FROM RUBY TO GOLANG

A RUBY PROGRAMMER'S GUIDE TO LEARNING GO

JOEL BRYAN JULIANO

From Ruby to Golang

A Ruby Programmer's Guide to Learning Go

Joel Bryan Juliano

This book is for sale at http://leanpub.com/rb2go

This version was published on 2021-06-10

ISBN 1080944001, ISBN-13 978-1080944002

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 - 2021 Joel Bryan Juliano

I dedicate this book to my loving, supportive and beautiful wife, and to my son.

Contents

Preface

In 2018, I was hired by a company that uses Go. After 8+ years of working with Ruby, the first thing I did to learn Go is to relate to what I know in Ruby. And I thought it would be a good idea to document my learning process. My initial intention is keep it as my personal documentation and notes, but after much careful thought and considerations, I decided to post it online. From one topic, it grew into a series of multiple online articles, discussing all about my research and learnings about the language, collecting it's analogies that I can compare with Ruby to help me learn it.

Go is a great language, it's easy to read, with clear syntax. And it compiles to a single binary file which makes apps fast and compact, and can compile to run on different platforms. And it's statically typed and garbage-collected making it efficient.

It's like a modern C with package support, memory safety, automatic garbage collection and concurrency baked-in. And you get all the nice features from a statically typed language, IDEs loves it, and so your development workflow.

In today's world of cloud native microservices, containerized architectures, You can be up-to-date with a knowledge in Go. Many notable open-source projects are built using Go (i.e. Docker, Kubernetes, Etherium and Terraform to name a few), and those platforms have APIs and SDKs readily available natively for you to use. And many global companies have been using Go in production (i.e. Google, Netflix, Dropbox, Heroku and Uber to name a few), proving that it has been battle-tested and powerful mature language to based your work into.

This book was made with a Rubyist in mind, all the learning methapors are based on Ruby and I think it will help you to learn Go programming language when you already knows Ruby. Go can be your second or third programming language and this book can help you get started.

Introduction

One of my favorite programming insights is in the book titled "Class Constructions in C and C++: Object-Oriented Programming Fundamentals" by Robert Sessions published in 1992, which reads "… object-oriented programming is really just a common-sense extension of structured programming". Fast forward today, this mindset still holds true not only for C programming language but also in Go. Go is not an OOP language by choice, and we can apply OOP techniques from OOP languages like Ruby to Go.

Go is an ideal programming language companies and developers choose to transition to, it is robust, fast, cross-platform, easy to learn, and a good choice for backend development.

It is also backed by Google, which ensures that it has supported developments in the long run.

The book is written in the way that it is easy and practical, and following through reading this book will help you to get up to speed on programming in Go fast.

And if you have prior Ruby experience, this book will help you learn Go **faster**, because the examples are written in Ruby, that you can relate easily.

About the Author

Joel has over 12+ years of experience as a software engineer in various IT domains namely Cybersecurity, OTT, Sports, PaAS, VoIP, Hospitality and E-commerce. He started programming when he was 12. He lives in Amsterdam, together with his family.

Setter/Getters, Attribute Accessors and Structs

Everything needs communication. Animals and humans communicate using physical gestures and sounds. Most insects have special antenna in them that they use to communicate to other insects, or emits a sounds or a scents that allows it to exchange information to other animals. Plants also have a special way to communicate to other plants through electric signalling or using others hosts to send their message.

From gigantic animals to small microscopic bacterias, viruses to cells, all have a special way to exchange message, they all have a way to coordinate information with each other.

Computers and software also requires a lot of communication.

Inside the computer hardware, electrical signals are sent to each individual components where they are task to perform a very specific function. And computers itself communicate to other computers via wire, wireless and The Internet.

It is hard to image a world without communication!

In programming, there are also various levels of ways to communicate to your code. In this chapter, we will discuss ways we can communicate to individual parts of code, one of them is variables.

If you need to pass, exchange and coordinate values that are bound to specific parts of the code, then we can use a variable.

Variables can either be a local variable that you can only access from inside the function, or global variable that you can globally access throughout your code.

For our first program example, we will create a function that takes a single word input, then output the word "Hello World".

Here is an example of a local variable in Ruby. In this example, we declared a local variable `hello` inside a function `say_hello`.

```ruby
def say_hello(message)
  hello = "Hello"
  puts hello + message
end

say_hello("World")
```

And when we run the Ruby code, it will output "Hello World".

```
$ ruby say_hello.rb

Hello World
```

However, if we want to change the `hello` variable to store a different message, we cannot directly change it from outside the function.

```
def say_hello(message)
  hello = "Hello"
  puts hello + message
end

hello = "Hi"
say_hello("World")
```

So even though we change the `hello` variable, this will still output a "Hello World".

```
$ ruby say_hello.rb

Hello World
```

Questions

1. What have we learned?
We learned that we can use a local variable to store a value that we can only access from inside the function.

2. Why do we need communication?
We need communication to exchange information.

3. What is a variable?
A variable is a name that you can use to store a value.

4. Why do you use a variable?
You use a variable to store a value that you can use later in your code.

5. How do you use a variable?
You use a variable by declaring it first, then you can assign a value to it.

6. How do we pass and coordinate information in our code?
We pass and coordinate information in our code by using variables.

7. What is a local variable?
A local variable is a variable that you can only access from inside the function.

8. What is a global variable?
A global variable is a variable that you can access from anywhere in your code.

Instance Variables

How do we change the information from inside different functions?

What we need is an instance variable. An instance variable is a variable that you can access via a single reference. In Ruby, instance variable starts with an `@` sign, followed by the variable name.

You can treat an instance variable like any normal variable, only that you can pass or modify information around it to other functions regardless of it's location.

So back to our example, we changed the local variable `hello` to `@hello`, then we can change the message of the variable.

```
def say_hello(message)
  @hello = "Hello"
  puts @hello + message
end

@hello = "Hi"
say_hello("World")
```

This will output a "Hi World" message.

```
$ ruby say_hello.rb

Hi World
```

This is very convenient, because we can pass values to other functions without having to initialize any object.

Questions

1. What have we learned?
We learned that we can pass values to other functions without having to initialize any object using instance variables.

2. Why is it useful?
It is useful because it is used to store data that is specific to an instance of a class or stores a value that is specific to a particular object.

3. What is an instance variable?
An instance variable is a variable that is associated with an instance of a class, that you can access via a single reference.

4. How do you create an instance variable in Ruby?
You can create an instance variable by using the `@` sign, followed by the variable name.

Struct

`struct` is a **type** that you can declaratively group and define data fields, which is accessible via a single reference. `struct` also provides a way to pass a value from within functions.

In Go, we can use structs to store a value from our functions. Here is an example of how to create a struct.

```
type Employee struct {
  FirstName string
  LastName  string
}
```

Using a struct, you can set a value and get a value from within your functions. This capability to exchange data throughout your code from within functions is useful for exchanging and coordinating data onto other parts of your code.

In Ruby, struct can be an equivalent to an **instance variable**.

Let's discuss this in detail in the following example, starting with a Ruby implementation on how do you pass values from within different parts of the code, then we can proceed on a Go example on how to pass values from function to another function.

In the following Ruby example, a class `Dog` has an initializer that accepts a parameter `breed` and set that parameter to an instance variable `@breed` which provides data access from within the class.

In this case, a public method `kind` directly returns the value passed to the initializer.

Ruby Instance Variables

```
class Dog
  def initialize(breed)
    @breed = breed
  end

  def kind
    @breed
  end
end

dog = Dog.new('Rottweiler')
dog.kind
```

```
$ ruby dog.rb

Rottweiler
```

The use of instance variables in the Ruby example can be rewritten in Go, using `struct`.

In the next example, a struct named `dog` defines a property `breed` with a value type `string` and inside `main()`. The struct `dog` initializes to a variable `kind` with its property `breed` filled in.

Go Private Struct

```go
package main

import "fmt"

type dog struct {
  breed string
}

func main() {
  kind := dog{
    breed: "Rottweiler",
  }

  fmt.Println(kind.breed)
}
```

```
$ go run dog.go

Rottweiler
```

Since `struct` can group similar data field types, you can extend this struct by adding multiple data fields.

```go
type dog struct {
  name  string
  breed string
  age   int
}
```

Then we can access multiple fields this way.

Go Private Struct

```go
package main

import "fmt"

type dog struct {
  name  string
  breed string
  age   int
}

func main() {
  pet := dog{
    name: "Maximus",
    breed: "Rottweiler",
    age: 5,
  }

  fmt.Printf("%+v", pet)
}
```

```
$ go run dog.go

{name:Maximus breed:Rottweiler age:5}
```

Questions

1. What have we learned?
We learned that struct is a type that you can declaratively group and define data fields, which is accessible via a single reference.

2. Why is it useful?
Struct is useful because it provides a way to pass a value from within functions.

3. What is a struct?
A struct is a user-defined type that is used to group together a number of variables of different types.

4. What is the equivalent of an struct in Ruby?
An instance variable.

5. How do you declare a variable in Go?
You declare a variable in Go by using the keyword var.

6. How do you create a struct in Go?
You can declare a struct using the `struct` *keyword, followed by the struct name, followed by the field name and type.*

7. What is a private struct in Go?
A private struct is a struct that is only visible to the package it is defined in.

8. What is a struct field in Go?
A struct field is a variable that is part of a struct.

Public Structs

Take note that the struct `dog` is only available internally, from within the package.

To make this public, the struct type alias name needs capitalization, in this case to `Dog`.

The same naming mechanics can also applicable to the *struct field names*, *functions* and *variables*. Go provide an option to make it's resource publicly available for other packages, by capitally them.

Go Public Struct

```go
package main

import "fmt"

type Dog struct {
  Breed string
}

func main() {
  kind := Dog{
    Breed: "Rottweiler",
  }

  fmt.Println(kind.Breed)
}
```

```
$ go run dog.go

Rottweiler
```

In most cases, the equivalent functionality of an *instance variables* in Go would be using `struct`. And you can extend `struct` by attaching it to a function, forming methods to a struct that you can perform *mutations* of existing values.

Questions

1. What have we learned?
Structs are a way to group data together, and you can also make it publicly available for other packages.

2. Why is it useful?
Sharing packages in Go is useful because it allows you to reuse code in different projects.

3. How do you make your Golang struct publicly available?
You can make your struct publicly available by capitalizing the struct type alias name.

4. What other Golang functionality that you can made public?
You can also make your functions and variables publicly available by capitalizing their names.

5. How do you mutate an existing struct values?
By attaching a struct to a function, you can perform mutations of existing values.

Attaching a Struct to a Function

Supposed we like to perform a method on a declared variable.

In Ruby, this is an OOP call, since Ruby can create methods in a class. This allows you to call those methods when you initialized the class using dot notation.

However, Go's first-class citizens are functions, and not an OOP language by design. Since, there's no concept of class, how do we attach a method from within a declared variable in a similar manner in Ruby?

In Go, we can associate a struct to a function, by specifying the type of that function as one of the declared structs. This will provide the capability to create methods from within functions.

In the following example in Ruby, an instance variable `produce` declares as a receiver. There are also three methods to operate on the variable, namely `add_item`, `change_item` and `items`.

In the `add_item` method, using a double-splat[1] parameter operator, we generate a hash from the method appending the values to `produce`.

The `change_item` method deletes the entry from the hash and then appends to `produce` as a new entry.

Operating On A Reciever Instance Variable in Ruby

```
class Basket
  def initialize
    @produce = []
  end

  def add_item(**entry)
    # Raise an error if name, flavour and
    # kind keys are not passed
    items = %w[name flavour kind]

    unless items.any? { |key| entry.key? key.to_sym }
      raise "Usage: add_item(name:    '..',
                             flavour: '..',
                             kind:    '..')"
    end

    @produce << entry

    puts "Entry #{entry[:name]} created!"
```

[1] https://ruby-doc.org/core-2.3.0/doc/syntax/calling_methods_rdoc.html#label-Hash+to+Keyword+Arguments+Conversion

```
  end

  def change_item(name, entry)
    # Delete existing record
    item = @produce
           .delete_if { |h| h[:name] == name }
           .first
           .dup

    item = entry

    # Add it to the instance variable
    @produce << item
    @produce.uniq!

    puts "Item #{name} changed!"
  end

  def items
    puts "There are #{@produce.count} number of items in
the basket"

    @produce.each do |entry|
      item(entry)
    end
  end

  private

  def item(entry)
    puts "Name: #{entry[:name]}"
    puts "Flavour: #{entry[:flavour]}"
    puts "Kind: #{entry[:kind]}"
  end
end

basket = Basket.new

basket.add_item(
  name: 'apple',
  flavour: "It's a little sour and bitter, but mostly
sweet, not at all salty, very juicy in general",
  kind: 'fruit'
```

```
)

basket.add_item(
  name: 'carrot',
  flavour: '',
  kind: 'veggies'
)

basket.change_item(
  'carrot',
  name: 'cucumber',
  flavour: 'Slightly bitter with a mild melon aroma,
and planty flavor.',
  kind: 'veggies'
)

basket.items
```

```
$ ruby basket.rb

Entry apple created!
Entry carrot created!
Item carrot changed!

There are 2 number of items in the basket

Name: apple
Flavour: It's a little sour and bitter, but slightly sweet,
not at all salty, juicy in general
Kind: fruit

Name: cucumber
Flavour: Slightly bitter with a mild melon aroma, and
planty flavor.
Kind: veggies
```

To rewrite this in Go, a struct needs to represent the `produce` with properties `name`, `flavour` and `kind`.

Another struct named `basket` will be an array of `produce`, since `basket` is an array of a struct that will contain the `produce` struct.

We will define a function and attach it to `basket` to interact with `produce`. We will call the defined `basket` functions as `add_item`, `change_item`, and `items`.

Operating on a Receiver Struct in Go

```go
package main

import "fmt"

type basket []produce

type produce struct {
        name    string
        flavour string
        kind    string
}

func (p *basket) add_item(entry produce) {
        *p = append(*p, entry)

        fmt.Printf("Entry %s created!\n", entry.name)
}

func (p basket) change_item(name string, entry produce) {
        for key, val := range p {
                if val.name == name {
                        p[key] = entry
                }
        }

        fmt.Printf("Item %s changed!\n", name)
}

func (p basket) items() {
        fmt.Printf("There are %d number of items in the basket\n",
                len(p))

        for _, val := range p {
                fmt.Printf(`Name: %s\n
                Flavour: %s\n
                Kind: %s\n`,
                        val.name,
                        val.flavour,
                        val.kind)
                fmt.Println("")
        }
}
```

```

func main() {
        basket := new(basket)

        basket.add_item(
                produce{
                        name: "apple",
                        flavour: `It's a little sour and bitter, but mostly
sweet, not at all salty, very juicy in general.`,
                        kind: "fruit",
                },
        )

        basket.add_item(
                produce{
                        name:    "carrot",
                        flavour: "",
                        kind:    "veggies",
                },
        )

        basket.change_item("carrot",
                produce{
                        name: "cucumber",
                        flavour: `Slightly bitter with a mild melon aroma
and planty flavor.`,
                        kind: "veggies",
                },
        )

        basket.items()
}
```

```
$ go run basket.go

Entry apple created!
Entry carrot created!
Item carrot changed!

There are 2 number of items in the basket

Name: apple
Flavour: It's a little sour and bitter, but slightly sweet,
not at all salty, juicy in general.
Kind: fruit

Name: cucumber
Flavour: Slightly bitter with a mild melon aroma and planty
flavor.
Kind: veggies
```

Questions

1. What have we learned?
We learned that we can attach a struct to a function in Go.

2. Why is it useful?
It's a way to pass data to a function.

3. How do you create a function with a struct in Go?
You can create functions with a struct by specifying the type of the function as one of the declared structs.

4. Is Golang an OOP language?
*No, Go is not an OOP language.

5. What is a double-splat parameter operator?
The double-splat operator is a way to pass multiple arguments to a function.

6. How do you create a double-splat parameter operator in Ruby?
*In Ruby, we can create a splat functionality by using the *args and **kwargs*

Pass-by-value and Pass-by-reference

In our above basket example, notice that there are two types of functions attached to `basket`?

It might be a new concept coming from Ruby, but when we are dealing with pointers and references, there are two semantics of passing a value to a variable.

Let's discuss them in detail.

Those are the two types of parameter passing in Go, namely *pointer receiver* and *value receiver*.

- *Value receivers* are *pass-by-value*, which means that the variable will use an actual value or a resulting value.
- *Pointer receivers* are *pass-by-reference* which takes in the memory address pointing to the value and pass it to the variable.

Ruby by default is pass-by-value, and in Go, we can use those two types of semantics of passing and using a variable value. We will discuss them further in detail.

When deciding the proper way of passing variables, the Go community had created a documentation on how to utilize the proper semantics between pointer receivers and value receivers. See the official Golang code review comment[2] for receiver types.

Questions

1. What have we learned?
We learned that there are two types of semantics of passing variables in Go.

2. Why is it useful?
It is useful because it helps us to understand the difference between the two types of semantics of passing variables.

3. What is pass-by-value?
Pass-by-value means that the value of the variable is passed to the function.

4. What is pass-by-reference?
Pass-by-reference means that the address of the variable is passed to the function.

5. Is Ruby a pass-by-value or pass-by-reference?
It's pass-by-value.

6. Is there a proper semantic in between pointer receivers and value receivers in Golang?
Yes, there is a community supported semantic in between pointer receivers and value receivers in Golang.

[2] https://github.com/golang/go/wiki/CodeReviewComments#receiver-type

Pointer Receiver

Pointer receiver is pass-by-reference, which means that we pass the reference to the memory address of the resulting value to a variable.

Pointer receivers will create a copy to a new variable referencing the memory address of the value per each assignment. When you make modifications to the parameter values, it will be a modification referencing to the memory address of the original variable's value.

To explain this further, let's go back to our previous example. In the previous example, `add_item` expects a `produce` argument, and the receiver of this function is `basket`.

Notice that there is an asterisk before the receiver type, `p *basket` which means that the receiver `basket` can be *accessed* and *mutated* directly within the function.

Pointer receivers are both setters and getters in Go, in the sense that the receiver `basket` is settable and gettable directly.

Pointer receivers can be an ideal use case when you are looking for an `attr_accessor` analog in Go, in comparison to the same behavior keen in Ruby. Here's the general form of constructing a pointer receiver.

Pointer Receiver in Go

```
func (receiverName *receiverType)
     funcName(paramName paramType) {
  // we can directly set the value of the
  // receiver type (setter)
  *receiverName = paramName

  // we can also access the value of the
  // receiver type (getter)
  fmt.Println(receiverName)
}
```

Questions

1. What have we learned?
Pointer receivers are a way to pass a pointer to a function.

2. Why is it useful?
It is useful because the function can use the pointer to access the data.

3. Is pointer receiver a pass-by-reference or pass-by-value?
Pointer receivers are pass-by-reference.

4. What is an equivalent attribute accessor in Golang?
Pointer receivers are an equivalent of attribute accessors in Go.

5. What is a pointer-receiver in Go?
A pointer-receiver is a function that takes a pointer as an argument.

6. Why there's an asterisk before the receiver type?
The asterisk before the receiver type is a pointer receiver, which means it can be accessed and mutated directly.

Value Receiver

Value receiver is pass-by-value, which means that we pass the actual value or a resulting value to a variable. Value receivers will create a new independent variable copying the original value per each assignment.

There are two functions in our previous example that utilize a value receiver function: those methods where the receiver type does not start with an asterisk, namely, the `change_item` and `items` functions which are also a method for `basket` struct.

You might ask: *Why does `change_item` mutate the value of the `basket` struct, does this mean that value receiver is also a setter? And the same with the pointer receiver?*

You can modify the parameter values but changes is not forwarded to the original variable. It is possible to have modification on existing variables with records, and value receivers can be setters of an initialized variable. However, any modifications to the variable will not reflect on new struct records. We still need to create the struct first, initialize it separately, before we can pass values to the new struct record.

Value receivers are ideal for concurrent applications because it does not modify the original reference, but creates a new copy of the reference, making value receivers thread-safe. To explain this much further, let's go back to our example above.

In our pointer receiver example, we used `append` to append a `produce` in a newly initialized `basket` array. The `change_item` mutates an already initialized `basket` variable but it cannot mutate the struct basket directly. Here's the general form of creating a value receiver.

Value Receiver in Go

```
func (receiverName receiverType)
     funcName(paramName paramType) {
  // we can set the value of an
  // _already initialised_ receiver type
  // but we cannot modify the receiver
  // type directly
  receiverName = parameterName

  // we can access the value of the
  // _already initialised_
  // receiver type (getters)
  fmt.Println(receiverName)
}
```

Questions

1. What have we learned?
Value receivers are a way to pass a value to a function, and creates a new copy of the original value

2. Why is it useful?
They are used when you want to pass a value to a function and not a reference to the value.

3. What is a value-receiver in Ruby?
A value-receiver is a class that has a method that accepts a value of another class as a parameter.

4. What is a value-receiver in Golang?
A value-receiver is a function that receives a value as an argument.

5. Are value receiver the same with pointer receiver?
No, value receivers are not the same with pointer receivers. Value receivers are pass-by-value, which means that we pass the actual value or a resulting value to a variable. Value receivers will create a new independent variable copying the original value per each assignment.

6. In value receivers, will any modifications to the variable reflect on new struct records?
No, any modifications to the variable will not reflect on new struct records. We still need to create the struct first, initialize it separately, before we can pass values to the new struct record.

7. Are value receivers thread-safe?
Yes, value receivers are ideal for concurrent applications because it does not modify the original reference, but creates a new copy of the reference, making value receivers thread-safe.

Decouple and Reuse Structs through Inheritance

One of the nicest feature of `struct` is the ability of decoupling the data structures into smaller chunks inheriting a common structure, allowing specific implementations for each data structures.

Modularizing your data structure is a good practice, by separately grouping your data structures into smaller chunks. When decoupling your structs into smaller chunks, you can maximize reusability of your struct, because it
allows you to interchange your struct into varied use cases, focusing on the content of each structure.

In the long run, your code will be easy to maintain, due to the modularized organization of your structures, adding the overall simplicity of using and maintenance of your code.

In the following example, we create a common `Animal` struct and a `Dog` struct. Specifying the name of the struct on top of the field inherits the struct and it's field properties. In this case, the Dog struct inherits the Animal struct and it's fields. Adding an asterisk before the name makes the inheritance as a pointer receiver, which allows modifications to the struct field values.

```
type Animal struct {
  Kind    string
  Habitat string
  Origin  string
  Diet    string
}

type Dog struct {
  *Animal
  Name  string
  Breed string
}
```

And we can also have the ability to inherit multiple structures. Here's the complete example where the `Dog` and `Cat` structs are inheriting both `Animal` and `Owner`, and in the `main()` we can add the values for each field.

```go
package main

import "fmt"

type Animal struct {
  Kind string
  Diet string
}

type Owner struct {
  Name    string
  Country string
}

type Dog struct {
  *Animal
  *Owner
  Name  string
  Breed string
}

type Cat struct {
  *Animal
  *Owner
  Name  string
  Breed string
}

func main() {
  var dog Dog

  dog.Name = "Maximus"

  dog.Animal = &Animal{
    Kind: "Dog",
    Diet: "Omnivorous",
  }

  dog.Breed = "Pitbull"

  dog.Owner = &Owner{
    Name:    "John",
    Country: "USA",
```

```
  }

  fmt.Printf("Name of %s is %s\n", dog.Animal.Kind,
dog.Name)
  fmt.Printf("%s is a %s with an %s diet\n", dog.Name,
dog.Breed, dog.Animal.Diet)
  fmt.Printf("%s is owned by %s who lives in %s\n",
dog.Name, dog.Owner.Name, dog.Owner.Country)
}
```

```
$ go run animal.go

Name of Dog is Maximus
Maximus is a Pitbull with an Omnivorous diet
Maximus is owned by John who lives in USA
```

Questions

1. What have we learned?
We learned that structs are a great way to organize your data, and you can use inheritance to create a common structure and then create specific structs that inherit the common structure.

2. Why is it useful?
It's useful because it allows you to create a common structure that can be used in multiple structs, and it allows you to create specific structs that inherit the common structure.

3. Can you decouple data structures in struct?
Yes, you can decouple data structures in struct.

4. Why do you want to decouple your structs in smaller chunks?
By decoupling your structs in smaller chunks, you can maximize reusability of your struct, because it allows you to interchange your struct into varied use cases, focusing on the content of each structure.

5. Why modularization a good practice?
In the long run, your code will be easy to maintain, due to the modularized organization of your structures, adding the overall simplicity of using and maintenance of your code.

6. What does adding an asterisk before a name in a struct field do?
Adding an asterisk before the name makes the inheritance as a pointer receiver, which allows modifications to the struct field values.

Anonymous Structs

An anonymous struct is a struct that you can declare and initialize on the fly without explicit declaration via `type`. It's declared in an inline manner along with your code, which provides a flexible way of declaration and usage of your data structures.

For a developer's perspective, this provides speedy invocations of your data structures when you need it because anonymous structs can immediately invoke your data structures as needed along with your code. Other cases where anonymous structs are useful is when a name is not needed for the operation.

Anonymous struct also provides a way to avoid needless declarations of type name alias which can cause namespace pollution for your struct. However, it's recommended to avoid deep nested anonymous structs, due to risk of unreadable code. This may result to a code that is hard to maintain.

Finally, anonymous structs is a good and cheap alternative to an empty `interface{}` type.

```go
package main

import (
  "fmt"
)

func main() {
  Animal := struct {
    Kind string
    Diet string
  }{"Dog", "Omnivorous"}

  fmt.Println(Animal.Kind, "-", Animal.Diet)
}
```

```
$ go run anonymous_struct.go

Dog - Omnivorous
```

Questions

1. What have we learned?
We learned that we can declare and initialize a struct on the fly without explicit declaration via type.

2. Why is it useful?
Anonymous structs are useful when a name is not needed for the operation.

3. What is an anonymous structs?
Anonymous structs are structs that you can declare and initialize on the fly without explicit declaration via `type`*. It’s declared in an inline manner along with your code.*

4. What is the advantage of anonymous structs for a developer?
Anonymous structs provides a speedy invocations of your data structures when you need it because anonymous structs can immediately invoke your data structures as needed along with your code. Other cases where anonymous structs are useful is when a name is not needed for the operation.

5. What problem does anonymous structs solve?
Anonymous structs provides a way to avoid needless declarations of type name alias which can cause namespace pollution for your struct.

6. What are the risks of using anonymous structs?
Anonymous structs are good and cheap alternative to an empty `interface{}` *type. However, it’s recommended to avoid deep nested anonymous structs, due to risk of unreadable code. This may result to a code that is hard to maintain.*

Anonymous Struct Fields

In OOP languages, properties of an object are collectively called attributes. Since Go is not an OOP language, attribute names in an OOP property referred to as fields.

Normally, a struct have a declared field name with an attached type, however, this is not always the case. You can also create struct without field names, leaving only its type.

The way to access the value of an anonymous field is by calling literal name of the field type of the struct, for example:

```go
package main

import (
  "fmt"
)

func main() {
  animals := struct {
    int
    string
  }{1, "Dog"}

  fmt.Println(animals.int)
  fmt.Println(animals.string)
}
```

```
$ go run anonymous_struct_fields.go

1
Dog
```

You can only access the value by calling the literal name of the type in the struct, and with this semantics, it comes with a limitation.

It is important to take note that when using anonymous field names, you have a limitation of having a declaration of the same type only once, since anonymous struct fields need to have unique value types, otherwise, there cannot be a distinction between each type.

Questions

1. What have we learned?
Anonymous struct fields are useful when you want to create a struct with a unique type, and you don't want to create a new type just for that.

2. Why is it useful?
Go's anonymous struct fields are a great way to pass data to functions without having to create a new type.

3. What is an anonymous struct fields?
Anonymous struct fields are struct fields without a name.

4. What is the limitations of an anonymous struct fields?
Anonymous structs are limited to a single type declaration.

5. What is a properties of an object in OOP?
In OOP, properties of an object are collectively called attributes.

6. What is the equivalent object attribute in Go?
An attribute is a property of an object. In Go, an attribute is a field of a struct.

7. How do you access the value of an anonymous field?
You can only access the value by calling the literal name of the type in the struct, and with this semantics, it comes with a limitation.

8. What is the limitation of an anonymous field?
It is important to take note that when using anonymous field names, you have a limitation of having a declaration of the same type only once, since anonymous struct fields need to have unique value types, otherwise, there cannot be a distinction between each type.

Chapter Questions

1. How do you create a function in Go?
You use the keyword func followed by the function name, a list of parameters in parentheses, and the function body.

2. How do you create a variable in Go?
You use the keyword var followed by the variable name and the type.

3. How do you create a pointer in Go?
*You use the keyword * followed by the type of the pointer and the variable name.*

4. What is a struct?
A struct is a collection of fields.

5. What is a field?
A field is a variable that is part of a struct.

6. How do you create a struct in Go?
You use the keyword struct followed by the struct name, a list of fields in curly braces, and the body.

7. What does capitalizing a struct, variable and function in Go does?
It makes it publicly accessible outside the package.

8. What is an anonymous struct?
An anonymous struct is a struct without a name.

9. Why use an anonymous struct?
Anonymous structs are useful when you want to create a struct on the fly.

10. Why do you modularized a struct?
Modularisation makes it easier to reuse the data structure defined in a struct.

Hash and Maps

If you need to store amounts of data programmatically in Go, that you can conveniently retrieve by its name and not by its index, then use a map. A map is storage of records that allows retrieval using a key mapping to a value. It is a good way to organize information for easy retrieval.

A key/value store database (i.e. Redis, Amazon DynamoDB, Riak, Tokyo/Kyoto Cabinet) works the same concept as a map.

Coming from Ruby, a map's equivalent is a hash. Hashes can be implicitly created using `{}` or idiomatically through `Hash.new`. Here's how to create a hash in Ruby. First, we initialize an empty hash by assigning `{}` to a variable named `basket`, then we iterate the contents and print the key name and its value.

Hash

```
basket = {}
basket[:fruits] = %w[apple mango avocado]
basket[:veggies] = %w[carrot cucumber kale]

basket.each do |key, val|
  puts "Key #{key} -- Value #{val.join(' ')}"
end
```

```
$ ruby basket.rb

Key fruits -- Value apple mango avocado
Key veggies -- Value carrot cucumber kale
```

In order for us to replicate this example in Go, we would use Go's built-in keyword called `map` and place it into a variable that maps a unique name to a value. But before we proceed on creating a `map`, we would need to understand two ways to initialize and allocate a `map`.

- Map by *declaration*, being able to initialize and allocate later.
- Map by *assignment*, a one-liner way to initialize and allocate a

map.

Questions

1. What have we learned?
We have learned that Go's map is a data structure that allows us to store key/value pairs.

2. Where this is useful?
This is useful when we need to store information that we can retrieve by its name and not by its index.

3. What is the benefit of using a map?
Go's map is a good way to organize information for easy retrieval.

4. What is a map?
A map is a collection of key-value pairs.

5. Why do you use a map?
You use a map to store data that you can retrieve by its key.

6. How do you create a map in Ruby?
You can create a map using the `{}` *literal or the* `Hash.new` *method.*

7. How do you initialize and allocate a map in Go?
The declaration of a map in Go is similar to the declaration of a struct. It is a composite type that is made up of a key and a value. The key is a string and the value is a type that can be any of the built-in types.

8. What is map by declaration?
Map by declaration is a way to initialize and allocate a map. It is done by declaring a map variable and assigning it a value.

9. What is map by assignment?
Map by assignment is a one-liner way to initialize and allocate a map. It is done by assigning a value to a map variable.

10. What examples of databases works like a map?
Redis, Amazon DynamoDB, Riak and Tokyo/Kyoto Cabinet

Maps by Declaration

One way to create a map is through declaration of the variable name using `var` and setting the *variable type* as `map[keyType]valueType`.

There are two ways to initialize a declared map, by *make* or by *literal type assignment*.

Initialization by Make

The most basic way to initialize a map is using `make`. Initializing your `map` this way offers an advanced usage because `make` can take a hint *size* argument for *reallocating* your `map` for resource and performance reasons. However, you don't need to set the hint *size* because it automatically handled by Go. Here is the general form and usage of declaring a map.

`map` hint *size* allocations are internally (and lazily) handled and value allocations are dependent on the architecture.

Maps by Declaration using Make

```
var variable map[keyType]valueType
variable = make(map[keyType]valueType)
variable[keyName] = Value

println(variable[keyName]) // Value
```

Here's an example of declaring a variable as a `map` and it's usage. In this example, we created a new variable `city` with a `map[string]string` type. Then we initialize the map via `make`, and created a string key `Netherlands` inside the `city` map, and assigned it a string value `Amsterdam`. Then we finally access the value of the key `Netherlands`, via `city["Netherlands"]`.

```
package main

import (
  "fmt"
)

func main() {
  var city map[string]string

  city = make(map[string]string)
  city["Netherlands"] = "Amsterdam"

  fmt.Println(city["Netherlands"])
}
```

```
$ go run map_init_by_make.go

Amsterdam
```

Questions

1. What have we learned?
We learned that we can declare a map variable and initialize it via make.

2. Why this is useful?
This is useful because we can initialize a map with a hint size, which is useful for performance and resource reasons.

3. Why do you want to use map initialization by `make`?
You want to use map initialization by make when you want to initialize your map with a hint size.

4. Do you always need to set a hint size?
No, you don't need to set a hint size. The hint size is automatically handled by Go.

5. How does Go allocate and initialize a map hint size?
Go allocates and initializes a map hint size by using the size of the key type.

Initialization by Literal Type Assignment

You can also create a variable by declaration and initialize its content at the same time by setting the type of the variable to `map[keyType]valueType`. Here the general form of maps by declaration and initialization in one line.

Maps by Declaration And Initialization in One Line

```
var variable = map[keyType]valueType{keyName: Value}

println(variable[keyName]) // Value
```

And here's a much specific example. In this example, we have assigned a map value to the declared `car` and set it's content to `{"Tesla":"Model 3"}`.

```
package main

import (
  "fmt"
)

func main() {
  var car = map[string]string{"Tesla": "Model 3"}

  fmt.Println(car["Tesla"])
}
```

```
$ go run map_init_by_literal_type_assignment.go

Model 3
```

If you would like to assign a value later, you can set an empty map value`{}` to the variable.

Maps by Declaration and Initialization on an Empty Map

```
variable := map[keyType]valueType{}
variable[keyName] = Value

println(variable[keyName]) // Value
```

And here's the same example as above, using empty map for later declaration.

```go
package main

import (
  "fmt"
)

func main() {
  car := map[string]string{}

  car["Tesla"] = "Model 3"

  fmt.Println(car["Tesla"])
}
```

```
$ go run map_init_by_literal_type_assignment_2.go

Model 3
```

Questions

1. What have we learned?
We learned that we can create a map by declaration and initialization in one line.

2. Why this is useful?
This is useful when you want to create a map and set its content at the same time.

3. How do you assign a value later?
You can set an empty map value `{}` to the variable.

4. How do you create a map by declaration?
You can create a map by declaration by declaring a variable name with the `map[keyType]valueType` type.

5. How do you initialize a declared map?
You can initialize a declared map by using `make` or by using `literal type assignment`.

6. How do you use 'make' in Go?
You can use `make` to create a map by declaration and initialize it at the same time.

7. How do you create a map via literal type assignment?
You can create a map via literal type assignment by declaring a variable name with the `map[keyType]valueType` type and setting the variable content to a map value.

Maps by Assignment

Maps can be both *initialized* and *created* on the go through *assignment*. There are two ways to store a value to an assigned map.

- Map By *key/value assignment*, being able to set the content of the assigned map variable in one expression.
- Map By *empty map assignment*, to assign value later.

Assignment with a Key/Value

In the following example, we created a map and assigned a value to a key in one expression.

Map by Assignment

```
variable := map[keyType]valueType{
  keyName: Value,
}

println(variable[keyName]) // Value
```

Assignment on an Empty Map

We can also create the variable, assign it into an empty map, then use it later like a declared map, for example.

Map by Assignment on an Empty map

```
variable := map[keyType]valueType{}
variable[keyName] = Value

println(variable[keyName]) // Value
```

Questions

1. What have we learned?
We learned that we can create a map on the go by assigning a key/value or an empty map.

2. What is maps by assignment?
Maps can be both initialised and created on the go through assignment.

3. What is map by key-value assignment?
Creates a map and assigned a value to a key in one expression.

4. What is map by empty map?
Creates the variable, assign it into an empty map, then use it later like a declared map.

Using Struct in Maps

Maps using struct is a useful way to contain your map information into data structures. We have discussed ways to create your maps *declaratively* or via *assignment*. In the following example, we will discuss how to create a map using a struct.

Struct Maps by Declaration

In this example, we first declare the struct `produce`, which we initialize via `make(map[string]produce)`. Then we assign a struct `produce` into a map key named `apple` and `kale`.

Map by Declaration using Struct Value Type

```
package main

import "fmt"

type produce struct {
	flavour string
	kind    string
}

func main() {
	var basket map[string]produce

	basket = make(map[string]produce)

	basket["apple"] = produce{
		flavour: `It's a little sour and bitter, but mostly
sweet, not at all salty, very juicy in general.`,
		kind: "fruit",
	}

	basket["kale"] = produce{
		flavour: `It boasts deep, earthy flavors that can range
from rich and meaty to herbaceous and slightly bitter.`,
		kind: "veggies",
	}

	for key, value := range basket {
		fmt.Printf(`Name: %s\n
		Kind: %s\n
		Flavour: %s\n\n`,
			key,
			value.kind,
			value.flavour)
	}
}
```

```
$ go run basket.go

Name: apple
Kind: fruit
Flavour: It's a little sour and bitter, but mostly sweet,
not at all salty, very juicy in general.

Name: kale
Kind: veggies
Flavour: It boasts deep, earthy flavors that can range from
rich and meaty to herbaceous and slightly bitter.
```

Struct Maps by Assignment

Here's another way to use the struct map by assignment. In this example, we create and initialize the map `basket` and assign its contents in one expression.

Map by Assignment using Struct Value Type

```go
package main

import "fmt"

type produce struct {
	flavour string
	kind    string
}

func main() {
	basket := map[string]produce{
		"apple": produce{
			flavour: `It's a little sour and bitter, but mostly
sweet, not at all salty, very juicy in general.`,
			kind: "fruit",
		},
		"kale": produce{
			flavour: `It boasts deep, earthy flavors that can
range from rich & meaty to herbaceous and slightly bitter.`,
			kind: "veggies",
		},
	}

	for key, value := range basket {
```

```
                fmt.Printf(`Name: %s\n
                Kind: %s\n
                Flavour: %s\n\n`,
                        key,
                        value.kind,
                        value.flavour)
        }
}
```

Struct Maps with Array Values

In order to assign multiple values to a map key, the valueType must be an array, with the general form of:

```
map[keyType][]valueType
```

Noticed that we prepended the `[]` before the `valueType`. In Go, this is a general form to define an array. In Ruby, arrays are also defined with an open and closing square brackets `[]`, in Go, the brackets needs to be explicitly defined in front of the valueType to form an array.

This form of defining an array is not only applicable to maps, but can also be the same way to initialize an array in variables and function arguments.

Using map by assignment, we can assign multiple values to a key. In our Ruby and Golang comparison in the first section of this article, we can rewrite our Ruby code above to the following Golang syntax:

Map with Array Values

```
basket := map[string][]string{
  "fruits":  []string{
             "apple",
             "mango",
             "avocado"},
  "veggies": []string{
             "carrot",
             "cucumber",
             "kale"},
}

for key, value := range basket {
  fmt.Printf(`Key %s -- Value %s \n`, key, value)
}
```

Questions

1. What have we learned?
We learned that we can create maps via struct by declaring the struct and assigning it into a map key.

2. Why this is useful?
This is useful because it allows us to create maps with structs, which is a common way to create data structures in Go.

3. Why do you use a map by struct?
Maps using struct is a useful way to contain your map information into data structures.

4. How do you create an array in Go?
The general form of defining an array is `[]valueType`.

5. How do you assign multiple values to a map?
The valueType must be an array, with the general form of `map[keyType][]valueType`.

Maps with Dynamic Types

An empty `interface{}` value type provides the ability to mix *strings, integers* or *dynamic values through function returns* as keys or values in your maps, variables or functions arguments.

To initialize an empty `interface{}` for your maps, you need to set it as either your key or value type. You can also intialize it independently, by assigning it as your variable type or function return type.

Here's an example of using an `interface{}` for a map to accept *integers*, *strings*, and *function return values* as a key type.

Interface Map Value Types

```
package main

import (
  "fmt"
  "github.com/google/uuid"
)

func main() {
  variable := map[interface{}]string{}

  variable[1] = "from an integer key"
  variable["a"] = "from string key"

```

```
  uuid := uuid.New()
  variable[uuid] = "from a UUID key"

  fmt.Println(variable[1]) // from an integer key
  fmt.Println(variable["a"]) // from string key
  fmt.Println(variable[uuid]) // from a UUID key
}
```

```
$ go run empty-interface.go

from an integer key
from string key
from a UUID key
```

An empty `interface{}` can provide you the same Ruby's *duck-typing interface behavior*. However, this convenience comes with a cost, and a cheap and fast alternative to this behaviour is using anonymous struct which is previously discussed in Chapter 1.

Questions

1. What have we learned?
We learned that Go's empty interface{} is a powerful tool to provide a dynamic typing behaviour.

2. Why this is useful?
This is useful when you want to accept a dynamic type of values as a key or value in your maps, variables or function arguments.

3. What is an empty 'interface{}' in Go?
An empty `interface{}` provides the ability to mix strings, integers or dynamic values through function returns as keys or values in your maps, variables or functions arguments.

4. How do you use an empty 'interface{}' in Go?
To initialize an empty `interface{}` for your maps, you need to set it as either your key or value type. You can also intialize it independently, by assigning it as your variable type or function return type.

Deleting Map Values

In Ruby, the easiest way to delete a value from a hash is by,

```
basket = { fruits: ["apple", "grapes"],
           veggies: ["kale", "cabbage"] }

basket.delete(:veggies)
```

and to avoid deleting the original hash, you can do,

```
basket = { fruits: ["apple", "grapes"],
           veggies: ["kale", "cabbage"] }

newbasket = basket.dup.tap { |produce| produce.delete(:veggies) }
```

In Go, you simply use the `delete` command.

```
variable := map[keyType]valueType{
  keyName: Value,
}

delete(variable, "keyName")
```

and going back to our example, you can delete a specific value from a map via `delete`,

```
package main

import (
  "fmt"
)

type produce struct {
  flavour string
  kind    string
}

func main() {
  var basket map[string]produce

  basket = make(map[string]produce)

  basket["apple"] = produce{
    flavour: `It's a little sour and bitter, but mostly
sweet, not at all salty, very juicy in general.`,
    kind:    "fruit",
```

```go
  }

  basket["kale"] = produce{
    flavour: `It boasts deep, earthy flavors that can range
from rich and meaty to herbaceous and slightly bitter.`,
    kind:    "veggies",
  }

  fmt.Printf("Map before delete %+v\n", basket)

  delete(basket, "kale")

  fmt.Printf("Map after delete %+v\n", basket)
}
```

```
$ go run map_delete.go

Map before delete map[apple:{flavour:It's a little sour and
bitter, but mostly sweet, not at all salty, very juicy in
general. kind:fruit}
kale:{flavour:It boasts deep, earthy flavors that can range
from rich and meaty to herbaceous and slightly bitter.
kind:veggies}]

Map after delete map[apple:{flavour:It's a little sour and
bitter, but mostly sweet, not at all salty, very juicy in
general. kind:fruit}]
```

Questions

1. What have we learned?
We learned that Go has a `delete` *command that can be used to delete a value in a map.*

2. Why this is useful?
This is useful because it is easy to use and it is not necessary to create a new map.

3. Why do you want to use map's delete command?
You want to use map's delete command when you want to delete a value from a map without creating a new map.

4. How do you delete a map entry in Go?
You use the `delete` *command.*

Reading a Non-Present Key from a Map

You can safely read a non-present key from a map in Go, and it will just return a zero values. Zero values are not literaly a zero integer, but are default values corresponding to each type. Depending on the type, there are different default values, for example:

- `false` for booleans
- `0` for integers
- `0.0` for floats
- `""` for strings
- `nil` for functions, interfaces, slices, pointers, channels, and maps

In this example, we are going to read a non-present key from a map with an `int` type, which will return the zero value (and the literal `0` value for int).

```go
package main

import (
  "fmt"
)

func main() {
  var basket map[string]int

  basket = make(map[string]int)

  basket["apple"] = 1
  basket["kale"] = 2

  if exists := basket["cabbage"] != 0; exists == false {
    fmt.Println("Item does not exists")
  }
}
```

```
$ go run map_read_non_present_key_1.go

Item does not exists
```

and we can rewrite this using the two form map assignment semantic

```
package main

import (
  "fmt"
)

func main() {
  var basket map[string]int

  basket = make(map[string]int)

  basket["apple"] = 1
  basket["kale"] = 2

  produce, exists := basket["cabbage"]

  if !exists {
    fmt.Println("Item does not exists")
  } else {
    fmt.Printf("%+v exists", produce)
    }
}
```

```
$ go run map_read_non_present_key_2.go

Item does not exists
```

with `zero` return values, you can do this operation safely without type checking

```
basket["cabbage"]++
```

If we need to create a more complex map that uses a struct, then a pointer is required to initialize the map, and the default return value will be nil. `nil` will be the default values for functions, interfaces, slices, pointers, channels, and maps.

For example,

```go
package main

import (
  "fmt"
)

type produce struct {
  flavour string
  kind    string
}

func main() {
  var basket map[string]*produce

  basket = make(map[string]*produce)

  basket["apple"] = &produce{
    flavour: `It's a little sour and bitter, but mostly
sweet, not at all salty, very juicy in general.`,
    kind:    "fruit",
  }

  basket["kale"] = &produce{
    flavour: `It boasts deep, earthy flavors that can range
from rich and meaty to herbaceous and slightly bitter.`,
    kind:    "veggies",
  }

  if exists := basket["cabbage"] != nil; exists == false {
    fmt.Println("Item does not exists")
  }
}
```

```
$ go run map_read_non_present_key_3.go

Item does not exists
```

Questions

1. What have we learned?
We can read a non-present key from a map, and it will return a zero value.

2. Why this is useful?
This is useful when we want to read a non-present key from a map, and we don't want to type check the key.

3. What is a zero values?
A zero value is a default value corresponding to each type.

4. What is the zero values of booleans?
`false`

5. What is the zero values of integers?
`0`

6. What is the zero values of floats?
`0.0`

7. What is the zero values of strings?
`""`

8. What corresponds a zero value of nil in Go?
Functions, interfaces, slices, pointer, channel, and map.

9. How do you read a non-present key from a map with 'int' type?
You can read a non-present key from a map with an `int` type, which will return the zero value (and the literal `0` value for int).

10. What other ways that you can safely read a non-present key from a map in Go?
You can use the `reflect.ValueOf` function to convert the map to a `reflect.Value` and then use the `reflect.Value.Elem` function to read the value.

Variadic Functions

Ruby has a way to capture arguments and convert to hash using double-splat[3] operator. In the following example, we have a method `add_item` that accepts a string and a double-splat, that appends a hash into the `basket` instance variable.

[3] https://ruby-doc.org/core-2.3.0/doc/syntax/calling_methods_rdoc.html#label-Hash+to+Keyword+Arguments+Conversion

Double Splat

```ruby
class Basket
  def initialize
    @basket = []
  end

  def add_item(kind, **item)
    @basket << item.merge(kind: kind)

    puts "Entry #{item[:name]} created!"
  end

  def print_items
    puts "There are #{@basket.count} item(s) in the basket."

    @basket.each do |entry|
      puts "Name: #{entry[:name]}"
      puts "Flavour: #{entry[:flavour]}"
      puts "Kind: #{entry[:kind]}"
    end
  end
end

basket = Basket.new

basket.add_item(
  'fruit',
  name: 'apple',
  flavour: "It's a little sour and bitter, but mostly sweet,
not at all salty, very juicy in general."
)

basket.print_items
```

```
$ ruby basket.rb

Entry apple created!
There are 1 item(s) in the basket.

Name: apple
Flavour: It's a little sour and bitter, but mostly sweet,
not at all salty, very juicy in general.
Kind: fruit
```

Using a variadic operation, Go expects the argument to be an array, however, there are various ways to do this. We will discuss using variadic on an empty interface to accomplish a similar behavior of a double-splat operator in Ruby that converts to a map.

Questions

1. What have we learned?
We learned that we can use variadic functions to emulate Ruby's double-splat behaviour.

2. Why this is useful?
This is useful if you want to pass multiple arguments to a function and you want to access them in a map-like way.

3. When this is not useful?
This is not useful when we want to accept a fixed number of arguments.

4. What is the purpose of double-splat operation in Ruby?
It is used to call a method on a receiver and pass arguments to it.

5. What is the equivalent of double-splat operations in Go?
A variadic map interface.

6. What is a variadic map interface in Go?
A variadic map interface is a map interface that can have any number of entries.

Variadic Interface

One way to do a functionality similar to Ruby's double-splat operator in Go is using variadic[4] functions to an empty interface[5].

Previously, I have discussed how you can use an `interface{}` in your map key type. The following example will explain using interface as a way to emulate Ruby's double-splat operator in detail.

[4] https://en.wikipedia.org/wiki/Variadic_function
[5] https://tour.golang.org/methods/14

In the following example, we created a `basket` function accepting a variadic function `...` to `map[string]interface{}` so that we can pass multiple arguments to the function. Then we pass a map and a string as an argument to the `basket` function, then we access the map's values using `item["name"]`, `item["flavour"]` and `kind`.

Double Splat Interface Example 1

```
1  package main
2  
3  import "fmt"
4  
5  func main() {
6          basket(map[string]string{
7                  "name": "apple",
8                  "flavour": `It's a little sour and bitter, but mostly
9  sweet, not at all salty, very juicy in general.`,
10         }, "fruit")
11 }
12 
13 func basket(args ...interface{}) {
14         item := args[0].(map[string]string)
15         kind := args[1]
16 
17         fmt.Printf(`Name: %s\n
18              Flavour: %s\n
19              Kind: %s\n`,
20                item["name"],
21                item["flavour"],
22                kind)
23 }
```

```
$ ruby basket.rb

Name: apple
Flavour: It's a little sour and bitter, but mostly sweet,
not at all salty, very juicy in general.
```

Noticed that on line 15, we have an expression `.(map[string]string)` to the first argument, this is because the variadic `...interface{}` will act as an array container of the arguments you pass into it and to access the argument, it needs to be *type asserted* back to `map[string]string`.

Here's another example, which is just an expanded version of the previous example but using pointer *receiver functions*[6].

[6] https://tour.golang.org/methods/4

Double Splat Interface Example 2

```
package main

import "fmt"

type basket []map[string]string

func main() {
        basket := new(basket)

        basket.add_item(map[string]string{
                "name": "apple",
                "flavour": `It's a little sour and bitter, but mostly
sweet, not at all salty, very juicy in general.`,
        }, "fruit")

        basket.print_items()
}

func (items *basket) add_item(args ...interface{}) {
        item := args[0].(map[string]string)

        item["kind"] = args[1].(string)
        *items = append(*items, item)

        fmt.Printf("Entry %s created!\n", item["name"])
}

func (items basket) print_items() {
        fmt.Printf("There are %d item(s) in the basket.",
                len(items))

        for _, item := range items {
                fmt.Printf(
                        `Name: %s\n
      Flavour: %s\n
      Kind: %s\n`,
                        item["name"],
                        item["flavour"],
                        item["kind"],
                )
        }
}
```

Questions

1. What have we learned?
We learned that Go has a variadic function that can accept an array of arguments.

2. Why this is useful?
This is useful when we want to accept a variable number of arguments.

3. What is the purpose of variadic?
The purpose of variadic is to accept a variable number of arguments.

4. Where do you put a variadic interface in Go?
You can put a variadic interface in a function's parameter list.

5. How do you create a variadic interface function in Go?
You can create a variadic interface function by using the `...interface{}` syntax in it's parameters.

6. How do you retrieve a variadic value?
You can retrieve a variadic value by using the `.(interface{})` syntax.

Maps with Variadic Interface

We can pass a variadic interface as an array to a method accepting a variadic argument by prepending triple dots on the variable name, i.e. `...variable` like on the following example.

Noticed that on the `basket` method, we have used an iterator `for` loop to process the `basket(produce...)` as an array, then using `fmt.Springtf("%T", item)` we can test if the passed variable is a map. This is similar to Javascript's `typeof` or Ruby's `is_a?(...)`. Then we can *type assert* the iterated `item`.

Variadic Array of Mixed Arguments

```
package main

import "fmt"

func main() {
	produce := []interface{}{
		map[string]string{
			"name": "apple",
			"flavour": `It's a little sour and bitter, but mostly
sweet, not at all salty, very juicy in general.`,
```

```
                        "kind": "fruit",
                },
                map[string]string{
                        "name": "cucumber",
                        "flavour": `Slightly bitter with a mild melon aroma
and planty flavor.`,
                        "kind": "veggies",
                },
                "water",
        }

        basket(produce...)
}

func basket(args ...interface{}) {
        for _, item := range args {
                item_type := fmt.Sprintf("%T", item)

                if item_type == "map[string]string" {
                        produce := item.(map[string]string)

                        fmt.Printf(`Name: %s\n
                  Flavour: %s\n
                  Kind: %s\n\n`,
                                produce["name"],
                                produce["flavour"],
                                produce["kind"])
                } else {
                        fmt.Println(`You have passed a non-fruit or
non-veggies argument`, item.(string))
                }
        }
}
```

```
$ go run variadic_array_of_mixed_arguments.go

Name: apple
Flavour: It's a little sour and bitter, but mostly sweet,
not at all salty, very juicy in general.
Kind: fruit

Name: cucumber
Flavour: Slightly bitter with a mild melon aroma and planty
flavor.
Kind: veggies

You have passed a non-fruit or non-veggies argument water
```

Hashes or Maps is a useful way to organize records in your program. Use it more and turn it into a workhorse of your project.

Questions

1. What have you learned?
We have learned how to pass a variadic interface as an array to a method accepting a variadic argument.

2. Where this is useful?
This is useful when you want to process a variadic interface as an array.

3. Why do you want to use a variadic interface?
You want to use a variadic interface when you want to accept a variable number of arguments.

4. When this is not useful?
When we need to pass a fixed number of arguments to a function.

5. How do you pass a variadic interface as array?
Use `...variable`

6. What is type assertion?
Type assertion is a way to test if a variable is of a certain type.

7. In Javascript and Ruby, how do you type assert a variadic interface value?
*Use `typeof` or `is_a?`

8. In Golang, how do you type assert a variadic interface value?
Use `fmt.Sprintf("%T", item)`

9. What's the difference between `...variable` and `variable...`?
`...variable` is a variadic interface, `variable...` is a variadic argument.

Chapter Questions

1. How do you create a function with a parameters and return values in Go?
func add(a, b int) int

2. How do you create a function with parameters as variadic type in Go?
func add(a, b, ...int) int

3. How do you declare a map in Go?
var m map[string]int

4. How do you initialize a map in Go via make?
m := make(map[string]int)

5. How do you initialize and assign a contents of a map in Go?
m := map[string]int{"one": 1, "two": 2}

6. Why do you use a map variable in Go?
To store a key-value pair.

7. Why do you use a variadic variables in Go?
To pass a variable number of arguments to a function.

8. Why do you use an empty `interface{}` in Go?
To pass a value of any type to a function.

9. How do you get the value of a map key?
m["key"]

Arrays and Slices

If you need a quick way to store a fixed number of data to a variable that you iterate over, then use array. Arrays are cheap and fast storage of a finite number of data that you can call sequentially, or by its index. Use it when you have to store data of the same type that you need to process for looping.

In Computer Science, arrays are an O(n) solution[7], it's best for looping through multiple sets of data. The difference between arrays and maps, is that maps is an O(1) approach.

> Big O(n) notation[8] means the uncertainty of things you have to do in ***n*** number of ways. Which means processing them and searching them will take a certain degree of steps to come up its value, also every step involves growing complexities, resource consumption, and time spending.

Because arrays are performant and efficient way to process multiple amount of data, it is a preferred way of storage of data when you need to process them *sequentially*, by *batch* or by *bulk*.

Coming from Ruby, arrays can be created either by initializing a variable to a blank array `[]` or idiomatically using the array class name `Array.new`.

Here is an example of initializing and using an array in Ruby, we first initialized an array to a variable, and add `"foo"` to the first index, then use the array append operator `<<` to append `"bar"` to the array, and finally prints the value of the array based on their array index which always starts with `0`.

Array in Ruby

```
array = []

array[0] = 'foo'
array << 'bar'

puts array[0] # > foo
puts array[1] # > bar

array # > ["foo", "bar"]
```

[7]https://en.wikipedia.org/wiki/Search_data_structure#Asymptotic_amortized_worst-case_analysis
[8]https://en.wikipedia.org/wiki/Big_O_notation

```
$ ruby array.rb

foo
bar
```

In Go, there are two types of array classifications with a different type of initializations, namely the *Fixed Arrays* and *Slices*. Both *Fixed Arrays* and *Slices* do the same thing what an array does, they both store amounts of data that can be accessed by index. However, there are a number of differences between the two that we will discuss further.

It is easy to be overwhelmed with unexpected behaviors of an array as experienced in high-level programming languages like Ruby. It's better to recognize the difference and its usage between a *fixed array* and a *slice* in Go, which will be discussed in this article.

Questions

1. What have we learned?
Arrays are a fast and efficient way to store a fixed amount of data that you can iterate over sequentially.

2. What is an array?
An array is a collection of data that can be accessed by index.

3. What are the array classifications in Go?
There are two types of array classifications in Go, namely the Fixed Arrays and Slices.

4. Why is it important to know the difference between fixed array and a slice in Go?
It is important to know the difference between a fixed array and a slice, because it will help you to choose the right data structure for the right job.

5. What is the difference between fixed and slice array in Go?
The difference between a fixed array and a slice array is that a fixed array is a constant size array, while a slice array is a variable size array.

6. How to sequentially retrieve the values of an array in Go?
Use the for loop.

7. What is a Big O Notation?
Big O Notation is a way to describe the complexity of a function or algorithm. It is a way to describe the uncertainty of things you have to do in n number of ways. Which means processing them and searching them will take a certain degree of steps to come up its value, also every step involves growing complexities, resource consumption, and time spending.

8. What is the Big O Notation of an Array?
The Big O Notation of an Array is O(n).

9. What is the Big O Notation of a Map?
The Big O Notation of a Map is O(1).

Fixed Array

If you have a box with partitions, you can only put a certain number of items in that box, which you can retrieve if you know on which part of the partition it is stored. This is how a *Fixed Array* works, this type of array in Go uses a *fixed array size* where you can specify the maximum *size* of the array. The following example is the Go rewrite of the Ruby example above. We have declared an array variable with a *string* type, with a max array *size* of 2. Then we store the strings `"foo"` and `"bar"` to the variable.

Fixed Array Size in Go

```go
const MAX_ARRAY_SIZE = 2

var array [MAX_ARRAY_SIZE]string

array[0] = "foo"
array[1] = "bar"

fmt.Println(array[0])
fmt.Println(array[1])
fmt.Println(array)
```

Questions

1. What have we learned?
We learned that we can use a fixed array to store a certain number of items.

2. Why do you want to use a fixed array?
If you know the maximum size of the array, you can allocate the memory for the array in advance.

3. When not to use a Fixed Array?
If you don't know the maximum size of the array, you can use a dynamic array.

4. How do you create a Fixed Array variable in Go?
You can create a fixed array variable by declaring the array variable with the type of the array and the size of the array.

5. What is a syntax of creating a Fixed Array variable in Go?
The syntax of creating a fixed array variable in Go is `var array [MAX_ARRAY_SIZE]string`.

Fixed-Array Automatic Size Calculation

Go can automatically calculate the size of the declared array by specifying a triple dot `...` notation to the size of the array inside the square brackets before the type. In the following example, we created a variable named `fruits` that was automatically set to a string array with a fixed size of 4, which allows us to assign the variable to the `flavours` string array with the same size.

Auto Array Size in Go

```go
var flavours [4]string

fruits := [...]string{"Apple",
                      "Mango",
                      "Orange",
                      "Banana"}

flavours = fruits

fmt.Println(flavours)
```

However this does not mean that it also automatically calculates the destination type, so the following example will throw an error.

Auto Array Size Mismatch

```go
var flavours [4]string

fruits := [...]string{"Apple",
                      "Mango",
                      "Orange",
                      "Banana"}

flavours = fruits

fmt.Println(flavours)
```

Questions

1. What have we learned?
Go can automatically calculate the size of the declared array.

2. Why is it useful?
It can save us from writing a lot of code.

3. How can Go automatically calculate the size of a declared array?
By specifying a triple dot `...` notation to the size of the array inside the square brackets before the type.

4. What are the caveats when using automatic size calculation?
It can only calculate the size of the declared array, but not the destination type.

5. When not to use automatic size calculation?
When the size of the array is not the same as the destination type.

Fixed-Array Sizes

In Ruby, you can assign any array variable to an array variable. However, in Go, the array type and element size needs to be exactly the same when assigning to another array.

Basic Array in Ruby

```
fruits = %w[apple mango kiwi avocado]
flavours = %w[pineapple tomato]

flavours = fruits

puts flavours
```

```
$ ruby array.rb

apple
mango
kiwi
avocado
```

When assigning an array to another array variable in Go, the destination array should be the same type of array from the source. For instance, the following example will give an error.

Array Types in Go

```
var flavours [6]string

fruits := [4]string{"apple",
                    "mango",
                    "kiwi",
                    "avocado"}

flavours = fruits
// error

fmt.Println(flavours)
```

This is because Go is a statically typed language[9], which means that the declared destination and source types should also be the same. This rule also applies on other variable types as well, like a string variable cannot be stored in an integer variable.

Questions

1. What have we learned?
We learn that Go is a statically typed language, which means that the declared types should be the same when assigning to another variable.

2. Why this is important?
This is important because it helps to prevent bugs from occurring.

3. What is statically typed language?
A statically typed language is a programming language that requires variables and function arguments to be declared with a specific type.

4. What is the caveat when assigning an array of different types?
The caveat is that you cannot assign an array of one type to an array of another type.

5. Why is the array size of the source should be the same as destination?
The reason is that the size of the array is used to determine the number of elements in the array. If the size of the source and destination are different, the destination array will be resized to the size of the source.

Fixed-Array Assignment Behaviour

Fixed Arrays are *value types*[10], which means that when you assign a fixed array to another fixed array, it will copy all the contents of the *source* as a new value. Any changes to the old array will

[9] https://en.wikipedia.org/wiki/Type_system#STATIC
[10] https://en.wikipedia.org/wiki/Value_type

be independent of the values from the new array. For instance, in the following example, we have declared two array variables, namely `oldArray` and `newArray`. Then we assigned the contents of `oldArray` to `newArray` on line 11. As expected we will get the `"foo"` value as the first index of `oldArray`, as `oldArray[0]` to `newArray[0]`. Then we assign the strings `"baz"` and `"bar"` to the second index of both arrays.

Fixed Array Value Type

```go
1  package main
2
3  import "fmt"
4
5  func main() {
6    var oldArray [2]string
7    var newArray [2]string
8
9    oldArray[0] = "foo"
10
11   newArray = oldArray
12
13   newArray[1] = "baz"
14   oldArray[1] = "bar"
15
16   fmt.Println(oldArray) // [foo bar]
17   fmt.Println(newArray) // [foo baz]
18 }
```

```
$ go run array.go

[foo bar]
[foo baz]
```

As you noticed, upon printing the values of both arrays, on line 16 and 17, the values of both arrays are independent of each other. We both inserted strings on both arrays and they both have independent values. This is the expected behavior of a *Fixed Array* being a value type, however, this can get really expensive on large data sets, and for this scenarios, we would require a flexible type of array, the *Sliced Array*.

Questions

1. What have we learned?
Fixed Arrays are value types, which means that when you assign a fixed array to another fixed array, it will copy all the contents of the source as a new value. Any changes to the old array will be independent of the values from the new array.

2. Why is this important?
This is important because it is a common mistake to think that a fixed array is a reference type, and that when you assign a fixed array to another fixed array, it will reference the same data.

3. How do you copy an array in Go?
You can use the built-in copy function to copy an array.

4. Why is fixed array expensive on large data sets?
Because it is a value type, and value types are copied by value, which means that the old array will be copied as a new value, and any changes to the old array will be independent of the values from the new array.

5. When to use fixed array?
When you have a small data set and you don't need to change the data set.

Sliced Array

Sliced Array is a much cheaper and flexible type of array, which allows you to resize the array using `append`, a feature that is not possible in a *Fixed Array*. Also, all of Go's standard library and public API uses *Sliced Array*. A sliced array can be initialized in two ways, using `make` or direct array assignment without a fixed size, `[]Foo{}`, `[]Foo{bar1...barN}`.

Sliced Array Initialization

```go
slicedArray1 := make([]string, 2)

var slicedArray2 []string

slicedArray3 := []string{}
slicedArray4 := []string{"foo", "bar"}
```

Fixed Arrays also allows you to use Go's built-in `append` and `copy` operations.

Go Sliced Array

```
const MAX_ARRAY_SIZE = 2

array := make([]string, MAX_ARRAY_SIZE)

array[0] = "foo"
array[1] = "bar"

fmt.Println(array[0])
fmt.Println(array[1])
fmt.Println(array)
```

Questions

1. What have we learned?
Sliced array is a much cheaper and flexible type of array, which allows you to resize the array using append, a feature that is not possible in a Fixed Array.

2. What is a Slice Array?
A slice is a reference to an array. Slices are used to represent arrays, or subsets of arrays, or multi-dimensional arrays.

3. Why is sliced array much cheaper?
Because it is a pointer to an array, not the array itself.

4. What is the syntax of a sliced array?
The syntax of a sliced array is the same as a normal array, but with a colon (:) after the array name.

5. How do we adjust the size of an sliced array?
We can use the built-in function append() to append elements to the end of a sliced array.

Sliced-Array Assignment Behaviour

Sliced Array is *reference types*[11] which means that when you pass or assign a value from a sliced array, only it's reference will be returned, not a new copy, so it is much cheaper and efficient.

For example, we have four sliced array variables, then add strings to each variable, but upon printing all variable contents, we saw that all four variables contain the same value. This means that when we modify a sliced array variable, we are also modifying the original source of the sliced array it is assigned from.

[11]https://en.wikipedia.org/wiki/Reference_type

Sliced Array Reference Types

```go
package main

import "fmt"

func main() {
  slicedArray1 := make([]string, 4)

  var slicedArray2 []string

  slicedArray3 := []string{}
  slicedArray4 := []string{"foo", "bar"}

  slicedArray1[0] = "foo"
  slicedArray2 = slicedArray1

  slicedArray2[1] = "bar"
  slicedArray3 = slicedArray2

  slicedArray3[2] = "baz"
  slicedArray4 = slicedArray3

  slicedArray4[3] = "quux"

  fmt.Println(slicedArray1) // [foo bar baz quux]
  fmt.Println(slicedArray2) // [foo bar baz quux]
  fmt.Println(slicedArray3) // [foo bar baz quux]
  fmt.Println(slicedArray4) // [foo bar baz quux]
}
```

```
$ go run array.go

[foo bar baz quux]
[foo bar baz quux]
[foo bar baz quux]
[foo bar baz quux]
```

Questions

1. What have we learned?
Sliced array is reference types, so when we assign a sliced array to a new variable, we are not creating a new copy of the original array, but instead, we are creating a new variable that points to the same array as the original variable.

2. Why is this useful?
This is useful because it is much more efficient and cheaper to use sliced array than creating a new copy of the original array.

3. What is the assignment behaviour of a sliced array?
When we assign a sliced array to a new variable, the new variable will point to the same array as the original variable.

Capacity

In Go, there is a third parameter when creating a Slice Array using `make`. *Make* takes the following parameters.

```
make(type, length, capacity)
```

By default, when you initialized a sliced array without specifying the third parameter, it would automatically take the second parameter as the value of the third, so

```
array := make([]string, 2)
```

is the same as

```
array := make([]string, 2, 2)
```

The third parameter is the *capacity*. So you might say '*Oh okay, it's the maximum capacity of the array. So I can do something like*':

Make Capacity Assumptions

```go
array := make([]string, 2, 3)

array[0] = "foo"
array[1] = "bar"

array[2] = "baz"

// oops, error
```

Well not exactly. In Go, capacity means, '*The remaining capacity of the new array initialized after slicing*'. So what does it mean?

So back in our example, we initialized a new slice with length 2 and capacity 3, and since we have a maximum length of two arrays as our second parameter in `make`, we can assign two values to `array`.

```go
array := make([]string, 2, 3)

array[0] = "foo"
array[1] = "bar"
```

So if I assign the third value to `array`, it would result in an error.

```go
// array[2] = "baz"
// error
```

Then we use the slice *capacity* parameter

```go
array2 := array[:3]
```

What we just did is that we slice using the `[:]` syntax and assign it to a new variable `array2`. We can now use the maximum capacity of `array`, so this makes the newly referenced array `array2` can hold the 3rd value.

```go
array2[2] = "baz"
```

And trying to assign the 4th value will result in an error.

```
// so this would results to an error
// array2[3] = "foobar"
```

Printing the results, we'll get something like `[foo bar]` for `array`, and `[foo bar baz]` for `array2`.

```
fmt.Println(array)
fmt.Println(array2)

// [foo bar]
// [foo bar baz]
```

Noticed that we have the existing elements of the old `array` to `array2`. Slicing a slice to a new variable will only reference the values of an old array to a new array variable using the existing memory address of the old array values, for example:

Go Array Memory Pointer

```
1  basket1 := make([]string, 1, 3)
2  basket1[0] = "apple"
3
4  basket2 := basket1[:2]
5  basket2[1] = "mango"
6
7  basket3 := basket2[:3]
8  basket3[2] = "banana"
9
10 fmt.Println(basket3) // [apple mango banana]
11
12 basket3[0] = "banana"
13 basket3[1] = "banana"
14 basket3[2] = "banana"
15
16 fmt.Println(basket3) // [banana banana banana]
17 fmt.Println(basket2) // [banana banana]
18 fmt.Println(basket1) // [banana]
```

In line 12 up to 17, what we intended is modify only the contents of `basket3`, but instead, the contents of `basket2` and `basket1` also changed. This is because we are sharing the same memory address of the previous array.

Questions

1. What have we learned?
We learned that when we slice a slice, we are not creating a new array, but instead, we are referencing the same memory address of the previous array.

2. Why is this important?
This is important because we can't assume that the capacity of the new array is the same as the length of the old array.

3. What is the third parameter of creating a sliced array using `make`?
The third parameter is the capacity of the new array.

4. Why do we set the capacity of an array slice in Go?
To avoid the error of trying to assign a value to an array that is out of bounds.

Deep Copy

If you want to copy the values of the old array to a new array variable and modify it without affecting the values of the old array, you can use a fixed array assignment or you would use a combination of `make` and `copy`.

In the following example, we will create a new basket array that can be modified without affecting the original array is referenced.

Deep Copy

```
basket1 := make([]string, 1, 3)
basket1[0] = "apple"

basket2 := basket1[:2]
basket2[1] = "mango"

basket3 := make([]string, cap(basket1))

copy(basket3, basket2)

basket3[1] = "pineapple"
basket3[2] = "kiwi"

fmt.Println(basket3) // [apple pineapple kiwi]
fmt.Println(basket2) // [apple mango]
fmt.Println(basket1) // [apple]
```

On line 7 and 8, we used `make` to initialize a new array and assign it to `basket3`, then on line 8 we `copy` the contents of `basket2` to `basket3`, then we can make modifications to an existing data in the new array as a new variable.

Questions

1. What have we learned?
We can use make to create a new array and copy the contents of an existing array to the new array.

2. What is array Deep Copy?
Deep Copy is a copy of an array that can be modified without affecting the original array.

3. Why is Deep Copy useful?
Deep Copy is useful when you want to make changes to an existing array without affecting the original array.

4. When not to use Deep Copy?
When you want to make changes to the original array.

5. How do you Deep Copy an Array in Go?
Use make and copy.

Append

We cannot go beyond the current capacity further when reassigning to another variable via slice syntax, for example:

Array Capacity

```
array := make([]string, 1, 2)
array[0] = "foo"

fmt.Println(array) // [foo]

array2 := array[:2]
array2[1] = "bar"

fmt.Println(array2) // [foo bar]

array3 := array[:3]

// error

```

```
array3[2] = "baz"
fmt.Println(array3) // [foo bar baz]
```

What we need is using Go's built-in `append` function to extend an array to a new variable.

Array Append

```
1  array := make([]string, 1, 2)
2  array[0] = "foo"
3
4  fmt.Println(array) // [foo]
5
6  array2 := array[:2]
7  array2[1] = "bar"
8
9  fmt.Println(array2) // [foo bar]
10
11 array3 := array[:2]
12 array3 = append(array3, "baz")
13
14 fmt.Println(array3) // [foo bar baz]
```

On line 10, we used `append` to array3, extending the length of array3.

Questions

1. What have we learned?
We can use append to extend an array.

2. Why is it useful?
Appending an array is useful when we need to extend an array to a new variable.

3. What are the dangers of using array append?
If we append to an array that is not large enough to hold the new values, the array will be reallocated and the old values will be lost.

Arrays with Variadic Types

In Ruby, there's a catch-all method argument that can be used to store multiple parameters into array or hash. In this example, we add a splat asterisk in the method argument, so that we can capture the parameters as an array.

Ruby Splat Operator

```
def basket(*fruits)
  fruits.each do |fruit|
    puts "#{fruit} fruit is in the basket"
  end
end

basket('Apple', 'Mango', 'Orange', 'Banana')
```

```
$ ruby splat.rb

Apple fruit is in the basket
Mango fruit is in the basket
Orange fruit is in the basket
Banana fruit is in the basket
```

In Go, the equivalent functionality similar to Ruby's catch-all splat argument is the triple dot, ... argument, or the variadic[12] functions.

Go Variadic Operator

```
package main

import "fmt"

func basket(fruits ...string) {
  for _, fruit := range fruits {
    fmt.Printf("%s is in the basket\n", fruit)
  }
}

func main() {
  basket("Apple", "Mango", "Orange", "Banana")
}
```

[12]https://golang.org/ref/spec#Passing_arguments_to_..._parameters

```
$ go run variadic.go

Apple is in the basket
Mango is in the basket
Orange is in the basket
Banana is in the basket
```

In this example, we have `fruits ...string` argument in the `basket` function, which acts as a similar to the splat operator in Ruby. Then we iterate over the fruits array and print them. If you would pass an array variable to another function accepting a variadic argument, you would need to pass the variable proceeded with three dots `...`. For example, on line 16, we pass the variadic `fruits...` to the `basket` function.

Variadic Passing

```
1  package main
2
3  import "fmt"
4
5  func basket(fruits ...string) {
6    for _, fruit := range fruits {
7      fmt.Printf("%s is in the basket\n", fruit)
8    }
9  }
10
11 func main() {
12   fruits := []string{"Apple",
13                      "Mango",
14                      "Orange",
15                      "Banana"}
16
17   basket(fruits...)
18 }
```

```
$ go run variadic-passing.go

Apple is in the basket
Mango is in the basket
Orange is in the basket
Banana is in the basket
```

Questions

1. What have we learned?
We learn that the splat operator in Ruby is similar to the variadic function in Go.

2. What is Ruby single-splat argument?
It is a catch-all argument that can be used to store multiple parameters into array or hash.

3. Why is this useful?
It is useful when you want to store multiple parameters into an array.

Empty Interface Array Type

Noticed that arrays and slices are only bound to their array types, those are Go's basic types, which consist of *strings*, *integer*, *float* and *boolean*. If we mixed non-string values to an array with basic types, it would result in an error.

Mixed Array Values on a Basic Type

```
var array [3]string

array = [3]string{"string", 1, uuid.New()}

// Error

println(array)
```

In some situations, you might want to have a *mixed* array content in one variable. That is where an *empty interface* comes in handy. By setting the type `interface{}` in your declarations, you can mix different values to the array. It is like a *duck-typing type* of behavior in Ruby.

Mixed Array Contents Using Empty Interface

```
package main

import (
  "fmt"
  "github.com/google/uuid"
)

func main() {
  var array []interface{}

```

```
  array = []interface{}{"apple", 1, uuid.New()}

  fmt.Println(array[0]) // apple
  fmt.Println(array[1]) // 1
  fmt.Println(array[2]) // XXXXXXXX-XXXX-XXXX-XXXX
}
```

```
$ go run mixed-array.go

apple
1
ba178f1d-ad1e-4fe5-920f-c5b9148de01d
```

An array is a good way to store a known number of items or multiple items with the same data type on a variable that can be processed *iteratively*, by *batch* or by *bulk*.

Questions

1. What have we learned?
We can use an empty interface array type to mix different values to the array.

2. Why is this useful?
It is useful when you want to have a mixed array content in one variable.

3. How do you create an empty interface array type?
You can create an empty interface array type by setting the type interface{} in your declarations.

4. When not to use empty interface array type?
You should not use empty interface array type when you want to have a specific type of array content.

Chapter Questions

1. What is an array?
An array is a collection of variables of the same type.

2. What is a fixed-array?
A fixed-array is an array whose size is specified at compile time.

3. What is a sliced-array?
A slice is a reference to an array. A sliced-array is a slice that refers to a subset of an array.

4. What is the difference between a fixed-array and sliced-array?
A fixed-array is a contiguous block of memory that is allocated at the time of declaration. A sliced-array is a contiguous block of memory that is allocated at the time of slicing.

Navigating your Arrays

In Ruby, there's an `each` keyword that is used to iterate a list of items in an array, hash or sets, however how do you navigate your arrays in Go?

In this Chapter, we are going to discuss what are the ways to navigate or iterate you're arrays in Go.

In Go, the equivalent functionality similar to Ruby's `each` functionality is the `for` loop. To understand this further, let's discuss how a basic array is iterated in Ruby.

In this example, we create an array with `4` items, then using `each`, we create a block iterating over the `fruits` array, with a local variable `fruit`, then we print the value of `fruit` using `puts`.

```ruby
fruits = ["Apple", "Mango", "Orange", "Banana"]

fruits.each do |fruit|
  puts fruit
end
```

```
$ ruby each_array.rb

Apple
Mango
Orange
Banana
```

Given this example, we will discuss the different ways to use Golang's `for` loop to iterate an array.

Questions

1. What have we learned?
We learn how to iterate an array using `for` loop.

2. Why is this useful?
There are many reasons why we iterate an array in Go, for example, to print the values of an array, to modify the values of an array, to remove the values of an array, to add values to an array, etc.

3. Is there a different ways of iterating an array in Go?
Yes, there are different ways of iterating an array in Go, for example, using the `for` loop and using the `range` keyword.

C-style Semantic Form

Probably the most prominent for loop example, the C style semantic form. This initializes `i := 0` as the initial index, then adds a condition that if the index `i` is less than the length of fruits, `i < len(fruits)` the for loop will continually run, the last statement continually iterates one to `i` index then stores the sum to itself, via `i++`.

```
package main

import (
  "fmt"
)

func main() {
  fruits := [4]string{"Apple", "Mango", "Orange", "Banana"}

  for i := 0; i < len(fruits); i++ {
    fmt.Printf("%d - %s \n", i, fruits[i])
  }
}
```

```
$ go run c_style_for_loop.go

0 - Apple
1 - Mango
2 - Orange
3 - Banana
```

Questions

1. What have we learned?
The `for` loop is a control structure that allows us to iterate over a range of values.

2. Why do we use C-style semantic form in Go?
It is the most common form of `for` loop, and is used in many languages.

Value Semantic Form

The next example will use the value semantic form, using the value syntax from the for-loop. For loop can take 2 parameters, the index and the initialized range of the array as a value, `for index, value := range array { ... }`.

```go
package main

import (
  "fmt"
)

func main() {
  fruits := [4]string{"Apple", "Mango", "Orange", "Banana"}

  for i, fruit := range fruits {
    fmt.Println(i, fruit)
  }
}
```

```
$ go run value_semantic_form_for_loop.go

0 Apple
1 Mango
2 Orange
3 Banana
```

Questions

1. What have we learned?
We can use the for-loop and `range` *to iterate over the array.*

2. Why is iterating with a range useful in Go?
It is useful because it is easy to read and understand.

3. Why do we use for-loop with range together?
The for-loop is used to iterate over a range of values, and the range keyword is used to create a range of values.

Value Semantic Form with Muted Parameter

Parameters can also be muted, in the example below, we can mute index by naming it as an _-underscore variable.

```go
package main

import (
  "fmt"
)

func main() {
  fruits := [4]string{"Apple", "Mango", "Orange", "Banana"}

  for _, fruit := range fruits {
    fmt.Println(fruit)
  }
}
```

```
$ go run value_semantic_form_muted_for_loop.go

Apple
Mango
Orange
Banana
```

Questions

1. What have we learned?
We can use underscore variable to mute a parameter.

2. Why is this useful?
This is useful when you want to iterate over a slice of values, but you don't care about the index.

Index Semantic Form for Range

We can also skip the value parameter of the for loop (the 2nd parameter) and initialize it directly to get the array index.

```go
package main

import (
  "fmt"
)

func main() {
  fruits := [4]string{"Apple", "Mango", "Orange", "Banana"}

  for i := range fruits {
    fmt.Println(i, fruits[i])
  }
}
```

```
$ go run index_semantic_form_for_loop_range.go

0 Apple
1 Mango
2 Orange
3 Banana
```

Questions

1. What have we learned?
We can use the index semantic form for a range to skip the value parameter of the for loop (the 2nd parameter) and initialize it directly to get the array index.

2. Why is this important to iterate an index?
This is important because it allows us to write more concise code.

3. How is index in for-loop iteration used?
It is used to access the value at that index.

Value Semantic Form with Pointer Access

We can also access the pointer to its memory address directly, but directly accessing the memory pointer is a bad practice, especially when we are doing mutations to the same pointer.

```go
package main

import (
  "fmt"
)

func main() {
  fruits := [4]string{"Apple", "Mango", "Orange", "Banana"}

  for i := range &fruits {
    fmt.Println(fruits[i])
  }
}
```

```
$ go run value_semantic_form_with_pointer_semantic_access.go

Apple
Mango
Orange
Banana
```

Questions

1. What have we learned?
We can access the value of a variable directly, but directly accessing the memory pointer is a bad practice, especially when we are doing mutations to the same pointer.

2. Why it's not a good idea to directly mutate the pointer in for-loop?
Because the pointer may be changed by the loop.

3. Why is this a bad practice to mutate the pointer directly in Go?
Because Go is a garbage-collected language, and the garbage collector will not know that the pointer has been mutated.

Chapter Questions

1. How do you iterate an array in Go?
You can use a for loop to iterate over an array.

2. What are other ways to iterate an array in Go?
You can use a for loop with a range clause to iterate over an array.

3. Why is a bad practice to iterate and mutate over a pointer in Go?
Because it is unsafe.

Package Management

If you were to install a package in Ruby, you would use the RubyGems utility to install a package. If you were to manage your packages, you would add the package in a `Gemfile` and then use **Bundler**. If you were to use a `module` defined methods in Ruby, you would use `extend` or `include` inside a `class`. However, what is the equivalent functionality in Go?

Questions

1. What have we learned?
RubyGems is a package manager for Ruby. It is used to install packages and manage dependencies defined in a Gemfile

2. Why is it useful?
Package management is important because it allows you to easily manage your dependencies.

Sharing Go Packages

Go allows you to package your Go functions and distribute it remotely or throughout your programs. It works by assigning the package into a namespace that you can use throughout your Go program. Using import[13], we can call the package namespace and access its functions. This is comparable to Ruby's `include`. Also, when you do an `import`, Go will look into the `GOPATH`[2] and require the package.

`GOPATH` is the environment variable that Go uses to store packages, libraries, binaries and Golang compiler itself. Using `GOPATH`, Go will store remote packages to `$GOPATH/src`, i.e. `$GOPATH/src/example.com/foo`. It is also the path where Go will search for packages.

In the following example, we have imported several packages and then used their namespace to access its functions. We also demonstrated assigning a custom name for a package.

[13]https://golang.org/doc/code.html#ImportPaths

greeter.go

```go
package main

import (
  "time"

  "github.com/foo/greeter"
  weather "github.com/bar/climate"
)

func main() {
  greeter.SayHello("John")       // Hello John!

  weather.SayWeather(time.Now())
  // It will be sunny today. Overall it's
  // a good weather!
}
```

Noticed that we imported `time` and `greeter` and use its name as-is to access the package's functions in our main function. We also imported `climate` and assign a name `weather`. In Ruby, this works like a `module` that contains methods that you can include in your `class`. Below is the following Ruby example that we will rewrite in Go using `package` and `import`.

Module and Include

```ruby
module Coffee
  def types
    %w[
      espresso
      cappuccino
      macchiato
    ]
  end
end

class Drink
  include Coffee

  def menu
    puts types
  end
end

```

```
drink = Drink.new

drink.menu
```

To rewrite this example in Go, we will create the package named coffee and put it in `$GOHOME/src/local/coffee.go`, this will be the `module` block in our Ruby example.

```go
package coffee

func Types() []string {
  types := []string{"espresso", "cappuccino", "macchiato"}

  return types
}
```

Noticed that the name of the function `Types` is capitalized, that's because Go will make any capitalized functions, structs and variables public. Next, we can use `import` to include the coffee package in our program and access the functions inside the coffee package. This will be the `class`block in our Ruby example.

Importing Packages in Go

```go
package main

import (
	"fmt"

	"local/coffee"
)

func main() {
	menu()
}

func menu() {
	for _, val := range coffee.Types() {
		fmt.Println(val)
	}
}
```

In summary, the equivalent of Ruby's `module` and `include` in Go would be using Go's built-in `package` and put it in `GOPATH` and use `import` to load it. Now that we know how to declare package elements to public and make it available on other projects, how do we declare them in a package file similar to `Gemfile` and use a similar package manager tool like `bundler`?

The default package manager for Go is **Go Modules**, however there are still many applications that uses `dep`, because `dep` for many years, was the most popular package manager before. We will focus on two ways to manage packages in Go, using built-in **Go Modules** and `dep` for legacy applications.

Questions

1. What do we learn about distributing Go package?
We learned that Go uses GOPATH to store packages and that we can use import to load packages. We also learned that we can use package to declare public functions and structs and that we can use import to access them.

2. What is GOPATH?
GOPATH is the environment variable that Go uses to store packages, libraries, binaries and Golang compiler itself. Using GOPATH, Go will store remote packages to $GOPATH/src, i.e. $GOPATH/src/example.com/foo. It is also the path where Go will search for packages.

3. How do we declare functions, structs and variables public in Go?
In Go, any capitalized functions, structs and variables are public.

4. How do you import packages in Go?
You can use `import` to load packages in Go.

5. What is Go `dep`?
Go `dep` is a dependency management tool for Go. It is similar to Ruby's `bundler` and Node's `npm`. It is a tool that will help you manage your Go packages and their dependencies.

6. What is Go Modules?
Go Modules is the official package manager for Go. It is a built-in package manager that will be part of the Go language. It is developed by the Go team and was released in 1.11.

Package Management using Go Modules

Go has a built-in package manager that is available on Go version 1.11 and higher. Type `go version` to check if your Go environment supports it. To use Go Modules on Go 1.11, you need to export an environment variable `GO111MODULE` with a value of `on`.

```
export GO111MODULE=on
```

You can either set the value of the variable to either `on`, `off` or `auto`. If the environment variable is unset, the default value to use is `auto`, but it must meet two conditions: 1. The project should **not** be on `$GOPATH/src` directory, 2. The project root contains a `go.mod` file.

Go modules requires your public package to be available remotely as a remote resource since turning `on` the environment variable means we will not rely on `GOPATH`. When using Go Modules, the API would need to be public and must be available online. In our previous example, we have our public package `coffee`, that we put in `$GOHOME/src/local/coffee.go`. We will create a repo with this file in Github as `github.com/foo/coffee`, then tag the release as version `v0.0.1` and change the import declaration to point to the remote path.

Going back to our `menu.go` example we put it in the `~/code/menu` folder. Then we need to create the `go.mod` file at the root of your project folder. There are two ways of creating this file, *manually* and automatically *generate* by source.

Questions

1. What have we learned?
We learned how to create a Go modules and how to use it in a Go application.

2. Why is Go Modules important?
Go Modules is important because it allows us to create a public API for our package.

3. How do we enable Go Modules?
We enable Go Modules by setting the environment variable GO111MODULE to on.

4. Do we need GOPATH when using Go Modules?
No, we don't need GOPATH when using Go Modules.

5. What files do we need to create when using Go Modules?
We need to create a go.mod file at the root of our project.

Manual go.mod Generation

We will first discuss how to manually create the `go.mod` file. In the terminal, we will be creating the `go.mod` file on the root of our `~/code/menu` directory.

```
$ cd ~/code/menu

$ cat >go.mod <<EOF
module menu

require github.com/foo/coffee v0.0.1
EOF
```

Inside the `go.mod` we need to specify the required package with the version number. For multiple requires, you can use the `require(...)` statement. Versions can support pattern matching via semantic version prefix, for example, having the version to `v1` means it will use the latest available tagged version with that prefix. It can also support version comparison via `<`, `<=`, `>`, and `>=`. Lastly, you can just specify `latest` as the version and it will pick the latest tagged or latest commit. After the `go.mod` file is created, we can now run our project via `go run .`, which will automatically download the dependencies.

```
$ cd ~/code/menu
$ go run .
go: finding github.com/foo/coffee v0.0.1
go: downloading github.com/foo/coffee v0.0.1

espresso
cappuccino
macchiato
```

You can also download the dependencies manually via `go get -u`. If you will make the `~/code/menu` API public and Go Modules ready, you have to change the module line in the `go.mod` file to point to a remote resource. The `module menu` needs to be pointing to a remote resource, i.e. `module github.com/foo/menu`.

Questions

1. What is go.mod file?
It is a file that contains the list of dependencies of a project.

2. What do we need to do to create a go.mod file?
We need to specify the required packages with the version number.

3. Why is this important?
This is important because it will allow us to have a single source of truth for our dependencies.

Automatic go.mod Generation Through Source-Code

To generate a `go.mod` through its source-code, you can use the `go mod -sync` which will automatically create the `go.mod` based on the package `import`.

Please note that it would require you to have the project's VCS tool installed, not only `git` but some packages might be using `bzr`, so take that into consideration too.

However, if you only import part of a package i.e. `github.com/quux/foo/baz` and not requiring the repo package `github.com/quux/foo` in your packages, `github.com/quux/foo/baz` will be treated as a repository. To make your apps *module-aware* in the future, `baz` either needs to be:

1. in its separate repository, i.e. `github.com/quux/baz`
2. marked as `// indirect` in the `go.mod` file, for example:

In your project file `foo.go` you have an import statement:

```
import "github.com/quux/foo/baz"
```

Then if running `go mod -sync` gives you the following error:

```
$ go mod -sync

go: finding github.com/quux/foo/baz latest
go: import "github.com/quux/foo/baz" ->
  import "github.com/quux/foo/baz":
  cannot find module providing package github.com/quux/foo/baz
```

We have to manually add the failing package in the `go.mod` file, but it is required to fill-in the semantic-version, so we will get the latest version via `curl` and `jq`.

```
$ curl --silent "https://api.github.com/repos/**quux/foo**/releases/latest"\
| jq -r .tag_name

v0.0.1
```

Once you got the version tag, add it to `go.mod` file and comment it as `// indirect`.

```
module github.com/foo/bar

require (
  ...
  github.com/quux/foo v0.0.1 // indirect
)
```

Then run `go mod -sync` again.

If you find that you have too much `// indirect` comment, you might want to consider option A, since if the package is used on different responsibilities, it might be logical for the package to have its own repo as a reusable package.[4]

Questions

1. What have we learned?
We learn that we can use go mod -sync to generate a go.mod file based on the package import.

2. Why is this useful?
This is useful if you want to have a go.mod file for your project, but you don't want to manually create it.

3. What is an indirect comment in go.mod?
An indirect comment is a comment that is not related to the package, but to the repository.

4. How do we fix a failing indirect dependency in Go Modules?
We fix an indirect comment by adding the semantic-version tag to the comment.

Automatic go.mod Generation Through dep Package Manager

Another way to automatically create your `go.mod` file is using `dep` package manager which will read the `Gopkg.lock` file to generate the `go.mod` file.

If it's your first time to use `dep` then you need to run `dep init` which will do three things:

1. Automatically generate the `Gopkg.toml`
2. Maintain package version locking via a `Gopkg.lock` file
3. Downloads the dependencies to the `vendor` folder.

After you have all the `dep` files, you can run:

```
$ dep ensure

$ go get -u

go: creating new go.mod: module github.com/foo/coffee
go: copying requirements from Gopkg.lock
go: finding github.com/foo/coffee latest
go: downloading github.com/foo/coffee v0.0.1

$ go run .

espresso
cappuccino
macchiato
```

Questions

1. What have we learned?
We learned how to use go get to download packages from the internet.

2. Why is it useful?
It's useful because it downloads packages from the internet and saves them in your vendor directory.

3. What is Gopkg.toml?
Gopkg.toml is a configuration file for go get.

4. What is Gopkg.lock?
Gopkg.lock is a file that contains the list of all the packages that you have downloaded.

5. How do we generate a go.mod file from using go `dep`?
We use dep ensure to generate the go.mod file.

Refresh Go Modules

Currently, Go Modules will store the libraries under `$GOPATH/src/mod` and will make the directory unwritable, but in any case, you need to delete the directory recursively, you would need to make the directory writable first.

Questions

1. What have we learned?
We learn that Go Modules is a great tool to manage dependencies, but it is not perfect. Libraries under $GOPATH/src/mod is read-only by default, we need to make it writablel and delete the directory recursively to refresh our packages.

2. Why is it useful?
So that we can refresh our Go modules when fixing failed dependencies.

Package Management using Dep

If you don't support Go Modules, or just looking for a stable production-ready package manager, then use `dep`. Dep is the most popular and easier package manager for Go. To start using `dep`, just run `dep init` and it will traverse through the project source code to look for imports and generate the files `Gopkg.toml`, `Gopkg.lock` and downloads the project's packages into the `vendor` folder.

TOML[14] files are similar to YAML files which are used extensively in Ruby or Rails, but it emphasizes to focus on simplicity[15]. Inside the `Gopkg.toml` you'll notice that each the packages are inside a `[[constraint]]` header. For example:

```
[[constraint]]
  name = "github.com/foo/bar"
  version = "0.0.1"
```

In TOML, this is called an Array of Tables[16]. In Ruby and JSON this will mean:

```
{
  "constraint": [
    {
      "name": "github.com/foo/bar",
      "version": "0.0.1"
    }
  ]
}
```

Going back to the `Gopkg.toml` file, we can replace `version` with `branch` pointing to the branch name, for example:

```
[[constraint]]
  name = "github.com/foo/bar"
  branch = "baz"
```

And if you have a fork of the package that you choose to use instead of upstream, you can specify `source` pointing to your package repo.

```
[[constraint]]
  name = "github.com/foo/bar"
  branch = "baz"
  source = "github.com/quux/baz"
```

Then after modifications, you can run `dep ensure` to update the `vendor` folder.

[14] https://github.com/toml-lang/toml
[15] https://github.com/toml-lang/toml#comparison-with-other-formats
[16] https://github.com/toml-lang/toml#array-of-tables

Questions

1. What have we learned?
If you don't support Go Modules, then you can use dep.

2. Why is it useful?
Go dep can be used for backward compatibility and legacy applications.

3. How do you create a Gopkg.toml file?
You can use the dep init command to create a Gopkg.toml file.

4. Why don't we use go dep anymore?
Because Go Modules is now the default Go package manager, which makes go dep obsolete.

Chapter Questions

1. What is a package manager?
A package manager is a software tool that automates the process of installing, upgrading, configuring, and removing software packages.

2. What is Go Modules?
Go Modules is a new way of managing Go packages. It is a replacement for the old Go package management tool, `dep`.

3. What is `dep`?
`dep` *is a tool for managing dependencies of Go projects, but is now obsolete.*

4. How do you start using Go Modules?
You can start using Go Modules by installing Go 1.11 or later.

5. What is `GOHOME`?
`GOHOME` *is an environment variable that specifies the location of the Go installation directory.*

6. What is TOML? And why TOML was created?
TOML is a configuration file format. TOML was created because the TOML authors wanted a configuration file format that was easy to read due to obvious semantics.

Collection Functions in Go

If you are a speaker in a conference and you are about to proceed, you might ask, "*is everyone's cellphone turned off?*", and later in the middle of your talk, you ask "*anyone wants to take a coffee break?*" Would it be really cool to ask this on large sets of data and avoid micro-managing every stuff in an array? Ask a condition to all elements in an array in one go? This feature is baked in Ruby, which is called the *enumerable methods*. The equivalent of asking the first question in Ruby is the `all?` method, that will check if a certain condition for every element is met. The equivalent of asking the latter is the `any?` method, which would meet the condition if every element is met. What if we need to apply them in Go? What can we can learn from Ruby's enumerable methods and how to apply them in our Go program?

Since Go doesn't have built-in enumerable methods similar in Ruby, we can learn from those Ruby built-in methods and apply them in Go in similar scenarios we need it. In this article, we will be discussing how we can learn and introspect those Ruby enumerable methods we enjoyed and create analog methods that we can use in our Go programs. We will discuss each Ruby's enumerable methods and make an equivalent Go version. Then we will discuss more on how to use those methods in practical applications. Ruby has many helper methods that are baked-in to the language to conveniently process collections of data to the desired result. The first Ruby enumerable method we will discuss is the `all?` method.

Predicate Method `all?`

In the first paragraph, we asked "*Is everyone's cellphone turned off?*", and if everyone's cellphone is off we can proceed with our talk. Programmatically in Ruby, we are going to use `all?`. This method returns true if the resulting outcome of the array of conditions is `true`. Here's the Ruby example, wherein the `all?` helper is readily available in an array. The first condition is we ask if the length of each item in the array is greater than 4, which results to `true`.

```
["Apple", "Orange"].all? { |fruit| fruit.length > 4 }
=> true
```

The second condition results to `false`, because "Apple" does not meet the condition that it must have the length of greater than 5.

```
["Apple", "Orange"].all? { |fruit| fruit.length > 5 }
=> false
```

Applying this on Go, we would have to loop over the length of each element in the array, then we store it in a *boolean* array.

```go
var basket = make([]bool, len(fruits))

for i := 0; i < len(fruits); i++ {
  basket[i] = len(fruits[i]) > length
}
```

Next, is check if all item in the array returns `true`, otherwise `break` if `false`.

```go
var cond bool

for v := range basket {
  if cond = basket[v] == true; cond == false {
    break
  }
}

return cond
```

And here's my implementation of `all?` in an example. We created a `is_lengthy(int, []string)` that accepts the expected length and the array of strings.

```go
package main

import "fmt"

func is_lengthy(length int, fruits []string) bool {
  var all = make([]bool,len(fruits))
  var cond bool

  for i := 0; i < len(fruits); i++ {
    all[i] = len(fruits[i]) > length
  }

  for v := range all {
    if cond = all[v] == true; cond == false {
      break
    }
  }

  fmt.Println(cond)

  return cond
```

```
}

func main() {
  fruits := []string{"Apple", "Mango", "Orange", "Banana"}

  is_lengthy(4, fruits)
  is_lengthy(5, fruits)
}
```

```
$ go run is_lengthy.go

true
false
```

Questions

1. What have we learned?
We learned how to emulate Ruby's `all?` predicate method in Go.

2. Why is this useful?
This is useful when we want to check if all the conditions are true.

Predicate Method `any?`

Remember our *anyone wants to take a coffee break?* example, the `any?` method would meet the condition if every element is met. In Ruby, this is simply using the `any?` predicate method in an array which returns `true` if any of the resulting conditions are met, otherwise it would returns `false`.

```
fruits = ["Apple", "Mango", "Orange", "Banana"]

fruits.any? { |word| word.length < 4 } # false
fruits.any? { |word| word.length > 5 } # true
```

And here is the Golang implementations. In this example, we have an array of fruits, and we created an array `any` to store boolean values for each string of the array `fruits`. Then we iterate through each of the array values to check if there's any `true` value, then `break` if found.

```go
length := 4
var any [length]bool

fruits := [length]string{"Apple", "Mango", "Orange",
"Banana"}

for i := 0; i < len(fruits); i++ {
  any[i] = len(fruits[i]) > length
}

for v := range any {
  if any[v] == true {
    fmt.Println(true)
    break
  }
}
```

Questions

1. What have we learned?
We learn how to emulate Ruby's `any?` predicate method which returns true if any of the resulting conditions are met, otherwise it would returns false.

2. Why is it useful?
It is useful when we want to check if any of the elements in an array meets a certain condition.

Collect Enumerable Method

If you want to collect the values that satisfy an expression or perform an operation on the accumulator and return it as an array, then use *collect*. Collect iterates through an array and returns the resulting values as a new array. Here's an example of how to use collect in Ruby.

```ruby
(1..4).collect {|i| i*i }   #=> [1, 4, 9, 16]
(1..4).collect { "cat"  }   #=> ["cat", "cat", "cat", "cat"]
```

And here's the Golang analog. In the example below, we iterate through all items in the array, then records the result to the `collect` variable.

```
var collect [4]int
integers := [4]int{1, 2, 3, 4}

for i := 0; i < len(integers); i++ {
  collect[i] = integers[i] * integers[i]

  if i == len(integers) - 1 {
    fmt.Println(collect)
  }
}
```

Questions

1. What have we learned?
We learn how to emulate Ruby's built-in function called `collect`.

2. Why is it useful?
It's useful when you want to collect the values that satisfy an expression or perform an operation on the accumulator and return it as an array.

Cycle Enumerable Method

Pretty much straightforward enumerable method, that cycle through a list of an array for *n* times. In Ruby, it can take an argument on how many cycles it will run.

```
a = ["a", "b", "c"]
a.cycle(2) {|x| puts x }  # a, b, c, a, b, c.
```

And here's the Golang analog which iterates through the array until the given count.

```
count := 2
a := [3]string{"a", "b", "c"}

for x := 0; x < count; x++ {
  for i := 0; i < len(a); i++ {
    fmt.Println(a[i])
  }
}
```

Questions

1. What have we learned?
We learn to emulate Ruby's `cycle` method, which returns an enumerator object that cycles through the elements of a collection.

2. Why is it useful?
In Ruby, it is useful because it allows you to cycle through a collection without having to write a loop.

Detect Enumerable Method

If you want to return the first item that satisfies a condition, use Detect. In the following example, detect returns the processed value in the block.

```
(1..10).detect  {|i| i % 5 == 0 and i % 7 == 0 }   #=> nil
(1..100).detect {|i| i % 5 == 0 and i % 7 == 0 }   #=> 35
```

And here's the Golang analog which iterates and performs an expression on the current array item, prints the value then `break`.

```
for i := 1; i < 100; i++ {
  if ((i%5 == 0) && (i%7 == 0)) {
    fmt.Println(i)
    break
  }
}
```

Questions

1. What have we learned?
We learn how to emulate Ruby's `detect` method which returns the first non-nil value of the given block.

2. Why is it useful?
It is useful when you want to do something with the first non-nil value of the given block.

Drop Enumerable Method

Drop the first *n* elements of an array, and returns the remaining values.

```
a = [1, 2, 3, 4, 5, 0]
a.drop(3) #=> [4, 5, 0]
```

And here's our Golang implementation drop.

```
a := [...]int{1,2,3,4,5,0}
drop := a[3:]

fmt.Println(drop)
```

Questions

1. What have we learned?
We learn to emulate Ruby's `drop` method, which drops the last item from an array.

2. Why is it useful?
In Ruby, it's useful when you want to drop the last item from an array, but don't want to use a `pop` method.

Drop While Enumerable Method

Similar to drop, drop_while returns the values that satisfy an expression.

```
a = [1, 2, 3, 4, 5, 0]
a.drop_while {|i| i < 3 }   #=> [3, 4, 5, 0]
```

And in Golang, we add a conditional expression inside the range, cut the value then break if it satisfies the condition.

```
a := [...]int{1,2,3,4,5,0}

for _, i := range a {
  if a[i] < 3 {
    drop := a[2:]
    fmt.Println(drop)
    break
  }
}
```

I'm sure there are many things we can learn from Ruby enumerable, not only this can potentially be applied to Golang but to any programming language that does not have the built-in enumerable methods. If you want to use a community supported collection method, check out Go By Example's collection-functions[17]

Questions

1. What have we learned?
We learn to emulate Ruby's `drop_while` method which drops elements from the array until the passed block returns true.

2. Why is it useful?
It can be used to remove elements from the array until a certain condition is met.

Chapter Questions

1. What is an enumerable?
An enumerable is a collection of items that can be iterated over.

2. What is an enumerable method in Ruby?
An enumerable method is a method that returns an enumerator.

3. Why is enumerable methods in Ruby useful?
Enumerable methods are useful because they allow you to iterate over a collection.

4. What is a predicate method in Ruby?
A predicate method is a method that returns a boolean value.

[17] https://gobyexample.com/collection-functions

Organizing your Functions using Interface

If you need to organize and scope your functions into specific implementations, provides a better structure for your programs, and a cleaner way to manage your components, then use `interface`. Interfaces provide a way to organize multiple implementations and behaviors into common APIs. One of the advantages of using `interface` is that it can enforce cleaner component management, allowing multiple implementations and behaviors to managed easily. In other words, treat `interface` as your API contracts in between your implementations.

There are no direct analogs of `interface` in Ruby, but there are several Ruby implementations that the concept of `interface` has been used. One of those implementations is organizing your Ruby methods using modules. In the following example, we are going to organize our class methods into manageable modules. We have a module called `Basket` which contains our methods for adding and removing items in the `items` accessor.

```ruby
require 'set' # We can collect unique unordered values

module Basket
  attr_accessor :items

  def setup_basket
    @items = Set.new
  end

  def add_item(item)
    @items << item
  end

  def remove_item(item)
    @items.delete(item)
  end
end
```

Then we are going to create a class that subclass to `String`, and includes our `Basket` module which includes the functionality that we defined for adding and removing items.

```ruby
class ProduceBasket < String
  include Basket

  def initialize
    super
    setup_basket
  end
end

basket = ProduceBasket.new

basket.add_item('Apple')
basket.add_item('Broccolli')

puts basket.items
```

```
$ ruby produce_basket.rb

#<Set: {"Apple", "Broccolli"}>
```

Interface as a Self-Documenting API Reference

You can use interface as the primary self-documenting API reference for your Go functions. In the following example, we will initialize all our functions from our defined interface. The first thing we need to do is to declaratively define our functions in the interface.

```go
type ProduceBasket interface {
  AddItem(entry Produce)
  RemoveItem(entry Produce)
  Items()
}
```

Then we can add the actual functions.

```go
type Basket []Produce
type Produce string

func (p *Basket) AddItem(entry Produce) {
  *p = append(*p, entry)

  fmt.Printf("%s Added\n", entry)
}

func (p *Basket) RemoveItem(entry Produce) {
  s := *p

  for i, v := range s {
    if v == entry {
      s = append(s[:i], s[i+1:]...)
      break
    }
  }

  *p = s

  fmt.Printf("%s Removed\n", entry)
}

func (p Basket) Items() {
  for _, v := range p {
    fmt.Println(v)
  }
}
```

Then to use our functions from the defined interface, we need to initialize our interface just like any other variable.

```go
var basket ProduceBasket = &Basket{}
```

At this point, we can then freely use the functions that we declared in the interface.

```
basket.AddItem("Apple")     // Apple Added
basket.AddItem("Mango")     // Mango Added
basket.AddItem("Broccolli") // Broccolli Added
basket.RemoveItem("Mango")  // Mango Removed

basket.Items()              // Apple
                            // Broccolli
```

Here's the Go implementation of our Ruby example using `interface`.

```go
package main

import (
  "fmt"
)

type ProduceBasket interface {
  AddItem(entry Produce)
  RemoveItem(entry Produce)
  Items()
}

type Basket []Produce
type Produce string

func (p *Basket) AddItem(entry Produce) {
  *p = append(*p, entry)

  fmt.Printf("%s Added\n", entry)
}

func (p *Basket) RemoveItem(entry Produce) {
  s := *p

  for i, v := range s {
    if v == entry {
      s = append(s[:i], s[i+1:]...)
      break
    }
  }

  *p = s

```

```go
  fmt.Printf("%s Removed\n", entry)
}

func (p Basket) Items() {
  for _, v := range p {
    fmt.Println(v)
  }
}

func main() {
  var basket ProduceBasket = &Basket{}

  basket.AddItem("Apple")     // Apple Added
  basket.AddItem("Mango")     // Mango Added
  basket.AddItem("Broccolli") // Broccolli Added
  basket.RemoveItem("Mango")  // Mango Removed

  basket.Items()              // Apple
                              // Broccolli
}
```

```
$ go run interface.go

Apple Added
Mango Added
Broccolli Added
Mango Removed
Apple
Broccolli
```

Questions

1. What have we learned?
We can use interface as a self-documenting API reference.

2. Why is it useful?
Go's interface is useful because it allows us to define a contract for our functions and then use that contract to define our functions. This allows us to define our functions in a declarative way.

Interface as Type contract

With `interface`, you can explicitly enforce coherence in the type definitions. For example, we can create groups of arrays that only use the defined functions.

```
fruits := new(Basket)

fruits.AddItem("Apple")      // Apple Added
fruits.AddItem("Mango")      // Mango Added

veggies := new(Basket)

veggies.AddItem("Broccolli") // Broccolli Added

fruits.RemoveItem("Mango")   // Mango Removed

var items []ProduceBasket

items = append(items, fruits)
items = append(items, veggies)

for _, v := range items {
  v.Items()                  // Apple
                             // Broccolli
}
```

In our Ruby example, we intended that our class to be a superclass of `String`, which explicitly define our contracts for the method in the class are expected to be of type `String`. And in our Go `interface` example, we can use `interface` to declaratively define our contracts for our types.

```
package main

import (
  "fmt"
)

type ProduceBasket interface {
  AddItem(entry Produce)
  RemoveItem(entry Produce)
  Items()
}

```

```go
type Basket []Produce
type Produce string

func (p *Basket) AddItem(entry Produce) {
  *p = append(*p, entry)

  fmt.Printf("%s Added\n", entry)
}

func (p *Basket) RemoveItem(entry Produce) {
  s := *p

  for i, v := range s {
    if v == entry {
      s = append(s[:i], s[i+1:]...)
      break
    }
  }

  *p = s

  fmt.Printf("%s Removed\n", entry)
}

func (p Basket) Items() {
  for _, v := range p {
    fmt.Println(v)
  }
}

func main() {
  fruits := new(Basket)

  fruits.AddItem("Apple")
  fruits.AddItem("Mango")

  veggies := new(Basket)

  veggies.AddItem("Broccolli")
  fruits.RemoveItem("Mango")

  var items []ProduceBasket

```

```
  items = append(items, fruits)
  items = append(items, veggies)

  for _, v := range items {
    v.Items() // Apple
              // Broccolli
  }
}
```

```
$ go run produce_basket.go

Apple Added
Mango Added
Broccolli Added
Mango Removed
Apple
Broccolli
```

Questions

1. What have we learned?
We can use interface to define contracts for our types.

2. Why is this useful?
We can use interface to enforce coherence in the type definitions.

Satisfying Return Values

We can use Interface as a contract for your function return values.

```
type Produce interface {
  Flavour() string
  Kind() string
}
```

And if we want to create an array of Produce that contains specific types defined in a struct, we can use interface to satisfy our return values.

```go
type Item struct {
  Name    string
}

func (i Item) Flavour() string {
  flavour := GetFlavour(i.Name)

  return fmt.Sprintf("Item %s flavour is: %s\n", i.Name,
flavour)
}

func (i Item) Kind() string {
  kind := GetKind(i.Name)

  return fmt.Sprintf("Item %s is a %s\n", i.Name, kind)
}
```

And here is our complete implementation. We can use our implementations like so,

```go
package main

import (
  "fmt"
)

type Produce interface {
  Flavour() string
  Kind() string
}

type Item struct {
  Name string
}

func (i Item) Flavour() string {
  flavour := map[string]string{
    "Apple": `It's a little sour and bitter, mostly sweet,
not salty, juicy in general`,
    "Kale": `It boasts deep, earthy flavors that can range
from rich, herbaceous and slightly bitter`,
  }

  return fmt.Sprintf("Item %s flavour is: %s\n", i.Name, flavour[i.Name])
```

```
}

func (i Item) Kind() string {
  kind := map[string]string{
    "Apple": "Fruit",
    "Kale":  "Veggies",
  }

  return fmt.Sprintf("Item %s is a %s\n", i.Name,
kind[i.Name])
}

func main() {
  var basket []Produce

  apple := Item{Name: "Apple"}
  kale  := Item{Name: "Kale"}

  basket = append(basket, apple)
  basket = append(basket, kale)

  for _, v := range basket {
    fmt.Println(v.Flavour())
    fmt.Println(v.Kind())
  }
}
```

```
$ go run produce.go

Item Apple flavour is: It's a little sour and bitter,
mostly sweet, not salty, juicy in general

Item Apple is a Fruit

Item Kale flavour is: It boasts deep, earthy flavors that
can range from rich, herbaceous and slightly bitter

Item Kale is a Veggies
```

Questions

1. What have we learned?
We can use Interface as a contract for your function return values.

2. Why is this important?
It allows us to create a contract for our return values.

Chapter Questions

1. What is Go's interface?
Go's interface is a way to specify the behavior of a type in a way that is independent of the type's implementation.

2. Why is it Go's interface useful?
It allows a type to be used in multiple contexts without having to change its implementation.

3. How do we define an interface in Go?
We use the keyword "type" followed by the name of the interface and the type's methods and fields.

4. When not to use an interface in Go?
When the type's methods and fields are not independent of the type's implementation.

Glossary

`\N`	a Control Character Sequence Signifying a Newline
`!`	The Logical NOT Operator, can be Applied to a Boolean Which Converts the Operand Into it's Anti Equivalent Value, and Will Return False Values for Non-boolean.
`""`	An Empty String, if the String is a Type, Enclose the String in Between Quotes Specifying a String
`#<Set: {"Apple", "Broccoli"}>`	the Printed Value of a Ruby set in it's Object Form Containing two Items
`$`	The Dollar Sign, in Shell Scripts, it is a Special Character That Preceded With a Name Containing a Stored Value
`$GOPATH`	the Library Lookup Path for go Programs Using old Module System, This Form it is Being Executed in a Shell Script or Terminal
`%+v`	a Special `fmt` Printing Construct That Prints the Value and Field Names of a Struct.
`%s`	a Special `fmt` Printing Construct That Prints the Value as a String Type
`%`	The Percent Sign, in go, it is a Special Character That Preceded With a Type Signifying an Expected Return Value Passed as Parameters
`&&`	A Logical AND Operator
`&MapName{}`	a go map Without any Values Initialized. can be Readily Assigned to a Value Like a Normal Variable

`//`	a comment (computer programming) indicator in several programming languages including C, C++ and Java. the root directory Path in Domain-OS.
=	The equals sign or equality sign (=) is a mathematical symbol used to indicate equality.
Ability	Suitableness
Accessibility	Accessibility in the sense considered here refers to the design of products, devices, services, or environments so as to be usable by people with disabilities. The concept of accessible design and practice of accessible development ensures both "direct access" (i.e. unassisted) and "indirect access" meaning compatibility with a person's assistive technology (for example, computer screen readers).
Accessible	Easy of access or approach; approachable
Accessor (Mutator method)	In computer science, a mutator method is a method used to control changes to a variable. They are also widely known as setter methods. Often a setter is accompanied by a getter (also known as an accessor), which returns the value of the private member variable.
Accumulator (computing)	in a CPU, a processor register for storing intermediate results
Accumulator	One who, or that which, accumulates
Alias	a pseudonym; Otherwise; at another time; in other circumstances; otherwise called
Allocate	To set aside for a purpose
Allocations	plural of allocation

Allocation	The process or procedure for allocating things, especially money or other resources
All	Every individual or anything of the given class, with no exceptions (the noun or noun phrase denoting the class must be plural or uncountable)
Alternative	Relating to a choice between two or more possibilities
Analog	variable physical quantity that can be measured (such as the shadow of a sundial)
Anonymous	Lacking a name; not named, for example an animal not assigned to any species
Any	To even the slightest extent, at all
Append	To hang or attach to, as by a string, so that the thing is suspended. In general, to append is to join or add on to the end of something. In computer programming, append is the name of a procedure for concatenating (linked) lists or arrays in some high-level programming languages.
Application programming interface (API)	An application programming interface (API) is an interface or communication protocol between a client and a server intended to simplify the building of client-side software. It has been described as a “contract” between the client and the server, such that if the client makes a request in a specific format, it will always get a response in a specific format or initiate a defined action.
Applications	plural of application

Application	The act of applying or laying on, in a literal sense. In mathematics and computer science, apply is a function that applies functions to arguments. It is central to programming languages derived from lambda calculus, such as LISP and Scheme, and also in functional languages. It has a role in the study of the denotational semantics of computer programs, because it is a continuous function on complete partial orders.
Apply	To lay or place; to put (one thing to another)
Apps	plural of app
App	An application (program), especially a small one designed for a mobile device
Architecture	In computer engineering, computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems. Some definitions of architecture define it as describing the capabilities and programming model of a computer but not a particular implementation.
Argument (logic)	In logic and philosophy, an argument is a series of statements (in a natural language), called the premises or premisses (both spellings are acceptable), intended to determine the degree of truth of another statement, the conclusion. The logical form of an argument in a natural language can be represented in a symbolic formal language, and independently of natural language formally defined "arguments" can be made in math and computer science.
Argument	A fact or statement used to support a proposition; a reason

Array	An array is a systematic arrangement of similar objects, usually in rows and columns.
As-is	As is, when employed as a term with legal effect, is used to disclaim some implied warranties for an item being sold. Certain types of implied warranties must be specifically disclaimed, such as the implied warranty of title. “As is” denotes that the seller is selling, and the buyer is buying an item in whatever condition it presently exists, and that the buyer is accepting the item “with all faults”, whether or not immediately apparent.
Assert	To declare with assurance or plainly and strongly; to state positively
Assignment	a type of modification to a variable
Assign	To designate or set apart something for some purpose
Asterisk	In computer science, the asterisk is commonly used as a wildcard character, or to denote pointers, repetition, or multiplication.
Attribute (computing)	a specification that defines a property of an object, element, or file
Attribute (generic)	A characteristic or quality of a thing
Automatic	Capable of operating without external control or intervention
Bar	A metasyntactic variable used to stand for some unspecified entity, typically the second in a series after foo
Basic	Necessary, essential for life or some process
Batch	The quantity of bread or other baked goods baked at one time

Baz	A metasyntactic variable used to stand for some unspecified entity, typically the third in a series after foo and bar
Behavior	Human conduct relative to social norms
Binaries	plural of binary
Binary file	A binary file is a computer file that is not a text file. The term “binary file” is often used as a term meaning “non-text file”. Many binary file formats contain parts that can be interpreted as text; for example, some computer document files containing formatted text, such as older Microsoft Word document files, contain the text of the document but also contain formatting information in binary form.
Binary	Being in a state of one of two mutually exclusive conditions such as on or off, true or false, molten or frozen, presence or absence of a signal
Blank	White or pale; without colour
Block	A substantial, often approximately cuboid, piece of any substance
Bool (generic)	A Boolean variable, one whose value is either true or false
Boolean	Boolean data type, a form of data with only two possible values (usually “true” and “false”). Boolean expression, an expression in a programming language that produces a Boolean value when evaluated
Bool	Any kind of logic, function, expression, or theory based on the work of George Boole is considered Boolean.
Bound	simple past tense and past participle of bind

Box	Senses relating to a three-dimensional object or space. A cuboid space; a cuboid container, often with a hinged lid. A cuboid container and its contents; as much as fills such a container
Break	To separate into two or more pieces, to fracture or crack, by a process that cannot easily be reversed for reassembly
Bulk	Size, specifically, volume
Bundler	Bundler is a Popular Ruby Package Manager
Calculate	To determine the value of something or the solution to something by a mathematical process
Calculation	A calculation is a deliberate process that transforms one or more inputs into one or more results, with variable change. The term is used in a variety of senses, from the very definite arithmetical calculation of using an algorithm, to the vague heuristics of calculating a strategy in a competition, or calculating the chance of a successful relationship between two people.
Call	A telephone conversation, in computing, this can be a function call to other functions, or a remote procedure call to other services
Capability	The power or ability to generate an outcome. A capability is the ability to do things and to choose for a way of life according to one's personal values. As it applies to human capital, capability represents performing or achieving certain actions/ outcomes in terms of the intersection of capacity and ability.
Capacity	The ability to hold, receive or absorb
Case	An actual event, situation, or fact

Catch-all	A catch-all or catchall is a general term, or metaphoric dumping group, for a variety of similar words or meanings.
Caveats	plural of caveat
Caveat	A warning. Caveat refers to Latin phrases: Caveat lector ("let the reader beware"), Caveat emptor ("let the buyer beware"). Caveat venditor ("let the seller beware")
Channels	plural of channel
Channel	The physical confine of a river or slough, consisting of a bed and banks. In Go, channels are pipes that connect to a goroutine.
Chunks	plural of chunk
Chunk	A part of something that has been separated
Chunk (information)	a fragment of information used in many multimedia formats
Class	A group, collection, category or set sharing characteristics or attributes
Coherence	The quality of cohering, or being coherent; internal consistency
Collect	To gather together; amass
Combinations	refer to the combination of n things taken k at a time without repetition.
Combination	The act of combining, the state of being combined or the result of combining
Command	An order to do something
Comment	A spoken or written remark
Commit	To give in trust; to put into charge or keeping; to entrust; to consign; — used with to, unto

Common	Mutual; shared by more than one
Community	A group sharing a common understanding, and often the same language, law, manners, and/or tradition
Comparable	Able to be compared (to). Comparative, in grammar, a word that denotes the degree by which an entity has a property greater or less in extent than another
Comparison (computing)	The act of comparing or the state or process of being compared
Compiler	A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language).
Complete	To finish; to make done; to reach the end
Complexities	plural of complexity
Complexity	The state of being complex; intricacy; entanglement
Complex	Made up of multiple parts; composite; not simple
Component	A smaller, self-contained part of a larger entity. Often refers to a manufactured object that is part of a larger device
Concept (generic)	abstract and general idea; an abstraction
Concepts	Concepts are defined as abstract ideas or general notions that occur in the mind, in speech, or in thought.
Concurrency (computer science)	the property of program, algorithm, or problem decomposition into order-independent or partially-ordered units. In concurrent computing, the overlapping execution of multiple interacting computational task

Concurrent	Happening at the same time; simultaneous
Conditions	plural of condition
Constructs (information technology)	a collection of logic components forming an interactive agent or environment
Construct	Something constructed from parts
Consumption	The act of eating, drinking or using
Container	Someone who contains; something that contains
Contract	An agreement between two or more parties, to perform a specific job or work order, often temporary or of fixed duration and usually governed by a written agreement
Convert	into another form, substance, state, or product
Coordinate	Of the same rank; equal
Copy (computing)	Cut, copy and paste, a method of reproducing text or other data in computing
Copy	The result of copying; an identical duplicate of an original
Cost	Price. In production, research, retail, and accounting, a cost is the value of money that has been used up to produce something or deliver a service, and hence is not available for use anymore.
Count	To recite numbers in sequence
Current	the part of a fluid that moves continuously in a certain direction, especially short for ocean current

Custom	Frequent repetition of the same behavior; way of behavior common to many; ordinary manner; habitual practice; method of doing, living or behaving
Cut	To incise, to cut into the surface of something. To perform an incision on, for example with a knife. To divide with a knife, scissors, or another sharp instrument
Cycle	An interval of space or time in which one set of events or phenomena is completed
Database (generic)	organized information in a regular structure, usually but not necessarily in a machine-readable format accessible by a computer
Database	A database is an organized collection of data, generally stored and accessed electronically from a computer system. Where databases are more complex they are often developed using formal design and modeling techniques.
Data	Plural of datum. Data is a set of values of subjects with respect to qualitative or quantitative variables.
Datum	(plural: data) A measurement of something on a scale understood by both the recorder (a person or device) and the reader (another person or device). The scale is arbitrarily defined, such as from 1 to 10 by ones, 1 to 100 by 0.1, or simply true or false, on or off, yes, no, or maybe, etc.
Declaration	A written or oral indication of a fact, opinion, or belief
Declaratively	In a declarative way
Decouple	to unlink; to take apart
Decoupling	Decoupling usually refers to the ending, removal or reverse of coupling.

Default	The condition of failing to meet an obligation
Defines	Third-person singular simple present indicative form of define
Definition (generic)	A statement of the meaning of a word or word group or a sign or symbol (dictionary definitions)
Definitions	plural of definition
Definition	A definition is a statement of the meaning of a term (a word, phrase, or other set of symbols).
Degree	A step on a set of stairs; the rung of a ladder
Delete	To remove, get rid of or erase, especially written or printed material, or data on a computer or other device
Dependencies	plural of dependency
Dependency	A state of dependence; a refusal to exercise initiative
Dependent	Relying upon; depending upon
Dep	deposit
Desired	simple past tense and past participle of desire
Destination	The act of destining or appointing
Detail	Something small enough to escape casual notice
Detect	to discover or find by careful search, examination, or probing
Developer	A person or entity engaged in the creation or improvement of certain classes of products
Difference	The quality of being different

Directory (computing), or folder, a file system structure in which to store computer file	
Directory	A list of names, addresses etc, of specific classes of people or organizations, often in alphabetical order or in some classification
Direct	Proceeding without deviation or interruption
Discuss (conversation)	Conversation is interactive communication between two or more people.
Discuss	To converse or debate concerning a particular topic
Distinction	The recognition of difference. That which distinguishes; a single occurrence of a determining factor or feature, the fact of being divided; separation, discrimination
Distribute	To divide into portions and dispense
Documentation (general)	is a set of documents provided on paper, or online, or on digital or analog media, such as audio tape or CDs. Examples are user guides, white papers, on-line help, quick-reference guides. It is becoming less common to see paper (hard-copy) documentation. Documentation is distributed via websites, software products, and other on-line applications.
Documentation	Something transposed from a thought to a document; the written account of an idea
Dots	plural of dot
Dot	A small, round spot
Double	Made up of two matching or complementary elements

Download (generic)	A file transfer to the local computer
Download	In computer networks, download means to receive data from a remote system, typically a server such as a web server, an FTP server, an email server, or other similar systems. This contrasts with uploading, where data is sent to a remote server. A download is a file offered for downloading or that has been downloaded, or the process of receiving such a file.
Do	A syntactic marker in a question whose main verb is not another auxiliary verb or be
Drop	A small mass of liquid just large enough to hold its own weight via surface tension, usually one that falls from a source of liquid
Dup	to open (a door, gate etc.). In Ruby, `dup` will duplicate an object, useful when iterating through a hash to avoid creating a deep copy of the object
Dynamic	Changing; active; in motion
DynamoDB	Amazon DynamoDB is a fully managed proprietary NoSQL database service that supports key-value and document data structures and is offered by Amazon.com as part of the Amazon Web Services portfolio.
Each	All; every; qualifying a singular noun, indicating all examples of the thing so named seen as individual or separate items (compare every). In Ruby, `each` is the iterator keyword for all enumerable types.
Easy	Comfortable; at ease

Efficiency	Efficiency is the extent to which time or effort is well used for the intended task or purpose.
Efficient	making good, thorough, or careful use of resources; not consuming extra. Especially, making good use of time or energy
Element (generic)	An element is a part or aspect of something abstract, especially one that is essential or characteristic.
Element	One of the simplest or essential parts or principles of which anything consists, or upon which the constitution or fundamental powers of anything are based
Else	Other; in addition to previously mentioned items
Empty	Devoid of content; containing nothing or nobody; vacant
Emulate	To attempt to equal or be the same as
End	The terminal point of something in space or time
Enforcement	Enforcement is the process of ensuring compliance with laws, regulations, rules, standards, or social norms.
Enforce	To keep up, impose or bring into effect something, not necessarily by force
Ensure	To make a pledge to (someone); to promise, guarantee (someone of something); to assure

Entry point	In computer programming, an entry point is where the first instructions of a program are executed, and where the program has access to command line arguments. To start a program's execution, the loader or operating system passes control to its entry point. (During booting, the operating system itself is the program). This marks the transition from load time (and dynamic link time, if present) to run time.
Entry	The act of entering
Enumerable	Capable of being enumerated; countable
Enumeration	An enumeration is a complete, ordered listing of all the items in a collection. The term is commonly used in mathematics and computer science to refer to a listing of all of the elements of a set.
Environment	The surroundings of, and influences on, a particular item of interest
EOF	End-of-file
Equivalent	Similar or identical in value, meaning or effect; virtually equal
Error	An error (from the Latin error, meaning "wandering") is an action which is inaccurate or incorrect. In some usages, an error is synonymous with a mistake. In statistics, "error" refers to the difference between the value which has been computed and the correct value. An error could result in failure or in a deviation from the intended performance or behaviour.
Error (generic)	The state, quality, or condition of being wrong
Every	All of a countable group (considered individually), without exception

Exactly	without approximation; precisely
Example	Something that is representative of all such things in a group
Exchange	An act of exchanging or trading
Exists	Third-person singular simple present indicative form of exist
Expanded	simple past tense and past participle of expand
Expected	Anticipated; thought to be about to arrive or occur
Expects	Third-person singular simple present indicative form of expect
Expensive	Having a high price or cost
Experienced	Having experience and skill in a subject
Experience	Experience is the knowledge or mastery of an event or subject gained through involvement in or exposure to it.
Explain	To make plain, manifest, or intelligible; to clear of obscurity; to illustrate the meaning of
Explanation	An explanation is a set of statements usually constructed to describe a set of facts which clarifies the causes, context, and consequences of those facts.
Explicit	Very specific, clear, or detailed. Refers to something that is specific, clear, or detailed.
Export	of or relating to exportation or exports
Expression	The action of expressing thoughts, ideas, feelings, etc
Extend	To increase in extent
Extensively	In an extensive manner, widely

Failing	present participle of fail
Failure	Failure is the state or condition of not meeting a desirable or intended objective, and may be viewed as the opposite of success.
False	Untrue, not factual, factually incorrect
Fast	Firmly or securely fixed in place; stable
Feature	One’s structure or make-up: form, shape, bodily proportions
Field	A land area free of woodland, cities, and towns; open country
File folder	a kind of folder that holds loose sheets of paper
File	A collection of papers collated and archived together
Fill-in	In numerical analysis, the entries of a matrix which change from zero to a non-zero value in the execution of an algorithm
Finally	At the end or conclusion; ultimately
Find	To encounter or discover by accident; to happen upon
Finite (generic)	Having an end or limit; constrained by bounds; whose number of elements is a natural number
Finite	is the opposite of infinite.
First class	First class (or 1st class, Firstclass) generally implies a high level of service, importance or quality.
First	Preceding all others of a series or kind; the ordinal of one; earliest

Fixed (mathematics)	Fixed point, a point that is mapped to itself by the function
Fixed	simple past tense and past participle of fix
Flexible	Capable of being flexed or bent without breaking; able to be turned or twisted without breaking
Float	Of an object or substance, to be supported by a liquid of greater density than the object so as that part of the object or substance remains above the surface
Focus	A point at which reflected or refracted rays of light converge
Folder	An organizer that papers are kept in, usually with an index tab, to be stored as a single unit in a filing cabinet
Foobar	The terms foobar (), or foo and others are used as metasyntactic variables and placeholder names in computer programming or computer-related documentation. They have been used to name entities such as variables, functions, and commands whose exact identity is unimportant and serve only to demonstrate a concept.
Foo	A metasyntactic variable used to stand for some unspecified entity, typically the first in a series before bar
For loop	In computer science, a for-loop (or simply for loop) is a control flow statement for specifying iteration, which allows code to be executed repeatedly.
Form	is the shape, visual appearance, or configuration of an object. In a wider sense, the form is the way something is or happens.

Forwarded	simple past tense and past participle of forward
Freely	Free; frank
Front	The foremost side of something or the end that faces the direction it normally moves
Functionality	The ability to perform a task or function; that set of functions that something is able or equipped to perform
Function	What something does or is used for
Func	Abbreviation of function
Future	The time ahead; those moments yet to be experienced
General	Including or involving every part or member of a given or implied entity, whole etc.; as opposed to specific or particular
Generate	To bring into being; give rise to
Gettable	Able to be gotten
Getters	plural of getter
Getter	One who gets
Get	To obtain; to acquire
GitHub	is an American company that provides hosting for software development version control using Git. It is a subsidiary of Microsoft, which acquired the company in 2018 for $7.5 billion. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features.

Git	is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.
Go (programming language)	Go, also known as Golang, is a statically typed, compiled programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing, and CSP-style concurrency.
Golang	the Shortened Name for “Go Programming Language”
Group	A number of things or persons being in some relation to one another
Handled	simple past tense and past participle of handle
Hard	Having a severe property; presenting difficulty. Resistant to pressure
Hash	Food, especially meat and potatoes, chopped and mixed together. corn-beef hash
Header	layout
Helper	One who helps; an aide

Heredoc	In computing, a here document (here-document, here-text, heredoc, hereis, here-string or here-script) is a file literal or input stream literal: it is a section of a source code file that is treated as if it were a separate file. The term is also used for a form of multiline string literals that use similar syntax, preserving line breaks and other whitespace (including indentation) in the text.
High-level and low-level	High-level and low-level, as technical terms, are used to classify, describe and point to specific goals of a systematic operation; and are applied in a wide range of contexts, such as, for instance, in domains as widely varied as computer science and business administration.
High-level	describe those operations that are more abstract in nature; wherein the overall goals and systemic features are typically more concerned with the wider, macro system as a whole.
Hint	A clue
Hold	To grasp or grip
HTML	Hypertext Markup Language (HTML) is the standard markup language for documents designed to be displayed in a web browser. It can be assisted by technologies such as Cascading Style Sheets (CSS) and scripting languages such as JavaScript.
Idiomatically	In an idiomatic manner
If	Supposing that, assuming that, in the circumstances that; used to introduce a condition or choice
Implementation (generic)	is the realization of an application, or execution of a plan, idea, model, design, specification, standard, algorithm, or policy.

Implementation	The process of moving an idea from concept to reality. In business, engineering and other fields, implementation refers to the building process rather than the design process
Implicitly	In an implicit or implied manner
Import	Something brought in from an exterior source, especially for sale or trade
Include	part or member
Increment and decrement operators	Increment and decrement operators are unary operators that add or subtract one, to or from their operand, respectively. They are commonly implemented in imperative programming languages. C-like languages feature two versions (pre- and post-) of each operator with slightly different semantics.
Independent	Not dependent; not contingent or depending on something else; free
Index	An alphabetical listing of items and their location
Indirection	the ability to reference something in computer programming
Indirect	The opposite of direct. Not direct; roundabout
Information (generic)	can be thought of as the resolution of uncertainty; it is that which answers the question of “what an entity is” and thus defines both its essence and nature of its characteristics. It is associated with data, as data represents values attributed to parameters, and information is data in context and with meaning attached.
Information	That which resolves uncertainty; anything that answers the question of “what a given entity is”

Inheritance (object-oriented programming)	a way to compartmentalize and re-use computer code
Inheritance	The passing of title to an estate upon death
Inherit	To take possession of as a right (especially in Biblical translations)
Initialization (Programming)	In computer programming, initialization (or initialisation) is the assignment of an initial value for a data object or variable. The manner in which initialization is performed depends on programming language, as well as type, storage class, etc., of an object to be initialized. Programming constructs which perform initialization are typically called initializers and initializer lists. Initialization is distinct from (and preceded by) declaration, although the two can sometimes be conflated in practice. The complement of initialization is finalization, which is primarily used for objects, but not variables.
Initialization	In computing, formatting a storage medium like a hard disk or memory. Also, making sure a device is available to the operating system. Booting, a process that starts computer operating systems.
Initialization (generic)	The process of preparing something to begin
Initializer	One who, or that which, initializes
Initialize	To assign initial values to something
Initial	Chronologically first, early; of or pertaining to the beginning, cause or origin

Init	In Unix-based computer operating systems, init (short for initialization) is the first process started during booting of the computer system. Init is a daemon process that continues running until the system is shut down. It is the direct or indirect ancestor of all other processes and automatically adopts all orphaned processes. Init is started by the kernel during the booting process; a kernel panic will occur if the kernel is unable to start it. Init is typically assigned process identifier 1.
Inline	Alternative spelling of in-line
Inserted	simple past tense and past participle of insert
Install	To connect, set up or prepare something for use
Instance Variable	in Ruby, Instance Variable are Instantiated and Readily Available Attributes That can Store and Retrieve Values and is Accessible from Within the Class and Modules
Instance	Urgency of manner or words; an urgent request; insistence
Integer	A number that is not a fraction; an element of the infinite and numerable set
Integer (generic)	An integer (from the Latin integer meaning “whole”) is a number that can be written without a fractional component. For example, 21, 4, 0, and −2048 are integers, while 9.75, 5+1/2, and √2 are not.
Interact	To act upon each other. To engage in communication and other shared activities (with someone)
Interchange	to switch (each of two things)
Interface	The point of interconnection or contact between entities

Internally	In an internal manner; within or inside of external limits; in an inner part or situation
Introspection	is the examination of one's own conscious thoughts and feelings. In psychology, the process of introspection relies exclusively on observation of one's mental state, while in a spiritual context it may refer to the examination of one's soul. Introspection is closely related to human self-reflection and is contrasted with external observation.
Introspect	To engage in introspection
Invocations	plural of invocation
Invocation	An invocation (from the Latin verb invocare "to call on, invoke, to give")
Invocation (generic)	The act or form of calling for the assistance or presence of some superior being, especially prayer offered to a divine being
Invoke	for help, assistance or guidance
In	Used to indicate location, inclusion, or position within spatial, temporal or other limits. Contained by
Iterate	to perform or repeat an action on each item in a set
Iteration	Iteration is the repetition of a process in order to generate a (possibly unbounded) sequence of outcomes. The sequence will approach some end point or end value. Each repetition of the process is a single iteration, and the outcome of each iteration is then the starting point of the next iteration. In mathematics and computer science, iteration (along with the related technique of recursion) is a standard element of algorithms.

Iteratively	In an iterative manner; using iteration
Iterator (computer programming)	an iterator is an object that enables a programmer to traverse a container, particularly lists. Various types of iterators are often provided via a container's interface. Though the interface and semantics of a given iterator are fixed, iterators are often implemented in terms of the structures underlying a container implementation and are often tightly coupled to the container to enable the operational semantics of the iterator. An iterator performs traversal and also gives access to data elements in a container, but does not itself perform iteration (i.e., not without some significant liberty taken with that concept or with trivial use of the terminology). An iterator is behaviorally similar to a database cursor. Iterators date to the CLU programming language in 1974.
Iterator	One which iterates
JavaScript	often abbreviated as JS, is a high-level, interpreted scripting language that conforms to the ECMAScript specification. JavaScript has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions.

JSON	In computing, JavaScript Object Notation (JSON) (“Jason”) is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common data format, with a diverse range of applications, such as serving as replacement for XML in AJAX systems.JSON is a language-independent data format. It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data. The official Internet media type for JSON is application/json. JSON filenames use the extension .json.
Keyword	Any word used as the key to a code
Key	An object designed to open and close a lock
Kind	A type, race or category; a group of entities that have common characteristics such that they may be grouped together
Language	A body of words, and set of methods of combining them (called a grammar), understood by a community and used as a form of communication
Last	Final, ultimate, coming after all others of its kind
Lazily	In a lazy manner
Length	The distance measured along the longest dimension of an object
Less	To a smaller extent

Library	An institution which holds books and/or other forms of media for use by the public or qualified people often lending them out, as well as providing various other services for its users
Limitation	The act of limiting or the state of being limited
Line	A path through two or more points (compare ‘segment’); a continuous mark, including as made by a pen; any path, curved or straight.
List (computing)	A list is any enumeration of a set of items
List	A strip of fabric, especially from the edge of a piece of cloth
Literal (computer programming)	value that is fixed by its coding within the program using it
Literal	Exactly as stated; read or understood without additional interpretation; according to the letter or verbal expression; real; not figurative or metaphorical
Load	A burden; a weight to be carried
Local	From or in a nearby location
Lock	Something used for fastening, which can only be opened with a key or combination
Logical	In agreement with the principles of logic

Logic	Logic (from the Ancient Greek: λογική, romanized: logikḗ) is the systematic study of the form of valid inference, and the most general laws of truth. A valid inference is one where there is a specific relation of logical support between the assumptions of the inference and its conclusion. In ordinary discourse, inferences may be signified by words such as therefore, thus, hence, ergo, and so on.
Long	Having much distance from one terminating point on an object or an area to another terminating point
Loop	A length of thread, line or rope that is doubled over to make an opening
Low-level	describes more specific individual components of a systematic operation, focusing on the details of rudimentary micro functions rather than macro, complex processes. Low-level classification is typically more concerned with individual components within the system and how they operate.
Maintenance	Actions performed to keep some machine or system functioning or in service
Main	Of chief or leading importance; prime, principal
Make	To create. To build, construct, or produce
Manageable	Capable of being managed or controlled
Management (generic)	Administration; The use of limited resources combined with forecasting, planning, leadership and execution skills to achieve predetermined specific goals

Management	Management (or managing) is the administration of an organization, whether it is a business, a not-for-profit organization, or government body. Management includes the activities of setting the strategy of an organization and coordinating the efforts of its employees (or of volunteers) to accomplish its objectives through the application of available resources, such as financial, natural, technological, and human resources. The term “management” may also refer to those people who manage an organization.
Manager	A person whose job is to manage something, such as a business, a restaurant, or a sports team
Manage	To direct or be in charge of
Manual	A handbook
`Map[String]String`	a Declared map With Type and Value Both Assigned as `String`
Map (generic)	A map is a symbolic depiction emphasizing relationships between elements of some space, such as objects, regions, or themes.
Map	A visual representation of an area, whether real or imaginary
Marked	Having a visible or identifying mark
Matching	present participle of match
Maximize	to make as large as possible
Maximum	The highest limit
Max	maximum function
Mechanics	The branch of physics that deals with the action of forces on material objects with mass

Memory	The ability of the brain to record information or impressions with the facility of recalling them later at will
Menu	The details of the food to be served at a banquet; a bill of fare
Method	A process by which a task is completed; a way of doing something (followed by the adposition of, to or for before the purpose of the process):
Middle	A centre, midpoint
Mild	Gentle and not easily angered
Mismatch	To match unsuitably; to fail to match
Mixed	simple past tense and past participle of mix
Mix	To stir together
Model	A person who serves as a subject for artwork or fashion, usually in the medium of photography but also for painting or drawing
Modification	the act or result of modifying or the condition of being modified
Modified	changed; altered
Modify	To change part of
Modularized	Having or made up of modules
Modularizing	present participle of modularize
Module	A self-contained component of a system, often interchangeable, which has a well-defined interface to the other components

Mod	An unconventionally modern style of fashionable dress originating in England in the 1960s, characterized by ankle-length black trenchcoats and sunglasses
More	comparative degree of many: in greater number
Multiple	More than one (followed by plural)
Mutate	To undergo mutation
Mutations	plural of mutation
Mutation	Any alteration or change
Muted	simple past tense and past participle of mute
Namespace	A conceptual space that groups classes, identifiers, etc. to avoid conflicts with items in unrelated code that have the same names
Name	Any nounal word or phrase which indicates a particular person, place, class, or thing
Naming	The process of giving names to things
Navigate	To plan, control and record the position and course of a vehicle, ship, aircraft etc on a journey; to follow a planned course
Needless	Not needed; unnecessary
Nested	simple past tense and past participle of nest
New	Recently made, or created
Next	Nearest in place or position, having nothing similar intervening; adjoining. Most direct, or shortest or nearest in distance or time
Nil	Nothing; zero

Normally	Under normal conditions or circumstances; usually; most of the time
Notation (generic)	The act, process, method, or an instance of representing by a system or set of marks, signs, figures, or characters
Notation	In linguistics and semiotics, a notation is a system of graphics or symbols, characters and abbreviated expressions, used (for example) in artistic and scientific disciplines to represent technical facts and quantities by convention.
Note	A symbol or annotation. A mark or token by which a thing may be known; a visible sign; a character; a distinctive mark or feature; a characteristic quality.
Not	Negates the meaning of the modified verb
Now	Present; current
Number	An abstract entity used to describe quantity
O(n)	Big O Notation, Indicating a Relative Order of Growth and Complexity `n` That is Attached to the Function `O`
Object	A thing that has physical existence
Once	One and only one time
Open	Not closed; accessible; unimpeded
Operate	To perform a work or labour; to exert power or strength, physical or mechanical; to act
Operating (generic)	In operation; that operates
Operation	The method by which a device performs its function

Operator	One who operates
Option	One of a set of choices that can be made
Order	Arrangement, disposition, or sequence
Organization	The quality of being organized
Organize	To arrange in working order
Original	relating to the origin or beginning; preceding all others
Origin	The beginning of something
Or	Connects at least two alternative words, phrases, clauses, sentences, etc. each of which could make a passage true. In English, this is the "inclusive or." The "exclusive or" is formed by "either [...] or"
Outcome	That which is produced or occurs as a result of an event or process
Overall	All-encompassing, all around
Package	Something which is packed, a parcel, a box, an envelope
Parameter	A value kept constant during an experiment, equation, calculation or similar, but varied over other versions of the experiment, equation, calculation, etc
Partition	An action which divides a thing into parts, or separates one thing from another
Part	A portion; a component. A fraction of a whole.
Passing	present participle of pass
Pass	To change place. To move or be moved from one place to another

Path	A trail for the use of, or worn by, pedestrians
Pattern	Model, example. Something from which a copy is made; a model or outline. A text string containing wildcards, used for matching. A representative example.
Performance	The act of performing; carrying into execution or action; execution; achievement; accomplishment; representation by action
Performant	Capable of or characterized by an adequate or excellent level of performance or efficiency
Perform	To do something; to execute
Perspective	A view, vista or outlook
Pick	A tool used for digging; a pickaxe
Place	An area; somewhere within an area
Pointer	Anything that points or is used for pointing
Point	A discrete division of something. An individual element in a larger whole; a particular detail, thought, or quality
Pollution	The desecration of something holy or sacred; defilement, profanation
Popular	Common among the general public; generally accepted
Possible	Able but not certain to happen; neither inevitable nor impossible
Potentially	In a manner showing much potential; with the possibility of happening in a given way
Practice	Repetition of an activity to improve a skill

Predicate	which states something about the subject or the object of the sentence
Preferred	simple past tense and past participle of prefer
Prefix	Something placed before another. A morpheme added to the beginning of a word to modify its meaning, for example as, pre- in prefix, con- in conjure, re- in reheat, etc
Prepended	simple past tense and past participle of prepend
Prepending	present participle of prepend
Previous	Prior; occurring before something else, either in time or order
Primary	first or earliest in a group or series
Print	Of, relating to, or writing for printed publications
Private	Belonging to, concerning, or accessible only to an individual person or a specific group
Proceeded	simple past tense and past participle of proceed
Proceed	To move, pass, or go forward or onward; to advance; to carry on
Processed	That has completed a required process
Processing	The action of the verb to process
Process	A series of events which produce a result (the product)
Produce	To yield, make or manufacture; to generate
Programmatically	In a programmatic manner

Programming	The designing, scheduling or planning of a radio or television program/programme
Program	A set of structured activities
Project	A planned endeavor, usually with a specific goal and accomplished in several steps or stages
Properties	plural of property
Property	Something that is owned
Proper	Suitable. Suited or acceptable to the purpose or circumstances; fit, suitable
Provide	To make a living; earn money for necessities
Publicly	In public, openly, in an open and public manner
Public	Able to be seen or known by everyone; open to general view, happening without concealment
Puts	Third-person singular simple present indicative form of put
Put	To place something somewhere
Quux	Metasyntactic variable
Range	A line or series of mountains, buildings, etc
Readily	Without unwillingness or hesitation; showing readiness
Reading	present participle of read
Ready	Prepared for immediate action or use
Read	To look at and interpret letters or other information that is written
Reallocating	present participle of reallocate

Reasons	plural of reason
Reason	A cause: That which causes something: an efficient cause, a proximate cause
Reassigning	present participle of reassign
Receiver	A person who or thing that receives or is intended to receive something. More formal, usually referring to one who receives such things as an award or medal
Recognize	To match something or someone which one currently perceives to a memory of some previous encounter with the same entity
Recommended	simple past tense and past participle of recommend
Record	An item of information put into a temporary or permanent physical medium
Recursively	In a recursive way or manner. (Can we add an example for this sense?)
Reference	A relationship or relation (to something)
Referred	simple past tense and past participle of refer
Reflect	from a surface
Refresh	To renew or revitalize
Release	free (e.g. hostages, slaves, prisoners, caged animals, hooked or stuck mechanisms)
Rely	to trust; to have confidence in; to depend
Remotely	At a distance, far away
Remote	At a distance; disconnected

Removed	Separated in time, space, or degree
Replace	To restore to a former place, position, condition, etc.; to put back
Repository	A location for storage, often for safety or preservation. a storage location for files, such as downloadable software packages, or files in a source control system
Represent	To present again or anew; to present by means of something standing in the place of; to exhibit the counterpart or image of; to typify
Requirements	plural of requirement
Requirement	A necessity or prerequisite; something required or obligatory. Its adpositions are generally of in relation to who or what has given it, on in relation to whom or what it is given to, and for in relation to what is required.
Require	for something; to request
Resize	To alter the size of something
Resource	Something that one uses to achieve an objective, e.g. raw materials or personnel
Responsibilities	plural of responsibility
Responsibility	The state of being responsible, accountable, or answerable
Resulting	Of something that follows as the result of something else
Result	To proceed, spring up or rise, as a consequence, from facts, arguments, premises, combination of circumstances, consultation, thought or endeavor
Retrieval	the act of retrieving or something retrieved
Retrieve	To regain or get back something

Return	To come or go back (to a place or person)
Reusability	The property or degree of being reusable
Reusable	able to be used again; especially after salvaging or special treatment or processing
Reuse	The act of salvaging or in some manner restoring a discarded item to yield something usable
Review	A second or subsequent reading of a text or artifact in an attempt to gain new insights
Rewrite	The act of writing again or anew
Rewritten	past participle of rewrite
Root	The part of a plant, generally underground, that anchors and supports the plant body, absorbs and stores water and nutrients, and in some plants is able to perform vegetative reproduction
Ruby	Ruby is an interpreted, high-level, general-purpose programming language. It was designed and developed in the mid-1990s by Yukihiro “Matz” Matsumoto in Japan. Ruby is dynamically typed and uses garbage collection. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. According to the creator, Ruby was influenced by Perl, Smalltalk, Eiffel, Ada, Basic, and Lisp.
Rule	A regulation, law, guideline
Run	To move swiftly. To move forward quickly upon two feet by alternately making a short jump off either foot

Satisfy	To do enough for; to meet the needs of; to fulfill the wishes or requirements of
Scenarios	plural of scenario
Scenario	An outline of the plot of a dramatic or literary work
Scope	The breadth, depth or reach of a subject; a domain
Search	An attempt to find something
Section	A cutting; a part cut out from the rest of something
Semantic	Of or relating to semantics or the meanings of words
Sensor (detect)	In the broadest definition, a sensor is a device, module, machine, or subsystem whose purpose is to detect events or changes in its environment and send the information to other electronics, frequently a computer processor.
Separate	Apart from (the rest); not connected to or attached to (anything else)
Sequentially	In sequence, in order
Settable	Capable of being set
Setter	One who sets something, especially a typesetter
Setting	present participle of set
Setup	Equipment designed for a particular purpose; an apparatus
Set	down, to rest
Similar	Having traits or characteristics in common; alike, comparable
Simple	Uncomplicated; taken by itself, with nothing added

Simplicity	The state or quality of being simple. The quality or state of being unmixed or uncompounded
Simply	In a simple way or state; considered in or by itself; without addition; alone
Size	An assize
Skip	To move by hopping on alternate feet
Slice	That which is thin and broad
Slicing	The action of the verb to slice
Solution	A homogeneous mixture, which may be liquid, gas or solid, formed by dissolving one or more substances
Source	comes or is acquired
Specific	explicit or definite
Speedy	rapid; swift
Stable	animals with hoofs, especially horses
Standard	Falling within an accepted range of size, amount, power, quality, etc
Start	The beginning of an activity
Statement	A declaration or remark
Statically	In a static manner
Step	An advance or movement made from one foot to the other; a pace
Storage	The act of storing goods; the state of being stored
Store	A place where items may be accumulated or routinely kept
Straightforward	Proceeding in a straight course or manner; not deviating

Struct (Record)	In computer science, a record (also called a structure, struct, or compound data) is a basic data structure. A record is a collection of fields, possibly of different data types, typically in fixed number and sequence. The fields of a record may also be called members, particularly in object-oriented programming; fields may also be called elements, though these risk confusion with the elements of a collection. Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type. A record type is a data type that describes such values and variables.
Structure	A cohesive whole built up of distinct parts
Style	Senses relating to a thin, pointed object
Subclass	from which it inherits a base set of properties and methods
Subroutine	In computer programming, a subroutine is a sequence of program instructions that performs a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed.
Summary	Concise, brief or presented in a condensed form
Sum	A quantity obtained by addition or aggregation
Superclass	A high-level class that passes attributes and methods down the hierarchy to subclasses
Super	Of excellent quality, superfine
Support	To keep from falling

Syntax	A set of rules that govern how words are combined to form phrases and sentences
Tagged	Having a tag; labeled
Tag	A small label
Take	To get into one's hands, possession, or control, with or without force
Tap	A tapering cylindrical pin or peg used to stop the vent in a cask; a spigot
Terminal	A building in an airport where passengers transfer from ground transportation to the facilities that allow them to board airplanes
Test	A challenge, trial
Throughout	In every part of; all through
Through	From one side of an opening to the other
Throw	To hurl; to cause an object to move rapidly through the air
Time	The inevitable progression into the future with the passing of present events into the past
Tool	A mechanical device intended to make a task easier
Top	The highest or uppermost part of something
Traverse	A route used in mountaineering, specifically rock climbing, in which the descent occurs by a different route than the ascent
True	Conforming to the actual state of reality or fact; factually correct
Type	A grouping based on shared characteristics; a class

Underscore	An underline; a line drawn or printed beneath text; the character _
Unexpected	Not expected, anticipated or foreseen
Unique	Being the only one of its kind; unequaled, unparalleled or unmatched
Unordered	Not having been ordered
Unreadable	That cannot be read or is not easy to read
Unset	Not set; not fixed or appointed
Until	Up to the time of (something happening)
Unwritable	Not writable
Upcoming	Happening or appearing in the relatively near future
Update	An advisement providing more up-to-date information than currently known
Upstream	in a direction against the flow of a current or stream; upriver
Usage	The manner or the amount of using; use
Utility	The state or condition of being useful; usefulness
Value	that renders something desirable or valuable
Variable	Able to vary. In computing, this is a can be a value storage referencing a memory address
Variadic	Taking a variable number of arguments; especially, taking arbitrarily many arguments
Vendor	A person or a company that vends or sells
Version	A specific form or variation of something

Which	What, of those mentioned or implied
Wikipedia	Alternative letter-case form of Wikipedia
Workhorse	Most heavily used horse; A horse used primarily for manual labor; a draft horse
Writable	Capable of being written
Zero	The cardinal number occurring before one and that denotes no quantity or amount at all, represented in Arabic numerals as 0

Acknowledgments

First of all, I give all my thanks to my God, Jehovah, who have given me all this knowledge and who have been my close friend and father throughout my life.

I thought writing a book is simple and easy, but really, it's not. However, now that I have completed the book, it gives me a sense of reward. I thank my wife, Herald Jane, who believed in me, and comforted me in times of difficulties, and my son who gives me great joy, love and happiness. In 2018, I was hired by a company that uses Go. The first thing I did to learn it, is to relate to what I know in Ruby. And I thought it would be a good idea to document my learning process. And what started as a series of online articles, with just a slight possibility of having those online articles be published one day as a book, just happened.

Looking back, I really thank all those people who helped me in all those articles.

To my friend Bas Van Essen, who have helped me proof-read all those articles and make necessary corrections.

To my family. To my little brother, Jan Carlo, who provided me the initial book cover artwork.

Finally, all those people who have been part of this process of creating a book: I thank you all.

Credits

Credit goes to Wikimedia Foundation for the definitions in the Glossary words some taken from Wikipedia, and Wiktionary. Also Thanks to Wikimedia Foundation for the free yellow Duck artwork used in the book cover.