci(20)
  }
}
```

```
[Module12-03]$ go test -bench=.
testing: warning: no tests to run
PASS
BenchmarkFib20-12          30000             40771 ns/op
ok      _examples/Module12/ex12-03 1.728s
```

In the result, the statement BenchmarkFib20-12, the -12 part refers to GOMAXPROCS which is value that controls the number of operating system threads allocated to goroutines in this program. You may see a different number on your machine when you run this.

The middle number is the final value of b.N so in this case, my test ran for 30,000 loops with an average time of of 40771 nanoseconds per loop.

12.3 Profiling

Another aspect of testing related to benchmarking is profiling which is used to measure the performance of critical code by sampling a number of events during the program execution and then extrapolating that sample to produce a statistical summary called a profile. There are several different kinds of profiling that are supported by the test package in Go.

1. A CPU profile identifies the functions whose execution requires the most CPU time.
2. A heap profile identifies the statements responsible for allocating the most memory.
3. A blocking profile identifies the operations responsible for blocking goroutines the longest.

These profiles can be generated by the use of the profiling flags. Example 12-04 demonstrates the use of the CPU profile using the same code that from example 12-03

```
[Module12-04]$ go test -bench=. -cpuprofile=cout.log
testing: warning: no tests to run
PASS
BenchmarkFib20-12          30000             40828 ns/op
ok      _/examples/Module12/ex12-04 1.671s
```

The profile is collected in the file cout.log which can then be examined using the pprof Go tool. This is illustrated on the next page.

Using the pdf output provides a more graphical format if the output of the pprof command in piped into a file to be opened in a pdf viewer.

One of the cautions of using the profiling tools is that one should not try to profile more than one of the values at a time since if more than one profile is selected, the sampling for each profile may interfere with the sampling of the others.

```
[Module12-04]$ go tool pprof -text ex12-04.test cout.log
1.67s of 1.67s total (  100%)
      flat  flat%   sum%        cum   cum%
     1.65s 98.80% 98.80%      1.65s 98.80%  Module12/ex12-04.Fibonacci
     0.01s   0.6% 99.40%      0.01s   0.6%  runtime.(*mspan).sweep
     0.01s   0.6%   100%      0.01s   0.6%  runtime.usleep
         0     0%   100%      1.65s 98.80%  Module12/ex12-04.BenchmarkFib20
         0     0%   100%      0.01s   0.6%  runtime.(*gcWork).get
         0     0%   100%      0.01s   0.6%  runtime.GC
         0     0%   100%      0.01s   0.6%  runtime.findrunnable
         0     0%   100%      0.01s   0.6%  runtime.gcDrain
         0     0%   100%      0.01s   0.6%  runtime.gcMarkTermination
         0     0%   100%      0.01s   0.6%  runtime.gcMarkTermination.func2
         0     0%   100%      0.01s   0.6%  runtime.gcStart
         0     0%   100%      0.01s   0.6%  runtime.gcSweep
         0     0%   100%      0.01s   0.6%  runtime.gchelper
         0     0%   100%      0.01s   0.6%  runtime.getfull
         0     0%   100%      1.66s 99.40%  runtime.goexit
         0     0%   100%      0.01s   0.6%  runtime.mstart
         0     0%   100%      0.01s   0.6%  runtime.mstart1
         0     0%   100%      0.01s   0.6%  runtime.schedule
         0     0%   100%      0.01s   0.6%  runtime.stopm
         0     0%   100%      0.01s   0.6%  runtime.sweepone
         0     0%   100%      0.01s   0.6%  runtime.systemstack
         0     0%   100%      1.66s 99.40%  testing.(*B).launch
         0     0%   100%      1.66s 99.40%  testing.(*B).runN
```

12.3.1 Profiling Non-Test Functions

Dave Cheney has written a nice interface to allow the use of profiling in non test environments in a simple way. The profiling tool he has created is at github.com/pkg/profile. Although we have not used the go get command yet, the example has set up a go workspace with the tool installed. To use the tool, the code is “wrapped” in a profiling function that produces the same sort of output as the seen in the last section.

```
// Profiling non-test functions

package main

import (
  "fmt"
  "github.com/pkg/profile"
)

func Fibonacci(n int) int {
  if n < 2 {
       return n
  }
  return Fibonacci(n-1) + Fibonacci(n-2)
}

func main() {
  defer profile.Start().Stop()
  n := 30
  fmt.Println("Fibonacci num for", n, "is", Fibonacci(n))
}
```

```
[Module12-04]$ go run ex12-04.go
2016/09/22 07:34:59 profile: cpu profiling enabled, /tmp/
profile457278947/cpu.pprof
Fibonacci num for 30 is 1346269
2016/09/22 07:34:59 profile: cpu profiling disabled, /tmp/
profile457278947/cpu.pprof
```

The file produced in this way is analyzed using go tool pprof as before

```
[Module12-04]$  go tool pprof -text ex12-04  /tmp/
profile457278947/cpu.pprof
10ms of 10ms total (  100%)
      flat  flat%   sum%        cum   cum%
      10ms   100%   100%       10ms   100%  main.fib
         0     0%   100%       10ms   100%  main.main
         0     0%   100%       10ms   100%  runtime.goexit
         0     0%   100%       10ms   100%  runtime.main
```

The various parameters to the profiling tool can be set using a configuration structure as demonstrated in this example from Cheney.

```
// Profiling non-test functions

package main
import (
  "fmt"
  "github.com/pkg/profile"
)

func main() {
        cfg := profile.Config {
                MemProfile: true,
                NoShutdownHook: true, // do not hook SIGINT
        }
        // p.Stop() must be called before the program exits
        // to ensure profiling information is written to disk.
        p := profile.Start(&cfg)
        ...
}
```

Module Thirteen

Packages

It's hard to read through a book on the principles of magic without glancing at the cover periodically to make sure it isn't a book on software design

Bruce Tognazzini

13.1 Packages

One of the motivating issue behind the development of Go was the problem of having to manage a number of large builds on a tight schedule. In previous modules we have mentioned packages and worked with them in passing so you should have a sense generally how they work. In this module, we will look at the package and build systems in more depth. We haven't needed the to use build system so far because the examples and labs have been simple enough that we do not need that sort of project organization. But is not generally true out there in the real world and more specifically, for the kinds of large scale projects that motivated the development of Go.

The Go package system bears some similarity to other programming languages so while many of the concepts may be familiar, there are a number of features that are quite unique to Go and which contribute to the speed with which Go programs compile and build.

As Brian Kernighan points out:

> *There are three main reasons for the compiler's speed. First, all imports must be explicitly listed at the beginning of each source file, so the compiler does not have to read and process an entire file to determine its dependencies. Second, the dependencies of a package form a directed acyclic graph, and because there are no cycles, packages can be compiled separately and perhaps in parallel. Finally, the object file for a compiled Go package records export information not just for the package itself, but for its dependencies too. When compiling a package, the compiler must read one object file for each import but need not look beyond these files.*

13.1.1 Package Visibility and Naming

The naming system of packages and symbols follows a set of very simple rules designed to simplify the organization of project source files and project structures. As mentioned earlier in the course, symbols are either public or private in Go. Pubic symbols begin with a capital letter, private symbols begin with a lowercase letter. Symbols referenced in code that belong to a different package have the package name prefixed. This is the full extent of how the visibility is managed in Go.

Package naming

Each package is named by its import path, which is a string that is used by the Go tool (not the compiler) to determine the physical location of the package. While it is not required by the language, the recommended naming for custom packages is to use the organization name as a prefix to the custom package. The protocol is the same for using name spaces in XML, if we are going to reuse or share or otherwise make the package available for general use, we want the package name to be globally unique.

13.2 Workspace Organization

While we can work out of an arbitrary directory structure, as we have been doing in this course so far since we have been working with small examples of code, in order to streamline the use and development of the Go build process and Go tool, there is a standard Go project structure and a set of assumptions made about how a Go project is organized.

These are:

1. Go programmers typically keep all their Go code in a single workspace.
2. A workspace contains many version control repositories (managed by Git, for example).
3. Each repository contains one or more packages.
4. Each package consists of one or more Go source files in a single directory.
5. The path to a package's directory determines its import path.

In other words, there is a single location dedicated to Go projects and each project's source files are under version control. The go tool looks for this workspace location in the GOPATH variable. This means we can have a number of workspaces that we use for Go and we can switch between them by resetting the value of the GOPATH variable.

13.2.1 Project Organization

Within a given project, the following directory structure is expected. Notice that the structure is a generalization of what programmers generally do in terms of organizing their projects.

A workspace is a directory hierarchy with three directories at its root:

`src` A directory that contains all of the Go source files,

`pkg` A directory that contains package objects, and

`bin` A directory that contains executable commands.

Having a standard directory structure for a project allows a simplified implementation of the tool-set rather than having to specify the directory structure in a make file or other kind of description file.

As a point of clarification, the pkg directory does not contain the source for the packges, that is still maintained under the src directory. The pkg directory contains subdirectories generated by the go tool which correspond to the target architecture of a specified platform. In these subdirectories are the compiled binary versions of the packages, what is referred to above as the package objects.

13.2.2 Hello World Example

To illustrate how this works, we will redo the Hello World example from module one as a Go project. The first thing we do is set the Go workspace. In this example, I have chosen “/home/projects/gospace” as my Go workspace. Since I am using a Unix file system, the path names will be in a Unix format, but you can make the appropriate translations to Windows where necessary.

The Go tool uses the GOPATH environmental variable to know where the workspace is located.

```
[Module13]$ export GOPATH=/home/projects/gospace
[Module13]echo $GOPATH
/home/projects/gospace
```

Now we add the hello project to the src directory and run go install hello from anywhere and we get the binary “installed” in the bin directory. Nothing new so far, but now we will add a library or package to the project called stringutil.

We add the package directory stringutils to the src directory and add the file listed on the next page to that directory (this example is from the golang.org website). Remember the code itself is not that important for this example, what is important the organization of the files and dependencies of the packages.

Our project now has the following structure:

```
src\
  hello\
       hello.go
  stringutil\
       stringutil.go
pkg\
bin\
```

```
// Example 13-01 Stringutils package

package stringutil

// Reverse returns its argument string reversed
// rune-wise left to right.
func Reverse(s string) string {
  r := []rune(s)
  for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
       r[i], r[j] = r[j], r[i]
  }
  return string(r)
}
```

The package import statement looks like this in the main package.

```
// Example 13-01 Hello.go

package main

import (
  "fmt"
  "stringutil"
)

func main() {
  fmt.Println(stringutil.Reverse("Hello World!"))
}
```

We can now test out the project by compiling the file and see if compiles correctly. As seen before, we locate to the package directory and run go build. No output is created if the compilation is successful and no executable is created because there is no main package.

```
[Module13]$ cd $GOPATH/src/stringutil
[stringutil] go build
[stringutil]
```

However when we execute go build hello, because of the import statement in the hello.go file, both the hello.go and stringutil.go files are compiled and the executable is deposited in the directory that we ran the build command from.

The go build command is primarily used as a quick check on how well the compilation works. All of the intermediate files are deleted right after so as not to pollute the project environment.This means that the build command will recompile all a file's dependencies even if there have been no changes to any of the dependencies.

```
[Module13]$ cd $GOPATH/src/hello
[stringutil] go build hello
[stringutil] ls
hello.go hello
```

13.2.2 Go install and Hello World

Now if we run the go install command on our hello project, two things happen that are different than what happened with go build.

1. The executable is now "installed" in the bin directory
2. A binary version of stringutil is now "installed" in the pkg directory. A sub directory has also been created in the pkg directory corresponding to the binary platform that the stringutil.go file was compiled to.

Our project now has the following structure:

```
src\
  hello\
       hello.go
  stringutil\
       stringutil.go
pkg\
  linux_amd64\
       stringutil.a
bin\
  hello
```

If we made a change to the hello.go file and ran install again, the Go tool would not re-compile the stringutil.go file since it has not changed since the binary form (stringutil.a) of the package was created. If we have a large project with only a couple of changes in one file, install becomes a lot faster than build since only the changed files are actually rebuilt.

Just as a note, this process also works best when we have a dev environment and a production environment. Once our dev version works and passes its tests, we can move only the changed files to the production environment where install creates the new version in optimal time. The further exploration of that aspect of Go project management is beyond the scope of this course.

13.2.3 Package aliases

Generally, the import path name for the package corresponds to the directory of package. However when we have a multi-part import path, we generally only use the last part of the path name as the prefix in our code. If our stringutil package had been a sub package of a "util" while the import path would be “util/stringutil” we still would have only used the last part as the package prefix for symbols – more specifically, our actual code remains unchanged.

Suppose though that we have two stringutil packages one of which is in $GOPATH/src/srtingutil and the other in $GOPATH/src/util/stringutil, and to make things even more worst case scenario, each one of these has a Reverse function, the only difference is that one just reverses the string and the other also converts the string to uppercase (just so we can tell which one we are calling in the examples).

```
// Example 13-03 util/stringutil package

package stringutil
import "strings'

// Reverse returns its argument string reversed
// rune-wise left to right.
func Reverse(s string) string {
  r := []rune(s)
  for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
       r[i], r[j] = r[j], r[i]
  }
  return strings.ToUpper(string(r))
}
```

Notice there is no path information in the package statement about where the source file is located. The actual location of the package source code is not bound to the code in any way. This means I can relocate my packages wherever I want to without having to modify the package source code.

Suppose that I want to reference each of these Reverse functions. Because of some poor code planning on my part, I have created an ambiguity.

```
// Example 13-04 Hello.go with abiguities

package main
import (
  "fmt"
  "stringutil"
  "util/stringutil"
)

func main() {
  fmt.Println(stringutil.Reverse("Hello World!"))
  fmt.Println(stringutil.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello.go:7: stringutil redeclared as imported package name
  previous declaration at ./hello.go:6
Error: process exited with code 2.
```

Just like using name spaces in XML, we can assign local aliases to packages to remove these ambiguous references.

```
// Example 13-05 Hello.go with aliases

package main
import (
  "fmt"
  s1 "stringutil"
  s2 "util/stringutil"
)

func main() {
  fmt.Println(s1.Reverse("Hello World!"))
  fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
!dlroW olleH
!DLROW OLLEH
```

13.3 Package Initialization

When a file is loaded, we can execute initialization code to do any initialization that cannot be handled through standard declarations. In each file there can be one or more functions named init() (this is the only exception to functions having unique names) which are all executed after:

1. All of the package variable have been initialized
2. All of the init() functions from the imported packages have run.

This is illustrated in the example below where we have added two init() functions to the hello main package.

```
// Example 13-06 Hello.go with init functions
package main

import (
  "fmt"
  s1 "stringutil"
  s2 "util/stringutil"
)

func init() {
  fmt.Println("Main init 1")
}
func init() {
  fmt.Println("Main init 2")
}

func main() {
  fmt.Println(s1.Reverse("Hello World!"))
  fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
Main init 1
Main init 2
!dlroW olleH
!DLROW OLLEH
```

Now adding an init() function into the package util/stringutil and re-running the build on on the same hello project would produce this output.

```
// Example 13-07 Dependent init functions
package stringutil

import "strings"
import "fmt"

func init() {
  fmt.Println("util/stringutil init")

... and so on
}
```

```
[hello]$ go build hello
./hello
util/stringutil init
Main init 1
Main init 2
!dlroW olleH
!DLROW OLLEH
```

According to the golang documentation one of the primary uses of the init() function is to verify and repair program state before execution as in the following example:

```
func init () {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath may be overridden --gopath flag on command line.
    flag.StringVar(&gopath, "gopath", gopath,
                                "override default GOPATH")
}
```

Sometimes we may want the init() function of an importedf package to execute but we don't want to use any of the symbols in the package. To avoid a compile time error, we use the blank variable as a package alias so the compiler will not panic if it can't find a symbol from that imported package.

```
// Example 13-08 Hello.go blank alias
package main

import (
  "fmt"
  "stringutil"
  _ "util/stringutil"
)

func main() {
  fmt.Println(stringutil.Reverse("Hello World!"))

}
```

```
[hello]$ go build hello
./hello
util/stringutil init
!dlroW olleH
```

13.4 The go get Option

Programming languages like Python, R and Ruby all allow for access to an extended set of resources over Internet type connections. The go tool also provides the capability to import packages and utilities from other remote locations.

In this example, we start with a clean project space so we can see the get command at work without any additional clutter. Before we begin, the src, bin and pkt directories are all empty.

Suppose we want to add a utility “golint” to our workspace. The utility is located in a standard Go project structure at https://github.com/golang/lint The “go get” command goes to the specificed location (in this case the url) and fetches the project, then runs “install” on the downloaded package or project.

```
go get github.com/golang/lint/golint
```

Once this command is finished, then the directory tree looks like this

```
bin\
  golint
pkg\
  linux_amd64\
       github.com\
            golang\
                 lint.a
       golang.org\
            x\
                 tools\
                      go\
                           gcimporter15.a
src\
  github.com\
       golang\
            lint\
  golang.org\
            x\
                 tools\
```

There is an additional package that needs to be installed because it is a dependency that is required to make golint, specifically the gcimporter15.a library.

Looking at the source for the lint tool, we can see why this dependency was included.

The go get operation went to the specific repository, downloaded the source code into the scr directory, then downloaded all the dependencies that are required, then compiles all of the source to the appropriate binary package objects.

There is a rich set of options that govern the go get tool which are documented in the go tool documentation.

```
package lint

import (
  "bytes"
  "fmt"
  "go/ast"
  "go/parser"
  "go/printer"
  "go/token"
  "go/types"
  "regexp"
  "sort"
  "strconv"
  "strings"
  "unicode"
  "unicode/utf8"

  "golang.org/x/tools/go/gcimporter15"
)
```

The last line of the import path refers to a local directory that is a copy of a repository at golang.org. When this code is compiled, Go tries to find the specified package locally, then a build error is generated. At this point the go get command has to be used to create the local copy.

13.5 Vendor Management

The go get tool creates a copy of the remote repository in the the local GOPATH workspace which is necessary for build to be repeatable. If the go install tool went out and copied the remote repository automatically, then changes to either the contents or location of the remote repository would cause the code to break. Go avoids this by creating a local copy to work with, a process called “vendoring.”

The import statement

```
import “github.com/golang/examples”
```

Would look only for the directory $GOPATH/src/github.com/golang/examples and if this directory does not exist, then any attempt to build an executable will fail because of a missing dependencies.

13.5.1 Problems with Versions

While this creates a nice clean build environment, there is a versioning problem. You can only have one version of a package in the workspace specified by GOPATH. In the examples we have been seeing, this has not been a problem, but consider the two following cases:

1. Two different developers are using two different versions of the same package.
2. A single developer has two projects, each of which use a different version of the same package.

In both of these cases, if we use go get for get the packages remotely, we generally are going to get the latest versions so that we may find ourselves breaking one of the projects.

13.5.2 Import Rewriting

One way that can be used to use different versions of a package is to do path rewriting, so that the installed path “github/golang/examples” can be changed to something like “github/golang/ver2/examples." A similar rewrite can take place for a different version. Al that needs to be done is to then rewrite any import statements necessary to ensure that we are pointing to the right location.

While this does solve the problem, it is not an effective solution. There is a lot of low level work being done that is tedious, critical and error prone. The whole solution works is quite against the whole philosophy of Go.

13.5.3 Third Party Vendoring Tools

An example of this sort of vendoring tool is godep. We won't exploring the third party tools in the course but it is illustrative to see how one of them typically works.

After executing go get command like

```
go get github/golang/examples
```

then godep snapshots the dependencies by running

```
godep save
```

which creates a json file that looks something like this

```
"ImportPath": "github.com/golang/prog",
    "GoVersion": "go1.6",
    "Deps": [
            {
            "ImportPath": "github.com/golang/examples/ex1",
            "Rev": "e0e1b550d545d9be0446ce324babcb16f09270f5"
            },
            {
             "ImportPath": "ithub.com/golang/examples/ex2",
             "Rev": "a1577bd3870218dc30725a7cf4655e9917e3751b"
                },
)
```

The Go commands are then executed via godep like this:

```
godep go build
```

which does the appropriate fixups.

13.5 The Vendor Folder

One of the innovations of the Go 1.6 release is the use of the vendor folder. The idea is that instead of sharing a common location for an external dependency, each project can declare have a vendor folder in the project tree.

Under the vendor folder is a local copy of an external dependency, however the difference is that when Go is building the project, it will look first in the local vendor folder to resolve the dependency rather than outside the project.

```
src\
  hello\
       vendor\
            stringutil\
                 version 1 code
  bye\
       vendor\
            stringutil\
                 version 2 code
  stringutil\
       version3 code
  whatever\
pkg\
bin\
```

In the example above, when resolving the stringutil dependency, the hello project will import the version 1 code, the bye project will import the version 2 code and the whatever project will import the version 3 code.

Currently most of the third party vendoring tools support the vendor folder.

Module Fourteen

Strings, Patterns and Files

The computing scientist's main challenge is not to get confused by the complexities of his own making.

E. W. Dijkstra

A program is like a poem: you cannot write a poem without writing it. Yet people talk about programming as if it were a production process and measure "programmer productivity"in terms of "number of lines of code produced".In so doing they book that number on the wrong side of the ledger: We should always refer to"the number of lines of code spent".

E. W. Dijkstra

14 Files, Strings and Patterns

This module is an exploration of several of the standard libraries found in Go:

1. **Strings, Characters and Runes:** Since Go uses UTF-8 encoding for strings, there is not a one to one mapping between bytes and characters, as we have seen in a few of the labs, which means that working with strings can be a bit more complicated than in Java or C++. There are several Go libraries that simplify dealing with Go strings which will be explored in this module.
2. **Patterns and Regular Expressions:** Regex is one of the basic tools for pattern matching across programming languages and is supported in Go with regex library which we examine in this module.
3. **File I/O:** We have only seen file I/O in passing in one of the labs so in this module we explore some more of the capabilities of the file I/O utilities in Go.

14.1 Characters, Unicode and Runes

In Go strings are implemented as immutable slices of bytes. There is no constraint on the contents of the individual bytes, they are arbitrary and can even have the numerical value of zero, which does not correspond to any character in any textual representations. One of the reasons of not defining a string to be an underlying array of characters is that by treating it as just a slice of bytes, it can be easily transmitted over any kind of connection without worrying about encoding and decoding – it's just a stream of bytes.

14.1.1 String Literals and Byte Slices

In example 14.1, two strings are defined using a string literal. The string "junk" contains an arbitrary sequence of bytes that is not valid UTF-8 nor is it even valid ASCII text even though it contains one ASCII character and one non-ASCII unicode character. But since strings are arbitrary, we can put whatever we want in them.

The second string "french" contains the UTF-8 string "Côté" which uses UTF-8 encoding for the accented characters. The string "ascii" contains a sequence of ASCII characters.

```
1  // Example 14-01 Basic Strings
2
3  package main
4
5  import "fmt"
6
7  func main() {
8      const junk = "\xbd\xb2\x3d\xbc\x20\xe2\x8c\x98"
9      const french = "Côté"
10     const ascii = "London"
11
12     fmt.Printf("junk =|%s| french=|%s| ascii=|%s|\n", junk, french, ascii)
13     fmt.Printf("Lengths junk=%d french=%d ascii=%d\n", len(junk), len(french), len(ascii))
14 }
15
uild Output
sr/local/go/bin/go run ex14-01.go [/home/gospace/Module14]
nk =|��=� ⌘| french=|Côté| ascii=|London|
engths junk=8 french=6 ascii=6
uccess: process exited with code 0.
```

The interesting part is when we do a comparison of the lengths of the strings. The string "junk" has a length of 8, which is to be expected since we initialized it with 8 bytes of data. The string "ascii" unsurprisingly has the same number of bytes as characters, however the string "french" has a length of 6 but only has 4 characters. This is because that we initialized the string with a sequence of UTF-8 characters, but it was stored as a slice of bytes.

14.1.1.1 String Literals

Just a few quick comments on string literals. As you might expect from the example,

there are actually two kind of string literals: raw string literals and interpreted string literals. From the Go language specification:

> *Raw string literals are character sequences between back quotes, as in `foo`. Within the quotes, any character may appear except back quote. The value of a raw string literal is the string composed of the uninterpreted (implicitly UTF-8-encoded) characters between the quotes; in particular, backslashes have no special meaning and the string may contain newlines. Carriage return characters ('\r') inside raw string literals are discarded from the raw string value.*

> *Interpreted string literals are character sequences between double quotes, as in "bar". Within the quotes, any character may appear except newline and unescaped double quote. The text between the quotes forms the value of the literal, with backslash escapes interpreted as they are in rune literals (except that \' is illegal and \" is legal), with the same restrictions. The three-digit octal (\nnn) and two-digit hexadecimal (\xnn) escapes represent individual bytes of the resulting string; all other escapes represent the (possibly multi-byte) UTF-8 encoding of individual characters. Thus inside a string literal \377 and \xFF represent a single byte of value 0xFF=255, while ÿ, \u00FF, \U000000FF and \xc3\xbf represent the two bytes 0xc3 0xbf of the UTF-8 encoding of character U+00FF.*

To summarize, raw string literals are NOT interpreted so if you want to put UTF-8 characters into a raw string literal, you have to ensure you provide the the actual printed form and not an encoded form since \U will not be interpreted. Interpreted string literals allow the use of the various escaping codes to allow us to enter UTF-8 characters directly or in with the \U Unicode values or the \x hex values.

In example 14-02 (taken from the Go language spec), the same UTF-8 string is initialized using different syntactic forms of the string literal.

```
1  // Example 14-02 String Literals
2
3  package main
4
5  import "fmt"
6
7  func main() {
8
9      const s1 string = "日本語"                                  // UTF-8 input text
10     const s2 string = `日本語`                                  // UTF-8 input text as a raw literal
11     const s3 string = "\u65e5\u672c\u8a9e"                   // the explicit Unicode code points
12     const s4 string = "\U000065e5\U0000672c\U00008a9e"       // the explicit Unicode code points
13     const s5 string = "\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e" // the explicit UTF-8 bytes
14     fmt.Printf("s1=|%s| s2=|%s| s3=|%s| s4=|%s| s5=|%s|\n", s1, s2, s3, s4, s5)
15     fmt.Println("Length of string: ", len(s1))
16 }
17
```

```
Build Output
/usr/local/go/bin/go run ex14-02.go [/home/gospace/Module14]
s1=|日本語| s2=|日本語| s3=|日本語| s4=|日本語| s5=|日本語|
Length of string:  9
Success: process exited with code 0.
```

14.1.1.2 Iterating over Strings

Example 14-03 demonstrates that we can iterate over a string in two ways. If we use a for loop, we will iterate over the underlying byte array. This may be exactly what we need if we want to treat the string as a slice or the purposes of transmission or some other similar sort of processing where we don't care about the meaning of the stream of bytes, but only care about the fact is an array of binary data.

```
1  // Example 14-03 Iteration over strings
2
3  package main
4
5  import "fmt"
6
7  func main() {
8
9      const french string = "Côté"
10     fmt.Println("Iterating with for loop")
11     for i := 0; i < len(french); i++ {
12         fmt.Printf("%x ", french[i])
13     }
14     fmt.Printf("\nIterating with range (%d bytes)\n", len(french))
15     for index, runeValue := range french {
16         fmt.Printf("%#U starts at byte position %d with length\n", runeValue, index)
17     }
18 }
19
```

```
Build Output
/usr/local/go/bin/go run ex14-03.go [/home/gospace/Module14]
Iterating with for loop
43 c3 b4 74 c3 a9
Iterating with range (6 bytes)
U+0043 'C' starts at byte position 0 with length
U+00F4 'ô' starts at byte position 1 with length
U+0074 't' starts at byte position 3 with length
U+00E9 'é' starts at byte position 4 with length
Success: process exited with code 0.
```

However, there are also times when we want to work with the string as a sequence of characters, such as when we are doing some kind of text processing. In this case, using the range operator does the appropriate interpretation of the bytes into their proper Unicode characters.

14.1.2 UTF-8, Runes and Characters

14.1.2.1 Unicode

We are not going to go into a discussion of Unicode in general, that is getting out of scope for the course. Unicode started in 1987 as a joint effort by programmers at Apple and Xerox built on previous Xerox work to extend character encoding to handle multiple alphabets. The result was a 16-bit encoding called Unicode88 which was published in 1988 and states:
Unicode is intended to address the need for a workable, reliable world text encoding. Unicode could be roughly described as "wide-body ASCII" that has been stretched to 16

bits to encompass the characters of all the world's living languages. In a properly engineered design, 16 bits per character are more than sufficient for this purpose.

After a number of other developers and organizations got involved in the project, the Unicode Consortium was established in 1991 and in 1996, Unicode 2.0 was released with a 32-bit standard to allow encoding for all orthographic systems both modern and historic. For example, Egyptian Hieroglyphics has a Unicode representation and even a Klingon Unicode representation was presented as a draft proposal.

One of the goals of Unicode was that textual encodings should be backwards compatible with ASCII so the first 256 Unicode values are the corresponding ASCII values. That means that an ASCII string is always also a Unicode string.

Unicode defines a universal character set in a very rigorous manner. If you are interested in pursuing this topic further, consult the Unicode consortium website at: unicode.org.

14.2.2.2.UTF-8

Unicode defines the values for the characters. Each Unicode encoded character is called a code point and is 32 bits long. Go uses the alias called “rune” of the int32 data type to hold Unicode code points. It would have been possible to define a string as an array of runes, but this would start to create performance issues since, as you recall from module 5, strings are stored in contiguous memory. If most of our textual data is ASCII with only occasional non-ASCII characters, we might start running into issues with all that wasted space in memory.

The encoding for a string that just describes – where each string is made up of fixed with 32 bit Unicode characters – is called UTF-32. Two of the developers of Go – Rob Pike and Ken Thompson – developed a variable length encoding of Unicode strings called UTF-8. Currently UTF-8 is a Unicode standard and is probably the most widely used encoding for web sites (according to Wikipedia as of 2017, 89.1% of all websites used UTF-8).

UTF-8 uses between 1 and 4 bytes to encode each Unicode code point. The basic strategy is to examine each byte (which in UTF-8 are called “code units”) in turn and use the fact that the first 256 Unicode values are also ASCII values. If the high order bit is 0, then that means that this byte represents an 7-bit ASCII character. A one in the first position indicates that the code point has more than one byte, and the the digits that follow describe how many bytes comprise the code point. In the multi-byte representation, all of the byte of the code point following first byte begin with the bits “10..”

The diagram following illustrates this. For and ASCII character, the first bit is 0 so the character uses one byte. This accounts for 128 basic characters. The two byte characters (1,920 to be precise) account for most of the common characters in use in major orthographic systems like Cyrillic and Arabic. In this case the first byte starts with “110..” which means that one byte following the first is needed to complete the encoding. The second byte starts with “10...” to indicate it is part of the encoding along

with the previous byte.

Character Length	First Byte	Second Byte	Third Byte	Fourth Byte
1 Byte	0xxxxxxx			
2 Bytes	110xxxxx	10xxxxxx		
3 Bytes	1110xxxx	10xxxxxx	10xxxxxx	
4 Bytes	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

The three byte characters add most of the symbols used in ideographic and syllabic orthographies used in languages like Korean, Chinese and Japanese. For these characters, the first byte starts with “1110..” which can be read as that the second the third bytes are part of the character, and the second and third bytes also begin with the same “10..”.

The four byte characters which include special symbols like emojis and mathematical notation as well as archaic writing systems follow the patter with the first byte being “11110..” and the following three bytes each starting with “10..”

The actual encoding is a bit more complex that what we have depicted, but this description has been simplified to provide a sense of the basic strategy. An accurate and detailed investigation of UTF-8 is beyond the scope of this class.

One of the problems that we can run into is that we are unable to be precise about what a “character” is. For example, taken from the Go blog, of the Unicode code point representing the lower case grave-accented letter 'A', à?

That's a character, and it's also a code point (U+00E0), but it has other representations. For example we can use the "combining" grave accent code point, U+0300, and attach it to the lower case letter a, U+0061, to create the same character à. In general, a character may be represented by a number of different sequences of code points, and therefore different sequences of UTF-8 bytes.

The concept of character in computing is therefore ambiguous, or at least confusing, so we use it with care. To make things dependable, there are normalization techniques that guarantee that a given character is always represented by the same code points.

We will return to this point briefly in the next section.

To summarize:

1. Go strings are expected to be UTF-8 encoded variable width Unicode characters, however strings can contain arbitrary data – they are not required to be valid Unicode code points.
2. Runes are Unicode “code points” and are always 32 bits wide.
3. We can work with strings either as sequences of runes or as slices of bytes.
4. There may be different UTF-8 representations for a Unicode character.

14.2.3 The unicode and unicode/urf8 Packages

To make working with text easier, Go has two libraries that provide definitions and tools for working with Unicode. These are defined in the Go documentation as:

1. unicode: Provides data and functions to test some properties of Unicode code points.
2. uncode/utf8: Package utf8 implements functions and constants to support text encoded in UTF-8. It includes functions to translate between runes and UTF-8 byte sequences.

14.2.3.1 The unicode Package

The unicode package defines a number of special classes that are used in Unicode. For example defining the names of the languages and defining the sets of Unicode code points that are used in that language. One example is the constant “Tibetan” which identifies all the code points used in writing Tibetan. Other definitions include providing more descriptive names for Unicode classes like diacritics and white space. Exploring all these ranges and groups are beyond the scope of this course.

What is more useful are the predicates that allow us to test what kind of code point a rune actually is. For example functions like IsDigit(), IsLetter(), and IsLower takes a single rune argument and returns a boolean result if the rune is belongs to that class as illustrated in example 14-04.

There are also conversion functions like ToUpper() that do the similar sorts of operations as strings.ToUpper() does for strings in general however the unicode.ToUpper() function can also take into account language specific rules for converting to upper case.

```
11
12      const french string = "語. Côté"
13      for index, runeValue := range french {
14          fmt.Printf("%c starts at byte position %d with length\n", runeValue, index)
15          fmt.Printf("  %c is uppercase? %t\n", runeValue, unicode.IsUpper(runeValue))
16          fmt.Printf("  %c is white space? %t\n", runeValue, unicode.IsSpace(runeValue))
17          fmt.Printf("  %c is letter? %t\n", runeValue, unicode.IsLetter(runeValue))
18      }
19
20  }
```

```
語 is white space? false
 語 is letter? true
. starts at byte position 3 with length
 . is uppercase? false
 . is white space? false
 . is letter? false
 starts at byte position 4 with length
  is uppercase? false
  is white space? true
  is letter? false
C starts at byte position 5 with length
 C is uppercase? true
 C is white space? false
 C is letter? true
ô starts at byte position 6 with length
 ô is uppercase? false
 ô is white space? false
 ô is letter? true
t starts at byte position 8 with length
 t is uppercase? false
 t is white space? false
 t is letter? true
é starts at byte position 9 with length
 é is uppercase? false
 é is white space? false
 é is letter? true
Success: process exited with code 0.
```

14.2.3.2 The unicode/etf8 Package

When we initialize a string with a literal using Unicode, the proper encoding at the byte level is done for us by Go, however very often we will be working with text in other contexts and will have to encode and decode the UTF-8 in code. The functions that are supplied in this package perform a lot of the low level work for us.

For example, we have functions that test is a string is a valid rune, count the number of runes is a string give us the length of a string in runes, and encode and decode runes for us.

Example 14-05 demonstrates some of these functions while example 14-06 demonstrates conversions of bytes to runes and vice-versa.

There is also a UTF-16 package that provides the same functionality for UTF-16 encoding which is the string encoding used by Java.

```go
// Example 14-05 The unicode/utf8 package
// Demonstrating utility functions.

package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {

    const test1 string = "\xff\xff"
    fmt.Printf("%x is valid Unicode? %t\n", test1, utf8.ValidString(test1))

    const test2 string = "Côté"
    fmt.Printf("%s is %d bytes and %d runes long\n", test2, len(test2),
        utf8.RuneCountInString(test2))

}
```

```
Build Output
/usr/local/go/bin/go run ex14-05.go [/home/gospace/Module14]
ffff is valid Unicode? false
Côté is 6 bytes and 4 runes long
Success: process exited with code 0.
```

```go
// Example 14-06 The unicode/utf8 package
// Encoding and Decoding Runes

package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {

    // creating a rune
    buf := []byte{228, 184, 150}
    r, size := utf8.DecodeRune(buf)
    fmt.Printf("Rune r is %c and is %d bytes wide\n", r, size)

    // converting rune to bytes
    buf2 := make([]byte, 3)
    _ = utf8.EncodeRune(buf2, r)
    fmt.Println(buf2)
}
```

```
Build Output
/usr/local/go/bin/go run ex14-06.go [/home/gospace/Module14]
Rune r is 世 and is 3 bytes wide
[228 184 150]
Success: process exited with code 0.
```

14.2 Pattern Matching and Regular Expressions

Regular expressions, or more colloquially “RegEx” is a formal syntax for describing patterns in strings. There are a number of formal standards for regular expressions although the implementation of RegEx in the PERL language tends to be more or less a defacto standard. Most programming languages have some form of RegEx implementation and Go does as well.

We will not be discussing the syntax of regular expressions, that is beyond the scope of the course. If you want to find out how to write regular expressions or learn the syntax of regular expressions, there are a number of excellent books and tutorials available. For the rest of this section, we will just focus on how Go implements regular expressions. As would be expected, all characters in regular expressions in Go are UTF-8-encoded code points.

14.2.1 Compiling the Regular Expression

One of the guarantees that Go makes about its regular expressions is that they are guaranteed to run in time linear in the size of the input. In order to do this, we have to convert a regular expression into a regex object that can be used for efficient pattern matching. In example 14-07 a simple regular expression search is demonstrated. Once the regex object is created, it can be used to search a number of different strings.

```
 1  // Example 14-07 Regular Expressions
 2  package main
 3  import (
 4      "fmt"
 5      "regexp"
 6  )
 7
 8  func main() {
 9
10      target := "Amal Singer: (613) 555-1212"
11
12      pattern := "6[0-9]+3"
13
14      regexObject, err := regexp.Compile(pattern)
15      if err != nil {
16          fmt.Printf("ERROR |%s| is not a valid regex", pattern)
17      } else {
18          fmt.Printf("Pattern /%s/ found in \"%s\" is %t \n", pattern, target,
19              regexObject.MatchString(target))
20          fmt.Printf("Matching string is %s \n", regexObject.FindString(target))
21      }
22
23  }
```

```
Build Output
/usr/local/go/bin/go run ex14-07.go [/home/gospace/Module14]
Pattern /6[0-9]+3/ found in "Amal Singer: (613) 555-1212" is true
Matching string is 613
Success: process exited with code 0.
```

There are a number of variations on the compile functions but all of which do the same basic task – create a regex object. For example CompilePOSIX() returns a regex object that is egrep POSIX compliant. Most of the variations have to deal with creating regex objects that are compliant in various ways with different standards for regular expressions.

The compile methods all return an error object as well which is generated if the regex object cannot be created. The most common reason for an error to be returned is that the string provided to the regex compile() function is not a valid regular expression.

14.2.2 Basic Matching

Example 14-07 also uses a basic matching function MatchString(str)which operates on a string and returns a Boolean value which is true if the pattern is found in the string to be searched. There is also a variation on this called FindString(str) which returns the first sub-string in the argument that matches the regex.

There are a variety of similar sorts of basic matching functions that all differ in minor ways but all do the same basic matching.

For simple matches, we can also use a regular expression directly without compiling it, however in this case, the Match() function can return an error unlike the Match() method on a regex which is guaranteed that the pattern is valid because it has been compiled. One of the reasons for compiling a regular expression though is improved performance and better error avoidance.

```go
// Example 14-08 Regular Expressions without Comiling

package main

import (
    "fmt"
    "regexp"
)

func main() {

    str1 := "Amal Singer: (613) 555-1212"

    pattern := "6[0-9]+3"

    if ok, _ := regexp.Match(pattern, []byte(str1)); ok {
        fmt.Println("Match found!")
    }
}

```

```
Build Output
/usr/local/go/bin/go run ex14-08.go [/home/gospace/Module14]
Match found!
Success: process exited with code 0.
```

Example 14-08 demonstrates the matching done in example 14-07 but without compiling the pattern into a regular expression object. Notice in this example, the regex.Match() is not a method on a regular expression object.

14.2.3 Finding Matches

The basic matching functions are usually enough for most simple applications where we are generally interested only in whether or not a match exists or what the first match is. However, we can perform more complex operations when we need more information about all the matches that take place in the target string.

In example ex14-09, the FindAllStrings(target,num) method returns a slice of strings representing all of the sub-strings in the target that matched the pattern. The integer argument “num” specifies how many matches to return: a -1 means all matches and any positive value n means return only n matches. If there are no matches, then nil is returned. It is possible to provide “0” as the numerical argument so no matches are returned, but that is pretty pointless.

```go
// Example 14-09 Finding Substrings

package main

import (
    "fmt"
    "regexp"
)

func main() {

    target := "Amal 555-1212 : Josh 247-1133 : Vu 998-7734"

    pattern := "[0-9]+-[0-9]+"

    rObj, _ := regexp.Compile(pattern)
    fmt.Println(rObj.FindAllString(target, -1)) // return all matches
    fmt.Println(rObj.FindAllString(target, 2))  // only return two matche
    fmt.Println(rObj.FindAllString("192.168.1.1 logged off", -1))
}
```

```
Build Output
/usr/local/go/bin/go run ex14-09.go [/home/gospace/Module14]
[555-1212 247-1133 998-7734]
[555-1212 247-1133]
[]
Success: process exited with code 0.
```

14.2.4 Replacement

In addition to finding sub-strings, we can also replace them which can be quite useful. For example, suppose that we want to mask out all phone numbers in a file with XXX-XXXX for privacy concerns. This could be a rather tedious programming problem if we had to use only strings. However the method ReplaceAllString(target, str) can replace all sub-strings in the target string with the string str. This is demonstrated in example ex14-10.

One thing to notice is that the replacement string does not have to be the same length as the sub-string that matched the pattern. That means the string returned by the method may be of different length than the original target string.

There are a number of other methods in the package that offer related and more complex functionality for finding and replacing but all methods in the package belong to one of the basic categories of Compiling regular expressions, searching for matches or replacing matches.

```go
// Example 14-10 Replacing Substrings

package main

import (
    "fmt"
    "regexp"
)

func main() {

    target := "Amal 555-1212 : Josh 247-1133 : Vu 998-7734"

    pattern := "[0-9]+-[0-9]+"

    rObj, _ := regexp.Compile(pattern)
    fmt.Println(rObj.ReplaceAllString(target, "XXX-XXXX"))
}
```

```
Build Output
/usr/local/go/bin/go run ex14-10.go [/home/gospace/Module14]
Amal XXX-XXXX : Josh XXX-XXXX : Vu XXX-XXXX
Success: process exited with code 0.
```

14.3 Data-String Conversions

One of the constant issues that we deal with in every programming language is the conversion between string representations of data like “123” and the actual numerical form. When we are dealing with input from various sources, the input may be a string but we want to extract some numeric value from that string. Of course, if we want to do any sort of pattern matching, since regular expressions only work on strings, we have to be able to convert numeric and other data to strings to use the regex package. And naturally, we also have to do conversions to strings for most data types for any kind of output.

The strconv package implements conversions to and from string representations of basic data types.

14.3.1 Basic Conversion

The two most basic conversions are names (following C tradition) “strconv.Atoi()” for “alphabetic to integer” and strconv.Itoa() for “integer to alphabetic. These are demonstrated in example ex14-11. Because these specific conversions are so common, it is assumed the arguments are in base 10 and that the numeric type is the Go int type. The Atoi() method also returns an err if the string cannot be converted to an integer. This is also demonstrated in example ex14-11.

```go
// Example 14-11 Atoi and Itoa

package main

import (
    "fmt"
    "strconv"
)

func main() {
    i, _ := strconv.Atoi("-42")
    s := strconv.Itoa(-42)
    fmt.Printf("i is of type %T, s is of type %T\n", i, s)

    i, err := strconv.Atoi("-42 degrees")
    if err != nil {
        fmt.Println("Atoi method failed")
    }
}
```

```
Build Output
/usr/local/go/bin/go run ex14-11.go [/home/gospace/Module14]
i is of type int, s is of type string
Atoi method failed
Success: process exited with code 0.
```

14.3.2 Formatting Functions

The FormatBool(), FormatFloat(), FormatInt(), and FormatUint() convert numeric and Boolean values to strings. For the integer and unsigned integer, the base of the string representation is provided. This is illustrated in example 14-12. Notice that since the argument for the FormatInt() function is an int64, we have to convert any integral value to an int64 before formatting. Unsigned integers work the same way.

Because floating point numbers are a bit more complex in terms of their formatting, the FormatFloat() function can specify the precision, size and exponent format. A detailed discussion of all of those options is beyond the scope of this section and is about as boring as anything you can imagine. If you are interested in the details, consult the Go documentation. There are several examples in example ex14-12. The formatting of a Boolean value is also demonstrated in example ex14-12.

```go
// Example 14-12 Formatting Numerics

package main

import (
    "fmt"
    "strconv"
)

func main() {

    var i int32 = 255
    var f float64 = 3.14159
    fmt.Println(strconv.FormatInt(int64(i), 10))
    fmt.Println(strconv.FormatInt(int64(i), 16))
    // -1 means use the smallest precision possible
    fmt.Println(strconv.FormatFloat(f, 'E', -1, 32))
    fmt.Println(strconv.FormatFloat(f, 'f', 10, 64))
    fmt.Println(strconv.FormatBool(true))
}
```

```
Build Output
/usr/local/go/bin/go run ex14-12.go [/home/gospace/Module14]
255
ff
3.14159E+00
3.1415900000
true
Success: process exited with code 0.
```

14.2.3 Parsing Functions

Going the other way, the parsing functions convert a string, if possible, into a specified numeric type. This means that we have to provide a few parameters to tell the parsing function how to interpret the string. We need to specify the base the string represents. If the base value is “0”, then the base is deduced from the prefix (ie. “0x” or “o”) if it is present. We also have to provide the size of the integer that the result should produce.

There are two kinds of errors that can be returned. The first error is if the string does not represent an integer. The second is that the size of integer variable specified is not large enough to contain the parsed result. However, the result returned is always an int64.

Example ex14-13 illustrates this.

```
 1  // Example 14-13 Parsing Numerics
 2  ...
 3  func main() {
 4
 5      var s1 string = "7386"
 6      var s2 string = "One Hundred"
 7      var s3 string = "0xff"
 8      var s4 string = "7889272162938"
 9      if i1, err := strconv.ParseInt(s1, 10, 16); err == nil {
10          fmt.Printf("%T, %v\n", i1, i1)
11      }
12      if i1, err := strconv.ParseInt(s2, 10, 16); err == nil {
13          fmt.Printf("%T, %v\n", i1, i1)
14      } else {
15          fmt.Println("Parse Error")
16      }
17      if i1, err := strconv.ParseInt(s3, 0, 16); err == nil {
18          fmt.Printf("%T, %v\n", i1, i1)
19      }
20      if i1, err := strconv.ParseInt(s4, 10, 16); err == nil {
21          fmt.Printf("%T, %v\n", i1, i1)
22      } else {
23          fmt.Println("Parse Error")
24      }
25  }
```

```
Build Output
/usr/local/go/bin/go run ex14-13.go [/home/gospace/Module14]
int64, 7386
Parse Error
int64, 255
Parse Error
Success: process exited with code 0.
```

14.4 File I/O and Terminal I/O

The basics of file and terminal I/O in Go are very similar to the way these operations are done in most C-style languages.

14.4.1 Console stdin Input and String Scanning

In keeping with how UNIX manages console input and output, C allowed access to the standard files stdin, stdout and stderr, which has now become fairly standard in most C or UNIX programming languages, and Go is no exception.

User input can be captured from the keyboard or standard input, which is accessible as os.Stdin. As with C, the simplest way is to use the Scanxx and Sscanxx type functions.

Scanln() reads from os.Stdin and tokenizes the input which means it stores successive white-space delimited strings values into successive arguments and stops scanning at a newline. Scan() does the same but treats the newline character as just white space.

```
1  // Example 14-14 Scanln() and Scanf()
2
3  package main
4
5  import (
6      "fmt"
7  )
8
9  func main() {
10
11     var firstName, lastName string
12     var age int
13
14     fmt.Println("Please enter your full name: ")
15     fmt.Scanln(&firstName, &lastName)
16     fmt.Println("Please enter your age: ")
17     fmt.Scanf("%d", &age)
18     fmt.Printf("Hi %s %s! you are %d years old \n", firstName, lastName, age)
19 }
20
```

```
Build Output
/usr/local/go/bin/go run ex14-14.go [/home/gospace/Module14]
Please enter your full name:
Albert Einstein
Please enter your age:
187
Hi Albert Einstein! you are 187 years old
Success: process exited with code 0.
```

Scanf() does the same, but the first argument is a Printf() style format string to “interpret” the string tokens being read so they can be assigned directly to variables of a particular data type. This is illustrated in example ex14-14. Notice that we use the address of the variable as locations where we are copying the data.

In example 14-15, we see the Sscanxx() analogues where we are reading from a string as opposed to stdin.

```
1  // Example 14-15 Sscanln() and Sscanf()
2
3  package main
4
5  import (
6      "fmt"
7  )
8
9  func main() {
10
11     var firstName, lastName string
12     var age int
13     var nameInput string = "Albert Einstein"
14     var ageInput string = "100"
15
16     fmt.Sscanln(nameInput, &firstName, &lastName)
17     fmt.Sscanf(ageInput, "%d", &age)
18     fmt.Printf("Hi %s %s! you are %d years old \n", firstName, lastName, age)
19 }
```

```
Build Output
/usr/local/go/bin/go run ex14-15.go [/home/gospace/Module14]
Hi Albert Einstein! you are 100 years old
Success: process exited with code 0.
```

14.4.2 Opening and Closing Files

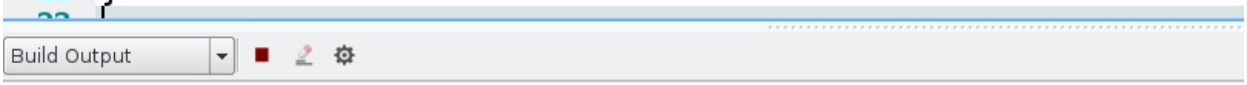
Files in Go are represented by pointers to objects of type os.File. These objects are essentially what is implemented in most operating systems as filehandles, which usually are blocks of data containing OS level information about the file that is necessary to access it. .In this UNIX derived model, the input and output streams are just two files called stdin and stdout, they just happen to function a bit differently than the usual files because of the interactive nature of the console interface.

In Go, a structure, called a file object, is created to hold the data needed to use the file and can be though of as a high level connection to the underlying filehandle. Once a file has been opened and the file object created, we can work with the contents of the file as a string object without having to deal with the low level raw bytes of the file.

To create the file object, a call to os.Open(filename) has to be made. As you would expect, if the file doesn't exist or cannot be opened (insufficient rights for example) the appropriate error is generated.

Files that are opened should also be closed. In Go, the typical way we do this is to use defer to ensure the file is closed properly no matter what happens during the file operations. This is illustrated in example ex14-16. Notice that because the file is opened successfully, the output is the file object that was created during the open operation.

```go
// Example 14-16 Opening and Closing Files

package main

import (
    "fmt"
    "os"
)

func main() {

    filename := "input.txt"

    myFile, err := os.Open(filename)
    if err != nil {
        fmt.Printf("Cannot open the file %s for input\n", filename)
        return // exit the function on error
    }
    fmt.Println(myFile)
    defer myFile.Close()

}
```



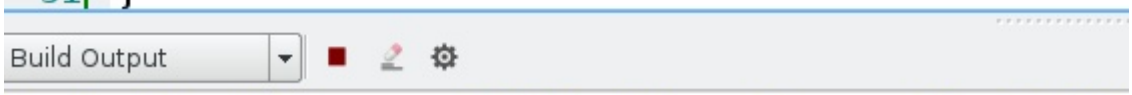

```
/usr/local/go/bin/go run ex14-16.go [/home/gospace/Module14]
&{0xc42000a520}
Success: process exited with code 0.
```

14.4.3 Reading and Writing

The bufio package allows us to wrap the file object in an interface that gives us access to the string like functionality that was mentioned in the previous section.

There are a couple of different ways that we can read from a file through a file Reader interface. The first is to read the file line by line with the ReadString() where each line is terminated but a new line (\n) character. This can produce problems with the Windows \r \n line termination. When the end of the file is encountered, the io.EOF error is generated which can be used to terminate the read. This is illustrated in example ex13-17.

```go
// Example 14-17 Buffered ReadLine()

package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)

func main() {

    filename := "input.txt"

    myFile, err := os.Open(filename)
    if err != nil {
        return
    }
    defer myFile.Close()

    reader := bufio.NewReader(myFile)

    for {
        line, err := reader.ReadString('\n')
        if err == io.EOF {
            return
        }
        fmt.Printf("File line: %s", line)
    }
}
```

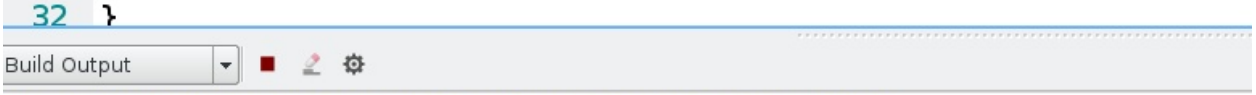



```
/usr/local/go/bin/go run ex14-17.go [/home/gospace/Module14]
File line: This is the first line in the file
File line: This is the second line in the file
Success: process exited with code 0.
```

Another method, which works for Windows as well is ReadString(delim) which reads from a file until the delimiter “delim” is encountered. There is also a similar function that can be used to read arbitrary data ReadBytes().

For non textual data or data without set delimiters, we can do a raw Read() from the file into a buffer of bytes. The method returns the number of bytes read or 0 if the EOF is reached. This is demonstrated in example 14-18.

```go
func main() {

    filename := "input.txt"

    myFile, err := os.Open(filename)
    if err != nil {
        return
    }
    defer myFile.Close()

    reader := bufio.NewReader(myFile)
    buffer := make([]byte, 1024)

    for {
        n, err := reader.Read(buffer)
        if err == io.EOF {
            break
        }
        fmt.Printf("File (%d bytes) is:\n%s", n, string(buffer))
    }
}
```

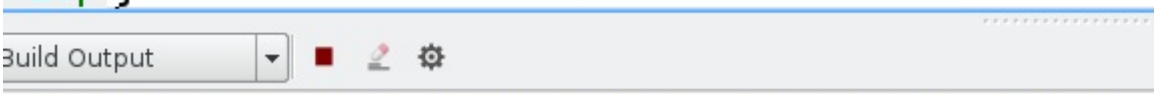



```
/usr/local/go/bin/go run ex14-18.go [/home/gospace/Module14]
File (71 bytes) is:
This is the first line in the file
This is the second line in the file
Success: process exited with code 0.
```

Finally,we can read an entire file in one shot into a buffer consisting of slice of bytes as demonstrated in example ex14-19. The function ReadFile is found in the package io/ioutil. Notice that this function does not require the file to be opened or the use of a file reader interface.

Writing to files will be covered in the labs.

```go
// Example 14-19 ReadFile()

package main

import (
    "fmt"
    "io/ioutil"
)

func main() {

    filename := "input.txt"

    buffer, err := ioutil.ReadFile(filename)
    if err != nil {
        fmt.Println("Unable to read from file")
        return
    }
    fmt.Printf("File is: %s", string(buffer))
}
```




```
usr/local/go/bin/go run ex14-19.go [/home/gospace/Module14]
ile is: This is the first line in the file
his is the second line in the file
uccess: process exited with code 0.
```