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.

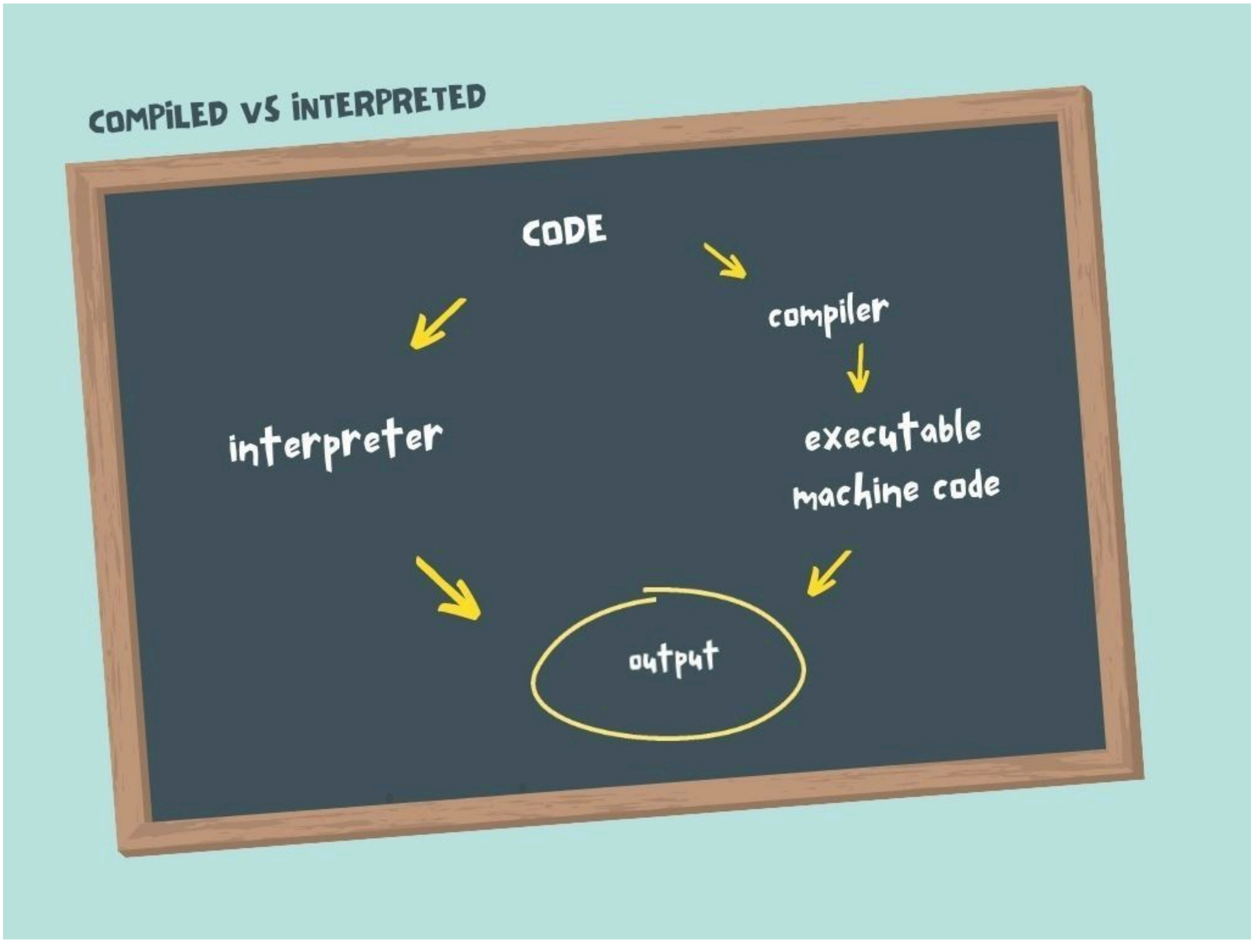
With Python and JavaScript, the code is interpreted at runtime by a separate program known as the "interpreter". Distributing code for users to run can be a pain because they need to have an interpreter installed, and they need access to the original source code.

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)
```

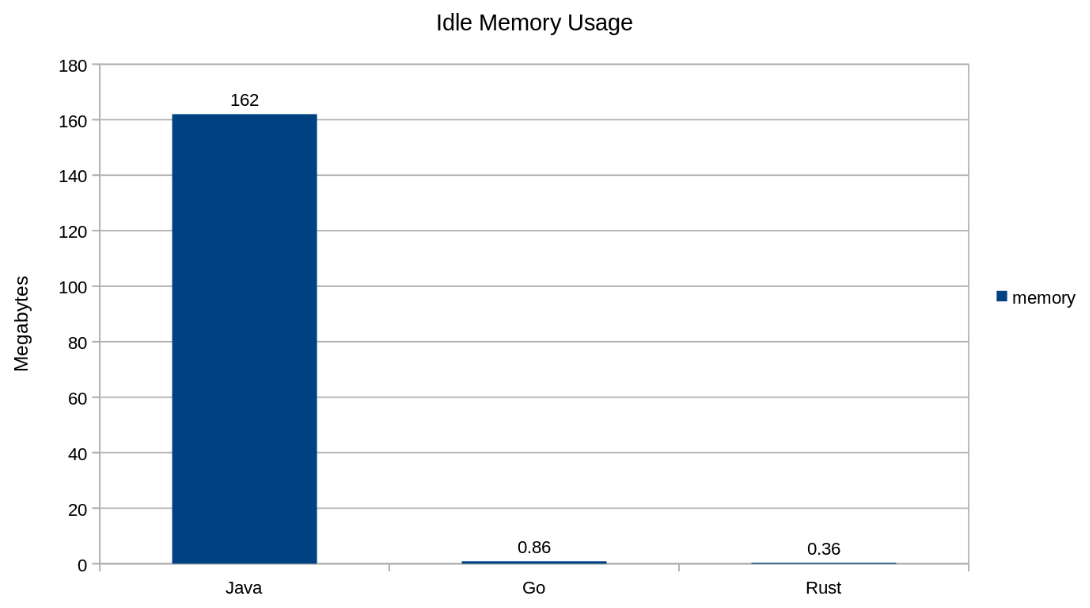
Go programs are lightweight

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:

```go
var number int
```

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

```go
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:

```go
const myInt = 15
```

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

```go
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:

```go
length := getLength(email)

if length < 1 {

    fmt.Println("Email is invalid")

}
```

We can do:

```go
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
```

It's nice for more complex signatures, as it makes them easier to read.

```
f func(func(int,int) int, int) int
```

How to Pass Variables by Value:

- Variables in Go are passed by value (except for a few data types we haven't covered yet). "Pass by value" means that when a variable is passed into a function, that function receives a *copy* of the variable. The function is unable to mutate the caller's original data.

```
package main


import f "fmt"


func main(){x:=5


    f.Println(x)//Here we can say That pass by value so x won't change
here


f.Println(inc(x))// here It's returning function value


    f.Println(x)//here it won't change so we can say pass by value only
sends its cop


}func inc(x int)int{


    return x+2


}
```

How to Ignore Return Values

A function can return a value that the caller doesn't care about. We can explicitly ignore variables by using an **underscore**: **_**

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

  return 3, 4

}

// ignore y value

x, _ := getPoint()
```

Even though **`getPoint()`** returns two values, we can capture the first one and ignore the second.

Why would you ignore a return value?

- There could be many reasons. For example, maybe a function called `getCircle` returns the center point and the radius, but you really only need the radius for your calculation. In that case, you would ignore the center point variable.
- This is crucial to understand because the Go compiler will throw an error if you have unused variable declarations in your code, so you *need* to ignore anything you don't intend to use.

Named Return Values

Return values may be given names, and if they are, then they are treated the same as if they were new variables defined at the top of the function.

Named return values are best thought of as a way to document the purpose of the returned values.

According to the tour of go:

"A return statement without arguments returns the named return values. This is known as a "naked" return. Naked return statements should be used only in short functions. They can harm readability in longer functions."

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

// x and y are initialized with zero values

  return // automatically returns x and y

}
```

Is the same as:

```
func getCoords() (int, int){
  var x int
  var y int
  return x, y
}
```

In the first example, `x` and `y` are the return values. At the end of the function, we could simply write `return` to return the values of those two variables, rather that writing `return x,y`.

Explicit Returns

Even though a function has named return values, we can still explicitly return values if we want to.

```
func getCoords() (x, y int){
  return x, y // this is explicit
}
```

Using this explicit pattern we can even overwrite the return values:

```
func getCoords() (x, y int){
  return 5, 6 // this is explicit, x and y are NOT returned
}
```

Otherwise, if we want to return the values defined in the function signature we can just use a naked `return` (blank return):

```
func getCoords() (x, y int){
  return // implicitly returns x and y
}
```

The Benefits of Named Returns

Good For Documentation (Understanding)

- Named return parameters are great for documenting a function. We know what the function is returning directly from its signature, no need for a comment.
- Named return parameters are particularly important in longer functions with many return values.

```
func calculator(a, b int) (mul, div int, err error) {
   if b == 0 {
     return 0, 0, errors.New("Can't divide by zero")
   }
   mul = a * b
   div = a / b
   return mul, div, nil
}
```

Which is easier to understand than:

```
func calculator(a, b int) (int, int, error) {
   if b == 0 {
     return 0, 0, errors.New("Can't divide by zero")
   }
   mul := a * b
   div := a / b
   return mul, div, nil
}
```

We know *the meaning* of each return value just by looking at the function signature: **`func calculator(a, b int) (mul, div int, err error)`**

Less Code (Sometimes)

- If there are multiple return statements in a function, you don’t need to write all the return values each time, though you probably should.
- When you choose to omit return values, it's called a *naked* return. Naked returns should only be used in short and simple functions.

Early Returns

- Go supports the ability to return early from a function. This is a powerful feature that can clean up code, especially when used as guard clauses.
- Guard Clauses leverage the ability to return early from a function (or continue through a loop) to make nested conditionals one-dimensional. Instead of using if/else chains, we just return early from the function at the end of each conditional block.

```
func divide(dividend, divisor int) (int, error) {
    if divisor == 0 {
        return 0, errors.New("Can't divide by zero")
    }
    return dividend/divisor, nil
}
```

Error handling in Go naturally encourages developers to make use of guard clauses. When I started writing more JavaScript, I was disappointed to see how many nested conditionals existed in the code I was working on.

Let’s take a look at an exaggerated example of nested conditional logic:

```
func getInsuranceAmount(status insuranceStatus) int {
  amount := 0
  if !status.hasInsurance(){
    amount = 1
  } else {
    if status.isTotaled(){
      amount = 10000
    } else {
      if status.isDented(){
        amount = 160
        if status.isBigDent(){
          amount = 270
        }
      } else {
        amount = 0
      }
    }
  }
  return amount
}
```

This could be written with guard clauses instead:

```
func getInsuranceAmount(status insuranceStatus) int {

  if !status.hasInsurance(){
    return 1
  }
  if status.isTotaled(){
    return 10000
  }
  if !status.isDented(){
    return 0
  }
  if status.isBigDent(){
    return 270
  }
  return 160
}
```

- The example above is much easier to read and understand. When writing code, it's important to try to reduce the cognitive load on the reader by reducing the number of entities they need to think about at any given time.
- In the first example, if the developer is trying to figure out when 270 is returned, they need to think about each branch in the logic tree and try to remember which cases matter and which cases don't.
- With the one-dimensional structure offered by guard clauses, it's as simple as stepping through each case in order.

Structs in Go

We use structs in Go to represent structured data. It's often convenient to group different types of variables together. For example, if we want to represent a car we could do the following:

```
type car struct {
  Make string
  Model string
  Height int
  Width int
}
```

This creates a new struct type called car. All cars have a **Make**, **Model**, **Height** and **Width**.
In Go, you will often use a struct to represent information that you would have used a **dictionary for in Python**, or **an object literal in JavaScript**.

Nested Structs in Go

Structs can be nested to represent more complex entities:

```
type car struct {
  Make string
  Model string
  Height int
  Width int
  FrontWheel Wheel
  BackWheel Wheel
}

type Wheel struct {
  Radius int
  Material string
}
```

The fields of a struct can be accessed using the dot . operator.

```
myCar := car{}
myCar.FrontWheel.Radius = 5
```

Anonymous Structs

An anonymous struct is just like a regular struct, but it is defined without a name and therefore cannot be referenced elsewhere in the code.

To create an anonymous struct, just instantiate the instance immediately using a second pair of brackets after declaring the type:

```
myCar := struct {
  Make string
  Model string
} {
  Make: "tesla",
  Model: "model 3"
}
```

You can even nest anonymous structs as fields within other structs:

```
type car struct {
  Make string
  Model string
  Height int
  Width int
  // Wheel is a field containing an anonymous struct
  Wheel struct {
    Radius int
    Material string
  }
}
```

When should you use an anonymous struct?

In general, prefer named structs. Named structs make it easier to read and understand your code, and they have the nice side-effect of being reusable. I sometimes use anonymous structs when I know I won't ever need to use a struct again. For example, sometimes I'll use one to create the shape of some JSON data in HTTP handlers.

If a struct is only meant to be used once, then it makes sense to declare it in such a way that developers down the road won't be tempted to accidentally use it again.

Embedded Structs

Go is not an object-oriented language. But embedded structs provide a kind of data-only inheritance that can be useful at times.

Keep in mind, Go doesn't support classes or inheritance in the complete sense. Embedded structs are just a way to elevate and share fields between struct definitions.

```
type car struct {
  make string
  model string
}

type truck struct {
  // "car" is embedded, so the definition of a
  // "truck" now also additionally contains all
  // of the fields of the car struct
  car
  bedSize int
}
```

Embedded vs nested

- An embedded struct's fields are accessed at the top level, unlike nested structs.
- Promoted fields can be accessed like normal fields except that they can't be used in composite literals

Struct Methods

While Go is **not** object-oriented, it does support methods that can be defined on structs. Methods are just functions that have a receiver. A receiver is a special parameter that syntactically goes *before* the name of the function.

```
type rect struct {
  width int
  height int
}

// area has a receiver of (r rect)
func (r rect) area() int {
  return r.width * r.height
}

r := rect{
  width: 5,
  height: 10,
}

fmt.Println(r.area())
// prints 50
```

A receiver is just a special kind of function parameter. Receivers are important because they will, as you'll learn in the exercises to come, allow us to define interfaces that our structs (and other types) can implement.

Interfaces in Go

Interfaces are collections of method signatures. A type "implements" an interface if it has all of the methods of the given interface defined on it.

In the following example, a "shape" must be able to return its area and perimeter. Both `rect` and `circle` fulfill the interface.

```
type shape interface {
  area() float64
  perimeter() float64
}

type rect struct {
    width, height float64
}
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perimeter() float64 {
    return 2*r.width + 2*r.height
}

type circle struct {
    radius float64
}
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perimeter() float64 {
    return 2 * math.Pi * c.radius
}
```

When a type implements an interface, it can then be used as the interface type.

Interfaces are implemented *implicitly*.

A type never declares that it implements a given interface. If an interface exists and a type has the proper methods defined, then the type automatically fulfills that interface.

Multiple Interfaces

A type can implement any number of interfaces in Go. For example, the empty interface, **`interface{}`**, is *always* implemented by every type because it has no requirements.

Naming interface args

Consider the following interface:

```
type Copier interface {
  Copy(string, string) int
}
```

Based on the code alone, can you deduce what *kinds* of strings you should pass into the `Copy` function?

We know the function signature expects 2 string types, but what are they? Filenames? URLs? Raw string data? For that matter, what the heck is that `int` that's being returned?

Let's add some named arguments and return data to make it clearer.

```
type Copier interface {
  Copy(sourceFile string, destinationFile string) (bytesCopied int)
}
```

Much better. We can see what the expectations are now. The first argument is the **`sourceFile`**, the second argument is the **`destinationFile`**, and **`bytesCopied`**, an integer, is returned.

Type Assertions in Go

When working with interfaces in Go, every once-in-awhile you'll need access to the underlying type of an interface value. You can cast an interface to its underlying type using a *type assertion*.

```
type shape interface {
    area() float64
}

type circle struct {
    radius float64
}

// "c" is a new circle cast from "s"
// which is an instance of a shape.
// "ok" is a bool that is true if s was a circle
// or false if s isn't a circle
c, ok := s.(circle)
if !ok {
    // s wasn't a circle
    log.Fatal("s is not a circle")
}

radius := c.radius
```

Type Switches in Go

A *type switch* makes it easy to do several type assertions in a series.

A type switch is similar to a regular switch statement, but the cases specify *types* instead of *values*.

```
func printNumericValue(num interface{}) {
    switch v := num.(type) {
    case int:
        fmt.Printf("%T\n", v)
    case string:
        fmt.Printf("%T\n", v)
    default:
        fmt.Printf("%T\n", v)
    }
}

func main() {
    printNumericValue(1)
    // prints "int"

    printNumericValue("1")
    // prints "string"

    printNumericValue(struct{}{})
    // prints "struct {}"
}
```

`fmt.Printf("%T\n", v)` prints the *type* of a variable.

Clean Interfaces

Writing clean interfaces is *hard*. Frankly, anytime you're dealing with abstractions in code, the simple can become complex very quickly if you're not careful. Let's go over some rules of thumb for keeping interfaces clean.

1. Keep Interfaces Small

If there is only one piece of advice that you take away from this article, make it this: keep interfaces small! Interfaces are meant to define the minimal behavior necessary to accurately represent an idea or concept.

Here is an example from the standard HTTP package of a larger interface that's a good example of defining minimal behavior:

```
type File interface {
    io.Closer
    io.Reader
    io.Seeker
    Readdir(count int) ([]os.FileInfo, error)
    Stat() (os.FileInfo, error)
}
```

Any type that satisfies the interface's behaviors can be considered by the HTTP package as a *File*. This is convenient because the HTTP package doesn't need to know if it's dealing with a file on disk, a network buffer, or a simple `[]byte`.

2. Interfaces Should Have No Knowledge of Satisfying Types

An interface should define what is necessary for other types to classify as a member of that interface. They shouldn't be aware of any types that happen to satisfy the interface at design time.

For example, let's assume we are building an interface to describe the components necessary to define a car.

```
type car interface {
    Color() string
    Speed() int
    IsFiretruck() bool
}
```

`Color()` and `Speed()` make perfect sense, they are methods confined to the scope of a car. `IsFiretruck()` is an anti-pattern. We are forcing all cars to declare whether or not they are firetrucks. In order for this pattern to make any amount of sense, we would need a whole list of possible subtypes. `IsPickup()`, `IsSedan()`, `IsTank()`… where does it end??

Instead, the developer should have relied on the native functionality of type assertion to derive the underlying type when given an instance of the car interface. Or, if a sub-interface is needed, it can be defined as:

```
type firetruck interface {
    car
    HoseLength() int
}
```

Which inherits the required methods from `car` and adds one additional required method to make the `car` a `firetruck`.

3. Interfaces Are Not Classes

- Interfaces are not classes, they are slimmer.
- Interfaces don't have constructors or deconstructors that require that data is created or destroyed.
- Interfaces aren't hierarchical by nature, though there is syntactic sugar to create interfaces that happen to be supersets of other interfaces.
- Interfaces define function signatures, but not underlying behavior. Making an interface often won't DRY up your code in regards to struct methods. For example, if five types satisfy the `fmt.Stringer` interface, they all need their own version of the `String()` function.

Errors in Go

Go programs express errors with `error` values. An Error is any type that implements the simple built-in error interface:

```
type error interface {
   Error() string
}
```

When something can go wrong in a function, that function should return an `error` as its last return value. Any code that calls a function that can return an `error` should handle errors by testing whether the error is `nil`.

```
// Atoi converts a stringified number to an interger
i, err := strconv.Atoi("42b")
if err != nil {
   fmt.Println("couldn't convert:", err)
```

```
    // because "42b" isn't a valid integer, we print:
    // couldn't convert: strconv.Atoi: parsing "42b": invalid syntax
    // Note:
    // 'parsing "42b": invalid syntax' is returned by the .Error() method
    return
}
// if we get here, then
// i was converted successfully
```

A `nil` error denotes success. A non-nil error denotes failure.

The Error Interface

Because errors are just interfaces, you can build your own custom types that implement the `error` interface. Here's an example of a `userError` struct that implements the `error` interface:

```
type userError struct {
    name string
}
func (e userError) Error() string {
    return fmt.Sprintf("%v has a problem with their account", e.name)
}
```

It can then be used as an error:

```
func sendSMS(msg, userName string) error {

    if !canSendToUser(userName) {
        return userError{name: userName}
    }
    ...
}
```

Go programs express errors with `error` values. Error-values are any type that implements the simple built-in error interface.

Keep in mind that the way Go handles errors is fairly unique. Most languages treat errors as something special and different. For example, Python raises exception types and JavaScript throws and catches errors.

In Go, an `error` is just another value that we handle like any other value – however, we want! There aren't any special keywords for dealing with them.

The errors Package

The Go standard library provides an "errors" package that makes it easy to deal with errors.

Read the godoc for the errors.New() function, but here's a simple example:

```
var err error = errors.New("something went wrong")
```

Loops in Go

The basic loop in Go is written in standard C-like syntax:

```
for INITIAL; CONDITION; AFTER{
  // do something
}
```

`INITIAL` is run once at the beginning of the loop and can create variables within the scope of the loop.

`CONDITION` is checked before each iteration. If the condition doesn't pass then the loop breaks.

`AFTER` is run after each iteration.

```
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
// Prints 0 through 9
```

How to Omit Conditions

Loops in Go can omit sections of a for loop. For example, the `CONDITION` (middle part) can be omitted which causes the loop to run forever.

```
for INITIAL; ; AFTER {
  // do something forever
}
```

No while loops in Go

Most programming languages have a concept of a `while` loop. Because Go allows for the omission of sections of a `for` loop, a while loop is just a for loop that only has a CONDITION.

```
for CONDITION {
  // do some stuff while CONDITION is true
}
```

```
plantHeight := 1
for plantHeight < 5 {
  fmt.Println("still growing! current height:",
plantHeight)
  plantHeight++
}
fmt.Println("plant has grown to ", plantHeight,
"inches")
```

Which prints:

```
still growing! current height: 1
still growing! current height: 2
still growing! current height: 3
still growing! current height: 4

plant has grown to 5 inches
```

Continue through a loop

The `continue` keyword stops the current iteration of a loop and continues to the next iteration. `continue` is a powerful way to use the "guard clause" pattern within loops.

```
for i := 0; i < 10; i++ {
  if i % 2 == 0 {
    continue
  }
  fmt.Println(i)
}
```

```
// 1
// 3
// 5
// 7
```

Break out of a loop

The `break` keyword stops the current iteration of a loop and exits the loop.

```
for i := 0; i < 10; i++ {
  if i == 5 {
    break
  }
  fmt.Println(i)
}
// 0
// 1
// 2
// 3
// 4
```

Arrays and Slices in Go

Arrays

Arrays are fixed-size groups of variables of the same type.

The type [n]T is an array of n values of type T.

To declare an array of 10 integers:

```
var myInts [10]int
```

or to declare an initialized literal:

```
primes := [6]int{2, 3, 5, 7, 11, 13}
```

Slices

99 times out of 100 you will use a slice instead of an array when working with ordered lists.

Arrays are fixed in size. Once you make an array like [10]int you can't add an 11th element.

A slice is a *dynamically-sized*, *flexible* view of the elements of an array.

Slices always have an underlying array, though it isn't always specified explicitly. To explicitly create a slice on top of an array we can do:

```
primes := [6]int{2, 3, 5, 7, 11, 13}
mySlice := primes[1:4]

// mySlice = {3, 5, 7}
```

The syntax is:

```
arrayname[lowIndex:highIndex]

arrayname[lowIndex:]

arrayname[:highIndex]

arrayname[:]
```

Where lowIndex is inclusive and highIndex is exclusive

Either lowIndex or highIndex or both can be omitted to use the entire array on that side.

How to Create New Slices in Go

Most of the time we don't need to think about the underlying array of a slice. We can create a new slice using the make function:

```
// func make([]T, len, cap) []T
mySlice := make([]int, 5, 10)

// the capacity argument is usually omitted and defaults to the
length
mySlice := make([]int, 5)
```

Slices created with make will be filled with the zero value of the type.

If we want to create a slice with a specific set of values, we can use a slice literal:

```
mySlice := []string{"I", "love", "go"}
```

Note that the array brackets *do not* have a 3 in them. If they did, you'd have an *array* instead of a slice.

Length

The length of a slice is simply the number of elements it contains. It is accessed using the built-in len() function:

```
mySlice := []string{"I", "love", "go"}
```

```
fmt.Println(len(mySlice)) // 3
```

Capacity

The capacity of a slice is the number of elements in the underlying array, counting from the first element in the slice. It is accessed using the built-in cap() function:

```
mySlice := []string{"I", "love", "go"}
fmt.Println(cap(mySlice)) // 3
```

Generally speaking, unless you're hyper-optimizing the memory usage of your program, you don't need to worry about the capacity of a slice because it will automatically grow as needed.

Variadic Functions

Many functions, especially those in the standard library, can take an arbitrary number of *final* arguments. This is accomplished by using the "..." syntax in the function signature.

A variadic function receives the variadic arguments as a slice.

```
func concat(strs ...string) string {
    final := ""
    // strs is just a slice of strings
    for str := range strs {
        final += str


    }
    return final
}



func main() {
    final := concat("Hello ", "there ", "friend!")
    fmt.Println(total)
    // Output: Hello there friend!
}
```

The familiar fmt.Println() and fmt.Sprintf() are variadic!
fmt.Println() prints each element with space delimiters and a newline at the end.
func Println(a ...interface{}) (n int, err error)

Spread operator

The spread operator allows us to pass a slice into a variadic function. The spread operator consists of three dots following the slice in the function call.

```
func printStrings(strings ...string) {

    for i := 0; i < len(strings); i++ {

        fmt.Println(strings[i])

    }

}
```

```
func main() {
    names := []string{"bob", "sue", "alice"}
    printStrings(names...)
}
```

How to Append to a Slice

The built-in append function is used to dynamically add elements to a slice:

```
func append(slice []Type, elems ...Type) []Type
```

If the underlying array is not large enough, append() will create a new underlying array and point the slice to it.
Notice that append() is variadic. The following are all valid:

```
slice = append(slice, oneThing)
slice = append(slice, firstThing, secondThing)
slice = append(slice, anotherSlice...)
```

Maps in Go

Maps are similar to JavaScript objects, Python dictionaries, and Ruby hashes. Maps are a data structure that provides key->value mapping.

The zero value of a map is nil.

We can create a map by using a literal or by using the make() function:

```
ages := make(map[string]int)
ages["John"] = 37
ages["Mary"] = 24
ages["Mary"] = 21 // overwrites 24

ages = map[string]int{
  "John": 37,
  "Mary": 21,
}
```

The len() function works on a map – it returns the total number of key/value pairs.

```
ages = map[string]int{
  "John": 37,
  "Mary": 21,
}
fmt.Println(len(ages)) // 2
```

Map Mutations

Insert an element

m[key] = elem

Get an element

elem = m[key]

Delete an element

delete(m, key)

Check if a key exists

elem, ok := m[key]
If key is in m, then ok is true. If not, ok is false.
If key is not in the map, then elem is the zero value for the map's element type.

Types of Valid Map Keys

Any type can be used as the *value* in a map, but *keys* are more restrictive.

You can read more in the following section of the official Go blog.

As mentioned earlier, map keys may be of any type that is comparable. The language spec defines this precisely, but in short, comparable types are boolean, numeric, string, pointer, channel, and interface types, and structs or arrays that contain only those types.

Notably absent from the list are slices, maps, and functions. These types cannot be compared using ==, and may not be used as map keys.

It's obvious that strings, ints, and other basic types should be available as map keys, but perhaps unexpected are struct keys. Struct can be used to key data by multiple dimensions.

For example, this map of maps could be used to tally web page hits by country:

hits := make(map[string]map[string]int)

This is map of string to (map of string to int). Each key of the outer map is the path to a web page with its own inner map. Each inner map key is a two-letter country code. This expression retrieves the number of times an Australian has loaded the documentation page:

n := hits["/doc/"]["au"]func add(m map[string]map[string]int, path, country string) {

```
    mm, ok := m[path]
    if !ok {
        mm = make(map[string]int)
        m[path] = mm
    }
    mm[country]++
}
add(hits, "/doc/", "au")
```

Unfortunately, this approach becomes unwieldy when adding data, as for any given outer key you must check if the inner map exists, and create it if needed:

On the other hand, a design that uses a single map with a struct key does away with all that complexity:

```
type Key struct {
    Path, Country string
}
```

```
hits := make(map[Key]int)
```

When a Vietnamese person visits the home page, incrementing (and possibly creating) the appropriate counter is a one-liner:
hits[Key{"/", "vn"}]++
And it’s similarly straightforward to see how many Swiss people have read the spec:

```
n := hits[Key{"/ref/spec", "ch"}]
```

Nested Maps

Maps can contain maps, creating a nested structure. For example:

map[string]map[string]int

map[rune]map[string]int

Map[int]map[string]map[string]int

Advanced Functions in Go

First-class and higher-order functions

A programming language is said to have "first-class functions" when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function, and can be assigned as a value to a variable.
A function that returns a function or accepts a function as input is called a Higher-Order Function.

Go supports first-class and higher-order functions. Another way to think of this is that a function is just another type – just like ints and strings and bools.

For example, to accept a function as a parameter:

```
func add(x, y int) int {
  return x + y
}

func mul(x, y int) int {
  return x * y
}

// aggregate applies the given math function to the first 3 inputs
func aggregate(a, b, c int, arithmetic func(int, int) int) int {
  return arithmetic(arithmetic(a, b), c)
}

func main(){
  fmt.Println(aggregate(2,3,4, add))
  // prints 9
  fmt.Println(aggregate(2,3,4, mul))
  // prints 24
}
```

Function Currying in Go

Function currying is the practice of writing a function that takes a function (or functions) as input, and returns a new function.
For example:

```
func main() {
  squareFunc := selfMath(multiply)
  doubleFunc := selfMath(add)

  fmt.Println(squareFunc(5))
  // prints 25

  fmt.Println(doubleFunc(5))
  // prints 10
}

func multiply(x, y int) int {
  return x * y
}

func add(x, y int) int {
  return x + y
}

func selfMath(mathFunc func(int, int) int) func (int) int {
  return func(x int) int {
    return mathFunc(x, x)
  }
}
```

In the example above, the selfMath function takes in a function as its parameter, and returns a function that itself returns the value of running that input function on its parameter.

Defer keyword

The defer keyword is a fairly unique feature of Go. It allows a function to be executed automatically *just before* its enclosing function returns.

The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

Deferred functions are typically used to close database connections, file handlers and the like.

For example:

```
// CopyFile copies a file from srcName to dstName on the local
filesystem.
func CopyFile(dstName, srcName string) (written int64, err error) {
  // Open the source file
  src, err := os.Open(srcName)
  if err != nil {
    return
  }
  // Close the source file when the CopyFile function returns
  defer src.Close()

  // Create the destination file
  dst, err := os.Create(dstName)
  if err != nil {
    return
  }
  // Close the destination file when the CopyFile function returns
  defer dst.Close()

  return io.Copy(dst, src)
}
```

In the above example, the src.Close() function is not called until after the CopyFile function was called but immediately before the CopyFile function returns.
Defer is a great way to make sure that something happens at the end of a function, even if there are multiple return statements.

Closures

A closure is a function that references variables from outside its own function body. The function may access and assign to the referenced variables.
In this example, the concatter() function returns a function that has reference to an enclosed doc value. Each successive call to harryPotterAggregator mutates that same doc variable.

```
func concatter() func(string) string {
    doc := ""
    return func(word string) string {
        doc += word + " "
        return doc
    }
}

func main() {
    harryPotterAggregator := concatter()
    harryPotterAggregator("Mr.")
    harryPotterAggregator("and")
    harryPotterAggregator("Mrs.")
    harryPotterAggregator("Dursley")
    harryPotterAggregator("of")
    harryPotterAggregator("number")
    harryPotterAggregator("four,")
    harryPotterAggregator("Privet")
    fmt.Println(harryPotterAggregator("Drive"))
    // Mr. and Mrs. Dursley of number four, Privet Drive
}
```