FUNDAMENTALS OF

GO FOR WEB DEVELOPERS

MAXIMILIANO FIRTMAN

MAXIMILIANO FIRTMAN

MOBILE+WEB DEVELOPER

HTML since 1996

JavaScript since 1998

AUTHOR

Authored 13 books and +70 courses

Published +150 webapps

@FIRT · FIRT.DEV

What we'll cover

- What's Go
- Fundamentals
- Standard Libraries
- Goroutines
- fmt Package
- http Package
- Templating
- Web Servers and Services

Pre-requisites

Go Runtime

Visual Studio Code

Code samples:

GitHub.com/firtman/go-fundamentals

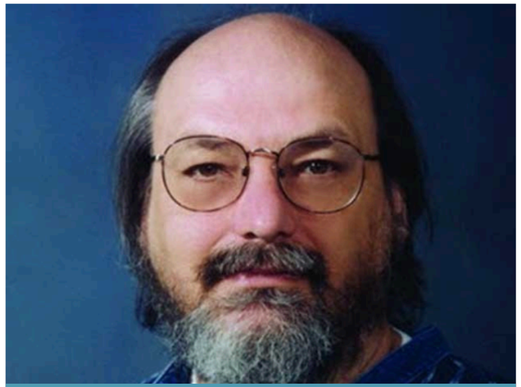
Questions?

What's Go?

What's Go?

- Created by Google in 2011
- Language created from scratch
- Open Source
- It's not a Google's product
- Multi-purpose
- Multi-platform
- Naming issues
- Golang?

Who are the creators?

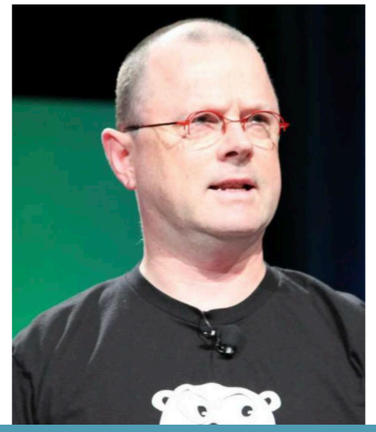


Ken Thompson

B and C languages

Unix

UTF-8



Rob Pike

Unix

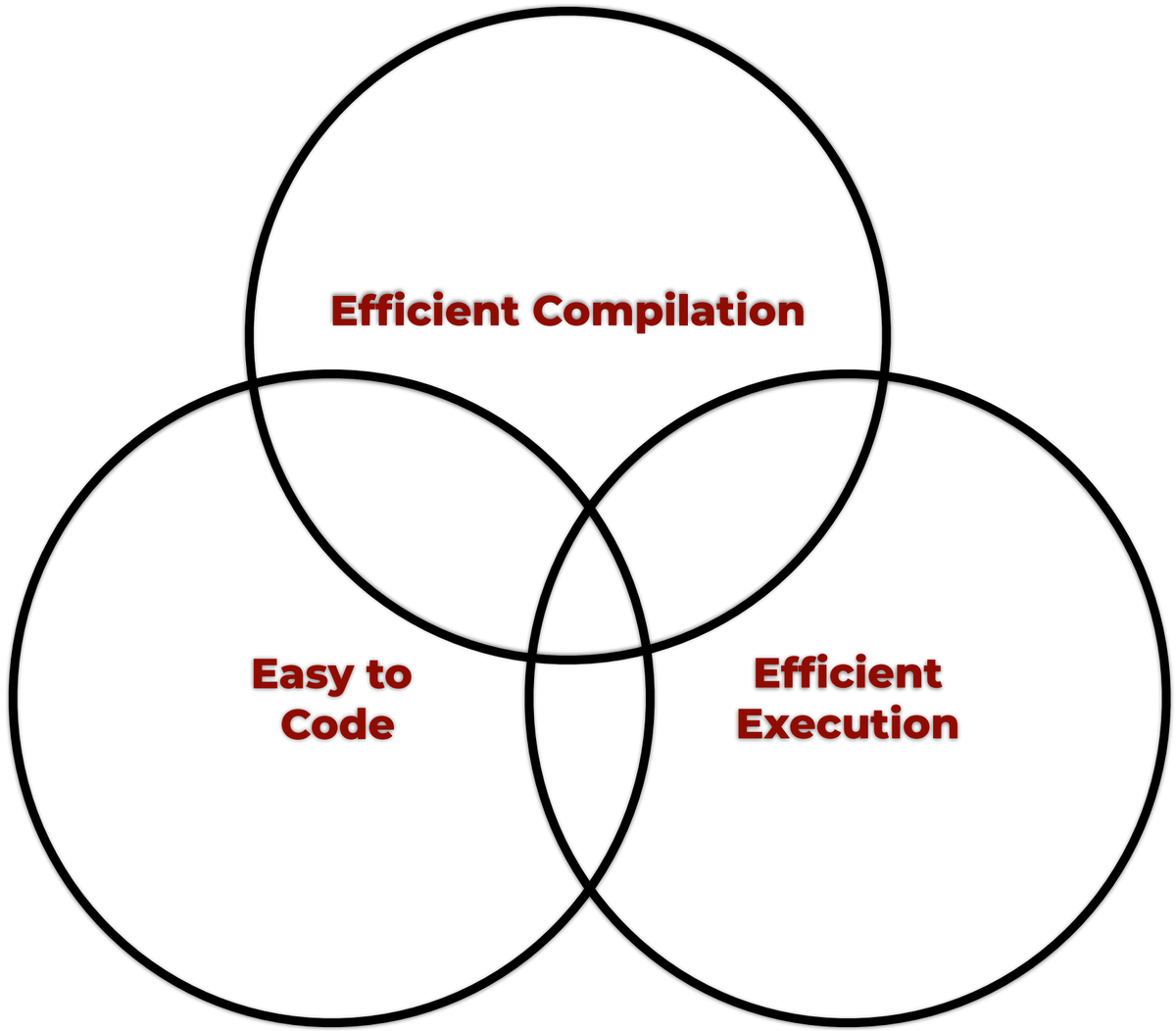
UTF-8

Google

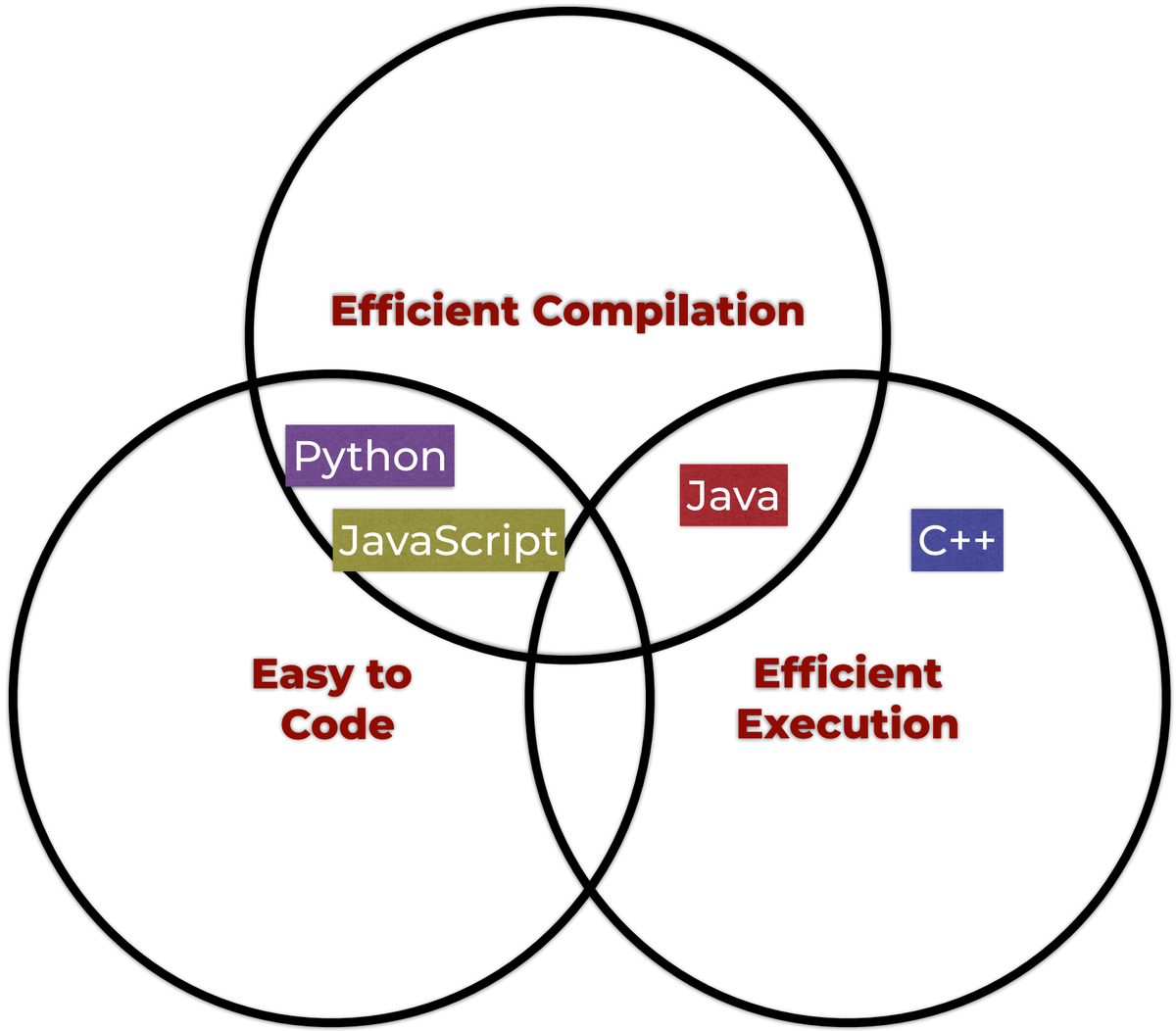
Robert Griesemer

Hotspot Java Virtual Machine

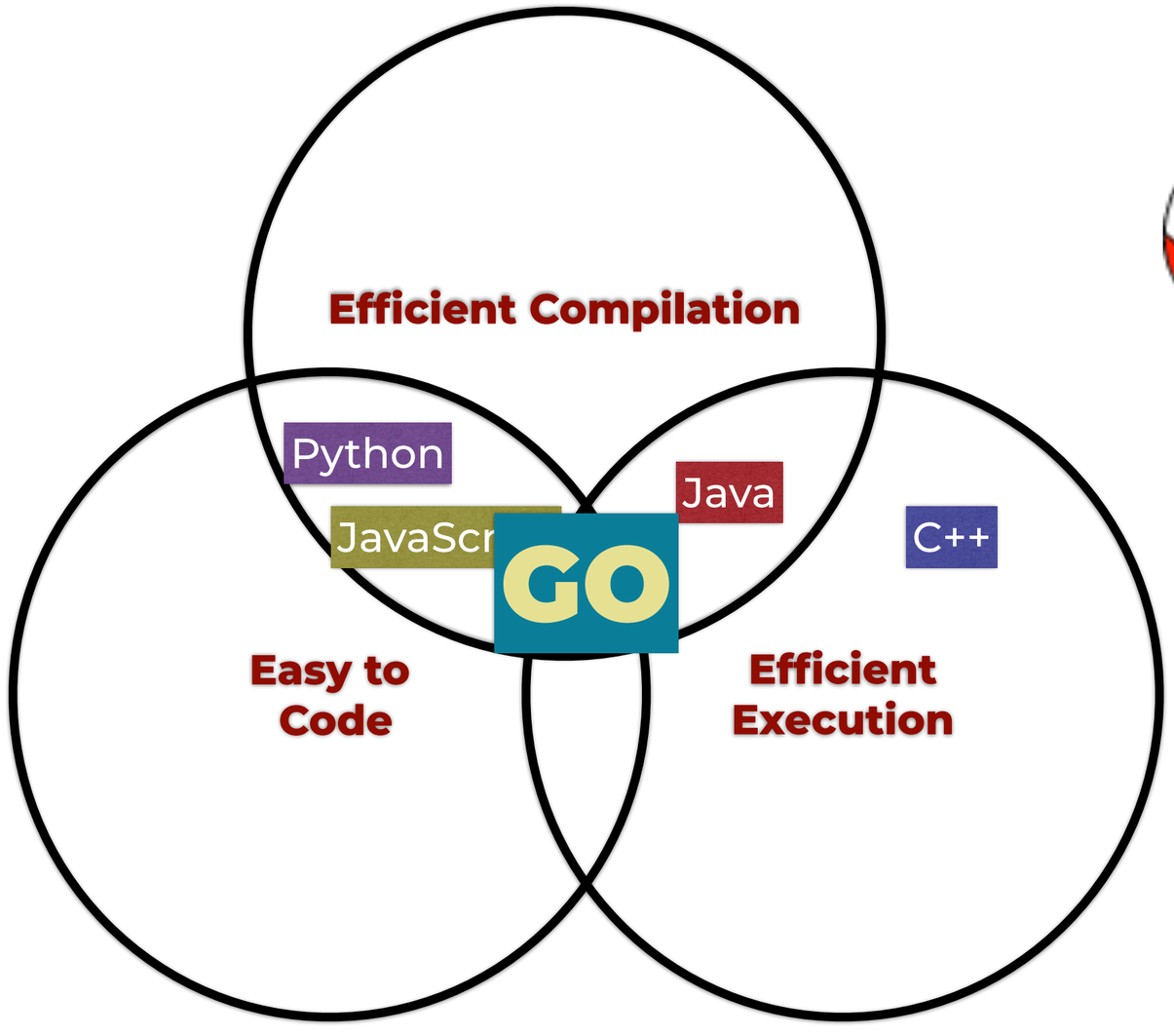
Why Go?



Why Go?


Efficient Compilation
Python
JavaScript
Java
C++
Easy to Code
Efficient Execution

Why Go?


Efficient Compilation
Python
Java
JavaScr
GO
C++
Easy to Code
Efficient Execution

Characteristics

- Strong, Static Type System
- C-inspired syntax
- Compiled
- Multi-paradigm
- Garbage-collected
- Fast
- Single binary compilation

Timeline

- 2007: Start of the project
- 2009: Publicly announced
- 2012: Go 1.0 released
- .....
- 2023: Go 1.20 released
- ???: Go 2.x?

Who is using Go?

Google

PayPal

American Express

Capital One

CLOUDFLARE

FACEBOOK

Microsoft

WILD LIFE

NETFLIX

RIOT GAMES

salesforce

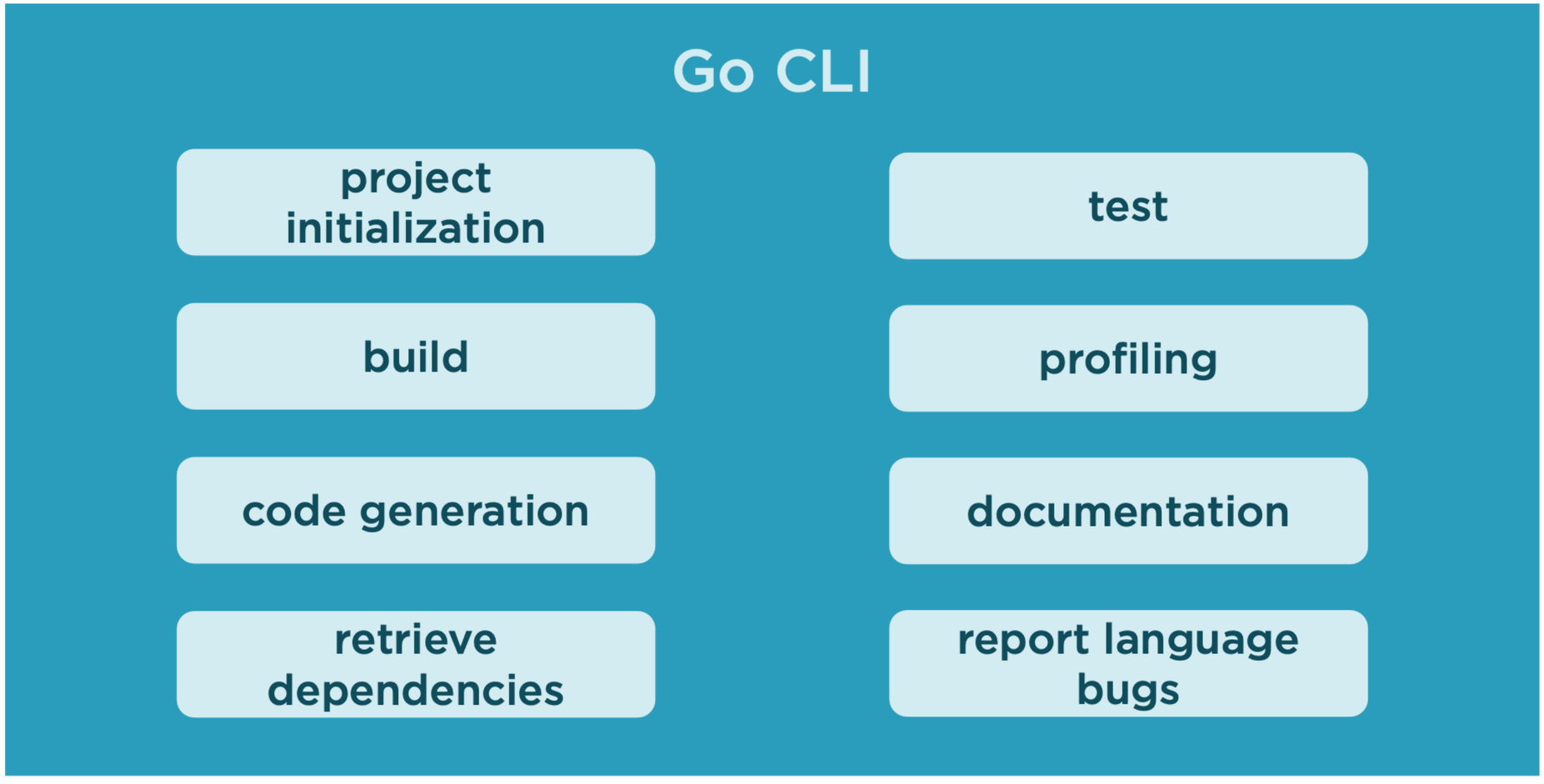
twitch

Uber

Values and Philosophy

- Simplicity
- Language that knows the existence of network requests and concurrency execution
- Library-free experience for: strings, network, compression, file management, testing
- Cross-platform
- Backwards Compatibillity
- Powerful Command Line Interface

GO CLI


Go CLI
project initialization
test
build
profiling
code generation
documentation
retrieve dependencies
report language bugs

Tools

- Go (including Go CLI): **go.dev**
- Visual Studio Code
- Go Plugins for VSC and tools installed by the Go Plugin
- Terminal

Multi-platform

- It can generate executable binary files for different platforms and operating systems.
- It can compile to WebAssembly (WASM)
- It can transpile to Frontend JavaScript (GopherJS)

- List: https://gist.github.com/asukakenji/f15ba7e588ac42795f421b48b8aede63

Common Use Cases

- Web services
- Web applications
- DevOps
- Desktop UI
- Machine Learning
- And much more

Language Types

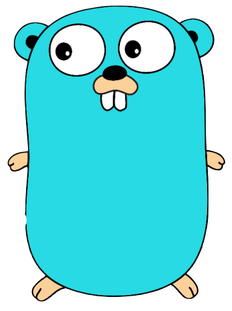
Developers write code in

Then ship

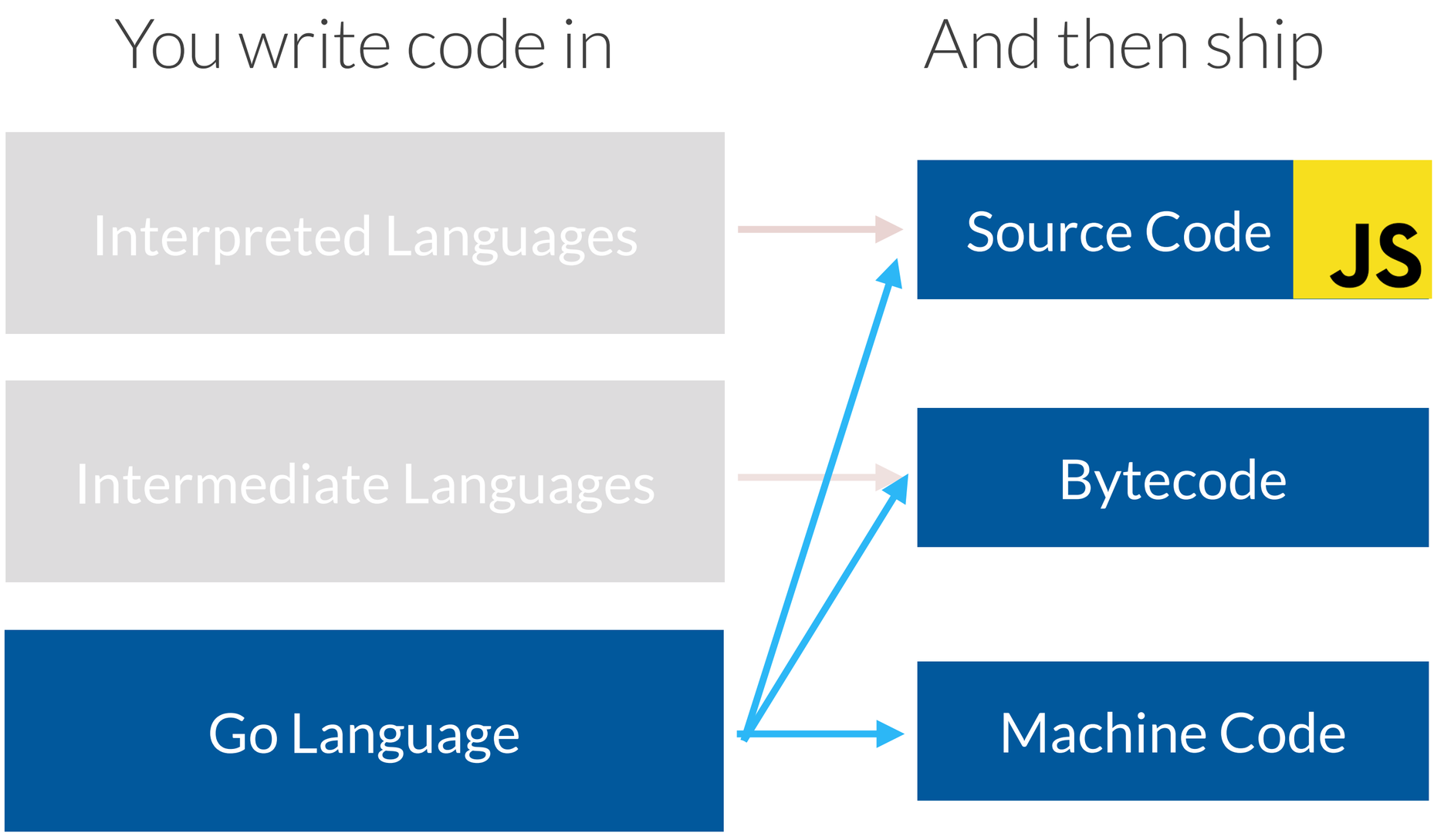
Interpreted Languages → Source Code

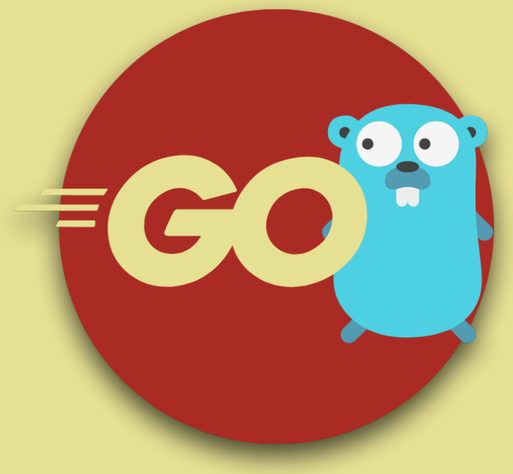
Intermediate Compiled Languages → Bytecode

Compiled Languages → Machine Code



Language Types





Fundamentals of Go Syntax

Basic Rules

- We use .go files
- Code Blocks in { }
- No styling freedom
- We do have semi-colon to separate sentences
- But it's optional :)
- Case-sensitive
- Strongly typed
- NOT an object-oriented language
- No classes, no exceptions

Basic Rules (continues)

- We have one file acting as the entry point with a main function
- A folder is a package
- Packages can have simple names (services) or URLs (github.com/fem/my-library)
- Within one go file, we can have:
  - Variables
  - Functions
  - Type declarations
  - Methods declarations

Modules and CLI

- A module is a group of packages
- It's our project
- It contains a go.mod file with configuration and metadata

- CLI manipulates the module
  - `go mod init`
  - `go build`
  - `go run`
  - `go test`
  - `go get`

Workspaces and CLI

- A workspace is a new kind of multi-module app concept from 1.18
- It contains a go.work file with configuration and metadata including which module to use
- CLI manipulates the workspace
  - `go work init`

Defining variables

```
var x int
var name string
const y = 2


var z int = 2

var text string
text = "Hello!"

otherText := "Bye!"
```

- Data types goes after identifier
- Variables have **`nil`** by default
- Constants can be only bool, string or numbers

- We can create variables with initialization
- Strings uses double quotes

- Variable Initialization shortcut

Built-in Data Types

- string
- Integer values: int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64
- Floating point values: float32, float64
- bool

Boolean operators: ==, !=, <, >, <=, >=, &&, ||, !

We can work also with pointers!

A package is a group of files in the same folder

```
package main

import "fmt"


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

◀ We define the name at the top

◀ We can import other packages

◀ Main app's entry point

Visibility

- What we write in a module:
  - If it's camelCase, it's private
  - If it's TitleCase, it's public and exported
- Variables and lambda functions can be:
- Module Scoped
- Function Scoped
- Block Scoped

Visibility

```
package main

import "fmt"

func notExported() {
}

func Exported() {
}
```

- If we use camelCase is private to the package
- If we use TitleCase it's public and exported to other packages

Numbers

We can easily convert between numbers by using a global function with the type name

```
id := 4

price := 45.4


priceAsInt := int(price)
idAsFloat := float32(id)
```

Strings

Multiline strings by default!

```
str1 := "This is just a text"

str2 := "Every text
is multiline
by default"
```

Collections

- Arrays: fixed length

  `[5]int`

- Slices: similar to a dynamic length array, but they are actually chunks of arrays

  `[]int`

- Maps: key/value dictionaries

  `map[keyType]valueType`

- Generics from 1.8

WARNING

Collections are not objects (nothing is an object, actually) so we use global functions to work with them, such as `len()` or `cap()`

Collections

Arrays have fixed length; Slices dynamic length

```
// Arrays
var countries [3]string
countries[0] = "Panama"

Prices := [2]int { 100, 150 }

// Slices
names := []string { "Mary", "John" }
names := append(names, "Carol")
println(len(names))
```

Maps

Be careful with the type definition

```
wellKnownPorts := map[string]int {"http": 80, "https": 443}
```

Functions

- Similar to other languages
- It can receive arguments
- Arguments can have default values
- The last argument can be of variable length
- The tricky part:
  - Functions can return more than one value (at once)
  - Functions can return labeled variables
  - Functions receive arguments always by value

Functions

We need to define types for arguments and return variables

```
func save() {}
func save(text string) {}

func add(a int, b int) int {
    return a+b
}

func addAndSubstract(a int, b int) (int, int) {
    return a+b, a-b
}
```

Functions receiving references

We need to receive pointers instead of the value

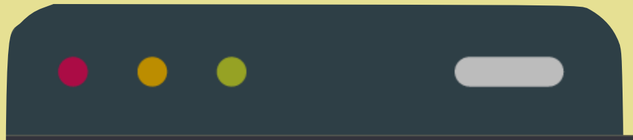
```
func increment(x *int) {
  *x++
}

func main() {
  i := 1
  increment(&i)
}
```

Some function curiosities

- Package init func
- panic
- defer

Errors design pattern



We don't have exceptions in Go, this is the typical design pattern when an action may trigger an error

```
func readUser(id int) (user, err) {
    // ... we proceed with the reading and see a bool ok value
    if ok {
        return user, nil
    } else {
        return nil, errorDetails
    }
}

func main {
    user, err := readUser(2)
}
```

Functions: Named automatic return

You have to save values in the variables defined as returned types and then call return when you are ready

```
func taxes(price float32) (stateTax, cityTax float32) {
    stateTax := price * 0.09
    cityTax := price * 0.015
    return
}

func main() {
    tax1, tax2 := taxes(100)
}
```

Control structures

- `if - else`
- `switch` (reloaded!)
- `for`

- `There is no while or do-while`

- No parenthesis are needed for boolean conditions or values
- Only one type of equality operators ==
- Other operators != < > <= >=

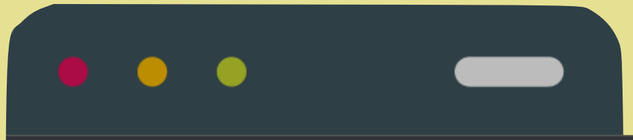
if, else

It can have multiple conditions
Have in mind, there is no ternary operator in Go

```
if user ≠ nil {
} else {
}

if message:="hello"; user ≠ nil {
} else {
}
```

switch



This is a simple switch operation. No break is needed; you can fallthrough to the next case, though.

```
switch day {
    case "Monday":
        fmt.Println("It's Monday! 💪")
    case "Saturday":
        fallthrough
    case "Sunday":
        fmt.Println("It's Weekend 🥳")
    default:
        fmt.Println("It's another working day 🤭")
}
```

switch with conditions

It can replace large ifs

```
switch {
    case user == nil:

    case user.active == false:

    case user.deleted == true:

    default:
}
```

for loop

It's a multi-purpose loop control structure

```
// Classic for
for i:=0; i<len(collection); i++ {
}

// For range, similar to "for in" in JS
for index := range collection {
}

// For range, similar to "foreach"
for key, value := range map {
}
```

for loop

We can emulate a while just by using a boolean expression

```
endOfGame := false
for endOfGame {
    // process Game loop
}

for count < 10 {
    count += 1;
}

for { // infinite loop
}
```

Type Definition

- New Types
- Aliases

Type - Aliases vs Definitions

You can add methods to new types, not to aliases!

```
// Type Alias
type distance = float32

// New type based on
type distance float32

// types have a constructor and conversion functions
d := distance(34.5)
```

Methods

A method is a function attached to a type, event built-in types!

```
func (d distance) toMiles() string {
    return d * distance(0.621371)
}

func main() {
    dist := distance(10)
    print(dist.toMiles())
}
```

Complex types for definitions

- **Structures**
  - They kind of replace the class idea
  - It's a data type with strongly typed properties
  - They have a default constructor
  - You can add methods to it
- **Interfaces**
  - A definition of methods
  - You emulate polimorsphism from OOP
  - Implicit implementation
  - We can embed interfaces in other interfaces

Type - Structs

If you want to export your struct remember to use TitleClase.
You will find two pre-built constructors, with and without name

```
type User struct {
    id int
    name string
}


func main() {
    var u1 User
    u1 = User {id: 1, name: "Frontend Masters"}
    u2 := User {2, "Frontend Masters"}
}
```

Type - Structs with methods

Structs are functions attached to a type declared outside of the structure

```
func (u User) PrettyPrint() string {
    return string(u.id) + ": " + u.name
}

func main() {
    u2 := User {2, "Frontend Masters"}
    msg := u2.PrettyPrint()
}
```

Type - Interfaces

It's just a list of methods that then we can use as a type

```
type PrettyPrinted interface {
    PrettyPrint() string
}

func saveObject(object PrettyPrinted) {
    save(object.PrettyPrint()
}

func main() {
    u2 := User {2, "Frontend Masters"}
    saveObject(u2)
}
```

Definition Blocks

- You can create definitions blocks within a module to define multiple:
  - Module imports
  - Variables
  - Types

Blocks

Using blocks is optional

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

var (
  x, y int
  Name string
  finished bool
)

type (
  User struct { id int }
)
```

Standard Library

Standard Library

- Code written and tested by Go team
- Backwards compatibility guaranteed
- Portability across all platforms
- The **fmt** packet is the first one we will see that manages input and output from the program
- Other packages are **http, strings, numbers, json**

Printing values with fmt package

It's much better than using print* functions

```
// stdout console output
fmt.Print()
fmt.Println()
fmt.Printf()

// prints to an external source (such as a file)
fmt.Fprint()
fmt.Fprintln()
fmt.Fprintf()

// prints to a character buffer
fmt.Sprint()
fmt.Sprintln()
fmt.Sprintf()
```

Goroutines

Goroutines and Channels

- A goroutine is the Go way of using threads
- We open a goroutine just by invoking any function with a go prefix.
- `go functionCall()`
- Goroutines can communicate through channels, an special type of variable
- A channel contains a value of any kind
- A routine can define a value for a channel and other routine can wait for that value
- Channels can be buffered or not

Using Channels

```
var m1 chan string

m2 := make(chan string)


m2 ← "hello"

message := ← m2
```

◀ Creates a channel variable

◀ Creates a channel variable with an instance

◀ Saves a value to the channel

◀ Creates a variable waiting for a channel value

Using Channels

```
logs := make(chan string, 2)
logs ←"hello"
logs ←"world"
fmt.Println(←messages)
fmt.Println(←messages)
```

- Creates a channel with a buffer (2 values)
- Se send two messages, even if no one is "listening"

- We read the two buffered values

WARNING

To avoid deadlocks you have to close the channels before ending the program with `close(chan)`

Testing

Testing

- Vanilla Go includes Testing
- A test is a file with suffix _test.go
- You define functions with prefix Test and with an special signature receiving a `*testing.T` argument
- The function inside calls methods of T
- You can create subtests as goroutines

Testing

- You use the CLI with `go test`
- TableDrivenTest Design Pattern
- Fuzzing from 1.19

DEFINITION

Fuzzing

Automated testing that manipulates inputs to find bugs. Go fuzzing uses coverage guidance to find failures and is valuable in detecting security exploits and vulnerabilities.

Go for Web Developers

Go for Web Developers

WebAssembly	Transpile to JS	Web Server	Web Services
Useful for mixing it with JavaScript and front-end	To write your frontend code directly in Go	Serve files and HTML including a template service	RESTful APIs or Microservices

Go Templates

- HTML file with Go code
- It's in the html/package package
- The template can include expressions in {{ }}
- Trimming spaces available
- Actions and pipelines
- if-else conditions
- range for loops
- You can call functions

Basic Go Template System

We create first a template and then execute it to replace the parameters

```
import "html/template"
type Person struct {
    Name string
}

func main() {
    t := template.New("my-template").Parse(`
        <!DOCTYPE html>
        <title>My Website</title>
        <h1>{{.Name}}</h1>
`)

person := Person{ Name: "Jane Doe" }
output, err := t.ExecuteToString(person)
```

Final Steps

Compiling

- Compiling the project

```
go build .
```

- Compiling in one specific output folder

```
go build . -o build/
```

- Compiling for other platforms and OSs

```
env GOOS=target-OS
    GOARCH=target-architecture go build .
```

- Compile and install

```
go install .
```

Packaging

- Go just produces a binary
- It doesn't provide any packaging solution
- If we want to embed assets for an app we need to use third-party or OSs tools, such as:
- Creating installers for Windows
- Create a DMG package for macOS
- Create RPM or DEB packages for Linux

What we've covered

- What's Go
- Fundamentals
- Standard Libraries
- Goroutines
- fmt Package
- http Package
- Templating
- Web Servers and Services

THANKS! 🤗

GO FOR WEB DEVELOPERS

MAXIMILIANO FIRTMAN