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

Introduction to Programming in Go

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.

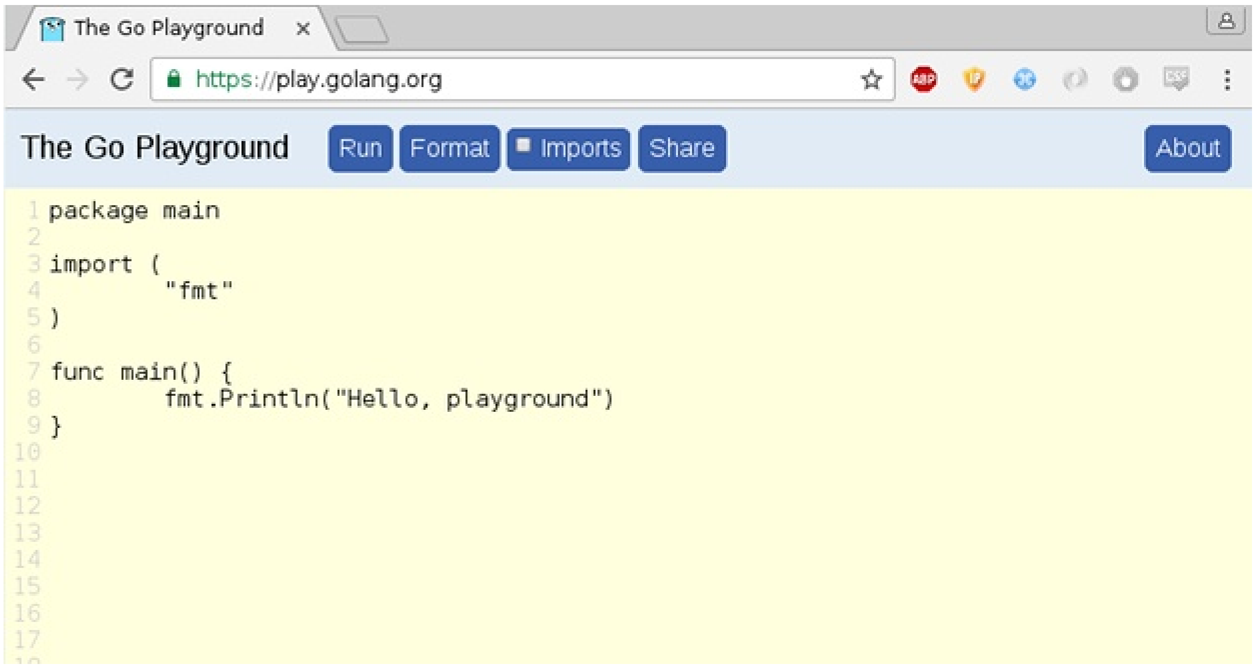
If you do choose to use the LiteIDE, installation of the LiteIDE is quite simple.

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/



```go
package main

import (
	"fmt"
)

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

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

```go
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

```go
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.