# Lecture 2 - Why Go, Learning a Language

More on Why Go - https://youtu.be/tGALot8nYXI
Learn Go - https://youtu.be/sxc-KS38WTk
Go Main Program - https://youtu.be/tHGRdyL2kU4
CLI Input and File Input - https://youtu.be/Qelw-XAsTvI
JSON Format - https://youtu.be/Qelw-XAsTvI
Read JSON data - https://youtu.be/VhDXh8YvLC8
Modules in Go and testing - https://youtu.be/spG8u6-FiAM

From Amazon S3 - for download (same as youtube videos)

More on Why Go
Learn Go
Go Main Program
CLI Input and File Input
JSON Format
Read JSON data
Modules in Go and testing

## Why Go?

Why use Go? Why is Go an important language to know?

A little bit of history on computer architecture.

About 1979 Bell Labs folks were porting the original vision of Unix from a PDP-7 to a PDP-11. This meant that they were going from 8k words of memory to 32k works of memory. The computer had a single CPU processor.

Computer languages were designed to reflect this “single processor” world. C, became the most popular language and is reflected in the syntax and constructs of other languages like C++, Java, JavaScript (ECMA Script 262) and a wide variety of other programming languages. Java’s virtual machine (JVM) was build as an emulator of a single processor system. There are fundamental assumptions in Java and many other programming languages that you are running on a single processor system.

By 2000 (30+ years later) multi processor systems were starting to be available. C was still the top language.

In 2020 you can get a server wit 96 CPU’s and 1 terabyte of memory for around $50k.

So we have 1-cpu designed languages with 96-cpu machines. For some applications in Java like IBM WebSphere and Python's threading package you can run 2 threads on each CPU. It is very hard to get threads to not step on each other - the code requires a LOT of effort to get correct. In the case of the JVM and the python internal "machine" there are assumptions that you are on a single CPU. For other languages that are compiled like C the assumption is at the compiler level.

The reason that I bring up Java and the JVM is that there are lots of languages that run on the JVM like Closure, Jython, Kotlin. I like and use some of these languages.

Three Languages have adapted or been designed to work in this multi-cpu world. JavaScript (node.js) is the weakest of the three. It allows for callbacks but still assumes that you are on a single CPU. This gives you asynchronous IO but fails to address other concurrency problems.

Rust addresses a number of the concurrency problems head on. It is a hard to learn language. This is because it detects all merry leeks at compile time. Rust (if you take the time to get good at it) is the fastest and best language.

Go is a simple easy to learn language. It addresses memory leeks with a concurrent garbage collector. This is really important. It addresses multi-CPUs with a built in set of super-light weight threads called Go-Routines. With JVM on a 96 CPU box we will get our best results with about 180 threads. With Go on the same box we can get 5 to 20 times the throughput with 25 to 100 million concurrent Go-Routines. This difference in performance is that the language handles the concurrency and the garbage collector.

The situation for multi-CPU is about to get much more dramatic. Apple M1 silicon has for it's low end MacBook Air a 4-fast,4-efficient CPU design. Apple is working on the M1X for it's high end laptops with a 20 CPU design and for its desktops a 32 to 128 CPU design. This is in your desktop computer! Amazon Web Services (AWS) is working on a 65 to 256 CPU design for servers. The same $50k U-6 box that you can get with 96 CPUs AWS will be building with 100+ 256CPU chips and 20 Tb of memory. The power consumption will be less that $\frac{1}{2}$ of current boxes - cutting the cooling (Air Conditioning) requirements in $\frac{1}{2}$. This meas that it will take $\frac{1}{4}$ the power to run. So you get a 25000 CPU box for $50K.

Single CPU languages like the JVM ones and others that rely on threads are going to slowly disappear.

The exception to this is C. C was designed to build supper efficient systems code for small CPU's. Remember that the first version of Unix ran on a PDP-7 with 8k words of memory. That was OS and the users progam in 8k. The rise of IoT systems like the Linux system in the light bulb above my head are ideal for a language like C. It is a single-CPU with only 64k of memory. This is why we see a resurgence in C programming.

And... yes - I can login to my light bulb and use the vi` editor on it. It is running a 32 bit ARM based CPU and I can cross compile C code and run it on my light bulb.

For server based systems languages that address this multi-CPU world are going to dominate. That is Go, Rust and node.js at the moment. Since Rust is hard to learn and node.js is only a ½ solution to the multi-CPU problem - we are going to learn Go.

We will be using Go for the first ½ of the semester - then move on to building some smart contracts in a JavaScript based language (Solidity). It is possible that we will do some work with IOHK's smart contract language - I am working on that.

So for this semester Go, some HTML/CSS/JavaScript, then a second or third language to tackle. The most important thing to walk away from this is - *How to Learn a Programming Language*

## How to Learn a Programming Language

There are a bunch of fundamental concepts in most languages. This is not how a language like Haskell or Closure works. Pure functional languages are quite different. This is how most C-Derived languages work. Go falls into this category. When you get this set of stuff down you know enough to write simple programs in the language. This list applies to Go, Kotlin, Java and 500 other languages. Integrated Development Environments (IDSs) go to huge lengths to hide some of these details - but usually fail. I am going to work through this list both at the command line using vi and using Visual Studio Code - an IDE. VSCode has the best Go debugger. On a daily basis I use go at the command line with VI as my primary editor. Remember all of those IoT devices. Most of them have a command line and a vi editor. Also note that I maintain systems that are remote - in Texas, Utah, Tokyo Japan and other locations. You cant use a GUI like VSCode on a remote system because it assumes that you have a mouse. A mouse is a real-time hardware system and you have to be physically close to the system for it to work. Between Wyoming and Tokyo the fastest communication is the speed of light - and about every 8 inches is 1 millionth of a second. There are a lot of 8 inch distances to the other side of the world. A mouse is basically unusable on a Graphical User Interface at that distance. Vi and a command line work just fine.

1. How to install / run / compile code (Hello World)
2. How to test code.
3. How are files organized
4. What is the main program
5. Objects or Interfaces
6. Functions or Methods
7. Package Management
8. if else

9. Loops
10. How data is passed to functions
11. Maps/Dictionaries
12. Structures
13. File IO
14. Network IO
15. What is special about this language.

## What is special about Go?

Go uses interfaces not objects. This is better system than object oriented programming. You can use interfaces to build object like code - but why bother? The interface based code is a better solution that is easier to understand and test.

Go has the worlds best garbage collector. Letting the language do the memory management turns out to be a huge productivity boost. You can cut development in time in ½ by using a garbage collector. The real advantage of Java over C is not the object oriented - it is the garbage collector. C++ shows that you can have objects in a C like language.

The problem with the JVM (and most other systems like Python) garbage collector is that you can not use the entire machine. In a realistic world you can use 20% of the machine so that there is enough left over when you garbage collect to that your application will not stall during garbage collection. Two years ago Java improved it's garbage collector and now you can use about 27% of the machine. Go's garbage collector let's you use 80% to 90% of the macing - and there is no noticeable lag when it garbage collects. Inf fact if you don't produce enough garbage you will get weird performance out of Go. This is an invisible but valuable feature of the language.

Go has built in concurrency and concurrency controls. This moves concurrency from a hot-patch like threads into the language itself. It is practical to run millions of Go-Routines at once. Go also has locks and communication channels. This let's you build fast reliable systems. Companies like Netflix and CloudFront have converted to all Go back ends.

This last important feature of Go is that it is really fast to compile. I can use Maven and build a Java 250,000 line system - when I re-compile I can probably go to lunch and come back and it will still be working on the compile. Go uses a modern compiler design and it will compile and link a 250,000 line progam in under 2 seconds. This has a huge impact on programmer productivity.