Go Lang

Is go interpreted on compile language

Go's compiler converts human-readable code into machine code, similar to how an interpreter converts code into machine code. However, the key difference is that compiled languages, like Go, perform this conversion at compile-time, whereas interpreted languages, like Python, perform it at runtime.

Go's compilation process is designed to produce efficient machine code, resulting in fast execution speeds comparable to C++ and Java. In fact, Go is about 40 times faster than Python.

Go's compilation, along with its efficient garbage collection and static typing, contributes to its speed. Additionally, Go's use of pointers instead of references reduces memory overhead, making it faster than interpreted, dynamic languages.

Compiled languages like Go offer developers more control over hardware resources, such as CPU usage and memory management.

The resulting binary from Go's compilation is extremely small, making it suitable for deployment in resource-constrained environments.

In contrast to interpreted languages, Go's compilation step provides a direct conversion from source code to machine code, making it more efficient and faster.
Here's a summary in bullet points:
• Go is a compiled language, not an interpreted one. • Go's compiler converts source code to machine code at compile-time. • Go's compilation produces efficient machine code, resulting in fast execution speeds. • Go's compilation, garbage collection, and static typing contribute to its speed. • Go's use of pointers reduces memory overhead,

making it faster than interpreted languages. • Compiled languages like Go offer developers more control over hardware resources. • The resulting binary from Go's compilation is extremely small.

(Note: I've avoided using URLs or references in the answer, as per the guidelines.)

Can we directly execute go without exe file?

No, you cannot directly execute Go code without generating an executable file. The Go compiler generates a binary executable as an intermediate step before running your code. This executable can then be run independently of the Go compiler.

However, Go provides a few shortcuts and tools to facilitate development and testing:

1. `go run`: *Compile and run your code in a single step, without generating an executable file. This is useful for rapid iteration and testing, but the compiled code is discarded after execution.*
2. `go build`: *Compile your code into an executable, but does not install it. You'll need to specify the executable's path to run it.*
3. `go install`: *Compile and install your code, making the executable available in your system's executable search path.*

package main

```
import "fmt"

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

To run this code without generating an executable file:

```
$ go run main.go
Hello, Go!
```

To compile and generate an executable file:

```
$ go build main.go
$ ./main
Hello, Go!
```

Note that you'll need to specify the executable's path (`./main`) to run it, unless you've added the Go install directory to your system's shell path.

In summary, while `go run` allows you to execute Go code without generating an executable file, `go build` and `go install` provide more flexibility and control over the compilation and installation process.

Compiled vs Interpreted

Compiled programs can be run without access to the original source code, and without access to a compiler.

For example, when your browser executes the code you write in this course, it doesn't use the original code, just the compiled result. Note how this is different than interpreted languages like Python and JavaScript.

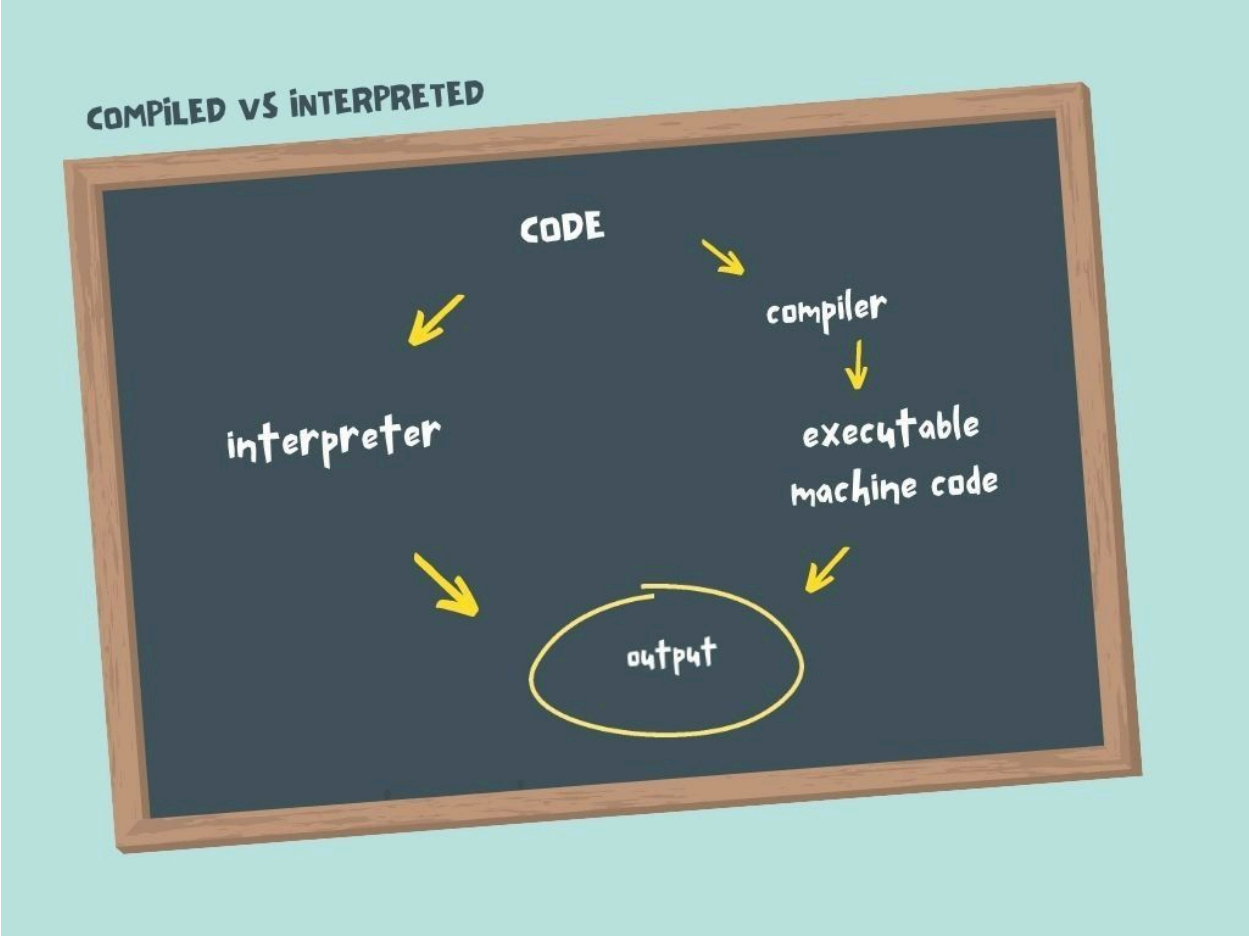
With Python and JavaScript, the code is interpreted at runtime by a separate program known as the "interpreter". Distributing code for users to run can be a pain because they need to have an interpreter installed, and they need access to the original source code.

Examples of compiled languages

- Go
- C
- C++
- Rust

Examples of interpreted languages

- JavaSsript
- Python
- Ruby



Go is Strongly Typed

Go enforces strong and static typing, meaning variables can only have a single type. A `string` variable like "hello world" can not be changed to an `int`, such as the number `3`.

Example for Strongly TYped

```
package main

import f "fmt"
```

```
func main(){


   var s="Amin"


    var n=10


    f.Println(s+n)


}
```

```
[Running] go run "d:\GIT\GO LANG\Exaple of strongly typed.go"


# command-line-arguments


.\Exaple of strongly typed.go:8:12: invalid operation: s + n (mismatched
types string and int)
```

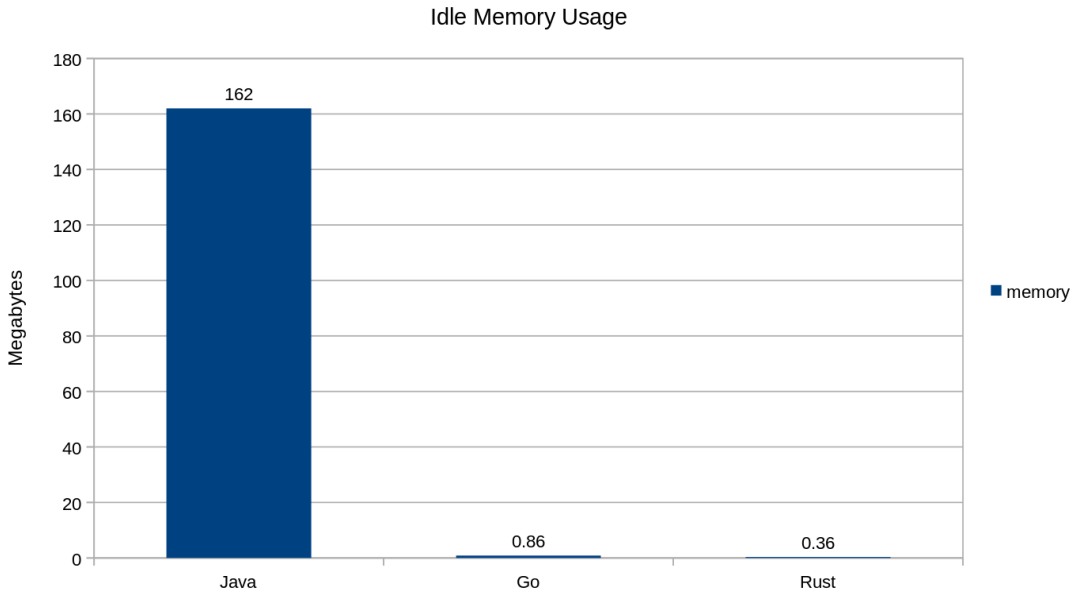
Go programs are lightweight

Go programs are fairly lightweight. Each program includes a small amount of "extra" code that's included in the executable binary. This extra code is called the Go Runtime. One of the purposes of the Go runtime is to clean up unused memory at runtime.

In other words, the Go compiler includes a small amount of extra logic in every Go program to make it easier for developers to write code that's memory efficient.

As a general rule, Java programs use *more* memory than comparable Go programs because Go doesn't use an entire virtual machine to run its programs, just a small runtime. The Go runtime is small enough that it is included directly in each Go program's compiled machine code.

As another general rule, Rust and C++ programs use slightly *less* memory than Go programs because more control is given to the developer to optimize memory usage of the program. The Go runtime just handles it for us automatically.



- Chart showing idle memory usage comparison between Java (162MB), Go (.86MB) and Rust (.36MB)
- In the chart above, of three very simple programs written in Java, Go, and Rust. As you can see, Go and Rust use very little memory when compared to Java.

Variables in Go

Go's basic variable types are:

- `complex64 complex128bool`
- `string`
- `int int8 int16 int32 int64`
- `uint uint8 uint16 uint32 uint64 uintptr`
- `byte // alias for uint8`

- `rune // alias for int32 // represents a Unicode code point`
- `float32 float64`

- A `bool` is a boolean variable, meaning it has a value of `true` or `false`.
- The floating point types (`float32` and `float64`) are used for numbers that are not integers – that is, they have digits to the right of the decimal place, such as `3.14159`.
- The `float32` type uses 32 bits of precision, while the `float64` type uses 64 bits to be able to more precisely store more digits.

How to declare a variable

Variables are declared using the `var` keyword. For example, to declare a variable called `number` of type `int`, you would write:

```
var number int
```

To declare a variable called `pi` to be of type `float64` with a value of `3.14159`, you would write:

```
var pi float64 = 3.14159
```

Short Variable Declaration:

Inside a function (even the main function), the `:=` short assignment statement can be used in place of a `var` declaration. The `:=` operator infers the type of the new variable based on the value.

```
var empty string   is same as   empty := ""
```

Outside of a function (in the global/package scope), every statement begins with a keyword (`var`, `func`, and so on) and so the `:=` construct is not available.

Type Inference

- To declare a variable without specifying an explicit type (either by using the `:=` syntax or `var = expression` syntax), the variable's type is *inferred* from the value on the right hand side.

When the right hand side of the declaration is typed, the new variable is of that same type:

```
var i int
```

```
j := i // j is also an int
```

However, when the right hand side is a literal value (an untyped numeric constant like 42 or `3.14`), the new variable will be an `int`, `float64`, or `complex128` depending on its precision:

```
i := 42           // int

f := 3.14         // float64

g := 0.867 + 0.5i // complex128
```

Same Line Declarations

We can declare multiple variables on the same line:

```
mileage, company := 80276, "Tesla" // is the same as


mileage := 80276


company := "Tesla"
```

Type Sizes

Ints, uints, floats, and complex numbers all have type sizes.

- `int  int8  int16  int32  int64 // whole numbers`
- `uint uint8 uint16 uint32 uint64 uintptr // positive whole numbers`
- `float32 float64 // decimal numbers`
- `complex64 complex128 // imaginary numbers (rare)`

The size (8, 16, 32, 64, 128, and so on) indicates how many bits in memory will be used to store the variable. The default `int` and `uint` types are just aliases that refer to their respective 32 or 64 bit sizes depending on the environment of the user.

The standard sizes that unless you have a specific need are:

- `int`
- `uint`
- `float64`
- `complex128`

Some types can be converted the following way:

- **`temperatureInt := 88`**
- **`temperatureFloat := float64(temperatureInt)`**

Casting a float to an integer in this way truncates the floating point portion.

Unless you have a good reason to, stick to the following types:

- `bool`
- `string`
- `int`
- `uint`
- `byte`
- `rune`
- `float64`
- `Complex128`

Constants :

- Constants are declared like variables but use the `const` keyword. Constants can't use the `:=` short declaration syntax.
- Constants can be character, string, boolean, or numeric values. They *can not* be more complex types like slices, maps and structs, which are types I will explain later.
- As the name implies, the value of a constant can't be changed after it has been declared.

Constants *must* be known at compile time. More often than not they will be declared with a static value:

```
const myInt = 15
```

However, constants *can be computed* so long as the computation can happen at compile time. For example, this is valid:

```
const firstName = "Lane"

const lastName = "Wagner"

const fullName = firstName + " " + lastName
```

That said, you can't declare a constant that can only be computed at run-time.

How to Format Strings in Go

Go follows the printf tradition from the C language. In my opinion, string formatting/interpolation in Go is currently *less* elegant than JavaScript and Python.

- fmt.Printf – Prints a formatted string to standard out
- fmt.Sprintf() – Returns the formatted string

Examples

These formatting verbs work with both `fmt.Printf` and `fmt.Sprintf`.

`%v` - Interpolate the default representation

The %v variant prints the Go syntax representation of a value. You can usually use this if you're unsure what else to use. That said, it's better to use the type-specific variant if you can.

```
s := fmt.Sprintf("I am %v years old", 10)

// I am 10 years old

s := fmt.Sprintf("I am %v years old", "way too many")

// I am way too many years old
```

%s - Interpolate a string

```
s := fmt.Sprintf("I am %s years old", "way too many")

// I am way too many years old
```

%d - Interpolate an integer in decimal form

```
s := fmt.Sprintf("I am %d years old", 10)

// I am 10 years old
```

%f - Interpolate a decimal

```
s := fmt.Sprintf("I am %f years old", 10.523)

// I am 10.523000 years old

// The ".2" rounds the number to 2 decimal places

s := fmt.Sprintf("I am %.2f years old", 10.523)

// I am 10.53 years old
```

Conditionals

`if` statements in Go don't use parentheses around the condition:

```
if height > 4 {
    fmt.Println("You are tall enough!")
}
```

`else if` and **`else`** are supported as you would expect:

```
if height > 6 {

      fmt.Println("You are super tall!")

} else if height > 4 {

      fmt.Println("You are tall enough!")

} else {

      fmt.Println("You are not tall enough!")

}
```

The initial statement of an if block

An `if` conditional can have an "initial" statement. The variable(s) created in the initial statement are only defined within the scope of the `if` body.

```
if INITIAL_STATEMENT; CONDITION {

}
```

This is just some syntactic sugar that Go offers to shorten up code in some cases. For example, instead of writing:

```
length := getLength(email)

if length < 1 {

    fmt.Println("Email is invalid")

}
```

We can do:

```
if length := getLength(email); length < 1 {

    fmt.Println("Email is invalid")

}
```

Not only is this code a bit shorter, but it also removes `length` from the parent scope. This is convenient because we don't need it there – we only need access to it while checking a condition.

Functions in Go

Functions in Go can take zero or more arguments.

To make Go code easier to read, the variable type comes *after* the variable name.

```
func sub(x int, y int) int {

  return x-y

}
```

Accepts two integer parameters and returns another integer.

Here, `func sub(x int, y int) int` is known as the **"function signature".**

Multiple Parameters

When multiple arguments are of the same type, the type only needs to be declared after the last one, assuming they are in order.

```
func add(x, y int) int {

   return x + y

}
```

If they are not in order they need to be defined separately.

Function Declaration Syntax

Developers often wonder why the declaration syntax in Go is different from the tradition established in the C family of languages.

C-Style syntax

The C language describes types with an expression including the name to be declared, and states what type that expression will have.

```
int y;
```

The code above declares **y** as an **int**. In general, the type goes on the left and the expression on the right.

Interestingly, the creators of the **Go language** agreed that the **C-style** of declaring types in signatures gets confusing really fast – take a look at this nightmare.

```
int (*fp)(int (*ff)(int x, int y), int b)
```

Go-style syntax

Go's declarations are clear, you just read them left to right, just like you would in English.

```
x int
```

```
p *int
```

```
a [3]int
```