An Overview of Go

CSE 486/586: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering

University at Buffalo

Writing Go

Go looks a lot like other languages (C, Java, *etc.*).

Go is not those languages.

If you pretend Go is Java (or C, or …), it will be difficult.
(This is true of other languages, as well)

Meet Go on its own terms and you will find it easier.

Ask questions about the topics in this lecture!

Go is Unforgiving

Unused variable? Won't compile.

Sloppy typing? Won't compile.

Duplicate declaration? Won't compile.

Wrong letter case? Won't compile.

Read error messages carefully and follow the rules.

Idiomatic Go

Go has many idioms.

Idiomatic language is how something is normally expressed.

Natural language has idioms:
If you see eye to eye with Go, it will be a piece of cake.

Programming idioms are commonly-used “phrases”.

```
if err := mp.Send(buf); err != nil {
        // handle error
}
```

Go Modules

A Go module is an installable unit.

It might be a program or a library.

Each of our projects is a module.

The `go.mod` file gives the module name and its dependencies:

```
module cse586.messageservice

go 1.19

require google.golang.org/protobuf v1.28.1
```

Packages

Each module contains packages.

All packages in a module start with the module name.

E.g., `cse586.messageservice/api`:

- Module `cse586.messageservice`
- Package `api`

Every `.go` file must have a `package` statement.

Packages correspond to directory names.

Arrays and Slices

Go has arrays much like arrays in Java.

```
var a [32]int //Array of 32 ints
```

Arrays are fixed in size and bounds checked.

Go also has slices, which are views into an array.

The array has a fixed size, but the slice length can change.

Slices are also bounds checked.

Making Slices

A slice can be taken from an array:

```
a := [32]int
s := a[:]
```

A slice can be allocated directly:

```
s := make([]int, 32)
```

A slice can be taken from a slice:

```
s1 := make([]int, 32)
s2 := s1[0:16]
```

Slice Length

Many Go functions and methods operate on slices.

Often the slice length is meaningful.

For example, `Read()`:

- accepts a `byte` slice
- attempts to read the slice length in bytes

Read 4 bytes into a 1024 byte buffer:

```
var buf [1024]byte
os.Stdin.Read(buf[:4])
```

Read the docs!

Maps

Go maps are like Python dictionaries.

Maps can only be created with `make`:

```
m := make(map[string]string)
```

Maps are unordered.

A map will resize itself as necessary.

Ranges

A range expression iterates maps, arrays, slices, strings, and channels.

```
for index, value := range variableName {
        // index is:
        //   key for maps
        //   array index for arrays
        //   slice index for slices
        //   unicode character position for strings
}
for value := range channel {
        // No index for channels!
}
```

Strong Typing

Go is strongly typed.

New types can be created with `type`:

```
type IntAlias int
```

Even structurally identical types are distinct:

```
var i int = 0
var ia IntAlias = i
```

```
cannot use i (type int) as type IntAlias in assignment
```

Structures

Go structures are sort of like C structures.

They can have both public and private members.

They can embed other structs.

```
type AStruct struct {
        privateField int
        PublicField string
}

type AnotherStruct struct {
        AStruct
        AnotherField []byte
}
```

Methods and Interfaces

We will cover these in detail later.

Methods provide object-like semantics to any non-interface type.

Interfaces provide polymorphism and encapsulation.

The empty interface (`interface{}`) is like C `void *` or Java `Object`.

Go Pointers

Pointers in Go are much like C pointers.

Go tries to make them safer, but they can still be abused.

You can create a pointer with &.

You can dereference a pointer with * or ..

There is no -> operator in Go.

Allocation and Reference Safety

Go is garbage collected, there is no `free()`.

Objects can be allocated with `new()`.

Local variables can be returned as pointers with &:

```
var i int = 42
return &i
```

Static initializers can have their address taken:

```
type Query struct { question string, answer int }
pq := &Query{"life, the universe, and everything",
   42}
```

Method Polymorphism

Go methods are polymorphic over receiver type.

More than one receiver can implement the same method name.

A method may accept different arguments on different receivers.

Only one method of a given name can be defined for a given receiver type.

Argument Polymorphism

Interfaces provide polymorphism for arguments.

If an argument requires an interface type, any implementation of that interface satisfies the argument.

Functions and methods are not polymorphic by signature.

This is used heavily in the Go standard libraries.
`Reader`, `Writer`, *etc.* are common.

Summary

- Go is unique, meet it on its own terms
- Go is a picky language
- Idioms are worth learning
- Go uses structural (“duck”) typing
- Go provides polymorphism through
  - Methods
  - Interfaces

Next Time …

- Our model of distributed systems

References I

Required Readings

[1] *Effective Go*. URL: https://go.dev/doc/effective_go.

To retrieve a copy of this material, or related materials, see `https://www.cse.buffalo.edu/~eblanton/`.